INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Guidelines for Testing Microservice-based Applications

DANIEL NUNES NOGUEIRA DA COSTA
outubro de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Guidelines for Testing Microservice-based Applications

Daniel Nunes Nogueira da Costa

Dissertation for obtaining the master's degree in Computer Engineering, area of specialization on Software Engineering

Adviser: Professor Nuno Silva

Porto, October 2022

Resumo

Há uma tendência no desenvolvimento de software de adotar uma arquitetura baseada em microserviços. Apesar de vários benefícios como maior modularização, escalabilidade e manutenibilidade, esta abordagem levanta outros desafios para a organização. Ao aplicar este padrão de arquitetura, a estratégia de teste precisa de ser ajustada. Um sistema baseado em microserviços é inerentemente distribuído e pressupõe que os vários serviços estejam em constante comunicação entre si, através de conexões de rede, para responder aos requisitos de negócio. Testar um microserviço por si só é mais fácil, pois este está naturalmente isolado do resto do sistema, mas a execução de testes de integração torna-se mais complexa. A utilização de microserviços também oferece várias opções sobre onde e o que testar.

Este trabalho tem o objetivo de estudar, comparar e sistematizar soluções e abordagens atuais para o desenvolvimento de testes em sistemas baseados em microserviços e propor um conjunto de diretrizes, métodos e boas práticas universais para facilitar o seu processo de testagem, ajudando as organizações a produzir testes com qualidade, mais valiosos e com menos custos.

De modo a perceber os problemas e desafios enfrentados a testar microserviços, um projeto em forma de prova de conceito (PoC) e utilizando uma arquitetura baseadas em microserviços foi planeado, desenhado e testes, relativos a alguns casos de uso foram investigados. Também foram sugeridos um conjunto de indicadores que pretendem medir a qualidade e valor da estratégia de testes. Para cada indicador foi proposto onde pode ser recolhido, um racional com a explicação do seu propósito e uma escala de medida.

Este trabalho concluiu que, apesar da existência de estratégias e *frameworks* de testes capazes de ajudar as organizações a testar as suas aplicações corretamente, é necessária a mentalidade certa para atingir uma estratégia de testes de qualidade. Deste modo, este trabalho propõe um conjunto de recomendações e boas práticas que promovem a mentalidade correta para desenhar e implementar testes sobre todas as camadas do sistema. São também sugeridos passos a seguir para definir e decompor cenários de teste, e soluções para os vários tipos de testes estudados.

Assim, este trabalho pode também ser considerado uma base de conhecimento na área de testes em microserviços e ajudar a acelerar a sua adoção.

Palavras-chave: testes; microserviços; recomendações para testes; aplicações baseadas em microserviços

Abstract

There's a trend in software development to adopt a microservice-based architecture. Despite several benefits such as increased modularization, scalability and maintainability, this approach brings other challenges to the table. When applying this architectural pattern, the testing strategy needs to be adapted. A microservice-based application presupposes that the various services that compose the system are communication with each other, across network boundaries, to fulfil business requirements and is inherently distributed. Testing a microservice by itself is easier, as it is naturally isolated from the rest of the system, but integration testing becomes more challenging. Microservices also offer several options about where and what to test.

This work focus on studying, comparing, and systemizing current solutions and approaches for testing in microservice-based systems and proposing a set of universal guidelines, methods, and best practices to facilitate microservice-based application testing, helping organizations produce more valuable and quality tests with less costs.

To understand the problems and challenges presented by microservices testing, a proof-of-concept (PoC) project, using a microservice-based architecture, was designed and tests for some use cases were explored. Furthermore, indicators to measure test quality and value were proposed, describing it source, rational and measurement scale.

This works concludes that, although many testing approaches and frameworks exist that can help organizations test their applications correctly, they need to be used with the right mindset. To achieved this, this work proposes a set of guidelines and best practices that promote the right mindset for designing and implementation tests at all system layers. It also proposes a workflow for test definition and decomposition, and solutions for the various studied testing types.

Keywords: testing; microservices; guidelines for testing; microservice-based applications

Agradecimentos

Um agradecimento a toda a gente que me acompanhou durante a minha jornada académica. Aos muitos amigos que me fizeram sentir em casa no ISEP. Aos docentes do Departamento de Engenharia Informática, especialmente aqueles que souberam transpor a sua paixão pelo que ensinam para os alunos. Em particular, agradeço ao Professor Nuno Silva, que me acompanhou como orientador deste trabalho, pela disponibilidade que sempre me mostrou quando necessitei e orientação ao longo deste projeto.

A título mais pessoal, um agradecimento especial à minha família por me darem tudo o que sempre precisei e mais, por me formarem em quem sou hoje e estarem sempre ao meu lado. Obrigado, Mãe, Pai e David! Outro agradecimento especial à minha namorada Beatriz por estar sempre presente e me apoiar, e, em especial, por me aturar durante toda esta jornada que não foi nada fácil. Um agradecimento também para os meus amigos mais próximos, onde se inclui os meus colegas de banda. A vida sem amigos (e música) não faz sentido.

Table of Contents

1 Introduction 1

1.1 Context 1

1.2 Problem 3

1.3 Objective 5

1.4 Approach 5

1.5 Results 6

1.6 Document structure 6

2 Microservice Architecture 8

2.1 Key Characteristics 8

2.2 Services anatomy 9

2.2.1 Layered architecture 9

2.2.2 Onion Architecture 10

2.2.3 MVC 11

2.3 Architectural patterns 12

2.3.1 Event Sourcing 12

2.3.2 CQRS 12

2.3.3 SAGA 12

2.4 Communication mechanisms 13

2.4.1 Technologies 13

2.4.2 Synchronous communication 14

2.4.3 Asynchronous communication 15

2.4.4 Challenges 15

2.5 Microservice lifecycle 16

3 State of the Art 19

3.1 Testing concepts 19

3.2 Testing typologies applied to Microservices 21

3.2.1 Unit tests 21

3.2.2 Integration tests 22

3.2.3 Component tests 23

3.2.4 Contract tests 24

3.2.5 End-to-end tests 25

3.2.6 Testing in production 26

3.2.7 Other testing types 26

3.3 Testing technologies 27

3.3.1 Unit testing frameworks 27

3.3.2 “Mocking” tools 30

3.3.3 Double HTTP server tools 31

3.3.4 Database testing tools 33

3.3.5 Asynchronous messaging testing 34
3.3.6 Event Stores testing 34
3.3.7 Contract testing tools 35
3.3.8 End-to-end testing tools 37
3.3.9 Containerization-based testing tools 37
3.4 Summary 38
4 Value analysis 39
4.1 Terminology 39
4.2 New Concept Development Model 39
4.2.1 Opportunity Identification 40
4.2.2 Opportunity analysis 41
4.2.3 Idea generation and enrichment 42
4.2.4 Idea selection 43
4.2.5 Concept Definition 47
4.3 Value proposition 48
5 Microservices testing scenarios analysis 49
5.1 Project overview 49
5.1.1 Business description 49
5.1.2 Requirements 50
5.2 System architecture 50
5.3 Testing scenarios 52
5.3.1 Search Products by Category 52
5.3.2 Integrating stock with a partner 58
6 Guidelines elaboration 61
6.1 Testing pillars 61
6.2 Testing types 63
6.2.1 End-to-end tests 63
6.2.2 Contract tests 66
6.2.3 Component tests 69
6.2.4 Integration tests 71
6.3 Scenarios decomposition strategies 73
7 Experimenting and Evaluation 75
7.1 Problem description 75
7.2 Objective description 75
7.3 Investigation hypothesis 76
7.4 Indicators and information sources identification 76
7.5 Evaluation methodology 77
7.5.1 Projected 77
7.5.2 Done 78

8 Conclusions 79

8.1 Objectives 79

8.2 Presented challenges 80

8.3 Future work 80

Table of Figures

Figure 1 - Costs of fixing bugs in specific development phases (Bueno, et al., 2018) 4
Figure 2 - Microservice internal structure (Clemson, 2014) 10
Figure 3 - Onion Architecture 11
Figure 4 - Microservice Development Lifecycle (Megargel, et al., 2020) 17
Figure 5 - Use build pipelines to get your software automatically and reliably into production (Vocke, 2018) 18
Figure 6 - Sociable vs Solitary Tests (Fowler, 2014) 19
Figure 7 – The Test Pyramid (Vocke, 2018) 21
Figure 8 - Broad vs narrow integration tests (Fowler, 2018) 23
Figure 9 - Typical sequence for using MockServer (MockServer, 2022) 33
Figure 10 - How Pact works (Pact, 2022) 36
Figure 11 - The new concept development (NCD) model (Koen, et al., 2002) 40
Figure 12 - AHP hierarchic decision tree 44
Figure 13 – Suggested system architecture 52
Figure 14 - Product boundary domain 53
Figure 15 - Search Products by Category Sequence Diagram 54
Figure 16 - Product service design 55
Figure 17 - Stock Integration with a partner - Sequence Diagram 59
Figure 18 - E2E tests example pipeline 65
Figure 19 – Consumer-driven contract testing using a shared git repository from the consumer perspective 68
Figure 20 – Provider-driven contract testing using a Stub Storage from the consumer perspective 69
Figure 21 - Component tests approach 1 example sequence diagram 70
Figure 22 - Component tests approach 2 example diagram 71
Figure 23 - Integration test dependencies decision flowchart 73
Figure 24 - Testing scenarios decomposition 74

Table of Tables

Table 1 - Saaty AHP ratio scale (Nicola, 2018). 44
Table 2 - AHP criteria comparison 44
Table 3 - Normalized AHP matrix 45
Table 4 - AHP criteria "Estimated effort" alternatives comparison matrix with priority-vector 46
Table 5 - AHP criteria "Author personal experience" alternatives comparison matrix with priority-vector 46
Table 6 - AHP criteria "Exists in market" alternatives comparison matrix with priority-vector 46
Table 7 - AHP criteria "Technical success probability" alternatives comparison matrix with priority-vector 47
Table 8 - Search Products use case breakdown 54
Table 9 - Search Product Contract tests 56
Table 10 - Testing types characteristics 63
Table 11 - Test specific indicators 76
Table 12 - Global test strategy indicators 77

Acronyms and Symbols

List of Acronyms

SOA	Service-Oriented Architecture
CICD	Continuous Integration and Continuous Deployment
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
XML	Extensible Markup Language
ORM	Object Relational Mapping
API	Application Programming Interface
E2E	End-to-end
JDBC	Java Data Base Connector
SQL	Structured Query Language
CQRS	Command and Query Responsibility Segregation
AHP	Analytic Hierarchy Process
AMQP	Advanced Message Queuing Protocol
C2C	Consumer-to-Consumer
B2B	Business-to-Business
QA	Quality Assurance
DBMS	Database Management System

List of Symbols

$	Payload

1 Introduction

This chapter aims at introducing the work described in this document. It describes the context, the problem, the objective, and a summary of the adopted approach. Lastly, it presents the structure of the rest of the document.

1.1 Context

The software development world is in constant evolution. Over the years, there has been a tendency for service-based applications to shift from Monolithic and traditional Service-Oriented Architectures (SOA) to a Microservice Architecture (Clemson, 2014), either when building an application from scratch, in consequence of the requirements, or, most commonly, by a migration process, to overcome limitations in the initial system.

The Monolithic Architecture refers to "*a software application whose modules cannot be executed independently.*" (Dragoni, et al., 2017), a unit of software that contains all the necessary components of the application. There are several positive aspects to this approach (Brandon, 2012):

- Low interactions complexity between components.
- Ease of aggregating all the code base in one project.
- Simpler deployment and scaling processes.

Despite, there are also several limitations: (Richardson, 2018)

- Large code base can be difficult to understand and modify.
- Continuous deployment is difficult - to update one part of the system you must redeploy the entire system. This increases the time to delivery and, consequently decrease the capability to answer market needs and create value.
- Scaling is only possible in one dimension – cannot scale each component individually, so creating more instances of the application to answer performance needed in one

component will lead to all components being replicated and unnecessary resource consumption.

- It is hard to parallelize development throughout several teams – prevents teams from working independently, as they share the same code base and need to sync developments and redeploys.

The SOA methodology addresses those limitations by breaking an application into smaller, more manageable components designated as services. "*The SOA provides a set of design principles that provide structure for the creation of these components and their aggregation into fully-featured applications*." (Vieira, 2020). A service, in this context, represents a discrete business activity and provides an interface that acts as service contract to its consumers – other services of the system. This approach promotes loose coupling between components and results in reduced dependencies throughout the application (IBM Cloud Education, 2021).

Some authors defend that Microservices can be seen as an evolution of SOA (Vieira, 2020) or consider "*microservices to be one form of SOA, perhaps service orientation done right*" (Lewis & Fowler, 2014). The two approaches aim to promote similar concepts but enforce them in different ways. While SOA is an integration architectural style and defines how the different services should communicate and interact with each other, Microservices are focused on the application architecture and do not force communication strategies (IBM Cloud Education, 2021).

The Microservice Architecture defines an application as a group of smaller, more focused "micro" services, that can be developed, maintained, and deployed independently of each other. It enables organizations to answer increasingly demanding requirements of availability scalability, maintainability and reusability while also allowing reduced infrastructure costs, technology stack heterogeneity, development parallelization and true continuous deployment and delivery independency (Dehghani, 2018).

Microservices architecture emerged and gained steam with the rises of virtualization, cloud computing, Agile development practices, and DevOps (IBM Cloud Education, 2021). A mature infrastructure is essential for taking advantage of Microservices. Having an automated build pipeline that can run automated tests and deploys provides teams with Continuous Integration and Continuous Delivery (CICD) mechanisms and enables fast feedback and fewer time to delivery (Lewis & Fowler, 2014).

To solve the challenges and problems faced by legacy applications, different companies are migrating to a Microservice architecture (Neves, 2019). However, the Microservice Architecture brings other challenges to the table. The natural data decentralization, definition of domain boundaries, use of patterns as CQRS (Command Query Responsibility Segregation), Saga and Event Sourcing creates additional network communications between system components (microservices) and creates dependencies between internal and external systems (Clemson, 2014).

1.2 Problem

When applying this architectural pattern, the testing strategy needs to be reconsidered - especially when migrating from a monolithic-like design (Clemson, 2014). While a single microservice can still be tested very similarly to a monolithic system, and even simpler, the microservice architecture presupposes that the various microservices that compose the system are working together, exchanging messages, and interacting in a distributed and decentralized manner (Neves, 2019). As a result, whenever a microservice changes, it needs to be tested together with other microservices it interacts with. This creates a lot of dependencies between the microservices, which become a dependency for the tests themselves (Neves, 2019).

Thus, testing a microservice by itself becomes easier, as it is isolated from the other parts of the system, but integration testing becomes more challenging, especially when there are many connections with the other services (Dragoni, et al., 2017). Saša Baškarada defends that "*testing of distributed systems built using microservice architecture is inherently more difficult than testing of centralized monoliths*". He highlights that a distributed system is inherently less stable than a non-distributed one and testing all its failure scenarios, comprehensively, is practically impossible. He argues that non-functional failures may be difficult to automate in a DevOps microservice architecture, and even if automated, can run very slowly (Baškarada, et al., 2018). João Neves identified "Automation support for testing" as one of the main challenges of designing a new system when migrating from a monolithic architecture (Neves, 2019).

Microservices also offer several options about where and what to test (Clemson, 2014). Tests can be performed at the class or method level (unit) and exercise all the possible success and failure scenarios. Tests can also be executed against a fully deployed production-like system and exercise business use cases – those are typically the ones that provide more value but require more effort. What factors should be taken into consideration when making this decision?

Kent Beck says the cost of fixing a defect increases over the software lifecycle - "*the sooner you test after the creation of an error, the greater your chance of finding the error and the less it costs to find and fix the error*" (Beck, 2004). Figure 1 correlates the percentage of bugs found and their cost to repair with the several development phases of a software lifecycle (Bueno, et al., 2018).

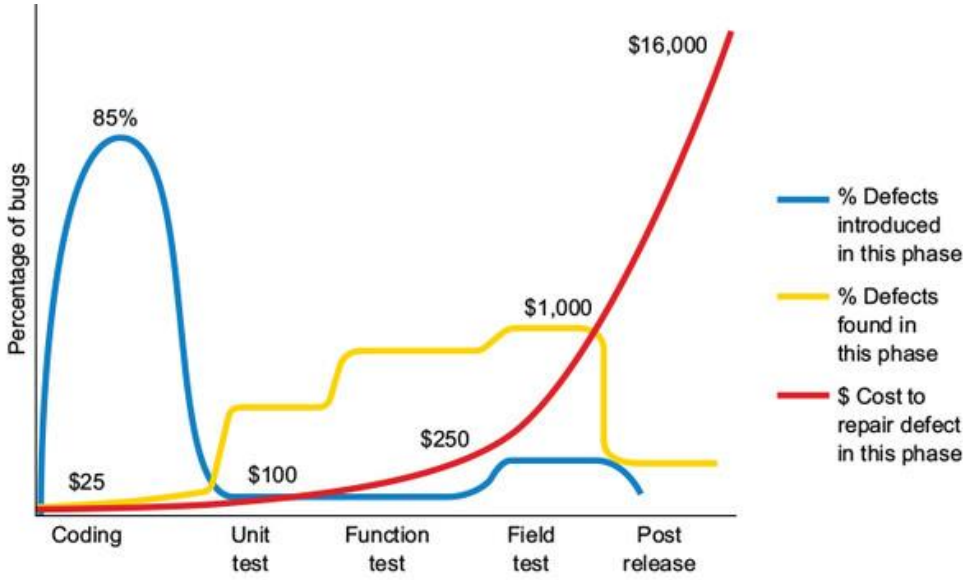

Figure 1 - Costs of fixing bugs in specific development phases (Bueno, et al., 2018)

From Figure 1 we can conclude that it is beneficial to detect defects as early as possible. In Microservices, addressing internal defects in the microservice logic early on can be achieved thought traditional testing techniques, such as Unit testing, but testing for such defects in the integration logic early on is a challenging task. While in a monolithic approach an incorrect assumption about another module would usually be caught at compilation time, in a Microservice approach this defect would not be immediately spotted – as each microservice compiles independently - and could, potentially, just be noticed in the production environment, when services start failing (Bueno, et al., 2018). This raises two questions: when should integration code (microservices) be tested? How can we guarantee excessive testing is not being performed?

While it is beneficial to find a defect early on, testing all possible scenarios to exhaustion can lead to high coupling between the test suite and the code, and testing for all possible scenarios at every layer will certainly lead to test duplicating (Vocke, 2018). Ideally, a test suite should be (Vocke, 2018):

- low coupled with the code – allowing for code refactoring with minimum test refactoring;
- automatable – automating test runs will result in tests being run more frequently and so, defects being found sooner; test results can be used as a promotion requirement;
- reliable – always fail for the same reason, i.e., don't fail for unrelated reasons to the test (e.g., dependency on external data, network failures);
- fast – running fast allows developers to run tests more frequently and so find defects sooner;
- maintainable – tests should bring value and not extra effort and time needs.

There are few or no microservice-based testing approaches and patterns that address these concerns.

1.3 Objective

This work focus on studying, analyzing, and systemizing current solutions and approaches for developing tests in microservice-based systems, highlighting its advantages and disadvantages, and comparing these approaches according to several criteria, such as number of tests produced by system layer, coupling with the code, capacity to be automated, technologic heterogeneous and maintainability. It should also state why the testing of such applications is considered a challenge.

The insights of such analysis should then be used to build a set of guidelines, approaches, and best practices for testing microservice-based applications with more quality and less costs, independently of their technology. This document should contain detailed information on how organizations can test their microservice-based applications, consisting of recommendations, possible approaches, focus points and examples. It should be a Kickstarter for the adoption of automated tests across the full system spectrum.

1.4 Approach

To understand the problems and challenges presented by microservices testing, the microservices architectural style and patterns will be analyzed. To reach conclusions about the current state of testing on microservice based systems, an analysis of current testing methodologies should be done, highlighting, and systemizing its best practices and limitations. The most used testing frameworks in microservices should also be analysed, as well as existing microservice-based systems that are recognized as references (e.g., Netflix, Spotify).

A testing approach, compose of guidelines, methods, and best practices, should be designed, and evaluated according to a set of evaluation methods. The evaluation should measure the benefits of applying such testing solution by collecting a set of metrics before and after its application. Examples of such metrics could be value (number of real vs false defects), reliability (fail always for the same reason), number of defects found vs number detects on production, cost by defect, time to detect defect (in project lifecycle), critically of defects found, test coverage, ratio between number of tests/execution time or test maintainability.

Ideally, a project that is already in production would be used to demonstrate the appliance of such testing patterns and measure its benefits. As this is not possible, proof-of-concepts (PoC) projects that use a microservice-based architecture should be designed, and possibly developed, to help gather knowledge about the challenges faced. Testing scenarios should be planned and solutions that satisfy the current test requirements suggested.

1.5 Results

The work contained in this document contributes to further developing methods and strategies to mitigate the challenges of testing microservice-based applications, especially broader scope use cases. Many testing approaches and frameworks exist that can help organizations test their applications correctly, they just need to be used with the right mindset. This is where this works ended up providing more value, as it promotes the right mindset for designing and implementation tests at all system layers. It proposes a workflow for test definition and decomposition, solutions for the various testing types, and indicates a methodology to collect metrics over the testing strategy evolution.

Additionally, this work resulted in a comprehensive guide to understand Microservices and their challenges, and in a knowledge base about several testing types and their role in the testing strategy. Furthermore, this work can be considered a kick starter for an organization who intends to adopt microservices, or that has already adopted it but is struggling with tests.

1.6 Document structure

The rest of this document is structured as follows. Chapter 2 presents the Microservice architecture and its characteristics, patterns and challenges. Chapter 3 consists of the state of the art, presenting a broad view over the existing Microservices testing strategies and technologies that aux those tests. Chapter 4 presents the value analysis made for this work. Chapter 5 presents an example Microservice project, its scope and possible testing scenarios. Chapter 6 contains the microservice-based application testing guidelines and its rationality. Chapter 7 is responsible for presenting how this work will be evaluated. Chapter 8 contains the conclusions that were possible to take from this work. At the end of the document, references details are provided.

2 Microservice Architecture

A microservice can be defined as an independently deployable component that is modelled around a business domain. Although, it isn't a microservice by itself that presents value, but its interaction with other microservices. As so, a microservice architecture can be described as "a distributed application where all its modules are microservices" (Dragoni, et al., 2017). Furthermore, "a microservices oriented system consists of a distributed application in which its behavior depends on the communication, composition, and coordination of its microservices via messages" (Neves, 2019).

This chapter focuses on characterizing a Microservice Architecture, understanding its principles, architectural patterns, and context. Those provide valuable information about the environment surrounding the testing strategy.

2.1 Key Characteristics

Martin Fowler defines the most important characteristics of Microservices as (Lewis & Fowler, 2014):

- Componentization via services – the software is broken down into services that run independently, share no resources, and communicate via remote calls, as messaging and APIs. Those should be highly maintainable and testable, loosely coupled and independently deployable.
- Organized around business capabilities – services are born from business capabilities decomposition, creating clear boundaries between business contexts and, consequently, between the services themselves.
- Products not Projects – a team should own a product during its full lifecycle, taking responsibility for development, maintenance, and production support.

- Smart endpoints and dumb pipes – microservices communicate using HTTP and light-weighted messaging, avoiding complex middleware, and applying business rules on the endpoints themselves.
- Decentralized Governance - possibility to use different technologies to build each service, using "the right tool for the job".
- Decentralized Data Management – microservices are the owners of their own data and data model, so those are decentralized and not shared between services; they typically have an independent database.
- Infrastructure Automation – microservices take advantage of Continuous Integration and Continuous Delivery principles, as automated build, tests, and deployment pipelines to provide fast feedback and promotion.
- Design for failure - applications need to be designed so that they can tolerate the failure of services at any time and automatically recover and restore the system.
- Evolutionary Design – microservices are ready to be easily modified and upgraded, providing a fast change mechanism to the organization.

"*The microservices approach is to organise cross-functional teams around services, which in turn are organised around business capabilities*" (Dragoni, et al., 2017). So, typically, a team focuses on a group of microservices which represents a business unit, and their responsibilities may include its design, maintenance, governance, and monitoring. If so, it's important that this team is cross-functional. This concept propagates into testing as well - each team is responsible for designing and implementing a testing strategy that guarantees its correct behaviour.

2.2 Services anatomy

"*Microservices architecture defines that every microservice should have specific and individual responsibility and are generally small and simple systems without significant complexity"* (Neves, 2019). This section focuses on exploring some architectural styles used to design and implement each microservice.

2.2.1 Layered architecture

Regarding a microservice internal structure, Toby Clemson defends they often follow remarkably similar structures, adjusted accordingly to the requirements. As displayed on Figure 2 - Microservice internal structure , he highlights the following layers (Clemson, 2014):

- Resources – maps between the resources exposed by the service and the objects that represents the domain; a resource receives a request and is responsible for validating the request/response accordingly to the business transaction and call the domain/service layer to handle the request. Includes Controllers.
- Service – coordinates operations across multiple domains.
- Domain – represents the business domains and contains the business rules and logic.

- Repositories – collections of domain entities that are often persisted on a database.
- Persistence Layer/ ORM – logic to persist the objects from the domain between requests; It Is usually encapsulated in objects utilized by the Repositories layer and takes advantage of Object Relation Mapping (ORM) or other data mappers to map the domain objects into persistable database objects.
- Gateways – encapsulates the necessary logic to communicate with an external service, as marshalling rules between request/response payloads and domain objects.
- HTTP Client – responsible for handling the communication protocol request-response cycle.

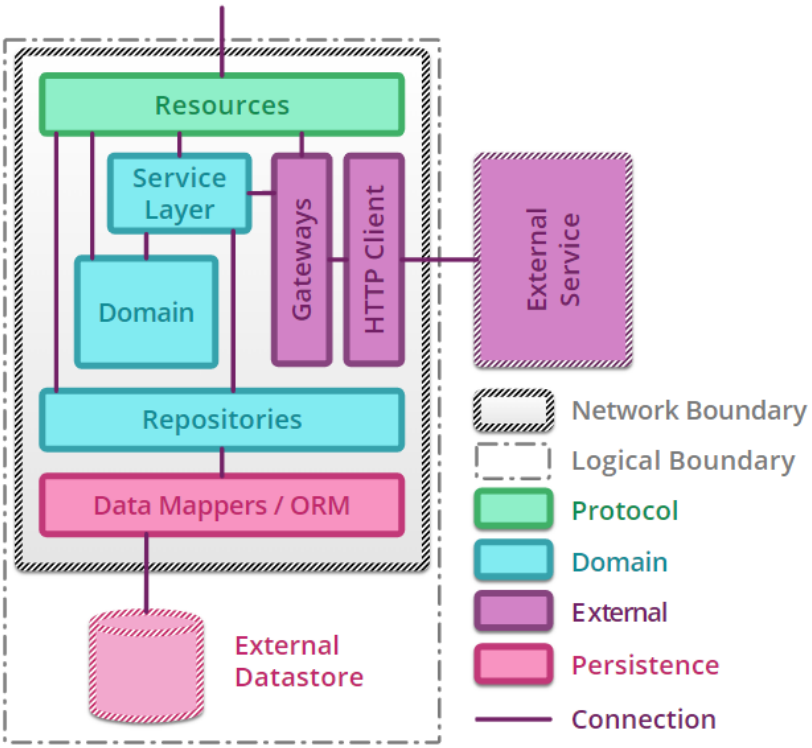

Figure 2 - Microservice internal structure (Clemson, 2014)

In Figure 2 - Microservice internal structure , the bigger grey box represents a microservice, and the boxes inside it its internal architectural layers. The connections represent the communication paths between the system layers. From those connections it's possible to understand the internal dependencies of such layers - how they interact and exchange information.

2.2.2 Onion Architecture

Onion Architecture is an architectural style that consists of designing a system as a set of concentric layers interacting with each other from the "inside-out", towards the core that holds the Domain. This architecture style was created by Jeffrey Palermo in 2008 as a solution for the unnecessary coupling of traditional n-tier layered architectures, especially regarding

the data access layer (Palermo, 2008). Palermo defined the main tenets of this architecture as: (Palermo, 2008)

- The application is built around an independent object model;
- Inner layers define interfaces. Outer layers implement interfaces;
- Direction of coupling is toward the center;
- All application core code can be compiled and run separately from infrastructure.

Figure 3 - Onion Architecture demonstrates this architecture style.

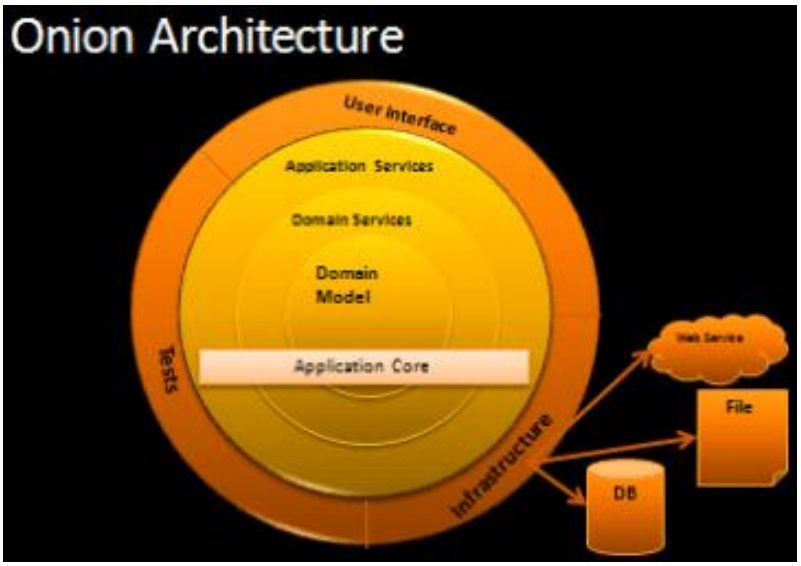

Figure 3 - Onion Architecture

2.2.3 MVC

The Model-View-Controller (MVC) architectural pattern uses separation of concerns to separate an application logic into three main components: Models, Views and Controllers (Smith, 2022).

- Model – the Model layer represents the business logic of the application. It is responsible for managing the application state and persisting it;
- View – the View layers are responsible for presenting content through the user interface (UI). It should only contain logic related to presentation content;
- Controller – the Controller layer is the point of entry to the application. It handles requests made by the user in the UI by manipulating a Model to fulfil the business need and a View to render it.

2.3 Architectural patterns

As microservice architectures present common characteristics there are some architectural patterns that aim to promote good practices and answer common challenges regarding areas such as Data Management, Security, Communication, Infrastructure and Deployment. In the section some patterns that may impact the testing strategy are explored.

2.3.1 Event Sourcing

Event sourcing is a pattern where the state of a business entity is persisted as a sequence of state-changing events, instead of simply storing the current state of each entity in a database. "*Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events."* (Richardson, 2021).

It was born to address a common microservice challenge: the need to atomically update a database and send messages/events to avoid data inconsistencies. This approach brings attentionally advantages as reliable auditing of all past events and built-in support for temporal queries (Richardson, 2021).

2.3.2 CQRS

The Command Query Responsibility Segregation (CQRS) pattern answers the need of implementing queries that join data stored at multiple services. It defends that a service with a replica read-only database should be developed to support queries. The replica must be kept to date though the use of sync mechanisms like snapshots or by subscribing to events published by the services that own the data. (Richardson, 2021)

The principal drawback of this implementation is the possibility of replication lag – having query data not synchronized with reality – as the system is only eventually consistent.

2.3.3 SAGA

The SAGA pattern is a solution for handling transactions that span multiple services. It aggregates a sequence of local transactions that belong to the same business transaction as a saga. "*Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions"* (Richardson, 2021).

2.4 Communication mechanisms

While in a non-distributed system, components communicate using method calls, in a Microservice architecture, this is not possible.

One of the biggest challenges of a microservice-based architecture is the communication strategy: microservices interact with each other over the network using common communication protocols as HTTP or messaging (Lewis & Fowler, 2014), using synchronous or asynchronous patterns accordingly with the necessities. Those communication strategies are applied when processing larger business requests, that span multiple components of the system to provide business valuable features.

2.4.1 Technologies

Here, some techniques related with communication are defined.

- **SOAP**

SOAP (Simple Object Access Protocol) is a network protocol for exchanging structured data, in the form of XML (Extensible Markup Language) and usually takes advantage of HTTP to expose its functionalities as a web service. It uses a WSDL (Web Services Description Language) to specify the operations, request's schema, and response's schema of the exposed service interface. SOAP also has a build-in error handling system, capable of verifying the request structure, matching it against the WSDL schema specification and returning insights about misconceptions on the response.

- **REST**

Representational State Transfer (REST) is an architectural style for building web APIs (often called RESTful APIs) that relies on HTTP for communication over the network. REST structures data using several possible formats, most commonly JSON (JavaScript Object Notation), but also CSV, XML, HTML, or plain text. REST focuses on resources, defined in the URI (e.g., host/**resource**) and defines a set of functions/ requests, being the most common: (SOAPUI, n.d.)

- GET – read or retrieve data from a resource;
- POST – create a new resource;
- PUT – update data that already exists in the resource;
- DELETE – delete the resource specified in the URI.

An interesting way of applying REST is by using Hypermedia. Hypermedia applies the concept of browsing the web using Hyperlinks to REST APIs. When requesting for a resource, the response will contain links for its related resources and possible actions to manipulate it. For example, instead of containing an ID of a related resource, the response will contain a link to its GET method and another to update it. When the possible actions change accordingly with the resource state **H**ypermedia **a**s **t**he **E**ngine **o**f **A**pplication **S**tate (HATEOAS) is applied (Fowler, 2010). This way the consumer can navigate through the API by using the returned

links instead of fixed URL paths, increasing flexibility for the server to refactor its URI schema without breaking clients. It also helps keep state management logic on the server-side, as the client does not need to have logic based on the response fields to know what actions are possible next, the server already contains the state logic and provides It as links. Some downsides of this approach are the extra effort needed on the server side and the natural increase in the response size.

- **Messaging**

Messaging is defined as the process of exchanging messages between software components. Messaging system typically use one of the below messaging styles: (Coppen, 2021)

- Peer-to-peer messaging – the message producer sends the message to a queue, which the message consumer is listening for;
- Publish/subscribe messaging – the message producer sends/publishes the message to a system (designated topic) that then delivers copies of this message to all interested consumer applications.

When integrating over messaging, a message broker (like e.g., RabbitMQ or Kafka) is commonly used to manage queues and topics.

- **gRPC**

gRPC is an open-source platform-independent Remote Procedure Call (RPC) framework that allows a client application to directly call a method/procedure on a server application, deployed on a different machine, as if it were a local object. It is used to connect systems in various scenarios as distributed applications and services, Internet of Things (IoT), to connect mobile devices and browser clients to backend services and to connect polyglot services in microservices style architectures (gRPC Authors, 2022).

gRPC works by defining a service – specifying its methods, parameters and returns types – as an interface that the gRPC server implements and the gRPC stubs expose to the client applications. The client side calls the gRPC stub as a local object, and this stub is then responsible for interacting with the server to perform the requests, abstracting integration logic. It also supports several languages to build their server and stub instances and provides load balancing, tracing, health checking and authentication out of the box (gRPC Authors, 2022).

gRPC uses Protocol Buffers – an open-source language-neutral mechanism for serializing structured data developed by Google – but can also be used with JSON. Protocol buffers work on top of a .**proto** file containing the structure of the data to serialize and uses its compiler protocol to create data access classes in the language being used.

2.4.2 Synchronous communication

In synchronous communication a client service makes a request to another service and waits for a response. This request/response pattern is simple and required in a lot of use cases,

where the client needs the server information to continue with its business flow. While the client can choose to process the response synchronously, having a blocking behaviour while waiting for the response, or asynchronously, by continuing its execution and just treating the response when it is indeed returned, the called service will always treat the request synchronously.

In microservices this call is typically done using HTTP or gRPC (Azure DevOps, 2022). Some of its major advantages are simplicity and ability to received error response discriminating what failed.

2.4.3 Asynchronous communication

In contrast to synchronous communication, in asynchronous communication the caller service sends the message without waiting for a response. Furthermore, the called service does not need to process the request right way. For this, Messaging is typically used. This message is often received by an Event Sourcing application, or a Message Broker, and later processed by a service (or more), providing a non-blocker behaviour to the caller. This is valuable in a microservice architecture because it allows: (Azure DevOps, 2022)

- Low coupling – the sender does not need to know about the request consumer.
- Allows multiple subscribers – a message can be propagated to several consumers (called subscribers in a publish-subscribe model).
- Failure isolation – if the consumer servicer is failing, the sender can still publish messages. Those will be processed once the consumer becomes available again.
- Responsiveness – when the consumer service needs to call other services synchronously during the request processing, it can have very high response times – also denominated latency. Using asynchronous processing the caller service is not impacted.
- Load levelling – when a service is receiving more workload than it can handle, the queue can act as a buffer to control the number of requests being made, delaying some messages to control the consumer health.

Nonetheless, asynchronous communication also adds a lot of complexity to a software infrastructure and communication flows. It requires a messaging infrastructure to be build and supported.

2.4.4 Challenges

As microservices communications are done across network boundaries, the system needs to be prepared for failure in remote components - “Design for failure” principle (Lewis & Fowler, 2014). *“Similarly, microservices communicate with external datastores through a network boundary, so they need to be designed to deal with latency and risk of outage.”* (Clemson,

2014). To maintain an overall system uptime, techniques like timeouts and circuit breakers are commonly used (Clemson, 2014).

As the system is compose of several microservices running across network boundaries, it is inevitable that, eventually, some parts of the system may be healthy while other are failing, overloaded with requests and unable to process incoming requests. Microservices need to be resilient on those scenarios and apply strategies to mitigate failure. Some options are: (gRPC Authors, 2022)

- Retries – consists of replaying a failed request a certain number of times. This makes sense for some error scenarios like network failures or server overload but should be carefully considered, as some request may be dangerous to replay and can lead to repeated operations/ objects in the server.
- Circuit Breaker – a pattern to prevent repeated failed requests from overloading the server, blocking its resources and, consequentially, creating failure. The circuit breaker object receives requests and monitors for failure. If failures reach a certain threshold the circuit closes and blocks incoming requests before they reach the server.
- Dead Letter Queue – a queue to hold messages which failed processing, so they can be reprocessed later, when the service is available.

Other challenge is related with managing the network address of all the services in the system. Microservices are typically deployed in a cloud environment using virtualization, containerization and running several instances. As consumers don't need to know about the deployment strategy of the services, its necessary to use strategies to abstract them from this logic. A Load balancer is a piece of the architecture that provides a stable IP address and URL for a service while distributing incoming requests (load) across the service running instances. They are the point of entry for a service.

Furthermore, consumers may require data that is stored in several microservices. While each consumer could orchestrate the calls to the microservices public APIs and aggregate the results, it would limit the capability of those services to change and cause a lot of chattiness over the public internet. An API Gateway is layer of the system architecture that exposes a single point of entry, abstracting clients from the system's internal structure (Richardson, 2015). It encapsulates responsibilities like authentication, monitoring, load balancing and request routing and orchestration. Clients call the API Gateway, which redirects the requests to the microservices, transforming and aggregating results when necessary. This is extremely helpful for use cases that required information from several microservices. An API Gateway offers custom APIs implementation for clients to retrieve the data they need without having to implement this logic themselves.

2.5 Microservice lifecycle

One of the advantages of adopting microservices is the ability to parallelize the full development lifecycle between different small teams (Lewis & Fowler, 2014). After agreeing

on an overall architecture for the distributed system, defining the communication paths as APIs specifications, contracts, and message coordination and segregating the business domain in separated services, each with individual responsibilities, teams can start working independently. Each team is responsible for managing their services through its lifecycle phases, but teams should apply similar concepts to take advantage of the architectural style. Figure 4 shows an example of a microservices development lifecycle and states some technologies that can help at each phase.

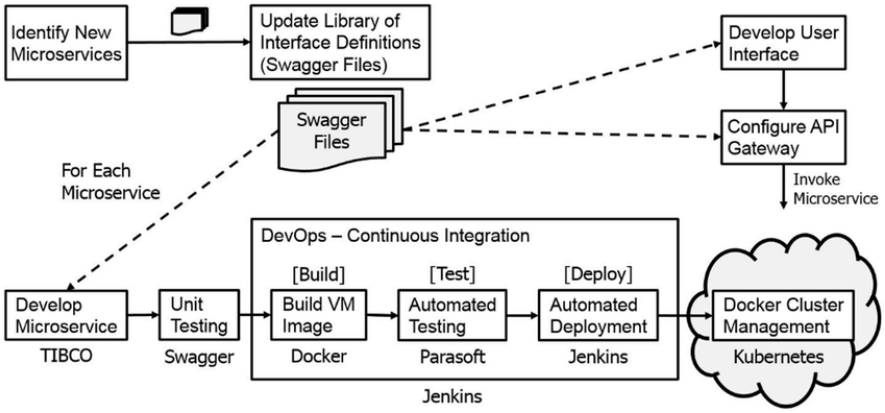

Figure 4 - Microservice Development Lifecycle (Megargel, et al., 2020)

Teams should take advantage of agile design cycles to enable Evolutionary Design (Lewis & Fowler, 2014). As the system is composed of several highly maintainable, easily modified, loosely coupled and independently deployable components, having a flexible development lifecycle will enable the organization with a fast change mechanism to adapt to the market (Lewis & Fowler, 2014). As change enablers themselves, microservices can change significantly throughout their lifecycles. Changes can vary from its internal structure, architecture, technology, and API to domain boundary decomposition or aggregation. As the organization keeps exploring and accumulating knowledge about the problem domain, the architecture needs to support this grown.

While the overall architecture keeps evolving over time, it's important to have confidence that changes made support both present and future use cases. CICD (Continuous Integration and Continuous Delivery/ Continuous Deployment) is a set of practices that uses automation to standardize a service lifecycle, bringing confidence and agility to the change process (RedHat, 2022). Continuous Integrations is the concept of continuously building and testing new code together to guarantee its compatibility. Continuous Delivery and Deployment enable teams to release their services in an automated manner thought the use of an automated build pipeline (Lewis & Fowler, 2014). *"A build pipeline that allows the automation of the build and release process, allows running unit tests and some types of integrated and system tests, allows automated deploy to a testing or production environment will provide fast feedback and fast promotion cycles."* (Vocke, 2018). Examples of software used to build such pipelines are Jenkins, Gradle and Helm. The Figure 5 displays an example of a CICD pipeline.

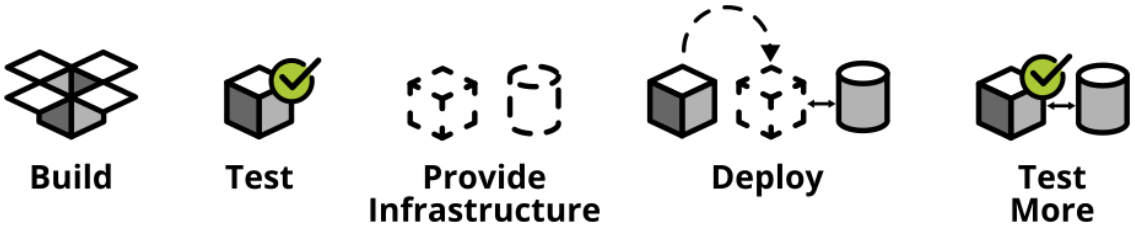

Figure 5 - Use build pipelines to get your software automatically and reliably into production (Vocke, 2018)

To take fully advantage of a microservice architecture, teams should be enabled to release their services in an automated manner (Lewis & Fowler, 2014). Teams enabled with automation will be more efficient, autonomous, and agile. They will be able to verify code changes with ease, deploy the software to some environment for further testing and make production changes with confidence. The CICD process can run automated tests on code submissions and use the results to decide if the changes should be promoted. Google highlights "*that all of the tests for a particular project must report a passing result before submitting code or releasing a project*" (Micco, 2016).

While this bringing additional complexity to the system it is necessary: the deployment stage in a microservice architecture is far more complex than in a monolithic system. The same system can be composed of several hundred services, written in a variety of languages and frameworks, each with different requirements for deployment, resources, and scaling. Furthermore, it is necessary to choose an infrastructure architecture: will it be deployed as-premise or in the cloud? Will It take advantage of containerization and virtualization? Most often Microservices are deployed in the cloud using a containerization technology, as Docker, and take advantage of deployment tools to manage the services deploys and resources.

3 State of the Art

This chapter exposes the research conducted on the current state of Microservices testing.

3.1 Testing concepts

Firstly, lets define some testing concepts:

- Solitary Test – style of testing that isolates the tested unit from its dependencies by replacing them with a Test Double, avoiding side-effects and possible complicated test environment setups (Fowler, 2014). This includes internal dependencies, as custom objects, and external dependencies, as databases and remote services.
- Sociable Test – style of testing where the tested unit relies on other units that are not subject of the test, and, so, can lead to faults in code that is not the focus of the test (Fowler, 2014).

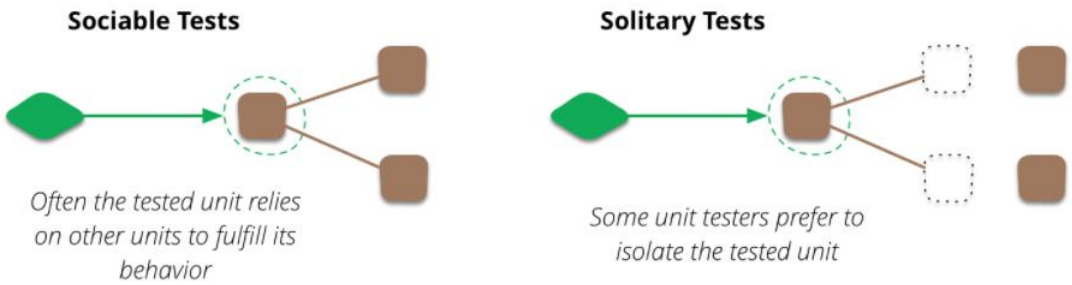

Figure 6 - Sociable vs Solitary Tests (Fowler, 2014)

- Test Double – term that represents an object that is used to replace a production object in testing that, while not behaving exactly like the real entity, provides the same API. It is used to replace parts of the system that are not the object-under-test,

but rather a dependency. It helps the tester verify the indirect output calls made to other components during the testing flow. It can be of several types: (Meszaros, 2007)

  - Dummy – object that replaces objects that the test does not really use but needs for its instantiation (e.g., a required parameter).
  - Fake – object, typically built for testing, that implements that same functionality as the production object but in a simpler way. It is commonly used when the real object is too slow or cannot be used in the testing environment (e.g., an in-memory database).
  - Stub – object that allows the definition of pre-defined response to calls made during the test execution, which can be valid values, or errors/exceptions.
  - Spy – stubs that furthermore allow the recording of information about the way they were called.
  - Mock – object that can be configured with a set of expected requests (or method arguments) and responses to be used during the test execution. It compares the actual arguments/request received with the excepted one and fails (with an error/exception) if they don't match (Fowler, 2014).

- Black Box Testing (Opaque Box Testing) – testing technique where the application internal structure is entirely unknown to the tester. The tester focusses solely on the verification that an input produces the desired output. The tester tests the system from a user perspective and taking in considerations the software requirements. It can also be called functional testing or behavioural testing. (Khan & Khan, 2012)
- White Box Testing (Transparent Box Testing) – testing technique where the tester has knowledge about the application internal architecture and has access to the source code, whose behaviour is analysed. This approach obtains a high-test coverage but requires skilled testers. (Khan & Khan, 2012)
- Grey Box Testing – testing technique that combines black box and white box testing to increase coverage and focus on all the application layers. Tester studies the interface definition and functional specification of the application and designs test cases for both layers. (Khan & Khan, 2012)
- Flaky test – Google defines a flaky test as a "*test that exhibits both a passing and a failing result with the same code*" (Micco, 2016). They pinpoint the reasons for this behaviour as concurrency problems, dependency on non-deterministic behaviours, infrastructure/network problems and third-party code (Micco, 2016).
- Test Pyramid - is often used as visual aid for developers and testers into understanding the several automated testing layers and how much tests should be performed for each one. It establishes a balance between system layers and number of tests, test independency, and test execution time. Its base principles are written tests with different granularity; write fewer tests, the more high-level you get (Vocke, 2018).

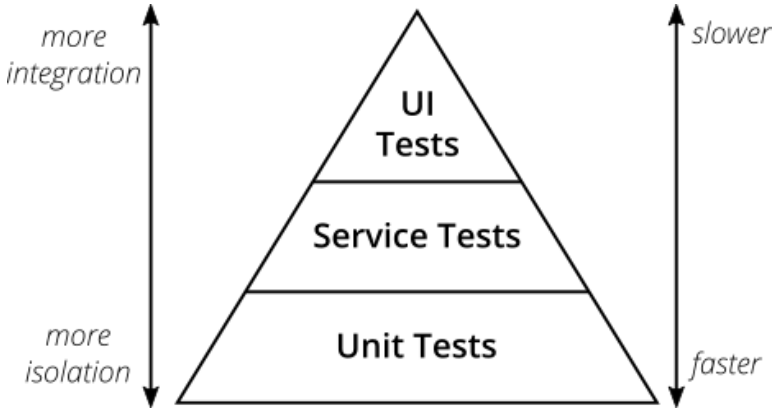

Figure 7 – The Test Pyramid (Vocke, 2018)

3.2 Testing typologies applied to Microservices

As microservices are independent components composing a larger system, they already provide clear boundaries and natural isolation between them. These characteristics provide teams with several testing options when designing their test suite. Teams can use a combination of several testing types and apply them to different application and architectural layers of the system, accordingly to the complexity of each service (Clemson, 2014). While some microservices may hold more critical business requirements and need to be comprehensively tested, others may be less crucial and applying some testing approaches will not bring many advantages (Clemson, 2014).

In this section, several testing types are defined in meaning and scope, and each's application to the Microservice architecture is explored, according to the layers presented on Figure 2 - Microservice internal structure .

3.2.1 Unit tests

Unit tests are the foundation of the test suite. They test the system at the smaller granularity possible, designated "unit" which typically is at the class or method level, and represent a large piece of the test suite. This testing type is usually focused on the code itself – is of small scope - and can be Sociable or Solitary (see section 3.1), depending on the unit-under-test requirements. Because they test the smaller units of the system, unit tests will run fast, providing fast feedback cycles about the code. They are typically automated on the build pipeline, and expose custom metrics, as code coverage, to help teams measure its quality. (Vocke, 2018)

A CICD process will likely run automated unit tests on code submission and use the results to decide if the changes should be promoted. Following this approach, tests are used as a *"a safety net for code changes"* (Vocke, 2018) – if they pass, it should be a guarantee that the code is doing what it was designed to. To follow this approach, it is important to guarantee

that tests are not deeply coupled with the internal code implementation. If they get too connected with the code logic, they are forced to change when the code changes and a big advantage is lost. Instead, an observable behaviour approach should be used: given x, the result should be y. (Vocke, 2018)

Unit tests are also a useful design tool. Test-driven development (TDD) is a software development methodology where "*unit tests guide your development*" (Vocke, 2018).

When applied to the Microservices layers presented on Figure 2 - Microservice internal structure , Clemson argues that the Domain layer is responsible for complex logic and state management, and so, there's little value in trying to isolate it from its dependencies. Sociable unit tests should be used to test this layer, where as far as possible, real collaborators of the unit under test are used (Clemson, 2014).

On the other hand, unit testing layers like Gateways and Repositories, that contain logic used for producing requests and mapping responses from external services should be done using the Solitary approach. Unit tests on this layer only verify the correct behaviour of this logic and do not focus on verifying the integration with the external dependency. The replacement of external dependencies for Test Doubles, allows the developer to simulate specific request-response scenarios and errors that otherwise would be hard to test and facilitates test automation. Unit testing at the Resources and Service layers should also be Solitary, as no advantages come from using the real collaborators. (Clemson, 2014)

3.2.2 Integration tests

Integration tests focus on verifying that the integration of an application with all its external components is working correctly (Vocke, 2018). In contrast with Unit tests, the goal is to test collaboration is working as expected and no incorrect assumptions were made about the collaborator (Clemson, 2014). This often means that it is necessary to run both the application and the component it is integrating with (e.g., a database or a third-party API).

There are different interpretations of what an Integration test really is. Some defend it is done by having all the system components running and connected, and testing using the full application stack. Others prefer to focus on one integration point at the time and replace the external dependencies with faithful test doubles (Vocke, 2018). Martin Fowler defines those approaches as Broad Integration tests and Narrow Integrations tests, respectively (Fowler, 2018).

While Broad Integration tests seem to have more advantages and guarantees over the correct system behaviour while cooperating, they also present other challenges. They require a complex test environment setup: working versions of all services dependencies need to be deployed and network access between them should be granted. This makes them slow and hard to automate. They also exercise code paths that are not responsible for integration alone. (Fowler, 2018)

On the other hand, Narrow Integration tests focus on the service boundary and only exercise code from the integration layer. They use test doubles to replace dependencies, facilitating test automation and speeding up their runtime, and rely on Contract tests to guarantee test doubles behave similarly to the real components. (Fowler, 2018)

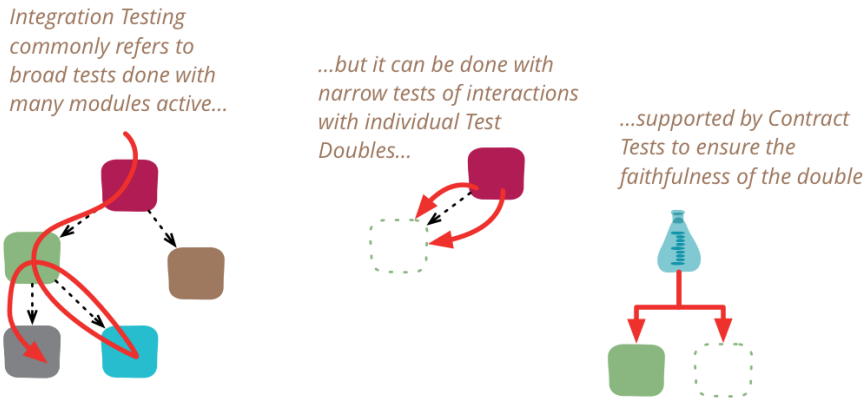

Figure 8 - Broad vs narrow integration tests (Fowler, 2018)

On the Microservice architecture the narrow approach is usually used and aims at verifying that a microservice can *"communicate sufficiently rather than to acceptance test the external component"* (Clemson, 2014). "As integration between microservices and their external components are done across a network, integration tests should verify that the integration modules can handle network failure gracefully, as timeouts and slow responses or other abnormal behaviors. To simulate those failures a test double, like a stub, is used and configured according to the testing scenarios. Furthermore, Integration tests focus on protocol level errors such as missing headers, invalid response codes or serialization in HTTP connections and can help unveil misconceptions such as invalid request or response formats." (Clemson, 2014)

3.2.3 Component tests

Although unit and integration testing can provide much confidence about the correctness of the internal logic of an individual component, they do not verify that the component modules work together to fulfil business requirements. While this could be achieved with broad end-to-end tests, component tests are a wise alternative. (Clemson, 2014)

A component is a part of the system that is coherent, independent, and replaceable. In a Microservice architecture, the components are the services themselves (Daya, et al., 2016). Component tests limit the testing scope to a single component by isolating it from its external dependencies using test doubles - avoiding any complex behaviours they may have, speeding up test runtime and feedback cycles, and facilitate automation, as any application error case can be simulated in a repeatable way (Clemson, 2014).

There are two common approaches to implementing component testing in microservices: (Daya, et al., 2016) (Clemson, 2014)

- In-process component tests - the microservice and its dependencies test doubles are instantiated in-memory and tests ran without any cross-network communication. While this minimizes test execution time and build complexity, it adds additional test logic - the service needs to be prepared to be configured in a "test" mode and have its dependencies injected at start-up time – and it does not cover integration points between dependencies.
- Out-of-process component tests – the microservice is deployed in a testing environment and its dependencies test doubles are managed by the test pipeline, which is responsible for setting up the environment, starting the services and coordinating network configurations between them. All interactions make use of real networks calls and so, more layers and integrations points are tested. *"As a result of the network interactions and use of a real datastore, test execution time is likely to increase"* (Clemson, 2014).

3.2.4 Contract tests

While with the previously testing strategies we can obtain a high-test coverage of the system individual components and have confidence about its implementation of business logic, there is still no tests that ensure the system components work together to fulfil larger business use cases (Clemson, 2014).

"A contract test is a test at the boundary of an external service to verify that it meets the expectation of the contract formed between the service and its consuming services." (Daya, et al., 2016). A contract is made between a component and each of its consumers, based on the requirements of each integration and consists of an agreement regarding the structure, format and attributes of the messages traded between them. Running contract tests guarantees that, if one component changes and breaks its contract, the teams involved will know as soon as possible. This provides consumers of external services with confidence about their integration. In a Microservice architecture contracts are ran against the public API exposed by each service. (Clemson, 2014)

Contract tests are also valuable for the maintainers of such services, as they can make changes knowing consumers will not be impacted. "*Ideally, the contract test suites written by each consuming team are packaged and runnable in the build pipelines for the producing services. In this way, the maintainers of the producing service know the impact of their changes on their consumers."* (Clemson, 2014).

Contract tests can also help when designing a new service. Each consumer builds a test suite with the expectation they need from the service and provide it to the team responsible for building the new service. In this approach the Consumer-Driven Contracts (CDC) are driving

the API design of the new service and can help make sure it is fulfilling its purpose in the system. (Clemson, 2014)

Vocke argues that contract tests are a type of narrow integration test, *"In fact the consumer test works exactly as the integration test, we replace the real third-party server with a stub, define the expected response and check that our client can parse the response correctly."* (Vocke, 2018) but they provide additional advantages: the contract tests can also be provided to the producer team and automated in its build pipeline (Vocke, 2018).

3.2.5 End-to-end tests

End-to-end tests treat the system as a black (opaque) box and aim to verify that the entire system works together to fulfil its requirements. *"The test boundary for end-to-end tests is much larger than for other types of tests, because the goal is to test the behavior of the fully integrated system."* (Daya, et al., 2016). The system is tested from the user perspective, either through its graphical interface (UI (User Interface)) or by utilizing its public API (Vocke, 2018). *"In this way, the correctness of the system is ascertained by observing state changes or events at the perimeter formed by the test boundary"* (Clemson, 2014). End-to-end tests also require system components to be fully deployed and configured as similarly as possible to the production environment.

In a Microservice architecture, end-to-end tests are important to provide extra confidence in the integration between its services, as those are communicating over network infrastructures like proxies, firewalls, load-balancers, etc., they ensure its correct configuration (Clemson, 2014). They also work as a safety net for architectural refactoring – as a microservice architecture is set to evolve over time, business domains may split or merge, and components change. End-to-end tests provide confidence that the business requirements are still being fulfilled. (Clemson, 2014)

Unlike the other testing approaches, end-to-end tests rarely rely on test doubles. Instead, they use the real collaborators of the component and, even external dependencies (managed by third parties), if possible, and that is why they are so valuable. When dependencies do not allow end-to-end tests to be reliably repeatable in a side effect free manner, it can be beneficial to replace them for test doubles, "losing confidence but gaining stability" (Clemson, 2014).

Although end-to-end tests give the most confidence about the system readiness for production, they have their own challenges. "*They are notoriously flaky and often fail for unexpected and unforeseeable reasons. Quite often their failure is a false positive."* (Vocke, 2018). Furthermore, end-to-end tests are difficult and expensive to write, require maintenance, run slowly, and have so much moving parts that can fail for very heterogenous reasons (Vocke, 2018). Clemson defines the following guidelines to help manage this test suite: (Clemson, 2014)

- Write as few end-to-end tests as possible – since other testing types already provide confidence about most business requirements and edge case behaviours, end-to-end tests should only focus on making sure everything ties together as expected and not duplicating test scenarios.
- Focus on personas and user journeys – design end-to-end tests around the user's journey thought the system, as those scenarios are the most valuable to the users and will provide the most confidence.
- Choose your ends wisely – remove unreliable external services from the test suite to reduce test flakiness, but make sure to test those using other testing approaches.
- Rely on infrastructure-as-code for repeatability – take advantage of infrastructure-as-code to create a fresh testing environment for the end-to-end test suite, in a repeatable manner, instead of relying on environments that may be non-deterministic.
- Make tests data-independent – stop relying on pre-existent data and, instead, automate data management before the test suite execution. This removes the possibility of data related failures happening.

"In a microservices world there's also the big question of who's in charge of writing these tests. Since they span multiple services (your entire system) there's no single team responsible for writing end-to-end tests." (Vocke, 2018).

3.2.6 Testing in production

If the test environment is different from the production environment, there are no guarantees about test validity in the production environment. A common way to guarantee full test validity and release software to production with more confidence is to execute tests in production (Richardson, 2019). For this, deploys in the production environment and releases of new versions of the application to users should be separated into two separated processes.

A solution is to deploy the updated version of the service alongside the production instance and run tests against it using intelligent routing logic. If tests succeed, traffic can start being shifted into the new release and the results should be monitored to understand the impact of the change. If error rates increase a rollback should be triggered, otherwise the rollout process can continue. Those steps should take advantage of automation for fast deployment, rollback and roll forward between both application versions. A mature monitoring and logging system is also recommended for this approach.

3.2.7 Other testing types

In this section some other testing types, considered relevant for the scope of this work, are presented.

- Load tests – testing technique which aims to measure the system capacity to respond to its expected usage in the production environment (load). It's done by simulating

multiple requests to the program concurrently, measuring its response times, and comparing them with the non-functional requirements of reliability, scalability, and availability.

- Smoke tests – testing process that verifies if a key feature of an application is working to determine if a system was built successfully. It runs a success scenario and aims to discover major problems in a simple, fast way, before more complex testing starts.
- Canary tests – consist of testing changes in the software using real users or processes of a live production environment. The version of the software with the code changes is only available for a small group of end users who are using the system normally, unaware they are taking part in a new version test.

3.3 Testing technologies

In this section, auxiliary tools, frameworks, and libraries used to help develop and run the previously presented testing types are described and its functionalities explained.

Testing technologies can be of several types:

- Tool – tools live outside of the source code. They are usually programs that provide valuable functionalities "out of the box" that help the user with achieving its goal more proficiently (e.g., faster, cheaper, with less effort);
- Library – a piece of code that provides a set of well-defined operations and functionalities to solve specific problems which can be called and used while coding;
- Framework – a framework defines a reusable skeleton for developing a set of functionalities. The user is then responsible for filling this skeleton with his set of specifications that the framework will utilize to build the functionalities. Martin Fowler says that frameworks often present Inversion of Control, as they are the ones controlling the flow and asking the developer for information (Fowler, 2005).

Libraries and framework are imported into a project as a build dependency. This means that, when the source code is compiled, the build agent will look for the dependency on the classpath. Often a dependency-management tool is used to automate the management of such dependencies, by providing a central repository where dependencies are kept. Examples of such tools are Maven, Gradle, NPM, and NuGet.

3.3.1 Unit testing frameworks

A Unit testing framework provides several functionalities for creating and running unit tests. It can suggest a test structure to be used for test development, a method for comparing the test results, and a motor, to run such assertions.

3.3.1.1 Junit

Junit is a *"programmer-friendly testing framework for Java and the JVM"* (JUnit, 2021) that encourages developers to write tests incrementally, during the development phase (Cheon & Leavens, 2001). It aggregates test together in a test suite, runs them automatically, compares test returns with provided expectations and summarizes the results (Beck, 2004). It is one of a family of unit testing frameworks known as xUnit, where x is related with the programming language or framework it aims to test.

Junit aims to make tests (Beck, 2004):

- easy to write – uses "ordinary" Java to write tests;
- easy to learn to write – as the target audience are non-professional testers but instead developers;
- quick to execute – as they can run several times a day;
- easy to execute – execute "with a push of a button" and present the results clearly;
- isolated – isolated tests provide high-quality, focused feedback;
- composable – ability to run anu number or combination of tests together;
- flexible – tests objects only using their public interface.

JUnit defines a testing structure and lifecycle to help keep tests in a standard format and repeated tasks to the minimum (Cheon & Leavens, 2001). In JUnit 5, the most recent version of the framework, the following concepts are defined: (JUnit, 2021)

- Test Class - Java class that aggregates test methods;
- Test Method - instance method annotated with @Test or any of its alternatives (@RepeatedTest, @ParameterizedTest, @TestFactory, or @TestTemplate). It uses assertions methods provided by the framework to decide if the test succeeds or fails (e.g., *assertEquals* will compare two objects using its *equals* method and report a test success if they match, or a failure, otherwise);
- Lifecycle Method - any method annotated with @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach, where:
    - @BeforeAll – executes once before all test methods in the current test class. This method is normally used to setup the testing environment and data;
    - @AfterAll – executes once all test methods in the current test class have run. It is used to clear the environment built for testing;
    - @BeforeEach – executes before each test method in the test class;
    - @AfterEach – executes after each test method in the current test class.

These concepts are displayed on the standard JUnit 5 test class displayed below.

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
```

```
class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

Standard JUnit 5 test class (JUnit, 2021)

JUnit is linked into a project in a JAR format at compile-time and is supported by most build tools (e.g., Gradle, Maven, and Ant) as a dependency (JUnit, 2021).

3.3.1.2 NUnit

NUnit is a unit-testing framework for all .NET applications. It started as a port from JUnit but has since been fully integrated amongst the .NET platform with updated semantics and new features (Poole & Prouse, 2022). Similarly, to JUnit, NUnit uses a set of custom annotations, renamed attributes, to identify and configure tests programmatically (NUnit, s.d.).

It also provides a rich set of assertions methods for verifying if a test result matches its expectation. "*In NUnit 3.0, assertions are written primarily using the* `Assert.That` *method, which takes constraint objects as an argument. We call this the Constraint Model of*

assertions" (NUnit, s.d.). Such constraints express conditions that the asserted object should match for the test to pass – otherwise the `Assert.That` will return an error.

3.3.2 "Mocking" tools

A "mocking" framework is a used for creating Test doubles in a programmable way. Those test doubles are used to isolate the unit under test within a controllable environment, allowing to test a specific class or method by itself, without having to worry about potential dependencies bugs and setting up those dependencies' dependencies (Clarke, 2020). Mocking frameworks simulate collaborators and replaced them with test doubles that should be configured with a set of mock actions – what it should be return If a method/property is called against the mock. Furthermore, many frameworks provide methods to verify that mocked action were indeed called during test execution, which proves useful to assert the correct behaviour of the unit under test (Clarke, 2020).

3.3.2.1 Mockito

Mockito is a mocking framework developed for Java that provides an API for the creation of test doubles (mocks) of methods and objects. The Mockito team highlights that Mockito mocks only "verify what you want" and do not require the setup of irrelevant behaviours of the unit-under-test. (Mockito, 2019)

Mockito does not enforce a specific test doubles type thought its API – "There is only one kind of mock". Instead, it allows the stubbing of information before execution and the validation of interactions afterwards (Mockito, 2019). This mocking type behaviours similarly to a Test Spy (see 3.1) – remembers all interactions and allows you to further verify whatever interactions you need for testing purposes (Mockito, 2021).

```
import static org.mockito.Mockito.*;

// mock creation
List mockedList = mock(List.class);

// using mock object - it does not throw any "unexpected interaction"
exception
mockedList.add("one");
mockedList.clear();

// selective, explicit, highly readable verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

Verifying interactions with Mockito (Mockito, 2021)

```
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

// the following prints "first"
```

```
System.out.println(mockedList.get(0));

// the following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

Stubbing method calls with Mockito (Mockito, 2021)

3.3.2.2 EasyMock

EasyMock is a testing framework for Java that allows the creation of Mock Objects, generating them using the Java proxy mechanism. A Mock Object is created in "recording" mode and saves all the actions made against it until "replay" mode is turned on. Then it replays the actions it recorded previously when requests match previously made ones. EasyMock defines mocks in 4 steps, for which it provides 4 key methods: (EasyMock contributors, s.d.)

1. **mock(…)** – creates a mock instance of the target class. EasyMock will record every action/call taken against the mock while it is in "recording" mode and replay them later, when "replay" mode is switched on.
2. **expect(…)** – sets expectations on the mock object, as calls, results, and exceptions.
3. **replay(…)** – switches a mock to "replay" mode. This is a prerequisite for running the actual test - calling the method/object under test. Onwards, actions will not be recorded. Instead, any action that matches a previously recorded one will cause a return of the results recorded during the "recording" phase, using the expect(…) method.
4. **verify(…)** – verifies that all expectations were met – everything that was supposed to be called was called and matched the behaviour set using expect(..) method.

EasyMock also supports the use of annotations to configure the test class and supports JUnit integration.

3.3.2.3 Moq

Moq is a popular mocking library for .NET applications developed to "*take full advantage of .NET Linq expression trees and lambda expressions*" (Moq, 2022). Just like previous presented frameworks it works by creating a Mock object, setting up expectations, calling the test method/class and assessing if, during test execution, requests made to the Mock object matched expectations. In contrast to EasyMock, it does not use any record/replay model, keeping mocking more straight to the point with a low learning curve (Moq, 2022).

3.3.3 Double HTTP server tools

A double server simulates an HTTP API by returning predefined data instead of running real functions. This is useful to avoid communicating with the real HTTP API and allows the definition of what the developer wants to receive for certain requests, facilitation the testing of specific failure scenarios.

3.3.3.1 WireMock

WireMock is a server-side simulator for HTTP-based APIs. It works as a test double, allowing the definition of stubbed responses to specific incoming requests and capturing of such requests, for further verifications. It can be used as a library for JVM applications or run as a standalone process, either on the same machine as the system under test or, in a remote server. WireMock helps developers test integration code, as real HTTP requests are performed. Its features can be configured via REST API, Java API or by using JSON files. (WireMock, 2022)

WireMock is distributed via Maven Central and can be included in your project using common build tools' dependency management (e.g., Maven, Gradle). WireMock integrates with JUnit and takes advantage of its lifecycle methods to manage its server. The following code shows a test example where a HTTP POST to /my/resource is stubbed and used in a verification. (WireMock, 2022)

```
@Test
public void exampleTest() {
    stubFor(post("/my/resource")
        .withHeader("Content-Type", containing("xml"))
        .willReturn(ok()
            .withHeader("Content-Type", "text/xml")
            .withBody("<response>SUCCESS</response>")));

    Result result = myHttpServiceCallingObject.doSomething();
    assertTrue(result.wasSuccessful());

    verify(postRequestedFor(urlPathEqualTo("/my/resource"))
        .withRequestBody(matching(".*message-1234.*"))
        .withHeader("Content-Type", equalTo("text/xml")));
}
```

Example test using WireMock and JUnit (WireMock, 2022)

There is also a project called WireMock.Net that provides similar functionalities to WireMock but for C# .NET projects.

3.3.3.2 MockServer

MockServer is another option for creating test doubles HTTP servers. It supports a variety of flexible options on how to run its server, including: programmatically, via a JAVA API or integration with JUnit; as a Docker container; as a Helm chart in Kubernetes; as a Node.js module; as a deployable WAR in an application server (MockServer, 2022). It also offers a user interface capable of displaying logs, active expectations, requests received, and proxied requests (MockServer, 2022). Similarly, to WireMock, its expectations can be set using Java code or by its REST API. MockServer allows requests to be matched with expectations by comparing properties like HTTP method, path, query parameters, headers, cookies, body, and security (MockServer, 2022). Matching of properties supports string values, regex expressions, json schemas and the matching of the body even supports JsonPath and XPath expressions (MockServer, 2022). Figure 9 shows the tyipical lifecycle for using MockServer.

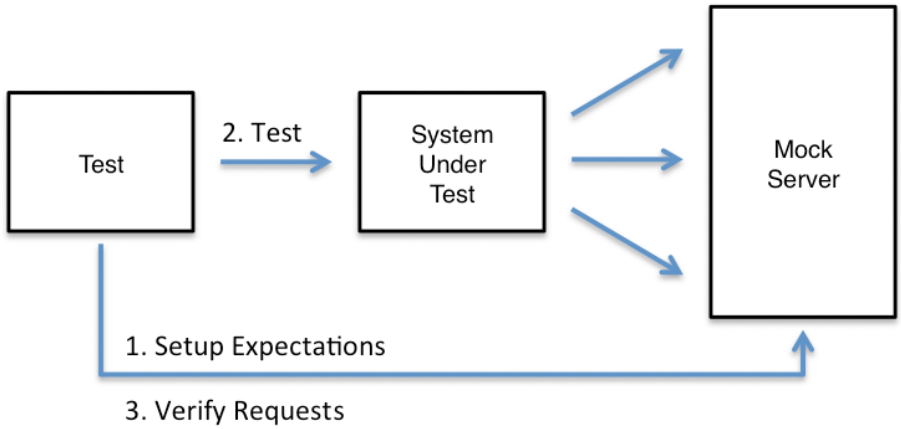

Figure 9 - Typical sequence for using MockServer (MockServer, 2022)

3.3.3.3 Postman MockServer

Postman is a tool for building and using APIs. It is commonly used by developers during an API lifecycle to design, build and test API's requests and responses. In Postman it is also possible to create a "mock"/double server by defining a set of example responses for each HTTP request in a collection. *"Postman will match the request configuration to the examples you have saved for the request and respond with the data you added to the example"* (Postman, 2022). Although Postman MockServer does not allow mocks to be created programmatically, they are quite simple and fast to create, providing a valuable option for local tests.

3.3.4 Database testing tools

To verify the integration of an application with a database, integration tests can run against a real database instance, or alternative replace it with a test double. While using a test double can make the tests faster, it also decreases the confidence in the correctness of the integration (Clemson, 2014). When using a real database instance, all the integration components are exercised and so there is more confidence the communication will work in production. Furthermore, there is the option to deploy a database instance across the network. This approach brings the most confidence about the integration but requires a more complex test environment setup, may fail to network issues, and hardens automation. Another option is to use a database deployed in the same environment as the application.

This section lists some Database Management Systems (DBMS) that can help make integration testing databases easier and faster.

3.3.4.1 H2 Database

H2 is an open-source lightweight relation database engine implemented in Java that supports standard SQL and the JDBC API (Java Data Base Connector API). It can be embedded in Java applications or deployed in server mode and can run as a disk-based or in-memory database. It also provides shell console and a browser interface to interact with the database (H2, n.d.).

An in-memory H2 database is often used to replace other databases implementations when running automated tests due to its light-weighted and easy-configurable character (Vocke, 2018). It makes tests easier to configure and faster to run – H2 is simply added to the classpath as a test dependency (H2, n.d.) – simplifying processes like test automation.

3.3.4.2 MongoDB

MongoDB is a cross-platform document-oriented database, often referred to as a NoSQL database program. It stores data as documents (JSON, BSON or XML) that resemble objects in the application code, instead of traditional tables or graphs, simplifying the data model and removing the need of a complex Object Relational Mapping (ORM) layer (MongoDB, 2022). MongoDB was created to offer a variety of advantages over traditional relational databases, as a data model that is directly mapped from objects in the code, a flexible-dynamic schema that naturally evolves with the application data model, and are distributed system which allows for high scalability and data distribution (MongoDB, 2022).

MongoDB has official drivers that support integration with the most popular languages and frameworks, e.g., C#, Java, Node.js and Python. It does not officially have an embedded implementation - although there are some community projects as the Flapdoodle's embedded MongoDB solution and the mongoUnit for JVM applications - so it is recommended to run integration tests against a local or remote mongoDB instance to faithfully test interactions (Schaefer, 2022).

3.3.5 Asynchronous messaging testing

Integration tests for asynchronous message-driven communication want to assure that the producer and the consumer applications can, in fact, connect to the message broker and, respectively, publish and consume messages using the correct queues.

The message broker can be instantiated as a local instance, an embedded instance (if support exists for it), as a container, using a containerization tool like Docker, or in a remote server.

3.3.5.1 Spring for Apache Kafka

Apache Kafka is an open-source distributed event streaming platform often used as a message broker. Spring for Apache Kafka is a boilerplate project for developing Kafka-based messaging solutions for Spring projects. Along many other helper classes, it has support for testing using an embedded Kafka server (Spring, 2022) – that does not need to be instantiated manually and configured – which helps speed up testing.

3.3.6 Event Stores testing

When testing microservices that implement the Event Sourcing architectural pattern (Richardson, 2021), it is necessary to test the code that integrates those applications with the event store where events logs are kept. But what is an event store?

An event store is described as a "*database of events*" - a relational or non-relation database that stores events in a sequential and immutable way – that also "*behaves like a message broker*" - takes advantage of messaging systems to deliver saved events to interested subscribers – to ensure transactions are atomical (Richardson, 2021). It provides an API for "*adding and retrieving an entity's events*" and for interested services to subscribe to events (Richardson, 2021). As only a few solutions support this out-of-the-box (EventStoreDB, Eventuate), organizations often build a custom component to manage this logic. So, to integration test an event store, similar concepts and technologies used to integration test databases and messaging systems should be used in conjunction.

3.3.6.1 EventStoreDB

EventStoreDB is database technology specifically built for Event Sourcing, optimized to store data in streams of immutable events (EventStoreDB, s.d.). It offers both an open-source edition (EventStoreDB OSS) and an enterprise edition (EventStoreDB Enterprise). It also comes with several useful tools for testing.

3.3.6.2 Eventuate Local

Eventuate Local is an event sourcing framework that consists of an event store and several client libraries for various languages and frameworks including Java, Scala, Java Spring, Micronaut, and Quarkus frameworks. It was specially designed for handling Microservices Event Sourcing and its related patterns, like CQRS and SAGA (Eventuate, 2021).

3.3.7 Contract testing tools

As described on section 3.2.4, contract tests assert that the contract a consumer develop with a producer is being followed. To perform contract tests, a contract definition needs to be shared between the "consumer" and the "producer". Contract testing tools aids teams with managing this process. They usually provide a canonical way to define a contract, create a sharing channel between the consumer and provider, and verify requests made against contract expectations.

3.3.7.1 Pact

"Pact is a code-first tool for testing HTTP and message integrations using contract tests" (Pact, 2022). Pact provides a platform for generating contracts and sharing them between the consumer and the producer. (Pact, 2022)

"During the consumer tests, each request made to a Pact mock provider is recorded into the contract file, along with its expected response." (Pactflow, 2022). Then a Pact simulated consumer will replay each request against the real provider and compare the actual and expected responses. In this way we can guarantee that, if expectations match, the simulated application (mock) behaves that same way as the real one, and so, that those applications will communicate successfully in the production world (Pactflow, 2022).

This process in demonstrated on Figure 10 - How Pact works .

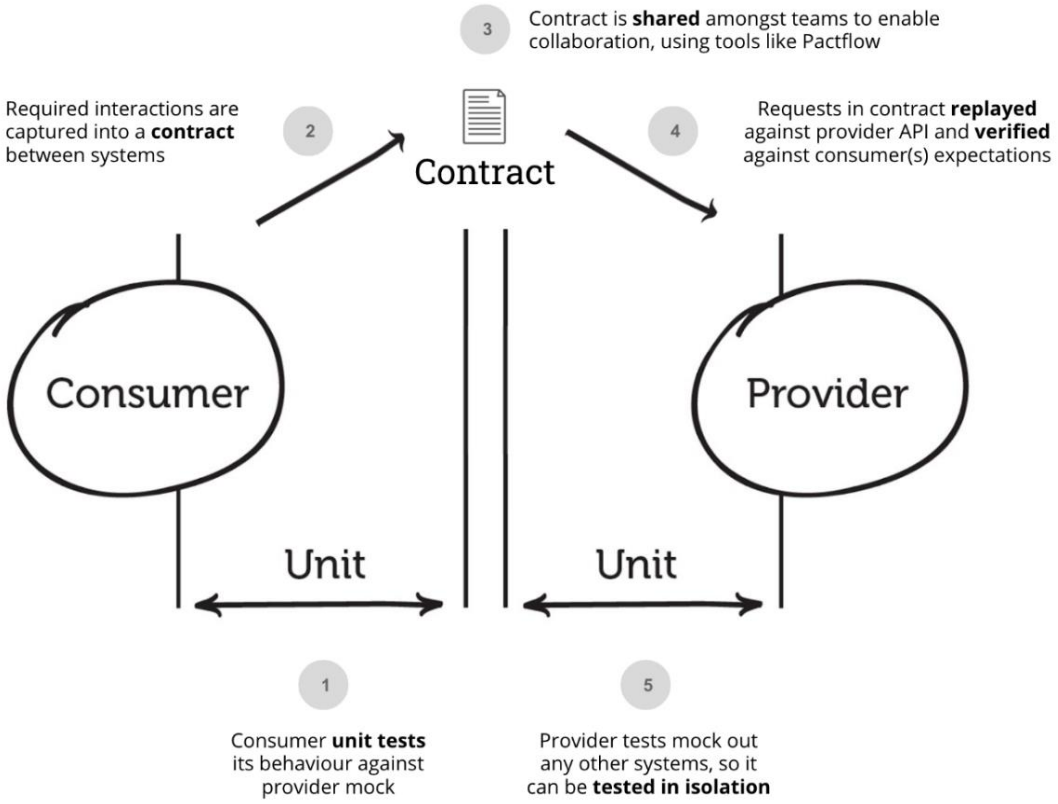

Figure 10 - How Pact works (Pact, 2022)

Pact also provides a remote platform to manage contracts called Pact Broker. It enables you to collaborate on contracts across teams and do CI/CD integration (manage contracts across git branches and environments; create rules for results-based promotion). (Pactflow, 2022)

3.3.7.2 Spring Cloud Contract

Spring Cloud Contract is a Java Spring boilerplate project with several helpers to implement consumer driven contract tests. It provides a Contract Definition Language (DSL), in Groovy or YAML, that is used to define contracts and generate test resources such as: JSON stub definitions, test data, messaging routes stubs and even acceptance tests for the producer application (Dudczak, et al., 2022). It supports contract testing for both HTTP communication and messaging.

It supports Producer-driven contract test where the producer application defines the contracts, uses Spring Cloud Contract to generate tests that verify its compliance with the contracts and, after the tests successfully pass, generate stub artifacts to be used by the consumers to mock the server in their local tests (Dudczak, et al., 2022).

But it also supports Consumer-driven contract tests, where consumers define the contracts and share them with Producers. Producers then use those contracts to generate acceptance tests and can verify which changes will break a consumer (Dudczak, et al., 2022).

3.3.7.3 Prism

Prism is an open-source tool that can generate Mock HTTP servers from OpenAPI v2/v3 documents. It allows developers and consumers to see how an API will work before it is even built (Spotlight, 2022). Based on the specification, Prism validates requests and generates dynamic examples for responses. “When the API has been built, Prism can continue to help by proxying requests to the real server and reporting if anything is different.” (Spotlight, 2022).

3.3.8 End-to-end testing tools

End-to-end testing tools should provide several functionalities for facilitating the creation and running of end-to-end tests. Those tools should be capable of exercising the full system, mimicking the end user in a production environment, and proving the overall system functionalities are working as expected.

3.3.8.1 Selenium

Selenium is an open-source tool for automating web-browser-driven application testing. It is core piece, called WebDriver, is an “API and protocol that defines a language-neutral interface for controlling the behavior of web browsers.” (Software Freedom, 2022), which allows developers to use it on testing scenarios to verify that web applications behave as expected and designed.

3.3.8.2 Postman

As referred in section 3.3.3.3, Postman is a tool for building and using APIs, commonly used by developers during an API lifecycle to design, build and test API’s requests and responses. It can be used for E2E testing back-end systems: coordinating requests to APIs, sharing data between them, and checking if responses match expectations. Postman feature called Flows allows for test scenarios to be design as a series of requests that propagate data between them and do configured validations (Postman, 2022). Postman also supports the automation of such test scenarios in CICD pipelines thought its extension program Newman.

3.3.9 Containerization-based testing tools

Containerization-based testing tools take advantage of container technologies to run dependencies in tests. They manage the configuration and setup of containers for us and don’t required developers to run a complex setup on their machines to run.

3.3.9.1 Testcontainers

Testcontainers is a Java library that integrates with JUnit that allows the use of Docker containers within our tests (North, 2021). It makes integrations tests easier by managing and setting the application dependencies as containers.

3.4 Summary

From what has previously discussed in this chapter it's possible to conclude that testing microservices based applications is a complex task. Unit testing, thought, is very similar to a monolithic architecture – possibly even simpler – so is considered a mature topic that doesn't represent a current challenge. On the other hand, Integrations testing – as well as other testing types with bigger scopes – are considered more complex due to the distributed and independent character of a microservice architecture.

This work will focus on the challenges present when Integration, Component, Contract and End-to-End testing microservice-based applications.

4 Value analysis

This chapter defines the value analysis proposed for this work. The following sections contains definitions related to value analysis, the appliance of the New Concept Development (NCD) model (Koen, et al., 2002) and the generated value proposition.

4.1 Terminology

Over the years "*value has been defined in different theoretical contexts as need, desire, interest, standard /criteria, beliefs, attitudes, and preferences*" (Nicola, et al., 2012). The creation of value in any business activity "*is about exchanging some tangible and/or intangible good or service and having its value accepted and rewarded by customers or clients, either inside the enterprise or collaborative network or outside*." (Nicola, et al., 2012). Therefore, value creation is driven from the relationship between the customer and supplier, as "*a trade-off between benefits and sacrifices perceived by customers during a supplier's offering*" (Walters & Lancaster, 2000).

The perceived value or value for the customer "is the consumer's overall assessment of the utility of a product based on perceptions of what is received and what is given" (Zeithmal & Bateman, 1990). Different customers will interpret the value of a product in diverse ways.

Value Analysis is a "systematic, formal and organized process of analysis and evaluation" (Rich & Holweg, 2000). Its primary objective is to increase the value provided to the customer at the lowest cost possible. "This can also be expressed as maximizing the function of a product relative to its cost: Value = (Performance + Capability)/Cost = Function/Cost" (Rich & Holweg, 2000).

Value Proposition "is an overall view of a company's bundle of products and services that are of value to the customer." (Osterwalder, 2004). The value proposition should answer the following questions:

- What is your product?
- Who is your target customer? For whom do you provide value?
- What value do you provide?
- Why is your product unique?

4.2 New Concept Development Model

According to (Koen, et al., 2002), "*the innovation process is divided into three processes: the fuzzy front end (FFE), the new product development (NPD), and the commercialization."*. The

New Concept Development Model (NCD model) is a theoretical construct proposed by Peter Koen to provide common terminology and a set of best practices for the innovation fuzzy front end (FFE) process (Koen, et al., 2002).

It consists of three key parts (Koen, et al., 2002):

- The engine portion drives the five front-end elements and is fueled by the leadership and culture of the organization;
- The inner area defines the five key elements composing the FFE: opportunity identification, opportunity analysis, idea generation and enrichment, idea selection and concept definition;
- The influencing factors consists of organizational capabilities, the outside world (i.e., distribution channels, laws, customers, and competitors), and the enabling science that will be utilized. These factors affect the entire innovation process and are considered relatively uncontrollable.

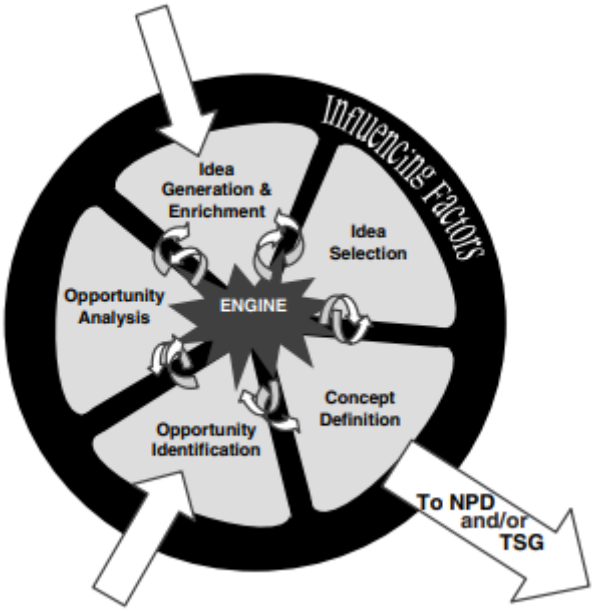

Figure 11 - The new concept development (NCD) model (Koen, et al., 2002)

The NCD model, shown in Figure 11, is represented in a circular form to visually suggest that ideas are expected to flow back and forward through the elements. (Koen, et al., 2002) defend that "*while the inherent looping back may delay the FFE, it typically shortens the total cycle time of product development and commercialization*" due to clearer definition of requirements, risk sources and of the business plan.

4.2.1 Opportunity Identification

Opportunity Identification is the element of the NCD where the organization identifies opportunities to pursue. Such business and technological opportunities may be driven by a

competitive thread, market growth or the possibility to operate more effectively and efficiently (e.g., speeding up processes, reducing costs) (Koen, et al., 2002).

This element proposes several methods to envision the future and assess an opportunity: "roadmapping, technology trend analysis and forecasting, competitive intelligence analysis, customer trend analysis, market research, and scenario planning" (Koen, et al., 2002). For this work, the technique chosen was technology trend analysis.

4.2.1.1 Technology trend analysis – trend to adopt a microservice architecture

Following the arguments exposed on the Context chapter (see 1.1), it is possible to identify a technological trend, in the software architecture world, to use a microservice architectural approach to design and develop new projects, and to migrate existing systems to, due to limitations in the initial system. Martin Fowler emphasizes that "*we've seen many projects use this style in the last few years, and results so far have been positive*" and "*for many of our colleagues this is becoming the default style for building enterprise applications*" (Lewis & Fowler, 2014). Although a microservice architecture may present many advantages (as enumerated in the previous chapters), it also raises various challenges to the organizations that choose to apply it.

Using this technique, several challenges related to the adoption of a microservice architecture were identified as opportunities and advanced further to the next phase of the NCD model.

4.2.2 Opportunity analysis

Opportunity analysis is when opportunities get assessed further, to confirm their viability - "*this involves making early and often uncertain technology and market assessments"* (Koen, et al., 2002). It uses many of the same techniques proposed on Opportunity Identification but with a different scope – before it was used to determine the existence of opportunities, now its goal is to define an opportunity with more detail and verify its appropriateness and attractiveness (Koen, et al., 2002).

4.2.2.1 Market segment assessment - regarding trend to adopt a microservice architecture

To understand the possible market size of such opportunities, multiple reports and systematic literature reviews were gathered and studied. O'Reilly ran a survey in the beginning of 2020 with more than 1500 responses and reports that "*Almost one-third (29%) of respondents say their employers are migrating or implementing a majority of their systems (over 50%) using microservices."* (O'Reilly, 2020). IBM, on a survey ran in 2021, reports that "*a majority of current microservices users and nonusers believe the benefits are real and they're likely to increase their reliance on microservices or adopt a microservices development approach in the next two years*" (IBM Market Development & Insights, 2021).

4.2.2.2 Customer assessment – research to understand customer needs

Market research, done by reviewing the literature acquired on microservices challenges, identified that there are few microservice-based testing approaches and patterns. A 2019

systematic literature review done on Microservice Testing Approaches identified that most studies use different models, frameworks, and tools to recommend a solution for microservices testing (Ghani, et al., 2019) – highlighting a lack of universal strategies, appropriate for different platforms and technologies, for microservice testing. João Neves also points out the Testing Complexity on the Microservices architecture as one of the most common problems when migrating over from a monolithic system (Neves, 2019).

As referred to in the Problem chapter (see 1.2), building a test suite capable of finding defects at the early stage of the software lifecycle is crucial to reduce its impact and, therefore, increase value and reduce costs. Organizations are reportedly struggling to define when, how and what to test when first in contact with a microservice architecture. The lack of clear guidelines for this process, as for its automation, is identified as a customer need not being met by current products.

Applying those techniques, it was possible to highlight a microservice challenge that currently is not being addressed thought a scientific approach and can provide organizations with more effective processes and reduced costs: testing microservice-based systems.

4.2.3 Idea generation and enrichment

This element of the NCD model consists of "*the birth, development, and maturation of a concrete idea"* (Koen, et al., 2002). In Idea generation, ideas are created, evolve, tear down, and change because of further examination and discussing within the organization. The author suggests techniques like brainstorming sessions and idea banks to accelerate idea identification, but highlights ideas may be generated by "*anyone with a passion for particular idea, problem, need, or situation*" or even from outside of the organization. "*Once the idea is identified, many different creativity techniques can be applied to generate and expand upon it*" (Koen, et al., 2002).

The brainstorming technique was used to generate ideas around the opportunity identified and analyzed. Several ideas were generated around the theme of testing microservice-based applications that try to answer, in some way, to the previously identified customer needs. The brainstorming session outcome (generated ideas) is described below.

1. Define a set of universal guidelines and best practices for integration testing microservice-based applications – this idea focusses on providing a guideline to help organizations tackle the difficulties faced when testing microservice-based systems, mainly integration features.
2. Develop a method for testing microservice-based applications regarding their quality attributes and non-functional requirements.
3. Implement a solution capable of utilizing the OpenAPI Specification (OAS) to generate Provider-driven contract tests, or even extended it to facilitate consumer contract tests.

4. Develop a solution to facilitate automated contract testing on messaged-based integrations and event sourcing systems.

4.2.4 Idea selection

In the idea selection element of NCD model the objective is "*selecting which ideas to pursue in order to achieve the most business value*" (Koen, et al., 2002). There is not a single formal process that guarantees a suitable selection - selection methods can vary from personal judgments and preferences to fully formal decision processes.

To decide about the most valuable idea, an Analytic Hierarchy Process (AHP) will be done taking into consideration the alternatives identified on Idea generation and enrichment element.

The Analytic Hierarchy Process (AHP) is a method, developed by Thomas L. Saaty, in 1980, to aid multi-criteria decision-making processes (Nicola, 2018). A multi-criteria decision method crosses alternatives with criteria using numeric techniques to help deciders choose an alternative. The AHP method divides the problem into a hierarchical decision tree to facilitate its assessment and comprehension. It also allows the use of quantitative and qualifying factors for the process evaluation (Nicola, 2018).

Figure 12 shows the hierarchic tree produced by phase 1 of the AHP method. At the first layer is the problem definition of this work. At the middle layer, several criteria to be used to evaluate each alternative are enumerated and are further explained below:

- Estimated effort – if the idea estimated effort fits the deadlines of this work;
- Author personal experience – if the author has more experience with the idea concepts, it is more likely to succeed;
- Exists in market – if the idea is completely new, exists in the market with flaws or meets all customer requirements;
- Technical success probability – if the idea is achievable taking into consideration the available resources.

The lower hierarchical level represents the alternatives - Idea Generation and Enrichment outcomes.

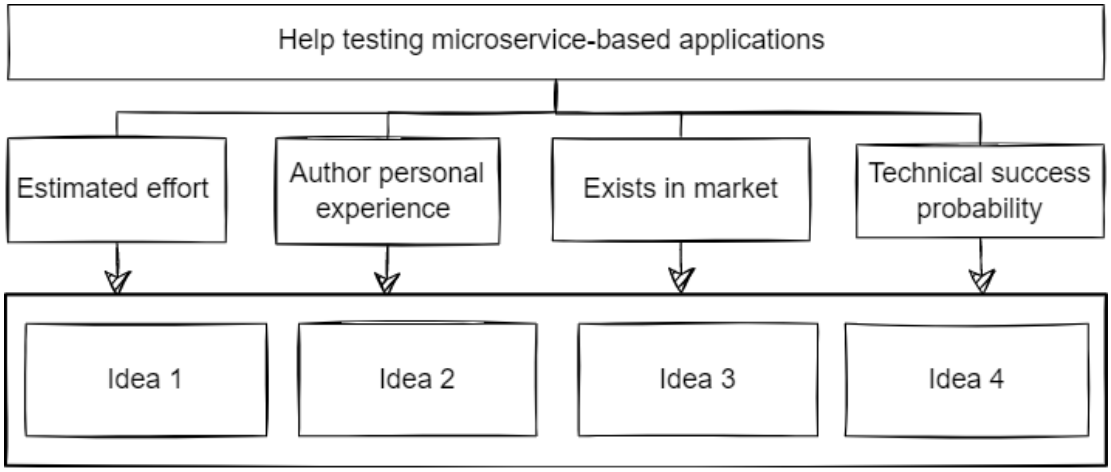

Figure 12 - AHP hierarchic decision tree

In phase 2, a pair-to-pair comparison matrix (A) was created, see Table 2, defining the importance of weight values between each previously identified choice criteria (C). The value chosen for each criteria pair (Ci, Cj) is based on the judgement scale proposed by Saaty, displayed on Table 1.

Table 1 - Saaty AHP ratio scale (Nicola, 2018).

Intensity of importance	Definition
1	Equal importance
3	Somewhat more important
5	Much more important
7	Very much more important
9	Absolutely more important
2, 4, 6, 8	Intermediate values

Criteria are positioned on the column (i) and row (j) headers and values must follow these rules: (Nicola, 2018)

- $if\ A_{ij} = \alpha, then\ A_{ji} = \frac{1}{\alpha}, \alpha \neq 0$
- $if\ C_i\ was\ the\ same\ relative\ weight\ as\ C_j, then\ A_{ij} = 1, A_{ji} = 1\ and\ A_{ii} = 1 \forall i$

Table 2 - AHP criteria comparison

	Estimated effort	Author personal experience	Exists in market	Technical success probability
Estimated effort	1	1/4	1/5	1/3
Author personal experience	4	1	1/4	1/2
Exists in market	5	4	1	3
Technical success probability	3	2	1/3	1
SUM	**13**	**7 + 1/4**	**1 + 47/60**	**4 + 5/6**

Next, the matrix A (Table 2) is normalized to represent all the criteria weight with the same unit value. This is achieved though the division of each value by the total sum of its correspondent column (i) and is displayed on Table 3 below. The last column of Table 3 "Relative priority" represents the weight of each criterion. It is designated priority-vector and is calculated by the row mean divided by the number of columns - 4 in this case (Nicola, 2018).

Table 3 - Normalized AHP matrix

	Estimated effort	**Author personal experience**	**Exists in market**	**Technical success probability**	**Relative priority/Mean**
Estimated effort	0.0769	0.0345	0.1121	0.0689	0.0731
Author personal experience	0.3077	0.1379	0.1402	0.1034	0.1723
Exists in market	0.3846	0.5517	0.5607	0.6207	0.5294
Technical success probability	0.2308	0.2759	0.1869	0.2069	0.2251
SUM	**1**	**1**	**1**	**1**	**1**

In the ATH method phase 4, the consistency of the previously calculated priority weights is evaluated. This is done by multiplying the initial matrix (Table 2) by the priority-vector (Relative priority column of Table 3) and then utilizing the resulting matrix (Ax) to calculate its λmax, using the following equation:

$$Ax = \mathrm{l}_{max}\, x$$

Calculation:

$$\begin{bmatrix}0.2963\\0.7096\\2.2594\\0.9637\end{bmatrix} = \mathrm{l}_{max} \begin{bmatrix}0.0731\\0.1723\\0.5294\\0.2251\end{bmatrix}$$

$$\mathrm{l}_{max} = avg(4.0533, 4.1184, 4.2678, 4.2812\ ) = 4.1802$$

$$\mathrm{l}_{max} = 4.1802$$

After calculating l_{max}, the Consistency Index (CI) can be calculated to calculate the Consistency Reason (CR).

$$CI = \frac{\mathrm{l}_{max} - n}{n - 1}$$

$$CR = \frac{CI}{RI}$$

The RI (Random Index) is obtained from a standard table, calculated for squared matrixes of n order, that defines RI values in function of the number of criteria factors used. For this work

$n = 4$, so $RI = 0.90$. If the resulting CR is less than 0.10 it means the values are consistent. Calculation:

$$CI = \frac{4.1802 - 4}{4 - 1} = \frac{0.1802}{3} = 0.06$$

$$CR = \frac{0.06}{0.90} = 0.0667$$

As CI is less than 10%, we can conclude the values are consistent.

In phase 5, a comparison matrix is built for each criterion, following calculations done on phases 1 to 3 again, but relative to the importance weight values between each alternative. The tables below (Table 4, Table 5, Table 6 and Table 7) Represent those calculations.

Table 4 - AHP criteria "Estimated effort" alternatives comparison matrix with priority-vector

Estimated effort	Idea 1	Idea 2	Idea 3	Idea 4	Priority vector
Idea 1	1	1/5	1/5	1/5	0.0618
Idea 2	5	1	1/3	1/2	0.1932
Idea 3	5	3	1	2	0.4572
Idea 4	5	2	1/2	1	0.2878
SUM	**16**	**6 + 1/5**	**2**	**3 + 5/7**	**1**

Table 5 - AHP criteria "Author personal experience" alternatives comparison matrix with priority-vector

Author personal experience	Idea 1	Idea 2	Idea 3	Idea 4	Priority vector
Idea 1	1	2	1/3	1/3	0.1339
Idea 2	1/2	1	1/5	1/4	0.0760
Idea 3	4	5	1	3	0.5194
Idea 4	3	4	1/3	1	0.2708
SUM	**8 + 1/2**	**12**	**1 + 6/7**	**4 + 4/7**	**1**

Table 6 - AHP criteria "Exists in market" alternatives comparison matrix with priority-vector

Exists in market	Idea 1	Idea 2	Idea 3	Idea 4	Priority vector
Idea 1	1	1/2	1/4	1/4	0.0909
Idea 2	2	1	1/2	1/2	0.1818
Idea 3	4	2	1	1	0.3636
Idea 4	4	2	1	1	0.3636
SUM	**11**	**5 + 1/2**	**2 + 3/4**	**2 + 3/4**	**1**

Table 7 - AHP criteria "Technical success probability" alternatives comparison matrix with priority-vector

Technical success probability	Idea 1	Idea 2	Idea 3	Idea 4	Priority vector
Idea 1	1	5	2	2	0.4420
Idea 2	1/5	1	1/3	1/3	0.0809
Idea 3	1/2	3	1	2	0.2783
Idea 4	1/2	3	1/2	1	0.1988
SUM	**2 + 1/5**	**12**	**3 + 5/6**	**5 + 1/3**	**1**

In phase 6 of the ATH model, we can build the final priority matrix using the previous alternatives matrixes priority vectors. Then, by multiplying this matrix against the criteria priority matrix, we obtain the compose priorities for the alternatives.

Calculation:

$$\begin{bmatrix} 0.0618 & 0.1339 & 0.0909 & 0.4420 \\ 0.1932 & 0.0760 & 0.1818 & 0.0809 \\ 0.4572 & 0.5194 & 0.3636 & 0.2783 \\ 0.2878 & 0.2708 & 0.3636 & 0.1988 \end{bmatrix} \times \begin{bmatrix} 0.0731 \\ 0.1723 \\ 0.5294 \\ 0.2251 \end{bmatrix} = \begin{bmatrix} 0.1752 \\ 0.1417 \\ 0.3781 \\ 0.3050 \end{bmatrix} \cong \begin{bmatrix} 0.17 \\ 0.14 \\ 0.38 \\ 0.31 \end{bmatrix}$$

In the final phase of the ATH, the best alternative is chosen. For this work, the ATH method suggests that the alternative “Idea 3” is the more indicated, taking into consideration the criterion used and their respective weight importance’s.

4.2.5 Concept Definition

Concept definition is the final element of the NCD model. To pass to the next phase of the innovation process an idea needs to make a compiling case for investment through the definition of a “win statement” (Koen, et al., 2002). The gatekeepers, whose decision dictates if an idea advances further, follow specific organization guidelines to assess ideas. Those guidelines may address business objectives, corporate vision, size of the opportunity and benefits, risk, and more organizational factors. “*Developing a business plan and/or a formal project proposal for the new concept typically represents the final deliverable for this element*” (Koen, et al., 2002).

For this work, the value proposition will be the concept definition. This is defined in the next section.

4.3 Value proposition

This work provides a set of guidelines, approaches and best practices that apply to any organization testing microservice-based applications and aims to help them produce tests with more quality and less costs, independently of the technology being used to build such applications.

5 Microservices testing scenarios analysis

In this section a microservice brownfield project will be analysed to better understand how functional and non-functional requirements are reflected on the testing strategy.

5.1 Project overview

The project described in this section is a fictional system designed to showcase testing scenarios in a microservice architecture. A brownfield project domain was chosen so the testing scenarios analysis is applicable to many production-like microservice-based applications. It was also important to choose a domain which has requirements that exercise several modules of a typical architecture, so tests can be designed for a bigger variety of scenarios.

The problem domain chosen was an e-commerce omnichannel platform as-a-service. In the following sections, a fictional system is presented, some of its requirements and use cases are enumerated, and an architecture is proposed for the system, based on microservices good practices and patterns. Since the focus here is the testing strategy the requirements gathering, and analysis phases will not be performed. Instead, general retail business use cases in functional areas as orders, products and inventory will be adopted for the project scope. Also, to simplify the business case, some problem areas will be ignored.

5.1.1 Business description

The e-commerce as a service platform is a business which intends to accelerate the digital transformation of shops and retailers by providing out-of-the-box solutions for their digital enablement. The platform will enable businesses to sell their products online while taking care of everything for them - all the way from the website to order fulfilment and delivery. Each business partner will have the possibility to have their own customizable website and

mobile applications, have automatically generated sales and business intelligence reports and integrate their own systems for extra customizability.

5.1.2 Requirements

The main stakeholders for the platform will be the partner and the partner customer base.

From the partners perspective, it needs to be able to:

- Manage products and catalogue;
- Manage orders and returns;
- Manage inventory and stock points;
- Manage prices and promotions;
- Integration of inventory and orders with their system.

From the customers perspective, the system needs to support standard e-commerce use stories:

- Searching products;
- Buy flow: add products to bag, checkout, shipping information.
- Account management;
- Share shipping updates;
- Return flow.

5.2 System architecture

The system will follow a microservice-based architecture and apply the key microservices characteristics shared in section 2.1. Below, for each characteristic, a brief description is given of how it will be reflected in the architecture: (Lewis & Fowler, 2014)

- Componentization via services – the system is composed of services that are independently deployable using containerization. “Although they work together to fulfil larger business operations, services only communicate via messaging and API calls, and do not know about other services internal implementation” (Lewis & Fowler, 2014).
- Organized around business capabilities – services are separated by the identified business areas, as Orders, Product, Price, etc., and have clear boundaries between them.
- Products not Projects – teams are cross-functional and usually assigned to a boundary of services, being fully independent from other teams regarding their applications.
- Smart endpoints and dumb pipes – services use REST API’s and event-driven messaging to communicate with each other.

- Decentralized Governance – each team has independency to choose the technologies in which they will develop their services, according to each service requirements and the team knowledge base.
- Decentralized Data Management – each microservice manages its own data and has authority about its data structures.
- Infrastructure Automation – the infrastructure team defined a set of guidelines and automations to be used by all applications under the company system. Those enable automated build and promotion of artifacts to shared repositories; configuration-based way to run tests and use results to decide if the build should be successful; automated deploy to the configured environment; possibility to run validations against the deployed application.
- Design for failure – services must be equipped with retry and state-saving mechanisms so failures in its dependencies can be handled gracefully (Clemson, 2014).
- Evolutionary Design – microservices are designed having in consideration the need of fast changes from the business side.

In Figure 13, an overall system architecture is suggested for the e-commerce as a service platform. The system is divided in 4 layers: the Front-end layer, containing the applications responsible for providing User Interfaces for the customers and partners; the API Layer, responsible for exposing the necessary operations that the Front-end layer applications need for their use cases – acting as an abstraction layer from the system; the Back-end layer, containing the microservices responsible for implementing the business logic; the Integrations layer, responsible for connecting the platform with 3rd party applications, providing B2B integration capabilities and customizations for partners.

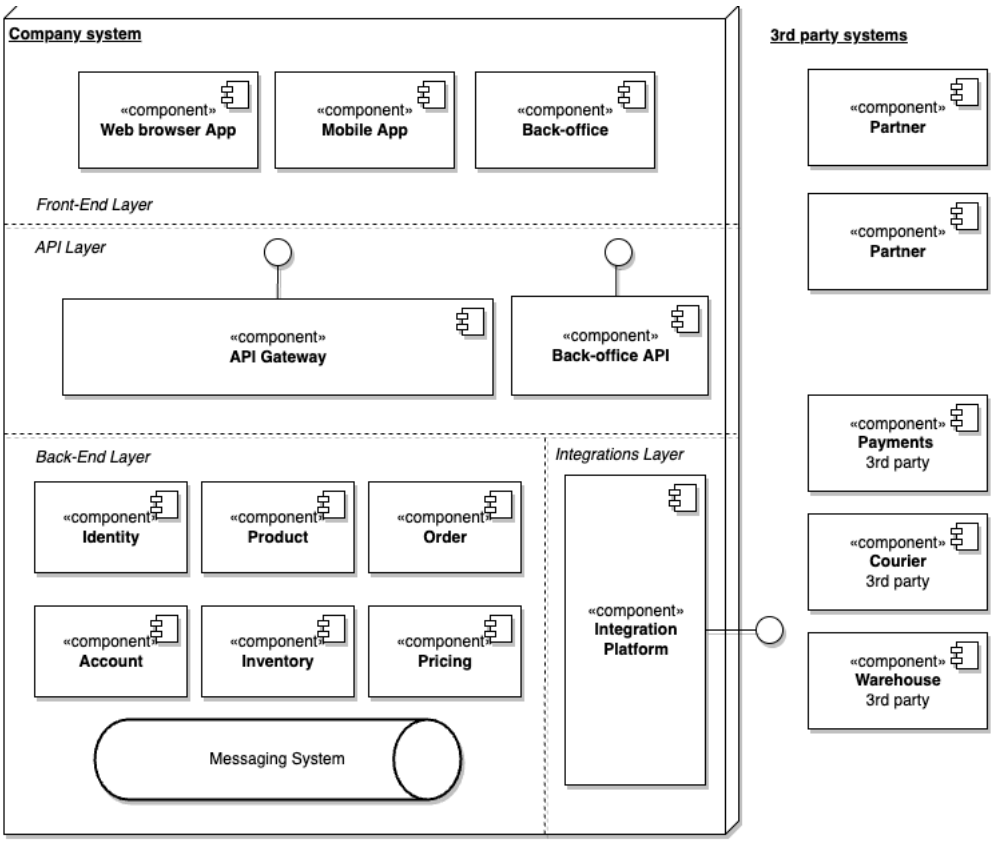

Figure 13 – Suggested system architecture

On the Back-end layer, each component represents a domain boundary. A team will be responsible for each component and can decide the technologies and design that better fits its requirements – while always having in mind the key characteristics of the system mentioned above.

5.3 Testing scenarios

In this section, user stories and use cases of the previously presented project are used to exemplify the identification and implementation of testing scenarios. For each, alternates testing strategies will be discussed and advantages and disadvantages presented. To cover the most scenarios possible, let us consider the testing types presented in section 3.2: Integration testing, Component testing, Contract testing and E2E testing.

5.3.1 Search Products by Category

Browsing the front-end applications are one of the main ways customers search for products in an e-commerce application. The Front-end provides the user with product categories and child-categories to choose from – a category tree. This is used as a filter to retrieve products and is associated with product characteristics (e.g., colour, screen size) to allow a more

detailed product search for each category. In Figure 14, a domain model for the Product boundary is proposed. This boundary contains the main objects that are needed for this use case.

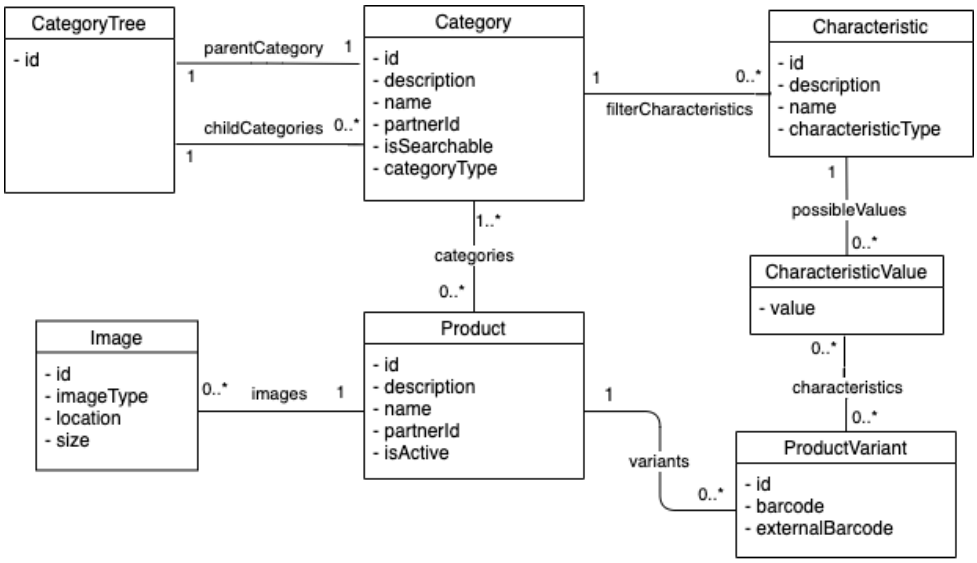

Figure 14 - Product boundary domain

In Figure 15 a sequence diagram for this use case is displayed. After the user/customer selects a category, the Front-end application makes HTTP requests to the API Gateway to retrieve more details about the chosen category. Based on the requests metadata and authentication details, the API Gateway validates the request and deduces the partner to which the front-end application belongs. Then it uses that information to enrich the request and redirects it to the Product boundary API, which should have the information needed to fulfil the request. In parallel to this flow, the front-end also requests a search for products in that category. As those queries impact the user experience, they need to be fast and highly available.

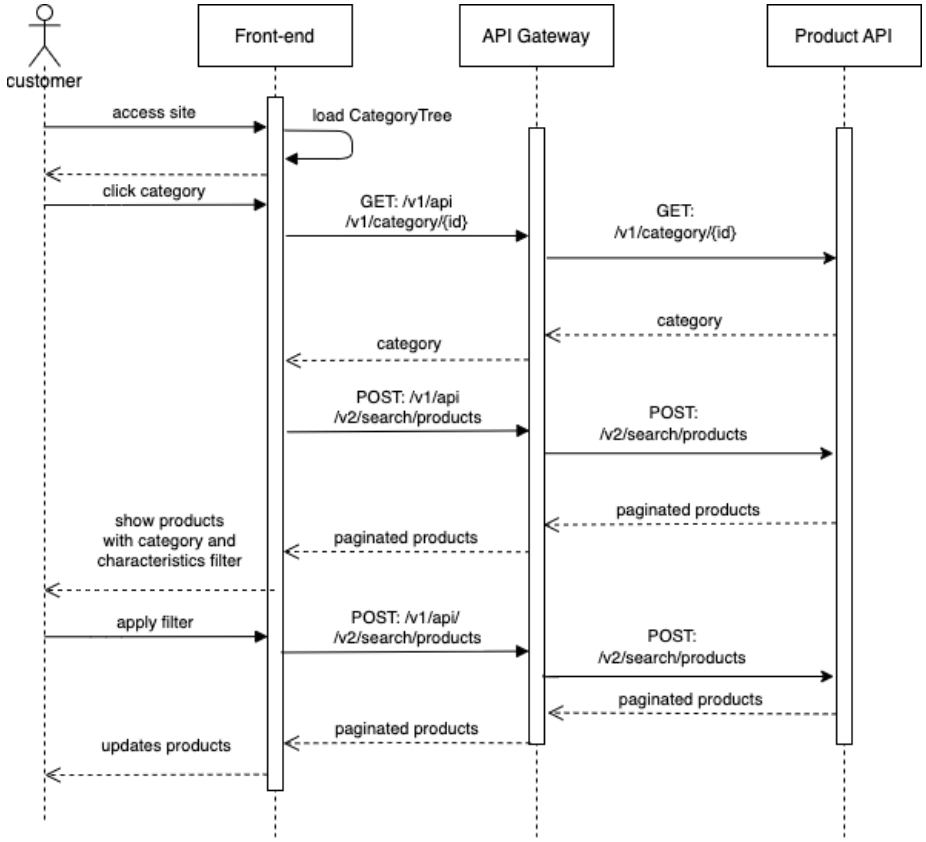

Figure 15 - Search Products by Category Sequence Diagram

To understand how this use case impacts each application and team individually, it is important to break it down into smaller requirements, specific to each one of the components involved. By identifying each component responsibilities and agreeing regarding communication paths and logic, teams can start developing independently – one of the main advantages of microservices. The same can be said about testing. Table 8 shows this breakdown analysis.

Table 8 - Search Products use case breakdown

#	Requirement	Component
1	Users must be able to see main product categories and its chill categories.	Front-end
2	Users can click a category and get redirected to a product search by category.	Front-end
3	Users can apply filters related to the product category when searching products.	Front-end
4	Get partner and location data from request metadata and enrich requests.	API Gateway
5	Create an endpoint that allows searching categories by id and returns its category tree and characteristics information.	Product
6	Create an endpoint for product search by categories and product characteristics that returns products and information about their characteristics, but also their price.	Product

Regarding the Product component, the team decided to use a CQRS-based approach, implementing queries in a read database and commands in a write database to allow queries to be highly available and making updates eventually consistent. Figure 16 shows a high-level design for the Product Service.

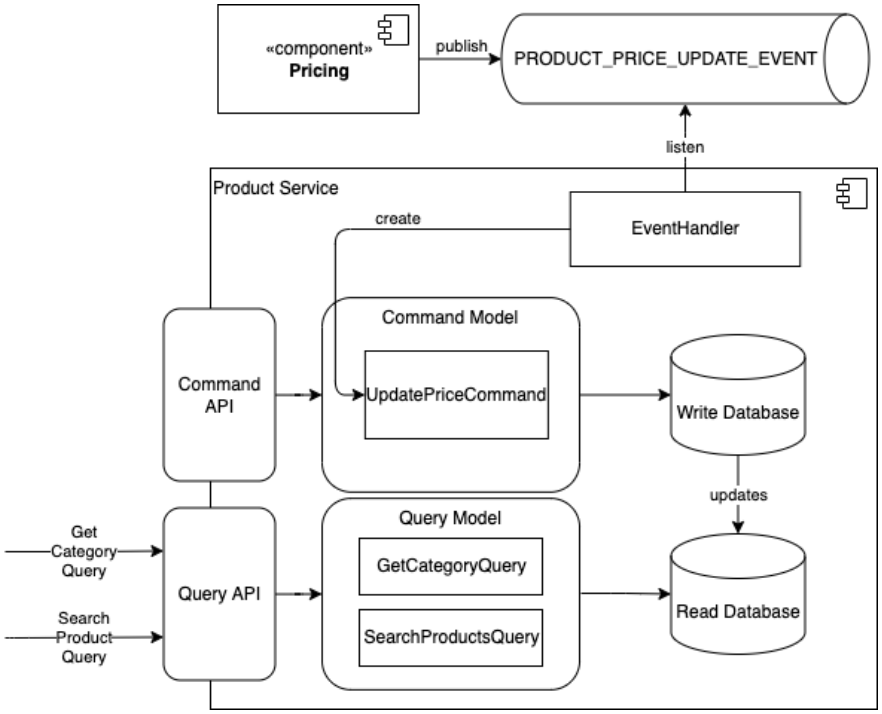

Figure 16 - Product service design

The Get Category Query retrieves category information by its id. The Search Product Query receives a set of parameters that are used to search for products and returns a list of products that match those parameters, but some information that it needs to return does not belong to the product boundary (e.g., price). To get this information, the Product service will subscribe to events from other boundaries that contain the information it needs and build a Projection of the Product entity. Figure 16 shows an example: the Pricing service will publish prices updates to a topic and Product service will consume this to update its product price.

5.3.1.1 End-to-end testing

To identify the E2E testing scenarios, it is necessary to look at the system from the user perspective. In this case, the user will manipulate the Front-end application so that's where end-to-end tests should act. Success and failure scenarios should be identified and designed (Clemson, 2014). Each test would require the following steps:

1. **Setup testing environment** – all applications are deployed and configured correctly.
    a. Run services snapshot build pipelines;
    b. Run services snapshot deploy pipelines pointing at a common environment.

2. **Load test data** – applications should be injected with testing data to make tests consistent and non-reliant on pre-existing data (Clemson, 2014).
    a. Use a tool such as Postman to build a collection that acts on the Product Boundary API to create the data necessary for tests (Categories, Products, Variants, Price);
    b. Automate running that collection for this test suite in a pipeline using Newman.
3. **Run automated tests** – front-end application is used; calls are made to other components and results are verified.
    a. Use a tool like Selenium to manipulate the Browser, that is accessing the Front-end application, in an automatable and repeatable way, validating its content;
    b. Run validations over the correctness of the returned data (it has injected by the test suite, so it was the full context).

5.3.1.2 Contract testing

To identify contract tests, the components and their communication routes should be well defined. Consumers and providers services should agree on a contract regarding the structure, format, and attributes of their communications. Looking at Figure 15, it's possible to identify the following communications:

Table 9 - Search Product Contract tests

#	Consumer	Provider	Resource
1	Front-end	API Gateway	GET: /v1/api/v1/category/{id}
2	Front-end	API Gateway	POST: /v1/api/v2/search/products
3	API Gateway	Product API	GET: /v1/category/{id}
4	API Gateway	Product API	POST: /v2/search/products

Let's analyse the example 4, for the Product API POST: v2/search/products. Its consumers, in this case the API Gateway, should share with it the Contract it uses for its requests and responses. This way the producer will know when running Contract tests if a change can break the API Gateway consumer.

Spring Cloud Contract can be used for this, as the Producer is implemented in Java Spring. The contract is defined using YAML in a shared repository organized using the following structure: "/producer-name/producer-version/consumer-name". The following Contract is proposed for this interaction:

```
request:
  method: POST
  url: /v2/search/products
  bodyFromFile: request.json
  headers:
    Content-Type: application/json
response:
  status: 200
```

```
  bodyFromFile: response.json
  headers:
    Content-Type: application/json;charset=UTF-8
```

Contract for Product API POST /v2/search/products

```
{
    "partner": 12345,
    "productsId": ["00000000-0000-0000-0000-000000"],
    "categoriesId": ["00000000-0000-0000-0000-000000"],
    "characteristicValue": [
        {
            "characteristicId": "00000000-0000-0000-0000-000000",
            "characteristicValue": "value"
        }
    ],
    "barcodes": ["barcode"],
    "externalBarcodes": ["externalBarcode"],
    "priceRange": {
        "low": -1,
        "high": 1000
    }
}
```

request.json file

Using Spring Contract Cloud the consumer can setup a remote repository as the source of Contracts. This framework will pull the consumer contracts and generated acceptance tests for the producer application. The Product API application must make those tests pass if it wants to comply with the Contract establish with the API Gateway.

5.3.1.3 Component testing

Component tests exercise a component through their external boundaries (Clemson, 2014). As identified in Table 8, the Product component is responsible for implementing 2 requirements under its boundary. As it makes available its functionalities using an HTTP REST API it can be component tested using a tool like Postman.

Two Postman collections can be created to validate that both endpoints exposed by the Product API are answering the requirements as expected – possible request/response scenarios should be identified. Those collections can later be automated in the Product Service pipeline using Newman. The same data collections that were created to populate data in the E2E tests can be used here.

To instantiate the Component for running the test, a docker-compose file should be developed containing all the component required internal dependencies, in this case one datastore (used for read and write operations) and a message broker.

5.3.1.4 Integration testing

For integration testing let's look at the Product service as well. In Figure 16 It is possible to identify dependencies of this service, specifically, its databases and the event bus.

Database

Regarding database integration tests, to faithfully test interactions and validate the logic within the integration classes, a real database instance should be instantiated and used in the Integration tests (Vocke, 2018). As there is no contract with the database, this is where the developer should validate that the developed queries and inserts are in fact doing what is expected.

As the application uses Spring JDBC to access PostgreSQL, it is possible to use an in-memory representation like H2 Database. It is also possible to use Testcontainers for Java to create a container for the database instance from Junit tests, exercising more integration points in the tests.

Event bus

Regarding the event bus integration tests, they should focus on validating that the integration logic can, in fact, consume events from the event bus and unmarshal its contents into their class representations.

As the application is developed in Java Spring, it uses the Spring Kafka to interact with the event bus. Spring Kafka Test can be used to test topic subscription and message publication. It is also possible to use Testcontainers to start a Docker container with a Kafka instance and test against it using Junit integrated tests.

5.3.2 Integrating stock with a partner

To have products available for customers to buy on the website it is necessary to have stock allocated. The Ecommerce as a service platform supports two different configurations regarding stock: either the partner uses is own warehouse and makes stock available to the website or the ecommerce platform does this for him (with a default warehouse). For the first use case, an integration of stock between the partner warehouse system and the ecommerce back-end is necessary to guarantee no over/under selling happens. Stock information will be exchanged daily via a CSV file containing all the warehouse stocks with barcode and absolute quantity information. Stock information needs to also be sent from the warehouse every time it loses or adds stock for a product – this is designated a stock movement and contains only a relative stock quantity (e.g., -1, +5).

The Inventory Service is the microservice responsible for managing stock in the ecommerce platform. It implements Event Sourcing and CQRS because it is a service that requires high availability on reads – when customers check product quantities on the websites – and because, from the business context, the logs that result from its event-driven approach are considered important for auditing purposes.

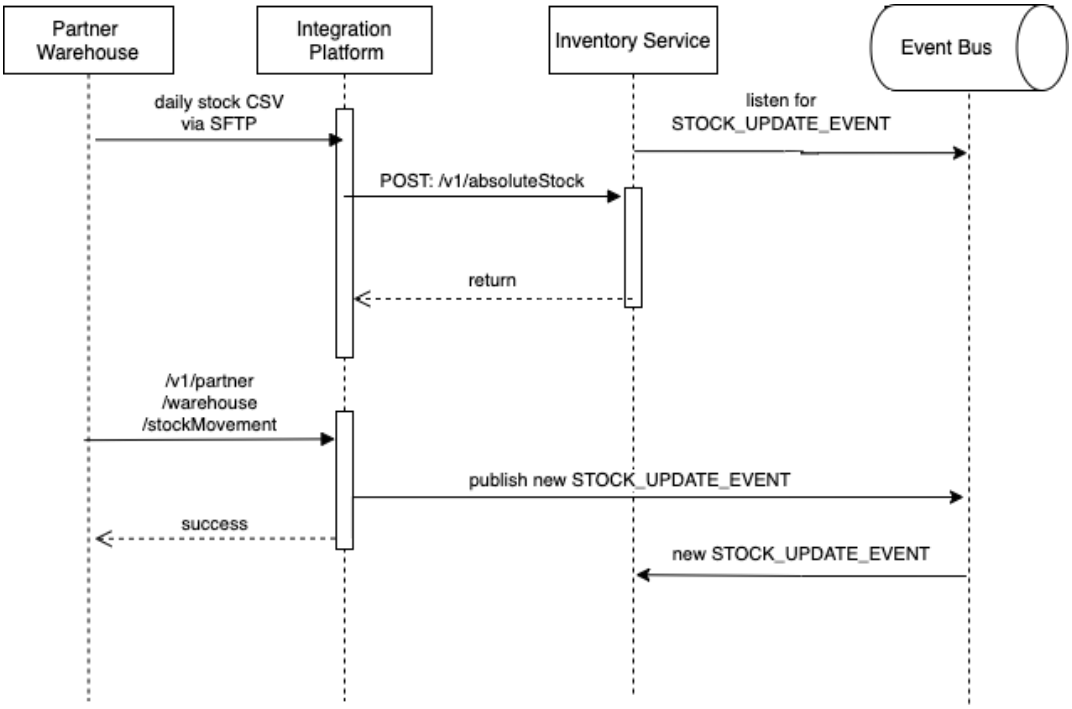

Figure 17 - Stock Integration with a partner - Sequence Diagram

5.3.2.1 End-to-end testing

In this scenario, automated E2E testing can be done by simulating being the partner and using SFTP to add daily stock files and the Integration Platform API to publish stock movements. Furthermore, queries to the Inventory Service API could be made to validate the changes that took place and reach its Read inventory values (that are being used to sell on the website).

5.3.2.2 Contract testing

Contract tests at the partner level are the news here. After agreeing on a Contract definition with the partner, the Integration Platform can use those contracts to validate that their implementation follow it. On its HTTP API, provider-driven contracts test could be used, using as acceptance tests the scenarios agreed with the partner warehouse.

5.3.2.3 Component testing

An interesting scenario to component testing here is the Integration Platform. As it works more as a proxy to the Inventory Service, with enrichments regarding the payloads it receives, component tests should validate this enrichment is done correctly. For the daily stock CSV, a file should be sent to SFTP, where it will be listening. Then, by using a Test Double, HTTP requests make by this component against the Inventory Service API should be recorded and validated regarding its content.

5.3.2.4 Integration testing

For integration testing this application is very similar to the previously presented use case, with the particularity of using Event Sourcing.

6 Guidelines elaboration

In this section guidelines for testing microservices are proposed and justified.

6.1 Testing pillars

In this section, key characteristics that should shape the testing strategy are enumerated and explained. Having these concepts in mind when designing and developing tests will impact their outcome positively. Those should be propagated through the organization if it wants to successfully improve its testing strategies.

- **VALUE**

Question: What does each test try to validate? If it passes successfully, will it bring assurance about the correctness of what?

Tests should bring value to the organization and not overload teams with additional work (Vocke, 2018). The purpose and scope of a test should be clear from its definition. The test description should also make this clear, so developers can easily understand what's wrong when a test fails. Tests should also be non-redundant (Vocke, 2018) – if a test fails, all redundant ones will also fail, resulting in nonadditional advantages – as it promotes duplication in test cases in different system layers.

Developers should have present that the real value of a test comes from guaranteeing that a software requirement is fulfilled correctly. Depending on the requirement, the test could spark the whole system, a single component, or a single class. Moreover, a test keeps guaranteeing a software requirement is correctly implemented over time, acting as a safeguard for refactoring under its layer (Clemson, 2014). By running at the finest granularity possible, a test will prevent duplicating, have more control over its requirement validation and find defects earlier (Vocke, 2018), and so add more value.

- **RELIABILITY**

Question: If this test fails, will it always be for the same reason?

Tests need to be reliable, or, in other words, to always fail for the same reason – a test scope related one. If a test fails for other unrelated reasons – such as network failures or dependency on data contained in external system – confidence in its results will be lost. A test that matches this description is often called flaky (Micco, 2016). Test should be isolated from dependencies that display this behavior (Clemson, 2014).

- **AUTOMATABILITY**

Question: Does the tests require manual intervention? Is it possible to run it automatically in a pipeline?

Automating tests as much as possible will result in tests being run more frequently – as less manual work is required. This will result in defects being found sooner. Developers should use tools and scripts to automate their tests and a step in the service pipeline should be defined to run tests without human interaction (Vocke, 2018). Results of such tests can be used as a promotion requirement for the service, serving a quality assurance (Vocke, 2018).

- **FAST (to run and fail)**

Question: Is the test suite fast? Will developers run it as often as needed?

The faster the test suite runs, the more often it will be run (Vocke, 2018). Developers should focus on keeping the test suite light, removing redundant and bad performing tests – unless they provide a lot of value. Tests should also be fast to fail, meaning, they should detect defects as early as possible in their execution.

- **MAINTAINABILITY**

Question: Will changes in requirements outside-the-scope of a test break it?

Test implementations should try to be as low coupled with the code as possible, allowing for code refactoring with minimum test refactoring (Vocke, 2018). If tests are too tightly coupled with the code, any change in the code will require changes in the tests and so require extra effort and time from teams. Focusing on the test value and scope also promotes maintainability, as each test will have a clear purpose and will not change due to unrelated requirements/code changes.

- **UNIVERSAL**

Question: Does this strategy make sense for every team in the organization? Is it technology independent?

The same advantages that lead the organization to adopt microservices should be applied to the testing strategy. As microservice architectures often follow a distribution approach that mimics the organizational structure (Dragoni, et al., 2017), this should be taken into consideration in the testing strategy.

The testing strategy should also be universal across the organization, technology independent and focused on the same concepts. For example, if an external organization is working with different teams of the same organization, they will expect a similar approach between teams. This also facilitates auditory, change control and quality control across the organization.

6.2 Testing types

An organization should focus on applying these guidelines in the narrow-scope testing types first and build their way to broader-scope tests. After all, those are simpler to implement, require less know-how and infrastructure maturity. Then, it can iteratively refactor broader-scope tests to apply these guidelines, identifying tests that don't add value and that repeat unnecessary validations. On the other hand, in a new project situation, test necessities need to be identified early on during the project Requirements, Design and Analysis phases, as they derive from artifacts and discoveries made in those phases. During development those necessities should be broken down and validated at the finest granularity level possible. Table 10 contains an overview over testing types definition phases, scope and where, in the development lifecycle, they should run.

Table 10 - Testing types characteristics

Testing Type	Definition Phase	Scope/Focus	Run locally	Run at service's pipeline
Unit	Development	Microservice internal logic	Yes	Yes
Integration	Development	Microservice dependencies communication	Yes	Yes
Component	Analysis	Component fulfills its requirements	Yes/No	Yes
Contract	Design	Validate microservices and their dependencies follow the same Contract	Yes	Yes
E2E	Requirements	The full system fulfills its requirements	No	No

This section explores how the presented testing types - that span different system layers and have different scopes - can take into consideration the previously presented test pillars to build a robust testing strategy for microservice-based systems.

6.2.1 End-to-end tests

As use cases and components requirements derive from user stories and global use cases decomposition, also should tests. As soon as the system requirements are defined, so should

the testing scenarios that validate such requirements be. This does not differ between software architectures. E2E tests should look to validate such requirements looking the at the system from the user perspective, and as a black box. (Clemson, 2014)

The challenges within E2E tests in a microservice-based architecture come from the distributed nature of the system. Setting one application and its dependencies may be relatively easy, but setting up various applications, their dependencies, the dependencies-dependencies and configuring them to communicate with one another is a challenge.

How to build an environment for E2E testing?

1. **E2E tests in a production-like environment** (often referred to as sandbox, canary, or QA environment)

A viable option to build an environment ready for E2E tests is to take advantage of the existing Infrastructure Automation, used to deploy applications into production, to deploy applications into a testing environment. This requires microservices to be configurable by environment. The automated CICD pipeline, triggered in a test branch, would build the application using a test configuration and then deploy it. Typically, by setting up all the microservices and their dependencies, we end up with an environment that is a replica of production (Clemson, 2014). This will cause increased costs that the organization may not want to support.

2. **E2E tests in production**

A good alternative to facilitate the environment setup is running E2E tests in production (Vocke, 2018). This also validates that the production environment is setup correctly but will require parts of the system to be configurable in a "test" mode (Richardson, 2019). For example, in tests we may not want to actually cause a physical shipping of an order – certainly not in automated tests - or we may want to create fake products to use in tests, but not make them available to the real customer.

As production environments are often more restrictive for security reasons, this approach brings less flexibility to developers when trying to identify a cause for an error. The same tools that are used to mitigate a production issue need to be used here. It may not be desirable to enable DEBUG logs, access databases to view values, etc.

3. **On-demand deploy and environment setup for E2E testing**

Another alternative is to develop a specific pipeline capable of building and deploying all the necessary components for a specific E2E test suite, inject configurations and run a set of automated tests. But this requires much more customization.

Where and when should E2E tests run?

Furthermore, E2E tests are the only testing type that do not make sense to include in a microservice CICD pipeline – they shouldn't be a requirement for a service promotion and doesn't make sense to call other microservices from one's pipeline. So E2E tests should be stored separately from other testing types, perhaps in a dedicated repository that is organized

around business areas or from the use case perspective. Each test suite should contain metadata about the microservices it requires and validates.

To automate them, a pipeline should be built capable of grabbing the test suite configuration, validating the testing environment state (depending on the chosen approach), loading the necessary test data, running E2E tests, extracting metrics and test results, and cleaning the environment from the loaded test data. Figure 18 displays a possible E2E testing pipeline.

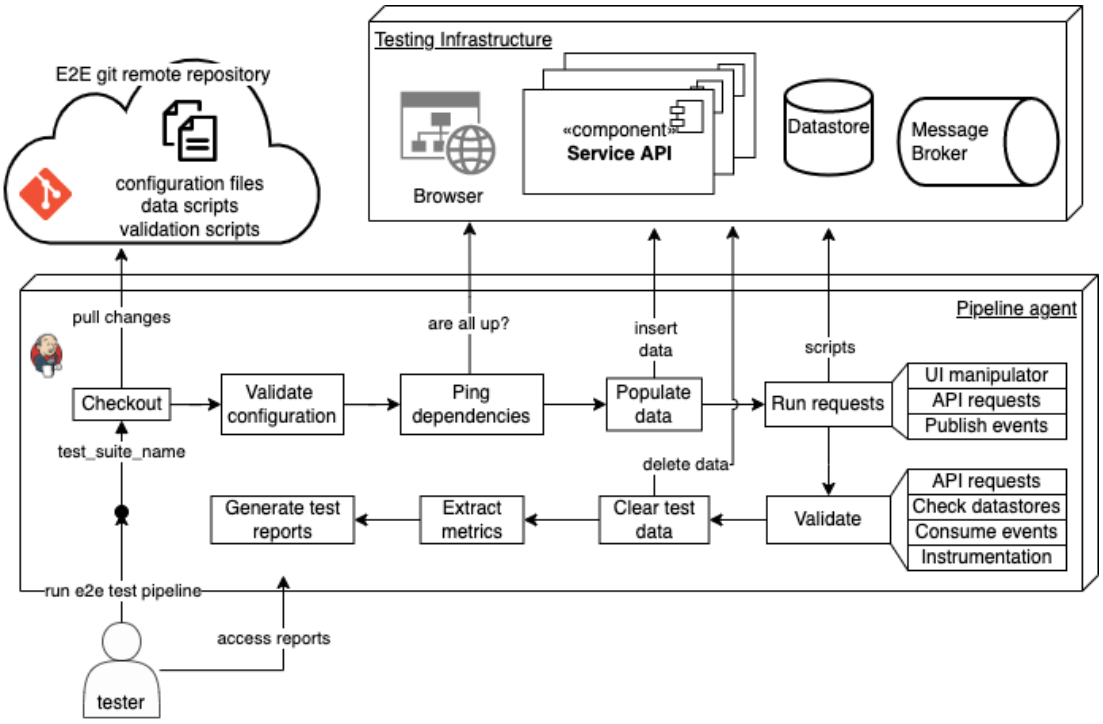

Figure 18 - E2E tests example pipeline

If an organization intends to use E2E tests as a safeguard for refactoring a component, it can use the E2E tests repository to identify the test suites where that component is required and so validate if their changes don't break any existing tests.

How to populate services with the necessary data for E2E tests?

This really depends on the nature of the microservices themselves and on how they store their data. Often, microservices public APIs have support for CRUD operations and can be used to populate them with data and later delete it (Clemson, 2014). If this is not the case, the team responsible for the service should provide a way to inject data into their service and delete it. Examples can be using an internal API or running database scripts.

For example, if a service implements the SAGA pattern, by definition it has a way to undo transactions safely. This can be used to latter delete the test data from the environment.

Who is responsible for E2E test execution?

Organizations also struggle to define who is responsible for E2E tests execution (Vocke, 2018). Obviously, this is dependent on the organization structure and teams' responsibilities, but every team involved in the development of components exercised in the tests should have a role in building and executing them. After all, those are the teams that have the knowledge to identify problems and errors efficiently.

Should all components be used in E2E tests?

If a component is unreliable or does not allow automation, it should be replaced for test doubles, "losing confidence but gaining stability" (Clemson, 2014). For example, if its owner is a 3rd party, like a partner - they may not have the means to provide a testing instance of their component. The team also doesn't have control over the status of this component which can cause tests to fail for the wrong reasons - increasing unreliability.

6.2.2 Contract tests

When a communication between components is identified, a contract should be defined consisting of an agreement regarding the structure, format and attributes of the messages traded between them (Clemson, 2014). Contract tests aim to validate that both consumers and providers comply with the contract. In a microservice-based architecture consumers and providers are the services themselves and, sometimes, an external system. Microservices communicate using HTTP and light-weighted messaging (Clemson, 2014), so those will be the focus of contract tests.

Contract tests should be automated both on the consumer and producer CICD pipelines (Vocke, 2018) and a repository for sharing contracts and contract tests resulting artifacts should be defined, perhaps following a folder structure based on "producer > consumer". There are two possible approaches to implementing consumer tests:

1. **Consumer-driven contract tests** - consumers build pipelines will run contract tests and publish their test results to the shared repository every time a new build is executed. Producers build pipelines will pull the existing consumers results and use them to acceptance test it generated messages.
2. **Provider-driven contract tests** – producers define the contracts and have no visibility over how consumers use them. Their build pipelines will run contracts tests, generate test stubs, and publish those stubs to a common repository from where consumers can pull and use them in their contract tests.

The choice between those approaches can depend on the framework chosen to implement contract tests and which approaches it supports, or, for example, on which service will be built first. Nonetheless, it is possible to co-exist with the two approaches, using one or another accordingly with the situation. Let's analyze the possible communication use case scenarios for contract tests:

- **Microservice -> microservice:** this will be the most common scenario. Here both contracts on the consumer-side and producer-side are applicable. The choice between the two may come to which service will be developed first.
    - **Consumer-driven contract tests** – the major advantage of this approach is that providers will have visibility over the consumers that will be impacted by their changes.
    - **Provider-driven contract tests –** this approach brings a bigger advantage when producers are developed first than consumers and so, consumers can use the stubs generated by the producers to validate their application when they are developing it.
- **Microservice -> external component**: when a microservice consumes from an external component, contract tests may lose value.
    - **Consumer-driven contract tests** - there are less advantages in running consumer-driven contracts tests if the producer will not use the results to validate its changes. Contract tests in this context are less valuable, as they only validate the consumer follows the contract and the provider will have no notion about which consumers may be impacted by its changes. If the producer is an external platform, it will probably just provide API documentation and that will serve as the Contract for the interaction. By translating this contract to the Contract language used in contract testing, the consumer service will be able to use contract test as safeguards for refactoring. This way, consumer contracts tests results are still generated and can be offered to the producer in the future when and if it adopts contracts tests.
    - **Provider-driven contract tests –** as the producer is not owned by the team, this approach can only be used if the external 3rd party also runs contract tests. An option is to create a Facade service that mimics the producer and use it to generate the test stubs to be used on the consumer side.
- **External component -> microservice**: when an external component consumes from a microservice, contracts tests also lose value
    - **Consumer-driven contract tests** – as consumers are not owned by the team or organization, it is not possible to run consumer-driven contract tests. So, the producer service will have no visibility over if a change to the contract will break consumers or not. A possible way to obtain this knowledge is by collecting some of the consumers requests in the production or testing environment, validate manually that those requests following the defined Contract and use them to create consumer-driven test results.
    - **Provider-driven contract tests –** contract tests at the producer side can still be run to guarantee that the producer is following the contract, but the generated stubs won't be used, unless the external

3rd party is also running contracts tests. Nonetheless, these stubs can be provided to 3rd party and act as a way to promote contract tests between the organizations.

As exposed above, both approaches have advantages and disadvantages that should be taken into consideration when planning a testing strategy. Nonetheless, teams should focus on running contract tests frequently to validate that changes don't break contracts. The testing strategy should contemplate running contract tests locally - on the developer machine - so defects can be spotted as early as possible.

Figure 19 shows a possible testing scenario from the consumer service perspective for Consumer-driven contract tests. In this example, contracts are kept in a remote Git repository that is shared between the consumers and the provider service. Note that contracts are only update by the service build and release pipeline. Producer will pull Contracts from the repository and validate their implementations.

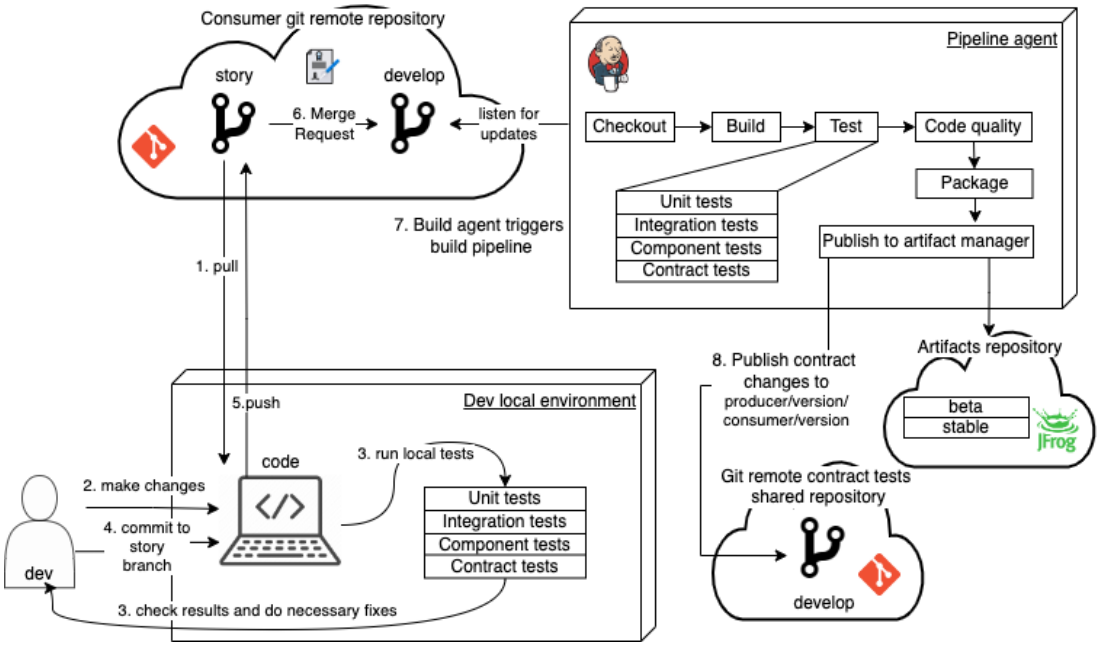

Figure 19 – Consumer-driven contract testing using a shared git repository from the consumer perspective

Figure 20 shows another possible testing scenario from the consumer service perspective but for Provider-driven contract tests. In this scenario Contracts are kept at the provider service repository and are packaged in its generated Stub artifacts. Consumers will download the Producer Stubs and use them to validate their assumptions about the producer.

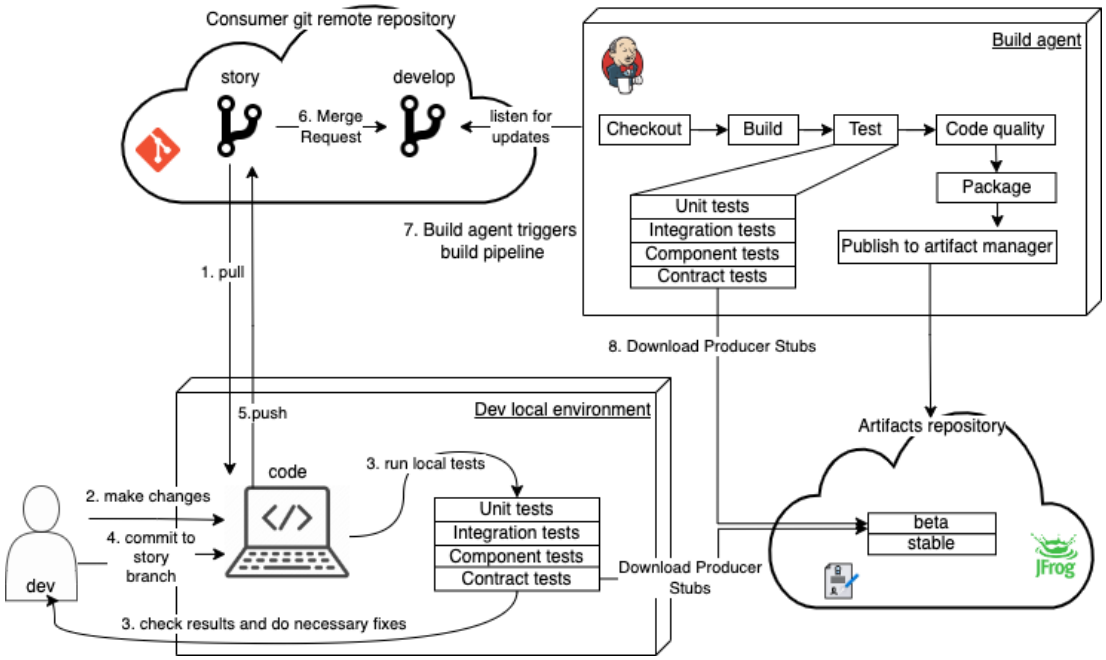

Figure 20 – Provider-driven contract testing using a Stub Storage from the consumer perspective

6.2.3 Component tests

Component tests should limit the scope of testing to a single component of the system (Clemson, 2014). As microservices apply "Componentization via services" (Lewis & Fowler, 2014), components typically represent a single microservice or a group of very tightly connected microservices that, from the outside, are indistinguishable from one.

Component tests should focus on verifying that a single microservice implements its requirements correctly without relying on any external components or dependencies that aren't under its control. Component tests exercise the component using its public API, or by publishes events it listens to (Clemson, 2014). They should not duplicate the validations that contract tests already provide. Furthermore, to make them more valuable, they should be automatable and ready to run both in the developer local machine and in the build pipeline.

Clemson claims that, by running component tests against a service deployed in a testing environment, with real instances of its internal dependencies, all interactions will make use of real networks calls and so, more layers and integrations points are tested (Clemson, 2014). While this is true, this still gives no guarantee that those communications will work in production if the testing environment does not exactly replicate production. This approach also makes it harder for developers to run component tests locally, as they would be required to manually configure and setup the test environment, using either local, or remote testing instances of the service-under-test' dependencies. Furthermore, component tests ran locally would not follow exactly the same steps as component tests ran in the build pipeline.

To keep a more consistent way to run component tests, two approaches are proposed:

1. Start the service-under-test locally with its dependencies replaced by Test Doubles. Build a framework capable of sending requests and validating responses from the service, intercepting requests to the service dependencies – HTTP, SQL, messages, SFTP, cache - and manipulating its context by injecting configurable responses. (e.g., taking advantage of WireMock (WireMock, 2022));

With this approach, component tests can be integrated with the service build manager and run-in conjunction with other testing types. A disadvantage is that the service-under-test doesn't really communicate with an instance of its dependencies, and so, less integrations paths are exercised. Figure 21 shows a sequence diagram for an example component test using this approach.

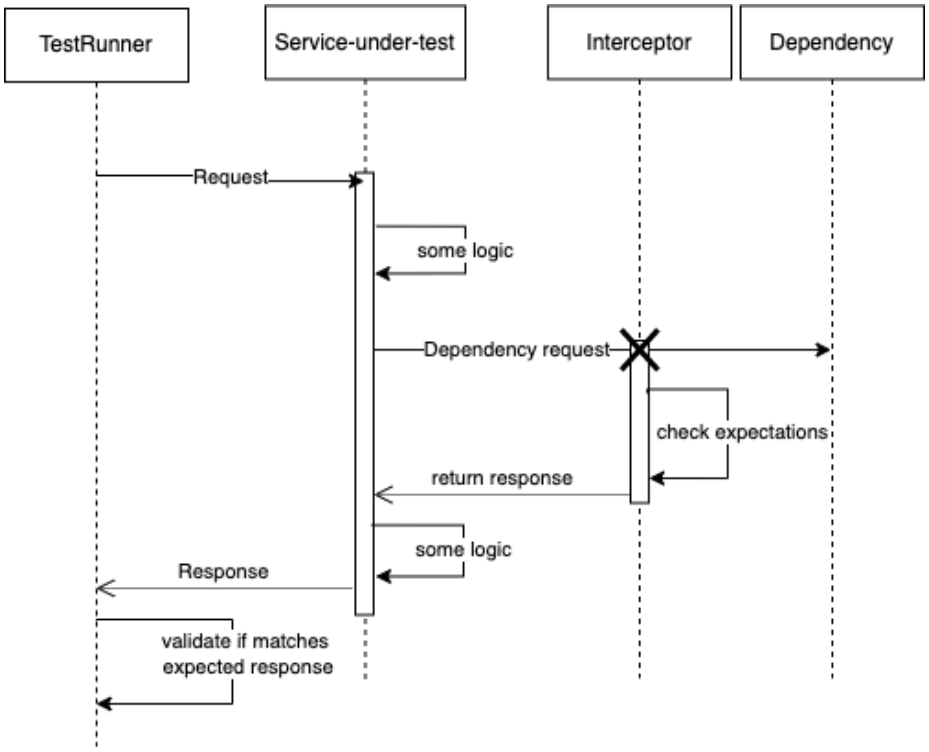

Figure 21 - Component tests approach 1 example sequence diagram

2. Take advantage of containerization to start the service-under-test and its internal dependencies as local containers and use Test Doubles to replace external dependencies. If, for example, Docker is the containerization tool, the official images that correspondent to the dependency technology should be used whenever possible. Another testing tool must be used to run tests against the service UI or public API and validate responses.

With this second approach, components tests are not integrated with other testing types - they must be run separately. Its main advantage is that the communication between the service and its internal dependencies happen between docker containers and so, more

communication logic is exercised and validated. In Figure 22 a scheme of how component tests would run in this approach is displayed.

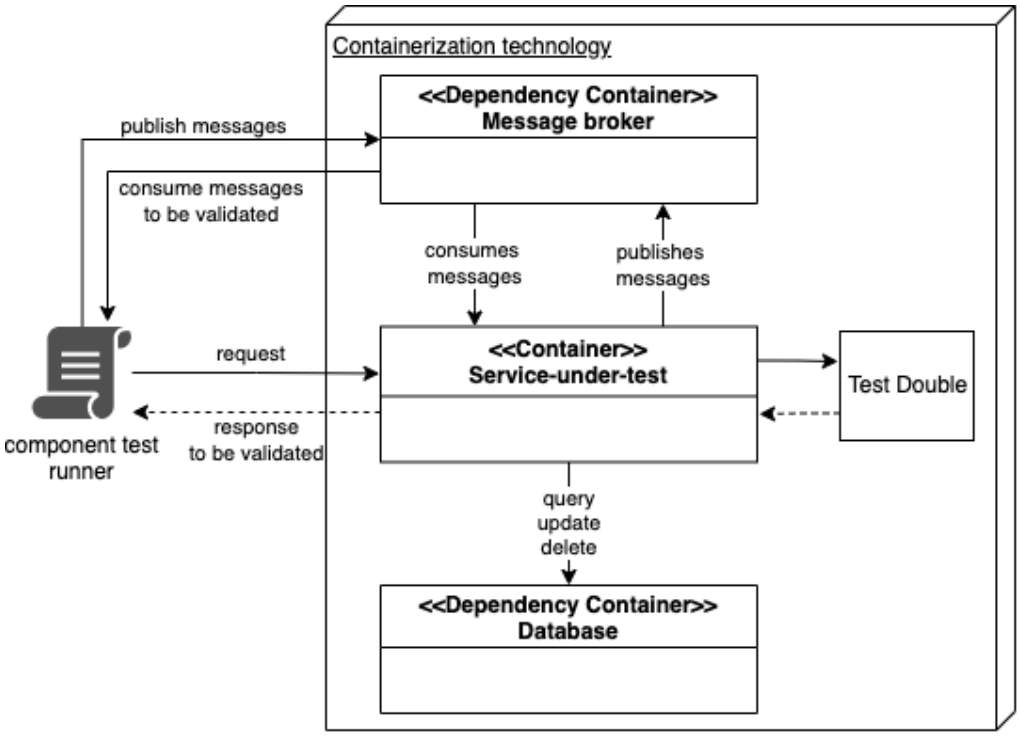

Figure 22 - Component tests approach 2 example diagram

6.2.4 Integration tests

Integration tests look to validate that an application can interact effectively with components that "live" outside of its boundaries (Vocke, 2018). Although they can have a more broad or narrow approach – as explained in section 3.2.2 – Clemson claims that, in a microservice architecture, the narrow approach is usually used (Clemson, 2014). As a microservice architecture implies lots of communication paths between services - to fulfill bigger business requirements due to its componentization nature and implementation of patterns like "Database per service", CQRS and Event Sourcing – integration tests are essential to provide confidence in all those integrations.

Nonetheless, an integration test should focus on "one integration point at a time" (Vocke, 2018), limiting it scope to one dependency only. When this dependency is another component it should try, as much as possible, to abstract from the communication Contract so, if it contents changes, the integration tests don't need to. Note that interactions without a Contract, like with a database or cache instance, should be tested regarding its contents.

Therefore, integration tests should "*focus on protocol level errors such as missing headers, invalid response codes or serialization in HTTP connections*" and "*verify that the integration modules can handle network failure gracefully, as timeouts and slow responses or other*

abnormal behaviors" (Clemson, 2014). This is where the "Design for failure" characteristic of microservices (Lewis & Fowler, 2014) should be tested - not acceptance testing the failure cases but rather test how services react to failure (Clemson, 2014) – do they fail as well? Do they throw unexpected errors?

Integration tests should also run both locally, in the developer machine, and in the service CICD pipeline. This needs to be taken into consideration when designing the testing strategy and choosing how dependencies will be represented. It is possible to characterize microservice dependencies into two types:

a. **External dependencies** – components external to the microservice as another service, a message broker, external databases, and APIs.
b. **Internal dependencies** – components that are part of the microservice as its dedicated database, cache instances, and file system.

To represent those dependencies in integration tests it is possible to:

1. **Use the real dependency**
    a. **External dependencies** – not recommended as they may be heavy to setup and run - increasing test complexity and execution time;
    b. **Internal dependencies** – a good solution as it gives the most assurance about the correctness of the integration. As microservices utilize containerization technology (Lewis & Fowler, 2014), their dependencies are often available as images which can be started up for integration tests and manipulated to simulate error scenarios (e.g., Testcontainers).
2. **Replace it for an in-memory representation** (Vocke, 2018)
    a. **External dependencies** – not possible for dependencies which require its own dependencies to start up;
    b. **Internal dependencies** – ideal for dependencies which have an official in-memory implementation. For those who don't it can still be doable. For example, if the application uses JDBC to access its database and its DBMS doesn't have an official in-memory implementation, other DBMS that supports JDBC can be used. Although, the same driver class is not used which can reveal incompatibilities in the integration in broader-scope tests.
3. **Replace it for a test double**
    a. **External dependencies** – recommended as it is the only way to have control over an external dependency state to force success and failure scenarios consistently (Clemson, 2014). Test confidence in the full integration is lost but stability and confidence in the integration reaction to simulated scenarios is gained;
    b. **Internal dependencies** – a possible solution when it's not possible to start a local instance of the real internal dependency for integration tests.

Figure 23 displays the logic presented above in a cleaner way, using a flowchart. Remember that developers want to have control over the dependencies used in integration tests to be

able to force failures. This must be managed by the Integration test logic, possibly, using a framework.

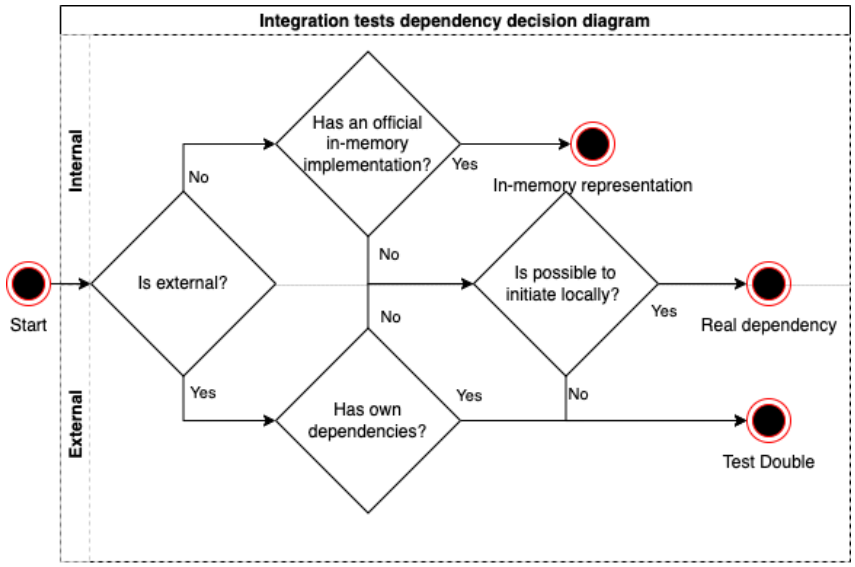

Figure 23 - Integration test dependencies decision flowchart

6.3 Scenarios decomposition strategies

To keep tests in-line with the previously presented testing pillars and strategies, it is necessary to break functional scenarios and decompose broader-scope test scenarios into smaller, more focused tests. Test identification starts in the Requirements phase with E2E testing scenarios identification and ends with the developer, which is writing the code and should test it for defects as soon as possible.

To decompose larger test scenarios into more focused ones it is important to understand how requirements impact each application and team. Typically, a requirement can result in the identification of several use cases and smaller, more specific requirements that are responsibility of individual system components. By identifying each component responsibilities and agreeing regarding communication paths and logic, teams can start developing independently – one of the main advantages of microservices. The same can be said about testing. Figure 24 displays a testing scenario decomposition where: blue parts should be tested with E2E tests; contracts are at the communication boundaries of components; red parts should be tested using component tests; green parts correspond to integration logic; and yellow boxes are the internal Units that are tested using Unit tests.

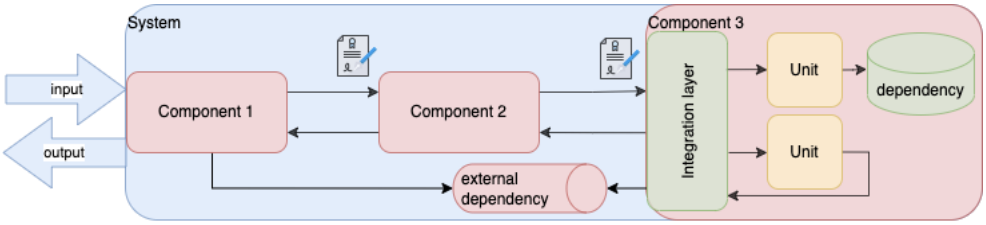

Figure 24 - Testing scenarios decomposition

7 Experimenting and Evaluation

In this section a plan is proposed for the evaluation and validation of this work.

7.1 Problem description

There is a trend in software development to adopt a microservice-based architecture. Despite several benefits such as increased modularization, scalability and maintainability, this approach brings other challenges to the table. When applying this architectural pattern, the testing strategy needs to be adapted.

A microservice-based application presupposes that the various services that compose the system are communication with each other, across network boundaries, to fulfil business requirements and is inherently distributed. Testing a microservice by itself is easier, as it is naturally isolated from the rest of the system, but integration testing becomes more challenging. Microservices also offer several options about where and what to test. As they are independently compiled and built, an incorrect assumption about another module could, potentially, just be noticed in the production environment, when services start failing. When should integration code (microservices) be tested? How can we guarantee excessive testing is not being performed?

7.2 Objective description

This work aims to reach conclusions about the current state of testing on microservice-based systems by analysing and systemizing the current testing methodologies, best practices, and benefits and limitations. The most used testing frameworks in microservices should also be analysed, as well as existing microservice-based systems that are recognized as references

(e.g., Netflix, Spotify) and testing types such as Unit, Integration, Component, Contract and End-to-End tests.

A testing approach, compose of guidelines, methods, and best practices, should be designed, and proposed with the focus of helping organizations emend the challenges they are facing when testing microservices and produce more quality tests.

7.3 Investigation hypothesis

With this work I hope to develop a universal testing approach that can clarify and improve the way a microservice-based system is tested. This testing approach should cover the most relevant microservices architectural patterns and the identified pain points in the problem section. Applying the proposed testing approach will hopefully make tests more valuable for the organization. This will reflect in the indicators identified in the next section.

7.4 Indicators and information sources identification

Table 11 presents the per test indicators that should be collected in order to measure the impact of the suggested testing strategy. It also contains information about the sources of those metrics, how they should be measured and a measure scale. Table 12 contains the same information for indicators that are globally applied to testing strategy.

Table 11 - Test specific indicators

Indicator	Information source	How to measure	Scale
Criticality	Tester/QA	Testers should define how critical a test is for the application. If it fails, this test is not passing in production, what is the impact?	1-100
Reliability	Test reports	Ratio between test real failures and false failures (fail for unrelated reasons to its goal validation)	0-1
Value	Tester/QA	Ratio between the number of real vs false defects. Testers should be responsible for validating if a test failed because of a real defect or not and taking note.	0-1
Speed	Test reports	Average time a test takes to run	n
Maintainability	Tester/QA	Number of changes in the test since its creation (not related with automation)	n

Table 12 - Global test strategy indicators

Indicator	Information source	How to measure	Scale
Fast Defect Detection	Tester/QA	Ratio between the number of defects found in testing vs number defects found on production	0-1
Cost by Defect	Tester/QA	A cost estimation for each defect found (when possible)	n
Test coverage	Test reports	Ratio between code who is covered by tests and code that is not.	0-1
Automation	Tester/QA	Ratio between the number of automated vs manual tests;	0-1
Test suite execution time	Test reports	Ratio between the number of tests per test suite and its execution time	0-1

Those indicators would be collected from a microservice-based project, for a sprint development cycle, before and after the appliance of the proposed testing strategy. The indicators moving in a positive direction would mean there are benefits to applying the proposed testing strategy.

7.5 Evaluation methodology

7.5.1 Projected

To evaluate the investigation hypothesis, the developed testing approach should be applied to a microservice-based project. Ideally, this project would be on-going, already in production and under a testing strategy. The identified indicators would be collected in a sprint cycle of development and analysed accordingly. Then the teams responsible for the design and development of the next sprint cycle of the application would be provided with the proposed testing approach and contextualized about how to apply it. After its appliance, the indicators would be collected again, in a sprint cycle of development, and compared to the ones previously collected. If they are better, the testing approach would be considered successful.

As this is not possible, one or more proof of concept (PoC) projects should be developed to demonstrate the appliance of such testing patterns. Such PoC's should focus on representing the most relevant problems that testing microservice-based applications raises and that are studied by the testing approach. On such PoC's the initial collection of microservice testing approaches should be applied to testing to obtain the initial values for the identified indicators. Then the proposed testing approach should be applied, and the indicators measured and compared to the previously collected. Those PoC's should be developed on different technologies, if possible, to prove the testing approach is technology agnostic.

Another possibility is to prepare a questionnaire about the contents presented in this document, including the microservice architecture, testing types, and the recommended guidelines and have a group of people from the area answer it. This would give a perspective on the usefulness of this guide and an understanding of its gaps.

7.5.2 Done

Due to time and resource limitations, only a PoC project were defined and used to apply guidelines, but no metrics were captured. Instead, in this document, the challenges found when designing tests for the PoC project were exposed and explored. Furthermore, in the guidelines' elaboration, those challenges were considered, and suggestions were proposed to overcome them.

8 Conclusions

This section presents an overview on the proposed and achieved objectives, challenges of this work and potential future work.

8.1 Objectives

The objective of this work was to develop a universal testing approach that can clarify and improve the way a microservice-based system is tested. This testing approach should cover the most relevant microservices architectural patterns and the identified pain points in the problem section. Applying the proposed testing approach will hopefully make tests more valuable for the organization.

It also had the objective to implement those guidelines in a project to measure its value. This would be done by collecting a set of metrics and indicators regarding the testing strategy and results, before and after the guidelines appliance – the proposed indicators and the way to measure they are also considered a contribution of this work.

This works concludes that testing microservice-based applications is really a challenge when comparing to monolith applications, especially broader scope tests – Unit tests are considered simpler. Although, many testing approaches and frameworks exist that can help organizations test their applications correctly, they just need to be used with the right mindset. This is where this works brings more value, as it promotes the right mindset for designing and implementation tests at all system layers. It also proposes a workflow for test definition and decomposition, and solutions for the various testing types.

From the proposed objectives some are considered achieved and others were not achieved at all. In fact, guidelines were proposed that contain lots of relevant information regarding testing microservice-based applications, but those were not validated and evaluate according to the proposed evaluating methodology. This was due to lack of a project to use. Instead,

only a PoC project was proposed, and its flows designed at a high-level to help gain knowledge about test necessities and challenges – using those in the guidelines' elaboration.

8.2 Presented challenges

The major challenge that this work very general-wide theme that it focusses on. The objectives were considered a bit vague from the beginning. It was defined that they would be better aligned over the course of this project but not having a final objective set from the beginning led to a difficulty in the thesis evolution and development. Instead of focussing on practical tasks, the work started to become a knowledge base about Microservices, their characteristics, possible testing types and technologies. This harmed the practical side of this work and so, it was more theorical.

Also, as this work as developed by a student on its own, it was impossible to analyse a real-world project and to apply the guidelines to it. Developing a PoC specific for it revealed not to be a good option as it would not mirror a real world situation (as the same person would implement all the system components and have knowledge about everything; no testing variety characteristic of different developers would exists) and it would require too much development time .Without having a project to implement the guidelines, it is hard to guarantee its quality and see the indicators and metrics proposed in action.

8.3 Future work

Much future work is possible in this area. The knowledge in Microservices is still growing everyday with more organizations adopting this architecture. Furthermore, testing tools that enable all the guidelines proposed in this work aren't completely develop and centralized. Some workarounds that are necessary right now could be removed if a specific tool would be developed for it.

Some ideas that would help organizations test applications and are related with this work are:

- Mind Map based decomposing of E2E testing scenarios
- Mind Map to help find the testing type responsible for testing a defect
- Defining a rigid structure for test metric collection
- Implementing a tool to help collect and measure test metrics automatically thought test reports/code exploration

References

Clemson, T., 2014. *Testing Strategies in a Microservice Architecture.* [Online]
Available at: https://www.martinfowler.com/articles/microservice-testing/
[Accessed 10 January 2022].

Richardson, C., 2018. *Microservices patterns: with examples in Java.* s.l.:Simon and Schuster.

Dehghani, Z., 2018. *How to break a Monolith into Microservices.* [Online]
Available at: https://martinfowler.com/articles/break-monolith-into-microservices.html
[Accessed 10 01 2022].

Brandon, L., 2012. *An empirical evaluation of an agile modular software development approach: a case study with ericsson.* s.l.:s.n.

Dragoni, N. et al., 2017. Microservices: yesterday, today, and tomorrow. In: *Present and ulterior software engineering.* s.l.:s.n., pp. 195-216.

Vieira, A., 2020. *What Is a Service-Oriented Architecture?.* [Online]
Available at: https://www.outsystems.com/blog/posts/service-oriented-architecture/
[Accessed 12 01 2022].

IBM Cloud Education, 2021. *SOA (Service-Oriented Architecture).* [Online]
Available at: https://www.ibm.com/cloud/learn/soa
[Accessed 12 01 2022].

Lewis, J. & Fowler, M., 2014. *Microservices.* [Online]
Available at: https://martinfowler.com/articles/microservices.html
[Accessed 13 01 2022].

Vocke, H., 2018. *The Practical Test Pyramid.* [Online]
Available at: https://martinfowler.com/articles/practical-test-pyramid.html
[Accessed 23 01 2022].

Baškarada, S., Nguyen, V. & Koronios, A., 2018. Architecting Microservices: Practical Opportunities. *Journal of Computer Information Systems,* pp. 428-436.

Fowler, M., 2014. *Unit Test.* [Online]
Available at: https://martinfowler.com/bliki/UnitTest.html
[Accessed 09 02 2022].

Meszaros, G., 2007. *xUnit test patterns: Refactoring test code.* s.l.:Pearson Education.

Khan, M. E. & Khan, F., 2012. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications,* 3(6), pp. 12-15.

Micco, J., 2016. *Flaky Tests at Google and How We Mitigate Them.* [Online]
Available at: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
[Accessed 12 02 2022].

Fowler, M., 2018. *Integration Test.* [Online]
Available at: https://martinfowler.com/bliki/IntegrationTest.html
[Accessed 15 02 2022].

Daya, S. et al., 2016. *Microservices from theory to practice: creating applications in IBM Bluemix using the microservices approach.* s.l.:IBM Redbooks.

JUnit, 2021. *JUnit 5.* [Online]
Available at: https://junit.org/junit5/
[Accessed 19 02 2022].

Beck, K., 2004. *JUnit Pocket Guide.* 1ª ed. s.l.:O'Reilly Media.

Cheon, Y. & Leavens, G. T., 2001. *A Simple and Practical Approach to Unit Testing: The JML and JUnit Way,* s.l.: Department of Computer Science, Iowa State University.

Neves, J., 2019. *Technical Challenges of Microservices,* Porto: Instituto Politénico de Engenharia do Porto, Departamento de Engenharia Informática.

Mockito, 2019. *Mockito Wiki: Features And Motivations.* [Online]
Available at: https://github.com/mockito/mockito/wiki/Features-And-Motivations
[Accessed 20 02 2022].

Mockito, 2021. *Mockito Javadoc.* [Online]
Available at: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html
[Accessed 20 02 2022].

WireMock, 2022. *WireMock User Documentation.* [Online]
Available at: https://wiremock.org/docs/
[Accessed 20 02 2022].

H2, n.d. *H2 Database Engine.* [Online]
Available at: https://www.h2database.com/html/main.html
[Accessed 22 02 2022].

Pact, 2022. *Pact: Introduction.* [Online]
Available at: https://docs.pact.io/
[Accessed 22 02 2022].

Pactflow, 2022. *How Pact contract testing works.* [Online]
Available at: https://pactflow.io/how-pact-works
[Accessed 20 02 2022].

Bueno, A. S., Gumbrecht, A. & Porter, J., 2018. *Testing Java Microservices.* s.l.:Manning Publications Co..

Nicola, S., Ferreira, E. P. & Ferreira, J. P., 2012. A Novel Framework for modeling value for the customer, an essay on negotiation. *International Journal of Information Technology & Decision Making,* 11(03), pp. 661-703.

Walters, D. & Lancaster, G., 2000. Implementing value strategy through the value chain.. *Management Decision*, 1 Abril.

Zeithmal, C. & Bateman, T., 1990. *Management : function and strategy..* s.l.:s.n.

Rich, N. & Holweg, M., 2000. *Value analysis. Value engineering.,* s.l.: s.n.

Koen, P. et al., 2002. Fuzzy front end: effective methods, tools, and. *The PDMA toolbook,* Issue 1, pp. 5-35.

Ghani, I., Wan-Kadir, W. M., Mustafa, A. & Imran, M., 2019. Microservice Testing Approaches: A Systematic Literature. *The International Journal of Integrated Engineering,* 11(8), pp. 65-80.

O'Reilly, 2020. *Microservices Adoption in 2020.* [Online]
Available at: https://www.oreilly.com/radar/microservices-adoption-in-2020/
[Accessed 25 02 2022].

IBM Market Development & Insights, 2021. *Microservices in the enterprise, 2021: Real benefits, worth the challenges,* s.l.: s.n.

Osterwalder, A., 2004. *The business model ontology a proposition in a design science approach,* s.l.: (Doctoral dissertation, Université de Lausanne, Faculté des hautes études commerciales)..

Nicola, S., 2018. *Moodle ISEP: TMDEI 2021/2022.* [Online]
Available at:
https://moodle.isep.ipp.pt/pluginfile.php/187507/mod_resource/content/1/An%C3%A1lise_Valor_Aula_4_21NOV_2018_1hora_AHP.pdf
[Accessed 25 02 2022].

Fowler, M., 2005. *Inversion of Control.* [Online]
Available at: https://martinfowler.com/bliki/InversionOfControl.html
[Accessed 27 02 2022].

Palermo, J., 2008. *The Onion Architecture : part 1.* [Online]
Available at: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
[Accessed 27 02 2022].

Smith, S., 2022. *Overview of ASP.NET Core MVC.* [Online]
Available at: https://docs.microsoft.com/en-

us/aspnet/core/mvc/overview?WT.mc_id=dotnet-35129-website&view=aspnetcore-6.0
[Accessed 27 02 2022].

SOAPUI, n.d. *SOAP vs REST 101: Understand The Differences.* [Online]
Available at: https://www.soapui.org/learn/api/soap-vs-rest-api/
[Accessed 27 02 2022].

Coppen, R. J., 2021. *What is Messaging?.* [Online]
Available at: https://developer.ibm.com/articles/what-is-messaging/
[Accessed 27 02 2022].

gRPC Authors, 2022. *gRPC About.* [Online]
Available at: https://grpc.io/about/
[Accessed 26 06 2022].

gRPC Authors, 2022. *What is gRPC.* [Online]
Available at: https://grpc.io/docs/what-is-grpc/introduction/
[Accessed 26 06 2022].

Azure DevOps, 2022. *Design interservice communication for microservices.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication
[Accessed 10 July 2022].

Richardson, C., 2019. *Testing microservices.* [Online]
Available at: https://microservices.io/testing/index.html
[Accessed 30 07 2022].

Fowler, M., 2010. *Richardon Maturity Model.* [Online]
Available at: https://martinfowler.com/articles/richardsonMaturityModel.html
[Accessed 07 08 2022].

Richardson, C., 2015. *Building Microservices: Using an API Gateway.* [Online]
Available at: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
[Accessed 13 08 2022].

Megargel, A., Shankararaman, V. & Walker, D., 2020. Software Engineering in the Era of Cloud Computing. In: *Migrating from monoliths to cloud-based microservices: A banking industry example.* s.l.:Research Collection School of Information Systems, pp. 85-108.

RedHat, 2022. *What is CI/CD?.* [Online]
Available at: https://www.redhat.com/en/topics/devops/what-is-ci-cd
[Accessed 24 08 2022].

Richardson, C., 2021. *Pattern: Event Sourcing.* [Online]
Available at: https://microservices.io/patterns/data/event-sourcing.html
[Accessed 24 08 2022].

Richardson, C., 2021. *Pattern: SAGA.* [Online]
Available at: https://microservices.io/patterns/data/saga.html
[Accessed 24 08 2022].

Richardson, C., 2021. *Command Query Responsibility Segregation (CQRS).* [Online]
Available at: https://microservices.io/patterns/data/cqrs.html
[Accessed 24 08 2022].

EasyMock contributors, n.d. *EasyMock - User Guide.* [Online]
Available at: https://easymock.org/user-guide.html
[Accessed 26 08 2022].

Clarke, D., 2020. *Comparing .NET Mocking Libraries.* [Online]
Available at: https://www.danclarke.com/comparing-dotnet-mocking-libraries
[Accessed 27 08 2022].

Poole, C. & Prouse, R., 2022. *nunit.* [Online]
Available at: https://nunit.org/
[Accessed 27 08 2022].

NUnit, n.d. *NUnit Documentation.* [Online]
Available at: https://docs.nunit.org/articles/nunit/intro.html
[Accessed 27 08 2022].

Postman, 2022. *Setting up mock servers.* [Online]
Available at: https://learning.postman.com/docs/designing-and-developing-your-api/mocking-data/setting-up-mock/
[Accessed 27 08 2022].

Moq, 2022. *moq.* [Online]
Available at: https://github.com/moq/moq4
[Accessed 28 08 2022].

MockServer, 2022. *Getting Started Mocking.* [Online]
Available at: https://www.mock-server.com/mock_server/getting_started.html#request_openapi_matchers
[Accessed 28 08 2022].

MongoDB, 2022. *What is a Document Database?.* [Online]
Available at: https://www.mongodb.com/document-databases
[Accessed 28 08 2022].

Schaefer, L., 2022. *How to Write Integration Tests for MongoDB Atlas Functions.* [Online]
Available at: https://www.mongodb.com/developer/products/realm/integration-test-atlas-serverless-apps/
[Accessed 28 08 2022].

EventStoreDB, n.d. *Introduction.* [Online]
Available at: https://developers.eventstore.com/server/v21.10/
[Accessed 31 08 2022].

Spring, 2022. *Spring for Apache Kafka.* [Online]
Available at: https://docs.spring.io/spring-kafka/docs/current/reference/html/#ktu
[Accessed 18 09 2022].

Eventuate, 2021. *Eventuate.* [Online]
Available at: https://eventuate.io/
[Accessed 20 09 2022].

Dudczak, A. et al., 2022. *Spring Cloud Contract Reference Documentation.* [Online]
Available at: https://docs.spring.io/spring-cloud-contract/docs/current/reference/html/index.html
[Accessed 20 09 2022].

Spotlight, 2022. *Prism.* [Online]
Available at: https://meta.stoplight.io/docs/prism/674b27b261c3c-overview
[Accessed 23 09 2022].

Software Freedom, 2022. *Selenium Documentation.* [Online]
Available at: https://www.selenium.dev/documentation/
[Accessed 23 09 2022].

North, R., 2021. *Testcontainers.* [Online]
Available at: https://www.testcontainers.org/
[Accessed 15 10 2022].