



# Detection of microservice smells through static analysis

**JOÃO AFONSO ALMEIDA SAMÕES**

Outubro de 2023

# **Detection of microservice smells through static analysis**

**João Afonso Almeida Samões**

**Dissertation to obtain a master's degree in informatics engineering, with  
a Specialization in Software Engineering**

**Supervisor: Isabel Azevedo**

Porto, 2023



# Declaration of Integrity

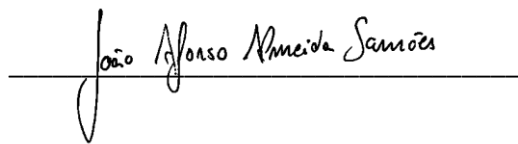
I declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of misuse of information or falsification of results throughout the process that led to its preparation.

Therefore, the work presented in this document is original and of my own authorship and has not been used previously for any other purpose.

I also declare that I am fully aware of P. PORTO's Code of Ethical Conduct.

ISEP, Porto, 13 October 2023

A handwritten signature in black ink, reading "João Afonso Almeida Samões", is written over a horizontal line.





Dedicated to my unwaveringly supportive parents and brother, who have consistently provided me with the strength and resources that have shaped the person I am today.



# Resumo

A arquitetura de microsserviços é um modelo arquitetural promissor na área de software, atraindo desenvolvedores e empresas para os seus princípios convincentes. As suas vantagens residem no potencial para melhorar a escalabilidade, a flexibilidade e a agilidade, alinhando-se com as exigências em constante evolução da era digital. No entanto, navegar entre as complexidades dos microsserviços pode ser uma tarefa desafiante, especialmente à medida que este campo continua a evoluir.

Um dos principais desafios advém da complexidade inerente aos microsserviços, em que o seu grande número e interdependências podem introduzir novas camadas de complexidade. Além disso, a rápida expansão dos microsserviços, juntamente com a necessidade de aproveitar as suas vantagens de forma eficaz, exige uma compreensão mais profunda das potenciais ameaças e problemas que podem surgir. Para tirar verdadeiramente partido das vantagens dos microsserviços, é essencial enfrentar estes desafios e garantir que o desenvolvimento e a adoção de microsserviços sejam bem-sucedidos.

O presente documento pretende explorar a área dos *smells* da arquitetura de microsserviços que desempenham um papel tão importante na dívida técnica dirigida à área dos microsserviços.

Embarca numa exploração de investigação abrangente, explorando o domínio dos *smells* de microsserviços. Esta investigação serve como base para melhorar um catálogo de *smells* de microsserviços. Esta investigação abrangente obtém dados de duas fontes primárias: *systematic mapping study* e um questionário a profissionais da área. Este último envolveu 31 profissionais experientes com uma experiência substancial no domínio dos microsserviços.

Além disso, são descritos o desenvolvimento e o aperfeiçoamento de uma ferramenta especificamente concebida para identificar e resolver problemas relacionados com os microsserviços. Esta ferramenta destina-se a melhorar o desempenho dos programadores durante o desenvolvimento e a implementação da arquitetura de microsserviços.

Por último, o documento inclui uma avaliação do desempenho da ferramenta. Trata-se de uma análise comparativa efetuada antes e depois das melhorias introduzidas na ferramenta. A eficácia da ferramenta será avaliada utilizando o mesmo benchmarking de microsserviços utilizado anteriormente, para além de outro benchmarking para garantir uma avaliação abrangente.

**Palavras-chave:** Architecture smells, microservices, detection tools, technical debt



# Abstract

The microservices architecture stands as a beacon of promise in the software landscape, drawing developers and companies towards its compelling principles. Its appeal lies in the potential for improved scalability, flexibility, and agility, aligning with the ever-evolving demands of the digital age. However, navigating the intricacies of microservices can be a challenging task, especially as this field continues to evolve.

A key challenge arises from the inherent complexity of microservices, where their sheer number and interdependencies can introduce new layers of intricacy. Furthermore, the rapid expansion of microservices, coupled with the need to harness their advantages effectively, demands a deeper understanding of the potential pitfalls and issues that may emerge. To truly unlock the benefits of microservices, it is essential to address these challenges head-on and ensure a successful journey in the world of microservices development and adoption.

The present document intends to explore the area of microservice architecture smells that play such an important role in the technical debt directed to the area of microservices.

It embarks on a comprehensive research exploration, delving into the realm of microservice smells. This research serves as the cornerstone for enhancing a microservice smell catalogue. This comprehensive research draws data from two primary sources: a systematic mapping research and an industry survey. The latter involves 31 seasoned professionals with substantial experience in the field of microservices.

Moreover, the development and enhancement of a tool specifically designed to identify and address issues related to microservices is described. This tool is aimed at improving developers' performance throughout the development and implementation of microservices architecture.

Finally, the document includes an evaluation of the tool's performance. This involves a comparative analysis conducted before and after the tool's enhancements. The tool's effectiveness will be assessed using the same microservice benchmarking as previously employed, in addition to another benchmark to ensure a comprehensive evaluation.

**Keywords:** Architecture smells, microservices, detection tools, technical debt



# Acknowledgements

First and foremost, I want to express my gratitude to my parents, who have been unwavering pillars of support throughout this academic journey, providing me with the best opportunities since I was a young boy, and instilling the invaluable lesson that hard work is the key to success in life.

I'd like to extend my heartfelt thanks to my brother, who has consistently believed in me, and provided invaluable guidance with the best advice, as any good older sibling would. He remains one of the central influences in my journey and he always be my role-model.

A special and heartfelt thank you is reserved for my love, my partner, who demonstrated patience in listening to everything I needed to share and for supporting me all these years. Perhaps without her, none of this would have been possible, and for that, I owe her my deepest gratitude. All the time I dedicated to this endeavour will soon be rewarded with countless beach strolls, catching sunsets and delicious sushi!

I want to express my immense gratitude to my friends, particularly to those who have been with me every step of the way in my academic journey, my friends João Dias and Diogo Barbosa, who were the best. Without them, this wouldn't be possible either. Also, thanks to those who shared the ups and downs of the bachelor's degree and kept sharing stories after it finished. Their support has played a crucial role in my growth, as they have consistently encouraged me to strive for the best and I wish them all the greatest success.

I owe a great debt of gratitude to my supervisor, Isabel Azevedo, whose tireless dedication, and unfailing support have been invaluable. She has consistently been available to address my questions and engage in discussions on all matters pertaining to this project. Without her guidance from the very outset, this achievement would undoubtedly not have been attainable.

Finally, I want to express my appreciation to all the survey respondents who generously devoted their time and patience to complete the lengthy survey and offer valuable insights.





# Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Context .....	1
1.2	Problem.....	2
1.3	Goals .....	2
1.4	Document Structure .....	3
<b>2</b>	<b>Background .....</b>	<b>5</b>
2.1	Microservice Architecture.....	5
2.1.1	Advantages .....	6
2.1.2	Challenges.....	8
2.2	Software Smells .....	10
<b>3</b>	<b>Microservice smells research .....</b>	<b>13</b>
3.1	Design.....	13
3.1.1	Requirements .....	13
3.1.2	Final Design .....	14
3.2	Data from the research literature .....	14
3.2.1	Protocol and definition of research questions .....	15
3.2.2	Conducting the search for primary studies .....	16
3.2.3	Screening .....	17
3.2.4	Classification System .....	18
3.2.5	Coding: data extraction and aggregation .....	18
3.2.6	Analysis and Report .....	20
3.3	Data from Industry .....	27
3.3.1	Introduction - Demographic Questions .....	27
3.3.2	Microservice Smells Catalogue Analysis .....	30
3.3.3	Open-answer questions.....	33
3.4	Threats to validity .....	34
<b>4</b>	<b>State of the Art .....</b>	<b>35</b>
4.1	Microservice architectural smells .....	35
4.1.1	Catalogues.....	36
4.2	Detection Tools for Microservice Architecture Smells .....	49
4.2.1	MSA-Nose .....	49
4.2.2	µTOSCA toolchain .....	53
4.2.3	MARS .....	58
<b>5</b>	<b>Solution .....</b>	<b>69</b>
5.1	Proposed Catalogue .....	69
5.1.1	Analysis .....	69

5.1.2	Catalogue Improvements.....	70
5.2	MSA Nose Improvement .....	73
5.2.1	Analysis .....	73
5.2.2	Requirements .....	74
5.2.3	Design and Implementation.....	75
<b>6</b>	<b>Evaluation.....</b>	<b>81</b>
6.1	Methodology .....	81
6.2	Case Studies .....	82
6.2.1	Train Ticket .....	82
6.2.2	Teacher Management System .....	84
6.2.3	Piggy Metrics.....	85
6.3	Threats to validity .....	87
<b>7</b>	<b>Conclusions .....</b>	<b>89</b>
7.1	Contributions .....	89
7.2	Difficulties along the way.....	90
7.3	Future Work.....	91
	<b>References .....</b>	<b>93</b>
	<b>Appendix A (Value Analysis) .....</b>	<b>97</b>
	<b>Appendix B (Additional Pitfalls) .....</b>	<b>103</b>
	<b>Appendix C (Survey).....</b>	<b>105</b>

# Table of Figures

Figure 1 - The scale cube defines three separate ways to scale an application. (Richardson, 2018). .....	6
Figure 2 - Z-axis scaling runs multiple identical instances of the monolithic application behind 7	
Figure 3 – Taxonomy of microservices security (a) properties, (b) smells, and (c) refactorings. For the sake of readability, the association between security properties and smells is represented by aligning the corresponding boxes, whilst that between smells and refactorings is represented with arrows (Ponce et al., 2022). .....	22
Figure 4 – Questionnaire: participant’s area of software engineering. ....	27
Figure 5 – Questionnaire: participant’s companies. ....	28
Figure 6 – Questionnaire: participant’s years of experience. ....	29
Figure 7 – Questionnaire: participant’s experience with microservices. ....	29
Figure 8 – Results of the participant’s answers depending on their area of software. ....	30
Figure 9 – Results of the participant’s answers depending on their experience in any software engineering area. ....	31
Figure 10 – Results of the participant’s answers depending on their experience with microservice architecture specifically. ....	32
Figure 11 – Results of the participant’s answers independently from any variable shown before. ....	32
Figure 12 - MSANose architecture diagram (Walker et al., 2020). ....	50
Figure 13 – Shared Persistency (Walker et al., 2020) .....	51
Figure 14 - The $\mu$ TOSCA toolchain (Soldani et al., 2021). ....	54
Figure 15 – TOSCA simple “Hello World” (OASIS, n.d.). ....	55
Figure 16 – Visual representation of the architectural smells and refactorings (Soldani et al., 2021). ....	56
Figure 17 - Microservice Antipatterns Research Software (Tighilt et al., 2023). ....	58
Figure 18 – Wrong Cut pseudo-code description (Tighilt et al., 2023). ....	59
Figure 19 – Cyclic Dependencies pseudo-code description (Tighilt et al., 2023). ....	60
Figure 20 – Mega Service pseudo-code description (Tighilt et al., 2023). ....	60
Figure 21 – Nano Service pseudo-code description (Tighilt et al., 2023). ....	60
Figure 22 – Shared Libraries pseudo-code description (Tighilt et al., 2023). ....	61
Figure 23 – Hardcoded Endpoints pseudo-code description (Tighilt et al., 2023). ....	61
Figure 24 – Manual Configuration pseudo-code description (Tighilt et al., 2023). ....	62
Figure 25 - No Continuous Integration/Continuous Delivery pseudo-code description (Tighilt et al., 2023). ....	63
Figure 26 – No API Gateway pseudo-code description (Tighilt et al., 2023). ....	63
Figure 27 – Timeouts pseudo-code description (Tighilt et al., 2023). ....	64
Figure 28 – Multiple Service Instances Per Host pseudo-code description (Tighilt et al., 2023). ....	64
Figure 29 – Shared Persistence pseudo-code description (Tighilt et al., 2023). ....	65
Figure 30 – No API Versioning pseudo-code description (Tighilt et al., 2023). ....	65

Figure 31 – No Health Check pseudo-code description (Tighilt et al., 2023).	66
Figure 32 – Local Logging pseudo-code description (Tighilt et al., 2023).	66
Figure 33 – Insufficient Monitoring pseudo-code description (Tighilt et al., 2023).	67
Figure 34 – Components Diagram.	73
Figure 35 – Functional requirements.	75
Figure 36 – Logical view of a components' diagram with a level 2 abstraction.	76
Figure 37 – Process view (UC1).	77
Figure 38 – Process view (UC2).	77
Figure 39 – Process View (UC3).	78
Figure 40 – Process View (UC5).	78
Figure 41 – Process View (UC6).	79
Figure 42 – The innovation process (Koen et al., 2002)	97
Figure 43 – Relationship diagram representing the NCD model (Koen et al., 2002).	98
Figure 44 - Duration of use of microservice from the survey respondents (Loukides & Swoyer, 2020).	100
Figure 45 - Ranking sources of technical debt. Choice 1 is represented by hatches; Choice 2, dashes; and Choice 3, dots (Ernst et al., 2015).	101

# Table of Tables

Table 1 – Microservice smells research requirements. ....	14
Table 2 – Research question 1 using the PICOC model (RQ <sub>1</sub> ). ....	15
Table 3 – Research question 2 using the PICOC model (RQ <sub>2</sub> ). ....	15
Table 4 – Research question 3 using the PICOC model (RQ <sub>3</sub> ). ....	16
Table 5 – Microservice smells systematic mapping study I/E criteria. ....	16
Table 6 – Papers collected after the screening phase. ....	18
Table 7 – Classification system.....	18
Table 8 – Microservice smells catalogue proposed by (Taibi & Lenarduzzi, 2018). ....	37
Table 9 – Internal microservice smells (Taibi et al., 2019). ....	38
Table 10 – Communication microservice smells (Taibi et al., 2019). ....	38
Table 11 – Other technical microservice smells (Taibi et al., 2019). ....	38
Table 12 – Team Oriented Microservice smells (Taibi et al., 2019). ....	39
Table 13 – Technology and Tool Oriented microservice smells (Taibi et al., 2019). ....	39
Table 14 – Missing microservice from other sources of research methods found by (Taibi et al., 2019) .....	40
Table 15 – Design Antipatterns (Tighilt et al., 2020). ....	42
Table 16 – Implementation Antipatterns (Tighilt et al., 2020). ....	43
Table 17 – Deployment Antipatterns (Tighilt et al., 2020). ....	44
Table 18 – Monitoring Antipatterns (Tighilt et al., 2020). ....	45
Table 19 – Design microservice smells (Ding & Zhang, 2022). ....	46
Table 20 – Deployment microservice smells (Ding & Zhang, 2022). ....	47
Table 21 – Monitor & Log microservice smells (Ding & Zhang, 2022). ....	47
Table 22 - Team & Tool microservice smells (Ding & Zhang, 2022). ....	48
Table 23 – Communication microservice smells (Ding & Zhang, 2022). ....	48
Table 24 – MSA-Nose improvement non-functional requirements. ....	74
Table 25 – Microservices information. ....	82
Table 26 – Train Ticket case study results (“B-Imp” means Before Improvement and “A-Imp” means After Improvement). ....	83
Table 27 – Teacher Management System case study results (“B-Imp” means Before Improvement and “A-Imp” means After Improvement). ....	85
Table 28 - Piggy Metrics case study results (“B-Imp” means Before Improvement and “A-Imp” means After Improvement). ....	86
Table 29 – Goals achievement. ....	89
Table 30 – Architecture Smells versus maintainability measures (Zhong et al., 2022). ....	102
Table 31 – The main pitfalls proposed in non-peer-reviewed literature and practitioner.....	103



# Abbreviations list

<b>API</b>	Application Programming Interface
<b>IDE</b>	Integrated Development Environment
<b>MSA</b>	Microservice Architecture
<b>RAM</b>	Random Access Memory
<b>SOA</b>	Service Oriented Architecture
<b>TOSCA</b>	Topology and Orchestration Specification for Cloud Applications
<b>URL</b>	Uniform Resource Locator





# 1 Introduction

This chapter has the objective of introducing the work described in this document. It contains the motivation context and the structure of the present document.

## 1.1 Context

The concept of code smells was first introduced in the late 90s by Kent Beck and Martin Fowler, who came up with the idea of using the phrase to describe patterns or practices in software development that negatively affect the quality of the code. (Fowler, 1999) helped to popularize the idea.

A software smell is an indicator of situations - such as undesired patterns, antipatterns, or bad practices - that negatively affect software quality attributes such as understandability, testability, extensibility, reusability, and maintainability of the system under development. There are many varieties of software smells like architecture smells (also known as bad smells), code smells, security smells, and others, which are related to different areas of software: bad smells to architecture (Azadi et al., 2019), code smells to code issues in general and security smells to coding patterns that are indicative of security weakness and can potentially lead to security breaches (A. Rahman et al., 2019). There are many tools to detect architecture and code smells in applications (Azadi et al., 2019). However, they are not adequate for detecting architecture smells oriented to microservices (Taibi & Lenarduzzi, 2018). Only recently the subject has been studied and some tools have been proposed (Pigazzini et al., 2020; Walker et al., 2020), but the topic remains underexplored.

A new architectural style based on a collection of tiny services, each with its process and interacting through lightweight mechanisms called microservices. It was given this name in May 2012 by a group of software architects who, a year earlier, had explored an architectural style at a workshop in Venice in May 2011 (Fowler & Lewis, 2014). For some practitioners seeking an architectural pattern even more "democratic" than what classic SOA could offer, microservices arose as a novel strategy (STRIMBEL et al., 2015).

Microservices are autonomous, typically automated procedures that are developed around business capabilities. In recent times, a significant number of systems have undergone or are contemplating a transition from monolithic applications to a distributed architecture composed of small, independent services. Due to the independence of these services, they may be created at various rates and with various technologies, which enables the implementation of some business capabilities in more suited ones. This aids in adapting to the market's current rapid speed (Ponce et al., 2019).

## **1.2 Problem**

Microservice architecture has become a popular topic of discussion in recent years as many organizations look to adopt or migrate away from traditional monolithic architecture in favour of a more modular approach. To ensure that the microservices architecture delivers the desired results, it is important to proactively identify and address any potential challenges and pitfalls, which are very likely to arise as some works were able to conclude (Lu et al., 2019; Taibi et al., 2017). Problems like these can be easily detected in monolithic applications with the usage of a set of tools and techniques, while it gets tougher to detect these in microservice applications due to a lack of exploration.

These challenges can have a significant impact on the overall performance and stability of the system, leading to increased downtime and reduced efficiency. Furthermore, the design and implementation of a microservices architecture can be complex and time-consuming, requiring specialized skills and expertise that may not be readily available within the organization. This can result in longer development cycles and increased costs, which can negatively impact the organization's ability to compete in the market.

## **1.3 Goals**

A tool enabling developers to identify issues and architectural smells alongside as they construct the microservices will be useful. A tool like this would be made expressly for the microservices architecture, comparable to other current tools that identify code-level or design-level smells in software development. Developers may enhance their microservices' architecture, performance, and maintainability by using this tool, which can offer insightful comments and recommendations. By having a tool that helps identify microservice smells, organizations can adopt a more proactive approach to software development, reducing the risk of technical debt and ensuring the successful implementation of their microservices architecture.

The work aims to thoroughly explore microservice smells, guidelines, and tools for their detection, thus contributing to this field of study. The objectives of this work are:

- explore and evaluate the comprehensiveness of existing catalogues of microservice smells, also gauging the acceptance of the included smells.
- improve a microservice smell detector that incorporates and improves upon existing applications. This detector aids in identifying smells in a microservice architecture, thus helping to improve the design and implementation of microservices.

## **1.4 Document Structure**

This document is divided into the following chapters:

- The first chapter introduces the theme to the reader, composed of the document's context, problem, goals, and structure.
- The second chapter is the background which describes the concepts that are key for understanding the remaining chapters.
- The third chapter comprises the microservice smell research, where it describes the systematic mapping study and the questionnaire done to the industry, as well as its results summary.
- The fourth chapter will show what is the State of the Art of the proposed theme – the existing catalogues for the microservice smells and the tools (and the way) they can detect these smells.
- The fifth chapter describes the solution, where the developments done to improve one of the detection tools found during the research and the update to a microservice smell catalogue is done, describing all the steps and analysis to its goal.
- The sixth chapter contains the evaluation, which outlines the criteria for assessing the quality of this work.
- The seventh chapter are the Conclusions, where all the conclusions are summarized, as well as the future works on this trend.
- The appendix contains one chapter that was created to accomplish one of the requirements of this document which is the Value Analysis where the reader will be able to check the value proposed by this document, containing the steps of the new concept development model. It also contains a table created by (Taibi & Lenarduzzi, 2018) and the survey used in the third chapter.



## 2 Background

This chapter presents to the reader key concepts related to microservice architecture (the advantages, challenges/pitfalls and antipatterns of this architecture) and software smells, such as what are they and which types exist.

### 2.1 Microservice Architecture

Microservice Architecture (MSA) has been defined by many different authors. The authors (Fowler & Lewis, 2014) started by defining it as “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API” (Fowler & Lewis, 2014). This is the most consensual definition as it is cited by many authors.

Another author (Dragoni et al., 2017) splits the definition between microservices and microservice architecture, where the second is a distributed application where all its modules are composed of the first. To this author “microservices (which are a cohesive, independent process interacting via messages) should be independent components conceptually deployed in isolation and equipped with dedicated memory persistence tools (e.g., databases) since all the components of a microservice architecture are microservices, its distinguishing behaviour derives from the composition and coordination of its components via messages” (Dragoni et al., 2017).

Both authors agree on the main characteristic of microservices, which is the ability to develop and deploy independently, and the communication between them via messages, which allows for improved scalability, flexibility, and maintainability of the software system.

### 2.1.1 Advantages

The microservices architecture presents many characteristics and benefits that have made it a trend in software development (Fowler, 2014; GitLab, n.d., 2022).

#### Scalability

To explain the scalability of the microservices, a scale model called the *scale cube* (as shown in Figure 1) can be used (Richardson, 2018). It was inspired by other authors Martin Abbott and Michael Fisher's book *The Art of Scalability*.

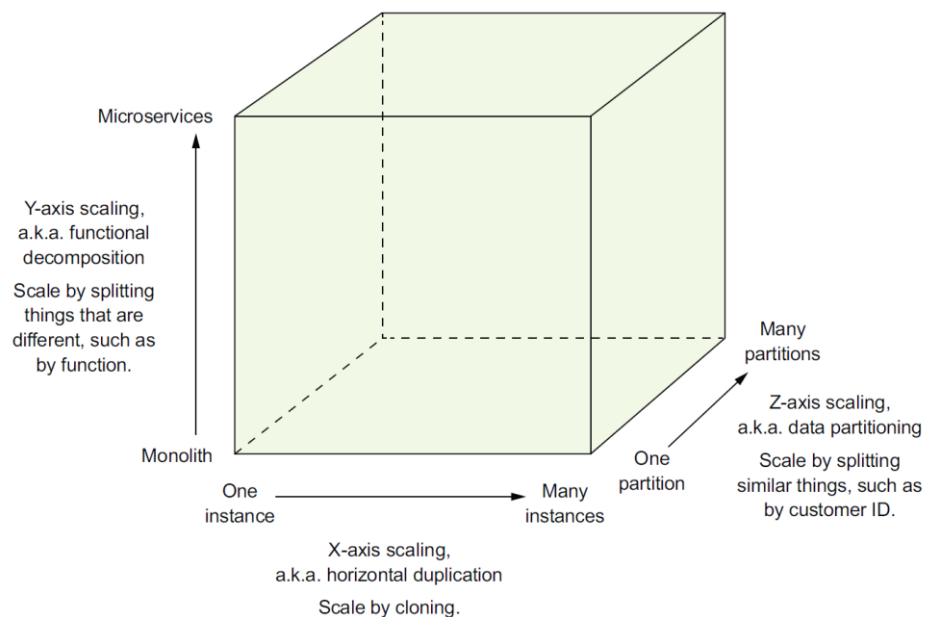


Figure 1 - The scale cube defines three separate ways to scale an application. (Richardson, 2018).

The X-axis scaling (or horizontal duplication) is the common way to scale a monolithic application which is by running many instances of one application behind a load balancer, which distributes the requests among the many identical instances of the application (Richardson, 2018).

The Z-axis scaling is where each instance is responsible for only a subset of data. which consists of having a router that forwards requests to instances that match its responsibility. As shown in Figure 2, there are  $N$  application instances that are identical, however, they are responsible for a subset of users. As soon as the request arrives at the router, it decides where it has to go, enabling the application to handle the increasing transaction and data volumes (Richardson, 2018).

Finally, the Y-axis is to scale by functional decomposition, or, by microservices. The application's capacity and availability are increased through X- and Z-axis scaling. However, neither strategy addresses the issue of growing application and development complexity.

By decomposing a monolithic application into a set of services, the application makes itself more controllable, because its services are independently scaled making it easier to handle increased traffic or demand, as they will have an X-axis and Z-axis scaling by each service instead of one for all the application.

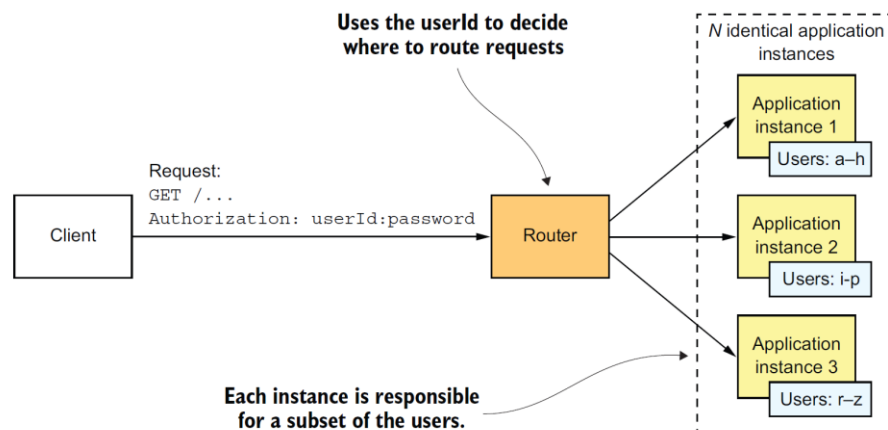


Figure 2 - Z-axis scaling runs multiple identical instances of the monolithic application behind a router, which routes based on a request attribute (Richardson, 2018)

### Flexibility and Technology Diversity

The microservices architecture allows for the development and deployment of each service independently, using a variety of languages, frameworks, and technologies. This flexibility brings many benefits to the development process, such as language diversity, framework diversity, technology diversity, team autonomy, best fit, innovation, and cost-effectiveness. Teams can choose the best tools for their specific needs, leading to a more efficient and effective development process, and ultimately, improving the overall quality of the software (Viggiato et al., 2018).

### Resilience and better fault isolation

The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks (CCSU, n.d.).

Microservices' capacity to continue operating even if one of them fails lessens the effect of outages on the entire system. This is accomplished by creating services that are highly coherent and loosely connected, allowing them to operate apart from one another.

### Improved Modularity

Improved modularity is one of the key benefits of microservices architecture. The isolation of microservices allows for a high degree of modularity, which makes them improve quality attributes of the application like understandability (each microservice is a small and self-



contained unit of functionality, which makes it easy to understand the purpose and functionality of each service), testability (each microservice can be tested independently, which makes it easy to detect and fix bugs), maintainability, and reusability.

### **Easy To Deploy and faster time to market**

The independence of microservices allows for a high degree of flexibility in the deployment process, which makes it easy to deploy new features. By isolating each service, teams can deploy them independently, without affecting the entire system. This allows teams to test new features and make changes to one service without affecting the rest of the system. It also allows teams to deploy new features and updates in a phased manner, which can reduce the risk of introducing new bugs or breaking existing functionality. Additionally, this independence allows teams to deploy and scale each service according to its specific needs, which can improve the overall performance and availability of the system.

### **2.1.2 Challenges**

The development team, to have all these benefits from the MSA, will face some challenges. If not overcome, then the advantages will not be achieved. Due to this, a study realized by (Söylemez et al., 2022) described what are the most described challenges of the adoption of the microservice architecture in the literature.

#### **Service Discovery**

The difficulties with service discovery are related to issues with design, implementation, and quality. Due to the numerous different service discovery techniques, including client-side, server-side, and hybrid service discovery, creating the service discovery is regarded as challenging at the design level. Based on the numerous demands and concerns for quality, the right choice must be selected. It is difficult to come up with a workable design option when several design alternatives might be found. The scale of the system and the architecture chosen have a direct impact on how service discovery is implemented; thus, high availability and scalability are the most crucial implementation criteria (Söylemez et al., 2022).

#### **Data Management and Consistency**

The distributed nature of MSA causes a challenge in terms of data management and consistency. These challenges include issues with distributed transaction management, backing up the system, and data integration. Architects and developers often use a database per service pattern for distributed transaction management, which MSA also favours for decentralized data management. This pattern has many benefits but also comes with difficulties in managing distributed transactions. Backing up the entire system in an MSA can be challenging and requires trade-offs, making it difficult for practitioners to decide on the best approach. There may not be a mature mechanism for data sharing and synchronization in some MSAs, which can lead to a more complex system. Sharing and synchronization

operations need to be handled in a way that does not impact other data (Söylemez et al., 2022).

### **Testing**

Testing is essential for making sure a system is prepared for deployment and enabling developers to proceed with confidence, but due to the dispersed nature of MSA, it may be difficult. The testing procedure is made more difficult by the fact that each microservice in MSA can be created using various technologies, languages, and infrastructures. Regression and acceptability testing, as well as testing for performance and robustness, provide significant issues. Also, MSA might make it challenging to build a thorough testing framework and automate tests. These issues need to be dealt with in an agile manner, automated, and as part of the continuous delivery process, while simultaneously making sure they don't compromise the dependability of the system (Söylemez et al., 2022).

### **Performance Prediction, Measurement and Optimization**

Performance is a crucial quality aspect that must be considered at various phases of the system's design, implementation, and operation. It is typically advantageous to predict how well a software system will function before it is put into use since changing the system after it has been put into use can be challenging or expensive. To meet the quality standards of MSA-based systems, performance assessment and optimization become crucial after deployment (Söylemez et al., 2022).

### **Communication and Integration**

It is challenging to guarantee that the communication infrastructure is trustworthy and that the protocol to be used for communication and integration can manage complicated processes, even when microservices interact using a more lightweight protocol. Dependability and durability are the most crucial requirements for both problems; if these requirements are not fulfilled, the system's correct functioning and reliability will be compromised, which might lead to cascade failures (Söylemez et al., 2022).

### **Service Orchestration**

Microservice deployment, scalability, scheduling, management, and networking are all included in the idea of service orchestration. Although several of these topics have been addressed by container orchestration technologies, certain research has noted difficulties in each of these areas. Scalability, dynamic and automated orchestration, storage service orchestration, deployment, load balancing, and scheduling are some of these difficulties. Sub-issues connected to these issues include dynamic resource adaptation for containers, persistent storage across containers, and creating an all-encompassing solution to handle workloads and resource issues. Decentralized deployments, load balancing, auto-scalability, resource allocation and scheduling, as well as comprehending container failure-recovery behaviour, are other difficult issues for practitioners to solve (Söylemez et al., 2022).

## **Security**

Given that data is flowing among microservices, the need exists to secure such communication through encryption techniques, but authentication mechanisms also must be implemented (Larrucea et al., 2018). However, comparing the applicability of these mechanisms in a monolithic architecture versus a microservice architecture, the second will of course have more complexity because of its distributed nature. Furthermore, it is challenging to integrate and use complex, non-lightweight frameworks comfortably, making the creation of a complete framework to create security across microservices challenging. Monitoring network traffic and implementing security rules that have been set up in accordance with standards is another crucial concern (Söylemez et al., 2022).

## **Monitoring, Tracing and Logging (MTL)**

These important activities, which have a role in satisfying availability, performance, and reliability concerns, consist of several challenging points related to identifying strongly coupled services, the root cause of anomalies and performance problems, and the heterogeneity of logs. It is essential to recognise these issues and respond quickly as soon as you can. If not, the system's fault tolerance, availability, and dependability will all suffer. The MTL process is expected to behave in a way whereby trouble areas are identified, and the system is then made more available, scalable, dependable, and fault-tolerant by taking prompt action or changing the architecture as needed (Söylemez et al., 2022).

## **Decomposition**

After opting to adopt MSA, the first difficult decision is determining the suitable scale of the business capability. If this cannot be done successfully, MSA will not be beneficial and may result in several issues, most notably with scalability, performance, availability, and reliability (Söylemez et al., 2022).

## **2.2 Software Smells**

Software smells (or code smells (Fowler, 1999)) are anomalies within the codebases which do not necessarily impact the performance or correct functionality of an application, however, they affect a wide range of attributes including reusability, testability, and maintainability. If these go unchecked the benefits of the selected architecture can be mitigated (Walker et al., 2020).

Software smells can be categorized based on their granularity, scope and impact: architecture (Garcia et al., 2009), design (Suryanarayana et al., 2014) and implementation (Fowler, 1999). The level of granularity refers to the scope and impact of the smell, with architecture-level smells having the highest scope and impact, and implementation-level smells having the lowest scope and impact, which means that architecture smells affect a set of components and require considerable effort to refactor (Sharma et al., 2020).

## **Implementation Smells**

Any indication that might potentially have a deeper negative impact on the software process and the quality and maintainability of software is referred to as an "implementation smell." (Fowler, 1999). Implementation smells are warning signs that show that there might be deeper problems with the implementation of a software system. These symptoms might be caused by a variety of things, including bad design, a lack of testing, or poor resource and code management. Implementation flaws can hinder the software development process and lower the software system's quality and maintainability. They frequently appear when developers give short-term objectives and rapid fixes precedence over long-term stability and dependability.

It is crucial to follow recognised best practices and apply a methodical approach to software development to prevent implementation smells, assure the quality, and maintainability of a software system. This involves following coding standards and rules, doing regular code reviews, and putting good testing techniques into practice.

One example of an implementation smell is the "Long Method" smell. This occurs, as the name can tell, when a method is too long and contains too many statements or operations. A long method can be difficult to understand, maintain, and test, and can also be a sign that the method is doing too much and violating the Single Responsibility Principle.

To improve this implementation smell, the long method could be broken down into smaller, more focused methods that each perform a single task. This can make the code easier to read and understand, as well as make it more modular and easier to maintain.

## **Design Smells**

Design smells are certain structures in the design that indicate a violation of fundamental design principles and negatively impact design quality (Suryanarayana et al., 2014). These smells can be caused by:

- Violation of design principles – Design principles guide designers in creating high-quality software solutions; when these principles are violated in the design, they manifest as design smells.
- Inappropriate use of design patterns – Design patterns are well-known solutions to problems in software design but applying them without fully understanding their consequences can negatively impact design quality. Architects and designers should use design patterns thoughtfully and carefully consider the specific consequences of each variant. Design smells and design patterns have a close relationship, as addressing a design smell can often involve using a specific design pattern, but misapplying a design pattern can also lead to a design smell.
- Language limitations.
- Procedural thinking in OO – Programmers with a procedural programming background may misunderstand the object-oriented paradigm and view classes as "doing" things

instead of "being" things which can lead to design smells such as imperative class names, functional decomposition, missing polymorphism with explicit type checks, and others.

- Viscosity – Developers may resort to hacking and introduce design smells due to the concept of viscosity, which is the resistance encountered when applying the correct solution to a problem.

One example of a design smell is the "Shotgun Surgery" smell (Refactoring Guru, n.d.). This smell occurs when a change in a feature requires modifications to be made in multiple unrelated parts of the codebase, indicating poor design or code coupling. In other words, a single logical change requires changes to be made in multiple classes or modules, leading to code duplication and maintainability issues.

### **Architecture Smells**

An architecture smell is “a commonly (although not always intentionally) used architectural decision that negatively impacts system quality” which “may be caused by applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviours, or applying design abstractions at the wrong level of granularity” (Garcia et al., 2009). Architectural smells can have a significant impact on the overall quality of a software system. They can negatively affect key properties of the software lifecycle such as performance and reliability, among others. The resolution of architectural smells requires a trade-off between different quality properties. System architects must carefully evaluate the situation and determine whether the correction of a particular smell will result in an overall improvement in the system. The process of addressing architectural smells involves making changes to the internal structure and behaviours of the system components, while maintaining the external behaviour of the system unchanged (Garcia et al., 2009).

One example of architecture smells is the “Cyclic Dependency”, also known as “Strong Circular Dependencies Between Packages” or “Shape detection”. This smell arises when two or more architecture components depend on each other directly or indirectly, violating the principles of Health Dependency Structure (the presence of this AS implies that the participating classes and packages cannot be deployed and maintained separately) and the Modularity (the presence of this AS implies that there are two pieces of code, that are highly coupled to each other directly or indirectly) (Azadi et al., 2019).

## 3 Microservice smells research

This chapter outlines the design of the research plan, presents its results, and concludes with potential threats to its validity.

### 3.1 Design

#### 3.1.1 Requirements

As discussed in the Problem section (see Section 1.2), microservices have gained immense popularity as an architectural approach, but they often present challenges during implementation. These challenges can be categorized as "microservice smells." There is a pressing need for research that focuses on cataloguing these microservice smells and developing effective tools for their identification. While literature has addressed various challenges in the realm of microservice applications, there is a noticeable research gap when it comes to systematically classifying these challenges as microservice architecture smells. Nevertheless, there have been instances where catalogues have emerged to classify these microservice smells.

Within the realm of tools, a selection of detection tools and mechanisms is available. Still, the scope narrows when it comes to tools that employ static analysis for detection, a subset that has also been expounded.

The requirements for this research were defined and shown in Table 1.

Requirement number	Description
1	Expose the problems of microservices development
2	Recognise the most common smells
3	Compare data between literature and industry
4	Contribute to the most recent catalogue found
5	Contribute to MSA-Nose to detect all possible microservice smells
6	Analyse all the data related to microservice smell detection via static analysis
7	Analyse data published since 2020 for the microservice catalogue contribution

Table 1 – Microservice smells research requirements.

### 3.1.2 Final Design

The design of this study embraces the utilization of a systematic mapping study as the chosen approach for conducting the literature research. The primary aim of employing this method is to not only enhance the credibility and reliability of the study's outcomes but also to forward an environment conducive to replication by the reader. It is noteworthy that, particularly within the domain of software engineering, systematic mapping studies are held in high regard for their consistency and inherent value (Sampaio, 2015). This stems from their adeptness in extracting patterns, trends, and knowledge from a diverse array of sources, ultimately contributing to a comprehensive understanding of the research landscape.

Regarding the industry research to gather data, the chosen method is the utilization of questionnaires. This deliberate selection is underscored by a myriad of advantages that this method brings to the forefront. Questionnaires offer an efficient and structured means of gathering data from a diverse pool of respondents. The standardized format ensures that all participants are presented with the same set of questions, eliminating potential biases that could arise from variations in interview or conversation styles. The scalability of surveys is also important. With the ability to deliver surveys to many participants at the same time, this approach is especially beneficial for obtaining a diverse viewpoint in a manageable period.

## 3.2 Data from the research literature

As mentioned earlier in this chapter, the chosen methodology to conduct the literature research was the systematic mapping study. (Sampaio, 2015) developed a detailed process for mapping studies based on guidelines from (Kitchenham, Barbara Charters et al., 2007; Petersen et al., 2008)(Barbara Kitchenham, 2004; Petersen et al., 2008) and others from social sciences, from MS studies in software engineering and orientations for systematic literature research.

Based on all those, (Sampaio, 2015) created a process consisting of 6 stages that go as follows:

1. Protocol and definition of research questions.
2. Conducting the search for primary studies.
3. Screening.
4. Classification system.
5. Coding: data extraction and aggregation.
6. Analysis and report.

### 3.2.1 Protocol and definition of research questions

The main purpose of this phase is to develop the protocol that will rigorously guide the mapping study effort. This will provide as outcomes a protocol able to guide the review and cover all the stages. (Sampaio, 2015) suggests that these research questions can be framed according to the PICOC (Population, Intervention, Comparison, Outcomes, Context) model, as this framework represents the “anatomy” of a well-focused research question. Table 2, Table 3 and Table 4 are presented the research question of this study.

<b>Research Question</b>	What are the available microservice smell catalogues?
<b>Population</b>	Reports of documents that assemble microservice smells and antipatterns
<b>Intervention</b>	Identify all the catalogues available to understand what is done.
<b>Comparison</b>	The number of new smells reported as new catalogues are found.
<b>Outcomes</b>	Identification of existing microservice smell catalogues and their characteristics.
<b>Context</b>	Research on microservice smell catalogues.
<b>Study Design</b>	Using different sources of information, it is expected that this RQ identifies most of the existing microservice smell catalogues that were created.

Table 2 – Research question 1 using the PICOC model (RQ<sub>1</sub>).

<b>Research Question</b>	What problems when developing using the microservice architecture style can be found in recent studies?
<b>Population</b>	Surveys, books about microservices and reports about issues found while developing using this architectural style
<b>Intervention</b>	Identify problems reported.
<b>Comparison</b>	Check if the issues reported are part of the microservice smell catalogues from RQ1.
<b>Outcomes</b>	Identification of issues that can be recognised using microservices.
<b>Context</b>	Research on microservice development issues.
<b>Study Design</b>	Some patterns should be followed when implementing microservices. This RQ is intended to explore the effects of not following microservice patterns.

Table 3 – Research question 2 using the PICOC model (RQ<sub>2</sub>).



<b>Research Question</b>	What are the available microservice smell detection tools?
<b>Population</b>	Papers about tools developed to detect microservice smells and/or antipatterns
<b>Intervention</b>	Identify tools that detect smells statically.
<b>Comparison</b>	Smells detected per tool.
<b>Outcomes</b>	Identification of tools created to detect microservice smells.
<b>Context</b>	Research on microservice smell detection tools.
<b>Study Design</b>	In this RQ the goal is to find the available tools to explore them and check what is the coverage of this area to find where to continue the research.

Table 4 – Research question 3 using the PICOC model (RQ<sub>3</sub>).

It is also in this phase that the inclusion/exclusion (I/E) criteria are developed, simultaneously with the PICOCS model. The purpose of this criteria is to simplify the screening process. Table 5 shows the defined criteria.

<b>Criterion</b>	<b>Description</b>
I1	Papers describing a microservice smell catalogue, independently from the year of publication.
I2	Technical reports describing developments in microservice and what problems were found.
I3	Papers describing microservice smells detection tools that use static analysis.
E1	Studies published before 2021 (for RQ2).
E2	Studies not written in English.
E3	Studies not available as full text.
E4	Studies that use only dynamic analysis to detect smells (RQ <sub>3</sub> ).

Table 5 – Microservice smells systematic mapping study I/E criteria.

### 3.2.2 Conducting the search for primary studies

This phase of the Systematic Mapping Study (Sampaio, 2015) serves the purpose of guiding the process to find primary studies that hold potential relevance for the review. The outcomes of this endeavour encompass all the papers selected through searches, accompanied by comprehensive recorded information about each search. This recorded information includes details such as the library source, search date, search restrictions, search queries, records retrieved, the count of records, and any other relevant information necessary for the search. It is in this phase that search strings are refined and, if not defined, they must be (Sampaio, 2015). For that reason, the search string goes as follows:

(detect \* OR " ") AND microser \* AND (problems OR antipatterns OR smells)

After searching on the different digital libraries (e.g.: IEEE Xplore and ACM Digital Library), 95 documents were identified.

### 3.2.3 Screening

From the documents identified, the I/E criteria shown in Table 5 were applied. In Table 6 we can see the documents found 13 documents that were found to help answer the research questions.

Document Title	Authors	Publication Year
On the Definition of Microservice Bad Smells	Davide Taibi Valentina Lenarduzzi	2018
Microservices Anti-Patterns: A Taxonomy	Davide Taibi Valentina Lenarduzzi Claus Pahl	2019
On the Study of Microservices Antipatterns: a Catalog Proposal	Rafik Tighilt Manel Abdellatif Naouel Moha Hafedh Mili Ghizlane El Boussaidi Jean Privat Yann-Gaël Guéhéneuc	2020
How Can We Cope with the Impact of Microservice Architecture Smells?	Xiang Ding Cheng Zhang	2022
On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study	Muhammad Waseem Peng Liang Mojtaba Shahin Aakash Ahmad Ali Rezaei Nasab	2021
Smells and refactorings for microservices security: A multivocal literature review	Francisco Ponce Jacopo Soldani Hernán Astudillo Antonio Brogi	2022
Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry	Xin Zhou Shanshan Li Lingli Cao He Zhang Zijia Jia Chenxing Zhong Zhihao Shan Muhammad Ali Babar	2023
Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review	Mehmet Söylemez Bedir Tekinerdogan Ayça Kolukısa Tarhan	2022
Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation	Chenxing Zhong Huang Huang He Zhang Shanshan Li	2022

Document Title	Authors	Publication Year
On the Way to Microservices: Exploring Problems and Solutions from Online Q&A Community	Menghan Wu Yang Zhangy Jiakun Liu Shangwen Wangy Zhang Zhangy Xin Xiax Xinjun Mao	2022
Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study	Andrew Walker Dipta Das Tomas Cerny	2020
The $\mu$ TOSCA toolchain: Mining, analysing, and refactoring microservice-based architectures	Jacopo Soldani Giuseppe Muntoni Davide Neri Antonio Brogi	2021
On the maintenance support for microservice-based systems through the specification and the detection of microservice antipattern	Rafik Tighilt Manel Abdellatif Imen Trabelsi Loïc Madern Naouel Moha Yann-Gaël Guéhéneuc	2023

Table 6 – Papers collected after the screening phase.

### 3.2.4 Classification System

(Sampaio, 2015) shows that the purpose of this phase is to define the classification system to be used to classify papers, which will retrieve a way to organize the papers in order to answer the research questions. In Table 7 is shown how the documents are going to be classified.

Group Number	Description
1	Catalogues/Lists of microservice architecture smells
2	Reports of problems faced while developing, industry surveys, solutions and approaches and microservice patterns
3	Detection tools for microservice smells/antipatterns.

Table 7 – Classification system.

### 3.2.5 Coding: data extraction and aggregation

The goal of this stage is to extract and record data from the relevant primary studies, and map these studies to the categories (of the system) developed previously the outcome of this stage is the map, that is, the relevant papers organized (classified) according to the classification system (Sampaio, 2015).

From the analysis, the catalogues of microservice smells (group 1) are:

- “On the Definition of Microservice Bad Smells” (Taibi & Lenarduzzi, 2018)
- “Microservices Anti-Patterns: A Taxonomy” (Taibi et al., 2019)
- “On the Study of Microservices Antipatterns: a Catalog Proposal” (Tighilt et al., 2020)
- “How Can We Cope with the Impact of Microservice Architecture Smells?” (Ding & Zhang, 2022)

The papers that belong to group 2 of the classification system will be presented in a different, having a small description of what is the paper about.

Starting with (Waseem et al., 2021) the authors conducted an empirical study on 1,345 issue discussions extracted from five open source microservices systems hosted on GitHub. Their analysis led to the first-of-its-kind taxonomy of the types of issues in open-source microservices systems, revealing that problems originating from Technical debt (321, 23.86%), Build (145, 10.78%), Security (137, 10.18%), and Service execution and communication (119, 8.84%) are prominent (Waseem et al., 2021).

(Ponce et al., 2022) conducted a multivocal review of the existing white and grey literature on the state of the art and practice in securing microservices. They systematically analysed 58 primary studies, selected among those published from 2011 until the end of 2020. The authors identified ten bad smells for securing microservices, which they organized into a taxonomy, associating each smell with the security properties it may violate and the refactorings enabling it mitigate its effects (Ponce et al., 2022).

(Zhou et al., 2023) carried out a series of industrial interviews with practitioners from 20 software companies. The collected data was then codified using qualitative methods which resulted in eight pairs of common practices and pains of microservices in industry after synthesizing the rich and detailed data collected and five aspects that require careful decisions were extracted to help practitioners balance the possible benefits and pains of MSA. Furthermore, five research directions that need further exploration were identified based on the pains associated with MSA.

(Söylemez et al., 2022) had as its main goal identifying the state of the art of microservices and describing the challenges in applying it together with the identified solution directions. A systematic literature review was performed using the published literature since the introduction of microservices. 3842 papers were discovered using a well-planned review protocol, and 85 of them were selected as primary studies and analysed regarding research questions. Nine basic categories of challenges were identified and detailed into 40 sub-categories, for which potential solution directions were explored.

(Zhong et al., 2022) had as a goal to bridge the gap by investigating the possible impacts, causes, and solutions of architectural smells in microservices-based systems. An industrial case study was conducted to gather repository data and practitioners' insights regarding six

typical architectural smells in a real microservice-based telecommunication system. Quantitative data was analysed using statistical analysis, while qualitative data was analysed using coding techniques. The results revealed that architectural smells affect various aspects of the microservice architecture-based system, including modularity, modifiability, analysability, and testability, leading to increased cross-team communication and the presence of change- and fault-prone microservices. To explore the causes of AS, the authors proposed a five-aspect conceptual classification, including technology, project, organization, business, and professional aspects, with a particular emphasis on the business and organizational factors.

(Wu et al., 2022) analysed 17,522 Stack Overflow posts related to microservices in a comprehensive study, creating the first taxonomy of microservice-related topics within the software development process. Their analysis highlighted a shortage of experts in the microservice domain, particularly in microservice design. They manually reviewed 6,013 accepted answers, identifying 47 general solution strategies for microservice-related issues, including 22 novel approaches.

Regarding the detection tools for microservice smells (group 3) the papers that were gathered during the analysis are:

- Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study (Walker et al., 2020)
- The  $\mu$ TOSCA toolchain: Mining, analysing, and refactoring microservice-based architectures (Soldani et al., 2021)
- On the maintenance support for microservice-based systems through the specification and the detection of microservice antipattern (Tighilt et al., 2023)

### **3.2.6 Analysis and Report**

In this section, the focus shifts to analysing the constructed map and generating a comprehensive report covering all study phases. The report aims to deliver informative insights, often using statistics presented through tables and charts to illustrate key patterns and correlations. The section also ensures that all research questions are thoroughly answered, contributing to the study's overall depth and quality (Sampaio, 2015).

#### **3.2.6.1 RQ1 - What are the available microservice smell catalogues?**

As indicated in Table 2, the objective of this research question is to discern the currently available microservice smell catalogues and their distinctive attributes. This information served as the foundation for conducting a comprehensive state-of-the-art analysis, which is elaborated upon in Section 4.1 of this document.

### 3.2.6.2 RQ2 - What problems when developing using the microservice architecture style can be found in recent studies?

This research question has as its objective the identification of issues that can be recognised using microservices, as can be seen in Table 3. To fulfil the aim of this research the selected documents from the screening phase will be analysed and it is intended to show what are the problems described in each.

#### **On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study (Waseem et al., 2021)**

In this study, (Waseem et al., 2021) used five open-source microservices systems hosted on GitHub to gather a total of 1345 issues. This resulted in a taxonomy of 17 categories, 46 subcategories and 138 types, which means that there are different issues on microservice systems, being the most discussed ones the ones related to Technical Debt (321 out of 1345, around 23.86%), Build (145 out of 1345, around 10.78%), Security (132 out of 1345, around 10.18%) and Service execution and communication (119 out of 1345, 8.84%) (Waseem et al., 2021). There are categories such as Exception Handling (8.77%), Compilation (6.91%), Documentation (4.53%), Testing (4.23%), Typecasting (3.71%), Configuration (2.75%), Updates and Installation (2.75%), Storage (2.6%), Performance (0.65%) and Networking (0.65%). However, the focus will be only on the top four, like what the authors do with their study.

Of these Technical Debt issues, the majority are linked to code debt (270 out of 321), underscoring the significance of code quality in microservices systems. In contrast, Service Design Debt, which pertains to the neglect of established best practices in designing open-source microservices systems, constitutes a smaller portion of the Technical Debt issues (51 out of 321). The authors discerned multiple issue types within the realm of Service Design Debt, which can be further categorized into aspects like business logic, service dependencies, missing functionality, and issues related to design patterns implementation and orphan responses (Waseem et al., 2021).

In the context of Build issues (which influences the process in which source code is converted into executable files for staging and production environments), the authors categorized them into three distinct groups: Build Errors (comprising 85 out of 145 issues), Broken or Missing Artifacts (consisting of 51 out of 145 issues), and Others (which encompassed 9 out of 145 issues). These issues predominantly revolve around challenges such as Build Errors, Docker Build Failures, issues associated with Broken or Missing Artifacts, and concerns regarding Obsolete APIs (Waseem et al., 2021).

In terms of Security issues (137 issues found), the authors created 5 different subcategories: Authentication and Authorization (41), Access Control (46), Encryption and Decryption (6), Secure Certificate and Connection (27) and Others (17). Issues such as Shared Authentication, API Key Security, Data Encryption, and HTTP Cookie can be found under this category (Waseem et al., 2021).

Finally, there are the Service Execution and Communication issues. In distributed environments, communication challenges can be misleading as services traverse multiple servers and hosts. These services engage in interactions utilizing various protocols, including HTTP, AMQP, and TCP, depending on the specific characteristics of the services. These are divided into two subcategories, Service Communication (102 out of 119) and Service Execution (17 out of 119).

### Smells and refactorings for microservices security: A multivocal literature review (Ponce et al., 2022)

In (Ponce et al., 2022) multivocal literature review, 58 primary studies were analysed, among those published from 2011 until the end of 2020. This review resulted in the gathering of ten bad smells for securing microservices, which were organised in a taxonomy, associating each smell with the security properties it may violate and the refactorings enabling it to mitigate its effects (Ponce et al., 2022).

The identified smells are as shown in Figure 3:

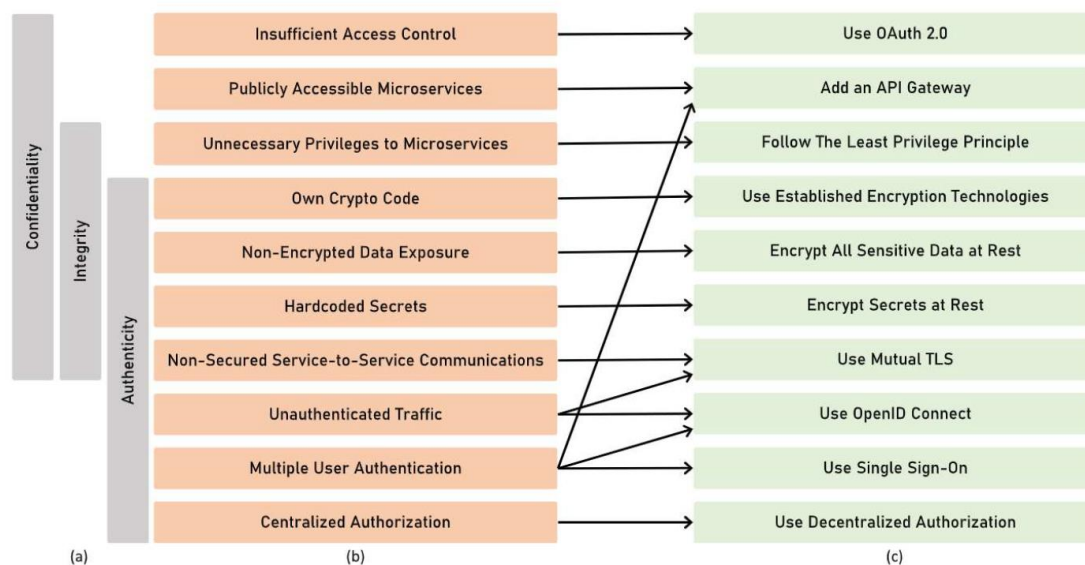


Figure 3 – Taxonomy of microservices security (a) properties, (b) smells, and (c) refactorings. For the sake of readability, the association between security properties and smells is represented by aligning the corresponding boxes, whilst that between smells and refactorings is represented with arrows (Ponce et al., 2022).

### Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry (Zhou et al., 2023)

(Zhou et al., 2023) carried out a series of industrial interviews with practitioners from 20 software companies and the collected data was then codified using qualitative methods. From the industry data analysis, eight sets of typical practices and challenges in microservices were identified. These findings highlight the importance of making informed decisions in five key

areas to strike a balance between the potential advantages and drawbacks of Microservices Architecture. Five research avenues were pinpointed, driven by the challenges associated with MSA, warranting further investigation and exploration.

The first set of practices and challenges pertains to componentization via services. The practice involves achieving independence through separation, while the challenge is managing chaotic independence. The next set focuses on organizing around business capabilities, where the practice involves structured organizational transformation, but the challenge lies in dealing with ad-hoc changes (Zhou et al., 2023).

There is the pair concerning smart endpoints and dumb pipes. Here, the practice entails choosing the right communication protocol, while the challenge arises from the complexity of API management. The concept of decentralized governance forms the basis of the next pair, with the practice emphasizing support for technology diversity, but the challenge emerges from excessive diversity (Zhou et al., 2023).

In the realm of decentralized data management, the practice centres on compromising with database decomposition, while the challenge manifests as data inconsistency. Infrastructure automation constitutes the following pair, where Continuous Integration/Continuous Deployment (CI/CD) is the recommended practice, yet the challenge often relates to inadequate automation (Zhou et al., 2023).

The pair tied to design for failure introduces microservices governance as the practice, while the challenge arises from unsatisfactory monitoring and logging. Lastly, evolutionary design concludes the list, with the recommended practice being stepwise evolution and the challenge relates to subjective decomposition (Zhou et al., 2023).

### **Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review (Söylemez et al., 2022)**

In their study, (Söylemez et al., 2022) performed a systematic literature review using the published literature since the introduction of microservices architecture in 2014. 3842 papers were discovered and 85 of them were selected as primary studies and analysed regarding research questions. Nine fundamental challenge categories were created (Service Discovery, Data Management and Consistency, Testing, Performance Prediction, Measurement and Optimization, Communication and Integration, Service Orchestration, Security, Monitoring, Tracing and Logging, and Decomposition), further breaking them down into 40 sub-categories, and delved into potential solutions. The authors also affirm that neglecting these identified challenges could hinder the realization of its anticipated benefits (Söylemez et al., 2022).



In their study, (Söylemez et al., 2022) identified a range of challenges across various aspects of microservices:

**Service Discovery Challenges:** These encompass issues such as Discovery Latency and Overhead, Design Choices and Decisions, Handling Service Discovery in Megascale Distributed Systems, Handling Service Discovery for Stateful Microservices, and Managing Unavailable Services.

**Data Management and Consistency Challenges:** This category includes Distributed Transaction Management, Data Sharing and Synchronization, and Backing-up Systems.

**Testing Challenges:** Challenges in this area involve Resilience Testing, Acceptance Testing, Test Automation, Defining a Comprehensive Testing Framework, Regression Testing, and Performance Testing.

**Communication and Integration Challenges:** (Söylemez et al., 2022) identified challenges related to Communication Infrastructure and Communication and Integration Protocols.

**Performance Prediction, Measurement, and Optimization Challenges:** These encompass Performance Prediction, Performance Measurement, and Performance Optimization.

**Service Orchestration Challenges:** This category includes challenges like Flow Control, Scalability, Storage Service Orchestration, Dynamic and Automated Orchestration, Understanding Failure-Repair Behaviour of Containers, Load Balancing, Resource Allocation and Scheduling, Communication and Collaboration, and Deployment. Specific challenges within Deployment encompass Heterogeneity of Functional and Non-Functional Requirements, Necessity of Deployment across Data Centre, Large Pulling Traffic and Long Response Times, and Deployment of Stateful Microservices and Service Recovery.

**Monitoring, Tracing, and Logging Challenges:** Challenges here involve managing a Large Number of Microservices, Distributed Tracing, Heterogeneity of Logs, Dependency Analysis, Architecture Extraction, and Root Cause Analysis for Anomalies and Performance Issues.

**Decomposition Challenge:** The sole challenge in this category is Identifying Microservices.

**Security Challenges:** Söylemez et al. (2022) identified challenges in Access Control, Providing a Comprehensive Framework, and Monitoring Network Traffic in the realm of microservices security.

Within each of these categories, (Söylemez et al., 2022) provided comprehensive descriptions of the challenges encountered, along with their corresponding solutions. This detailed analysis offers valuable insights into addressing the multifaceted challenges posed by microservices architecture.

## **Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation (Zhong et al., 2022)**

In their study, (Zhong et al., 2022) described five causes of architectural smells in microservices:

### **Cause 1: Business Aspects**

Among the practitioners in the MSA landscape, business aspects take centre stage as one of the primary drivers of architectural smells. This cause is intricately linked to the unique characteristics of the domain in which the microservices are applied and the development costs associated with microservice architecture. The inherent challenges posed by specific industries or business domains, coupled with the financial considerations involved in adopting microservice architecture, can catalyse the emergence of architecture smells (Zhong et al., 2022).

### **Cause 2: Organizational Structure and Culture**

Improper management of responsibilities within an organization and a reluctance to share code can create a fertile ground for the cultivation of architectural smells. This cause delves into the organizational aspects of microservice architecture implementation. The way responsibilities are structured and the prevailing culture within the organization play pivotal roles in the formation of architectural smells.

### **Cause 3: Technical Choices**

Technically, the choices made in terms of inter-service communication and the trade-offs among various quality attributes, such as performance, can significantly contribute to the occurrence of architectural smells. The intricacies of technical decisions within microservice architecture can inadvertently lead to architectural issues.

### **Cause 4: Project Management**

Effective project management is critical in MSA-based systems. Causes within this category encompass aspects related to the development process, such as the pursuit of development speed, and the delivery process, including deployment, maintenance, and deliverables. Flaws in project management can introduce architectural smells at various stages of the microservice architecture lifecycle.

### **Cause 5: Stakeholder Proficiency**

The proficiency and experience of stakeholders involved in designing microservices systems are essential factors in preventing architectural smells. Insufficient professionalism and expertise, particularly in the domain of microservices design, can lead to the introduction of architectural smells because of misinformed decisions and design choices.

Furthermore, (Zhong et al., 2022) present solutions for architectural smells, adapting current microservice-oriented decomposition approaches with additional considerations regarding addressing architectural smells in microservices.

### **On the Way to Microservices: Exploring Problems and Solutions from Online Q&A Community (Wu et al., 2022)**

(Wu et al., 2022) have developed a comprehensive taxonomy encompassing discussions related to microservices. This taxonomy comprises four distinct phases, delving into ten overarching categories, and further dissecting into sixteen specific topics. This structured framework provides a valuable roadmap for understanding and exploring the multifaceted landscape of microservices, offering insights into their various facets and intricacies.

As said previously, their taxonomy encompasses four distinct phases, each covering multiple categories and topics:

In the Architecture Design Phase (12.24% of the issues), the primary category is Microservice Design, which comprises one topic focusing on Design Strategy.

Moving to the Construction Phase (29.25% of the issues), it branches into the following categories:

- Microservice Communication, which includes topics on Inner-communication and Web Interaction.
- Failure Tolerance, with a specific focus on Exception Handling.
- Microservice Data Management, centring around Data Management.

In the Delivery Phase (25.82% of the issues), the taxonomy covers:

- Microservice Testing, with an emphasis on Testing when Deployment.
- Project Building, addressing Project Building itself.
- Project Deployment, encompassing topics such as Containers, Web Application Deployment, Deployment Pattern, and Deployment Platforms.

Lastly, in the Governance Phase (32.96% of the issues), the taxonomy branches into the following categories:

- Microservice Monitoring, which explores topics like Observability/Logging.
- Service Management, incorporating topics including Spring Cloud Components, Resource Management, and API Governance.
- Microservice Security, encompassing topics related to Authorization and Authentication.

(Wu et al., 2022) not only offer an elucidation of these phases alongside their associated categories and topics but also enrich the study by incorporating solutions to specific microservices-related problems.

### 3.2.6.3 RQ3 - What are the available microservice smell detection tools?

Much like the response to RQ1, the analysis of the documents selected during the screening phase and RQ3 is also included in the state-of-the-art section (Section 4.2) of this document. This comprehensive examination provides insights into the research landscape surrounding RQ3.

## 3.3 Data from Industry

A questionnaire was distributed to several software industry professionals who have experience with microservice architecture to gather data from the industry. This questionnaire was shared using different communication channels such as LinkedIn (private messages and with posts) and it was also spread in different companies (like Inditex) by using different proprietary business communication platforms like Microsoft Teams and Slack. A total of thirty-one answers were gathered and it can be seen in Appendix C (Survey).

### 3.3.1 Introduction – Demographic Questions

The initial section of the questionnaire focused on gathering information about the respondent's software engineering background. In a survey, the goal of inquiring about someone's experience with software engineering is to comprehend the context and viewpoint from which the respondent is providing feedback. This will make it more probable that the survey findings are insightful and that any recommendations or insights are founded on a thorough knowledge of the target population, as the people who don't have enough experience will not be able to answer the remaining questionnaire.

As can be analysed in Figure 4 most of the participants work in/have more experience in the area of software development (around 80.6%), with four participants (around 12.9%) in the area of software architecture and two participants (around 6.4%) in the area of software management.

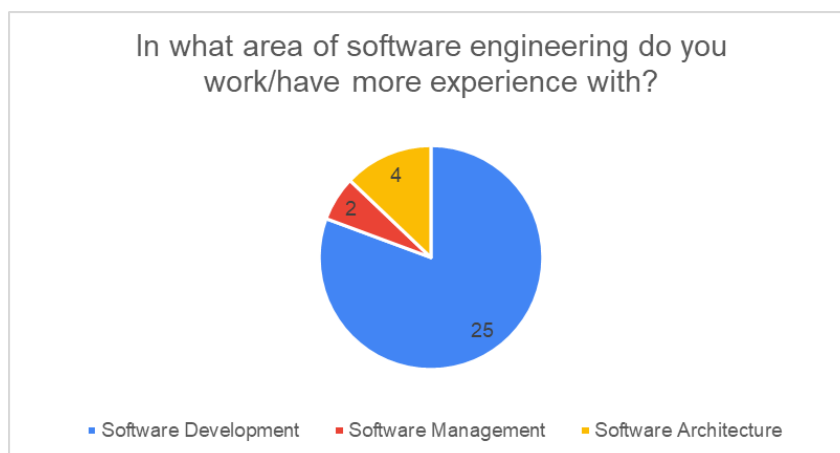


Figure 4 – Questionnaire: participant's area of software engineering.

The questionnaire was distributed and shared via LinkedIn, resulting in a diverse pool of respondents from various companies. Among the companies represented in the survey, Kodly had the highest number of respondents, followed by Inditex, as shown in Figure 5. Given that the majority of responses were submitted by consultancies, the survey results may provide insights from a variety of contexts, as these consultancies often work with multiple companies and industries.

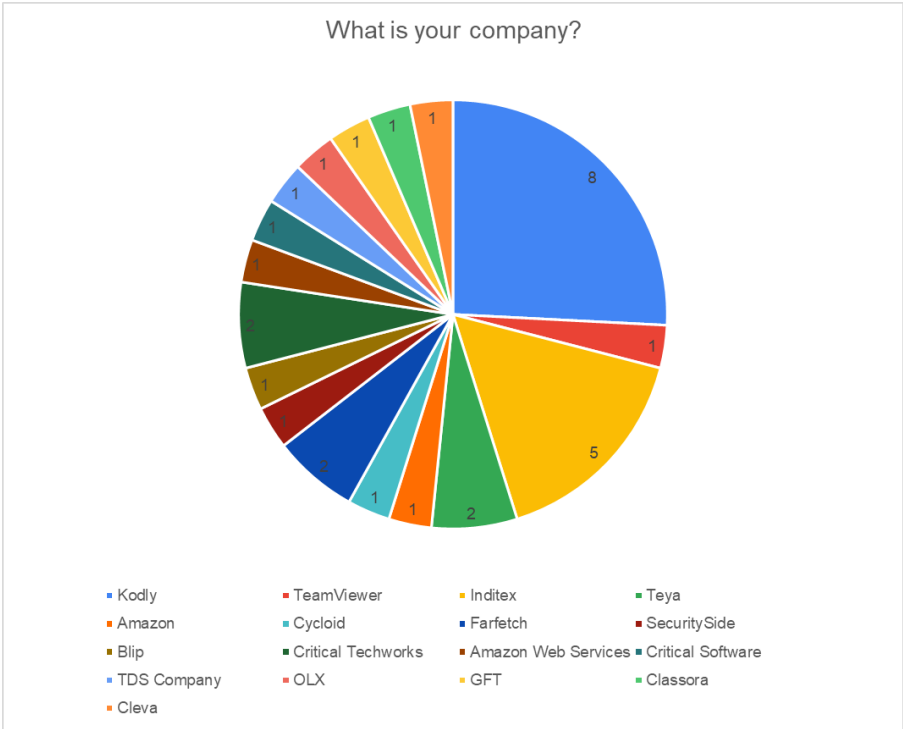


Figure 5 – Questionnaire: participant’s companies.

Figure 6 displays the distribution of participants' years of experience in software engineering. As shown, the majority of participants (approximately 64.5%) reported having more than 5 years of experience in the field, while 8 participants (around 25.8%) had 3 to 5 years of experience, and 3 had 1 to 2 years of experience (around 9.7%). Although participants were given the option to indicate experience levels below 1 year, no such responses were received. It is worth noting that any participant who indicated less than 1 year of experience would have been unable to complete the remainder of the questionnaire.

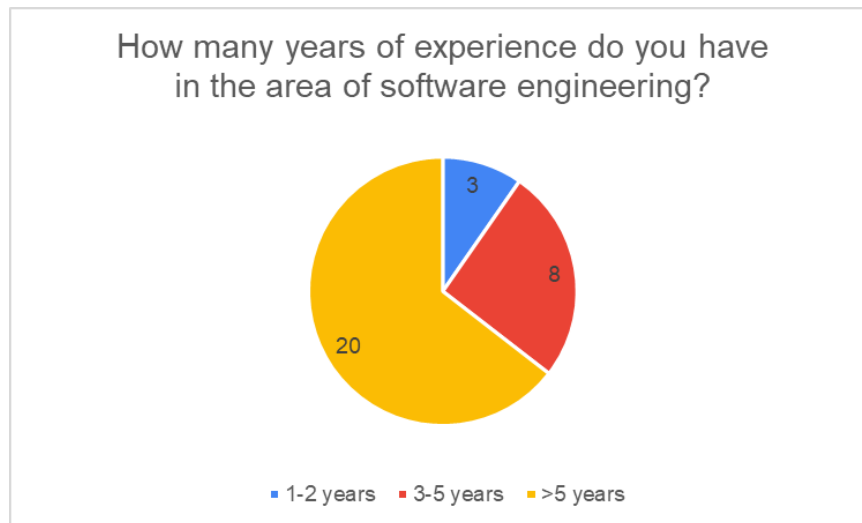


Figure 6 – Questionnaire: participant's years of experience.

To conclude the demographic questions, participants were asked to indicate their experience specifically with microservice architecture. As Figure 7 illustrates, this question revealed a range of experience levels among respondents. The majority of answers came from participants with significant experience in microservices, with the largest group comprising 10 participants who reported 3 to 5 years of experience. Close behind were 9 participants with more than 5 years of experience. The remaining participants reported having 1 to 2 years of experience (7 participants) or less than a year of experience (5 participants)

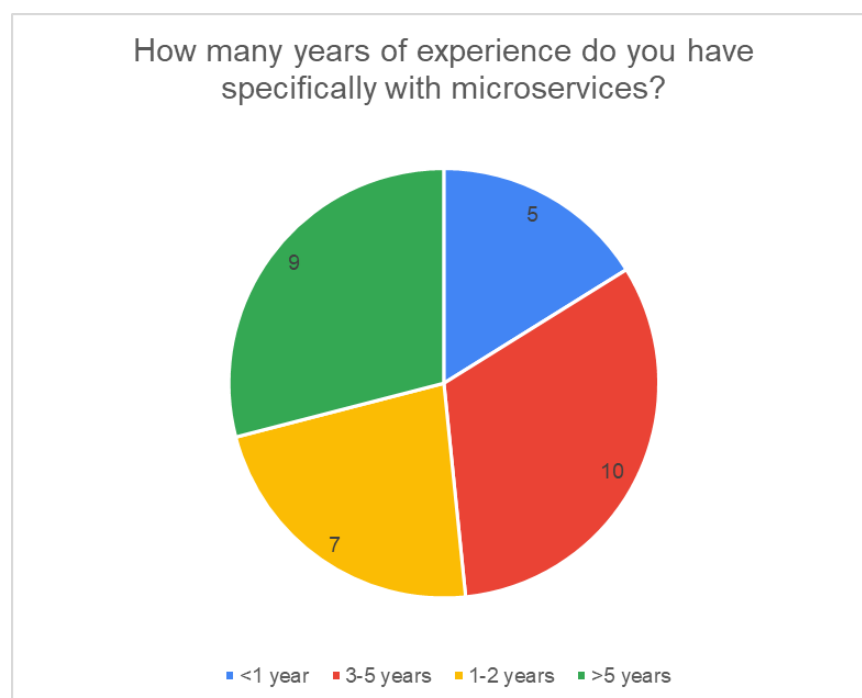


Figure 7 – Questionnaire: participant's experience with microservices.

### 3.3.2 Microservice Smells Catalogue Analysis

In this crucial section, participants were actively engaged in evaluating the effect of microservice smells as outlined in the catalogue provided by (Taibi et al., 2019). To ensure a comprehensive understanding, each participant was presented with a concise description of a specific microservice smell. They were then asked to evaluate and assess the importance of that smell in its potential impact on a microservice-based system.

By leveraging the collective expertise of these participants, the aim was to capture diverse insights and perspectives on the significance of each microservice smell. Their evaluations will contribute to refining the understanding of the potential risks and challenges associated with the smells that this catalogue provides in a microservice architecture.

Analysing the results based on participants' areas of software expertise, as depicted in Figure 8, reveals interesting discrepancies that can be attributed to their specific roles and responsibilities within the software domain. For instance, participants working in the area of Software Management (involving responsibilities such as overseeing project execution and coordinating teams) tend to place higher importance on microservice smells such as "Lack of Monitoring" or "No DevOps tools" than participants that work in the area of Software Development or Software Architecture.

It is important to note that the representation of Software Architecture and Software Management participants in this questionnaire was relatively small, with a total of 6 out of 31 participants. This limited sample size could be a contributing factor to the potentially inflated values observed in the evaluations.

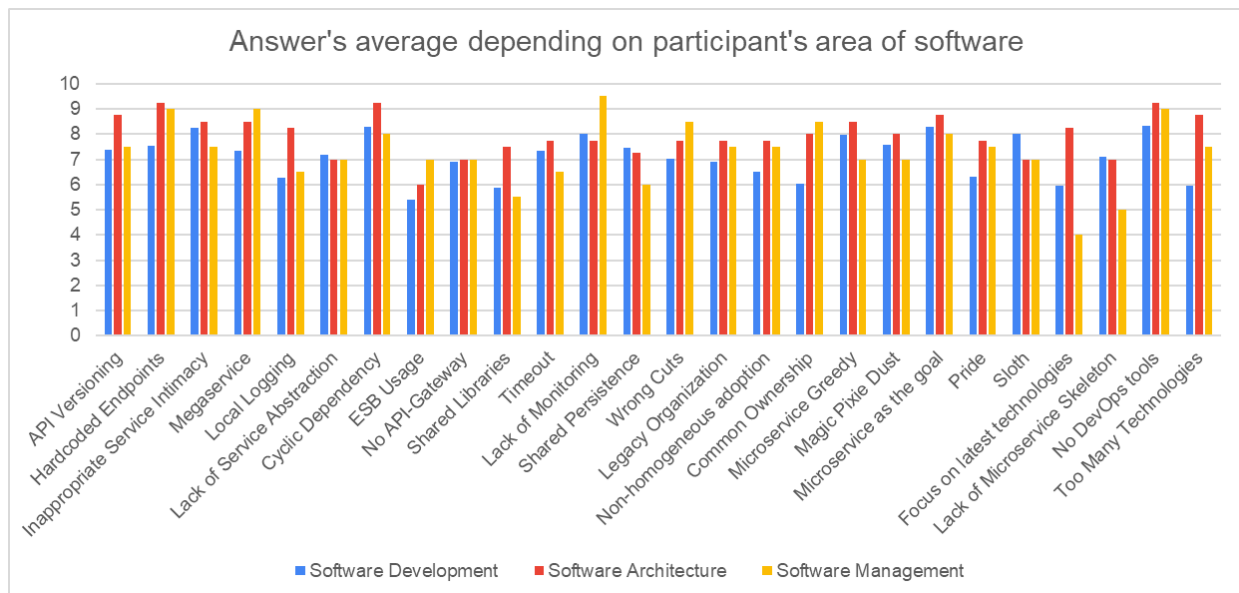


Figure 8 – Results of the participant's answers depending on their area of software.

Figure 9 illustrates the importance of microservice smells based on participants' years of experience in any software engineering area. Notably, participants with 3 to 5 years of

experience and those with more than 5 years of experience demonstrate a high degree of agreement in their evaluations. While there are a few instances where their perspectives diverge, overall, a pattern emerges as the opinions of these two groups align.

In contrast, participants with 1 to 2 years of experience present a distinct pattern and offer differing opinions compared to their more experienced counterparts. This disparity indicates that as developers gain more experience, their perspectives on the importance of microservice smells tend to converge.

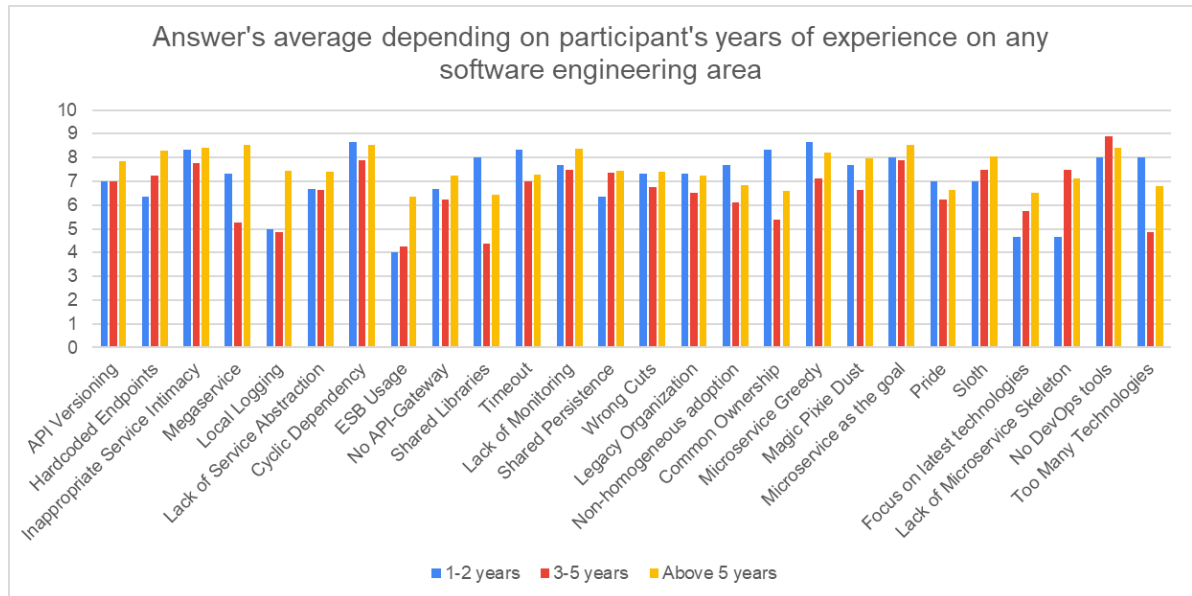


Figure 9 – Results of the participant's answers depending on their experience in any software engineering area.

The results from the survey's most varied group are shown in Figure 10. Due to the increased number of diverse participants in terms of experience with microservice architecture, it was expected that it would be difficult to draw unambiguous conclusions from the data. However, despite these challenges, some noteworthy patterns emerge.

Notably, participants with less experience in the field of microservices demonstrate consistent opinions across different years of experience. This finding suggests that their perspectives on certain microservice smells, such as "Inappropriate Service Intimacy" and "No API Gateway," remain relatively stable over time.

This observation raises interesting questions about the underlying factors influencing these participants' perceptions. It is possible that early experiences in microservices strongly shape their understanding and evaluation of specific smells, leading to consistent opinions regardless of increasing years of experience. By exploring these consistent patterns among less experienced participants, valuable insights into the long-term implications and potential challenges associated with specific microservice smells would be provided.



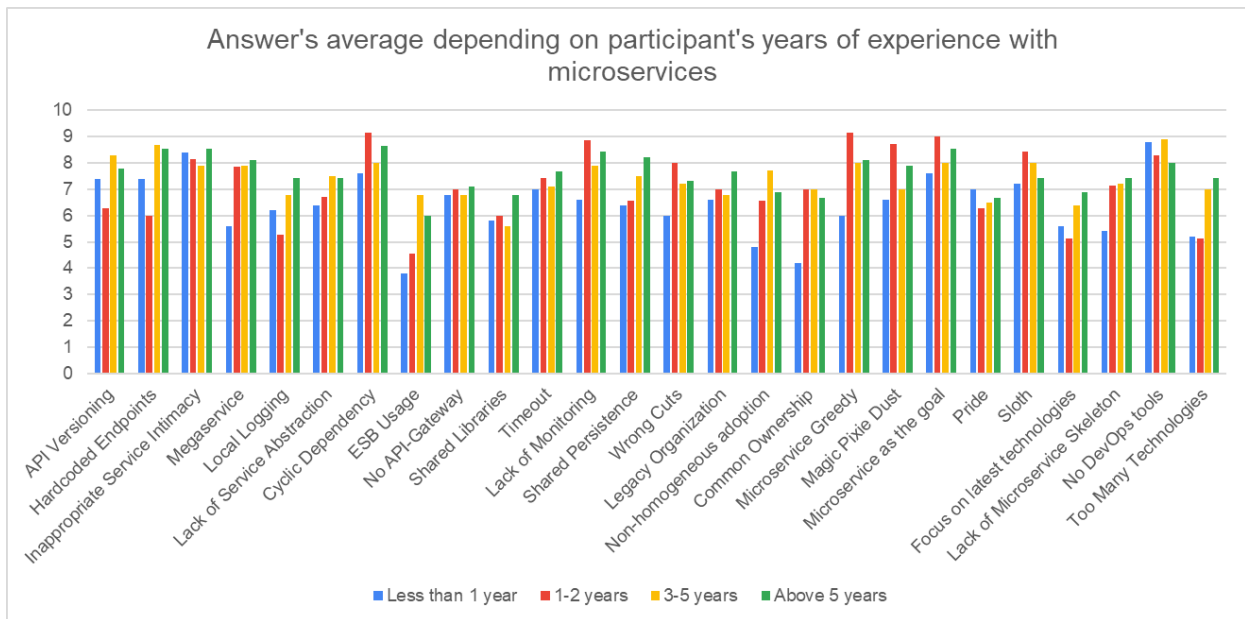


Figure 10 – Results of the participant's answers depending on their experience with microservice architecture specifically.

To conclude this section on the analysis of the catalogue, Figure 11 presents a graph displaying the average results independent of participants' experience levels and areas of expertise. The graph reveals that the highest rating given by participants is the No DevOps tools with a mean of 8.5 and the lowest being the ESB Usage with a mean value of 5.6.

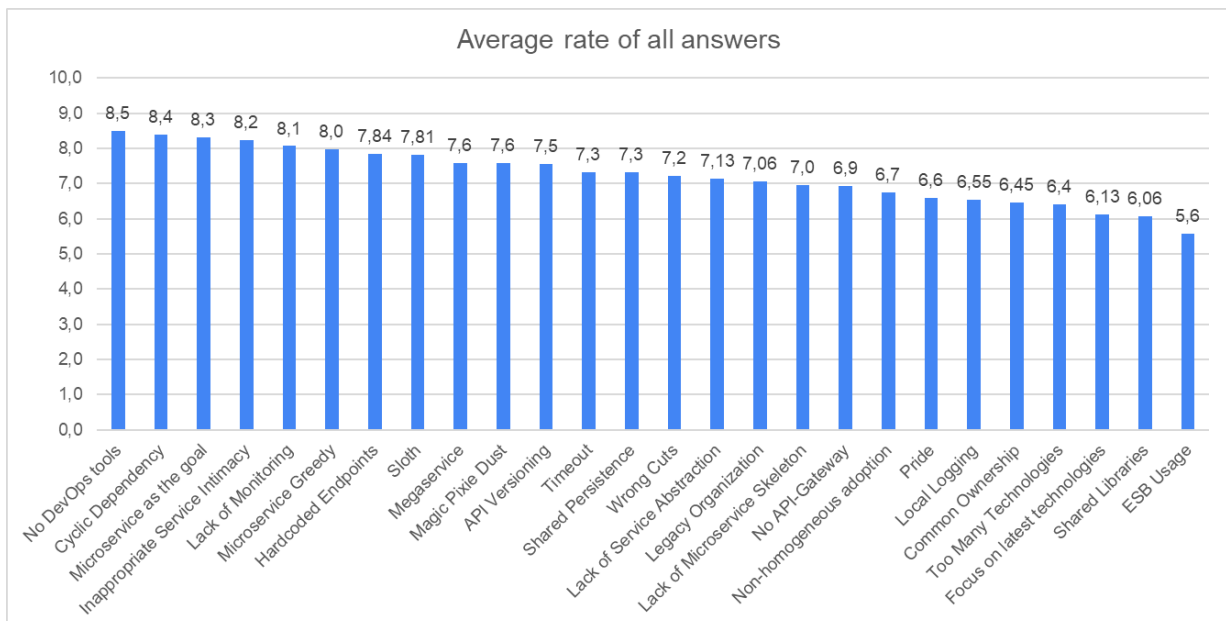


Figure 11 – Results of the participant's answers independently from any variable shown before.

The respondents' choice of "No DevOps Tools" as the top microservice smell likely reflects their awareness of how essential DevOps tools are for successful microservice architectures.

The absence of these tools can significantly impact operational efficiency, scalability, security, and overall system reliability, making it a critical concern for those evaluating microservice smells.

### **3.3.3 Open-answer questions**

After the questionnaire, participants were allowed to contribute any additional smells they deemed relevant to the catalogue. The goal of this open-answer question was to provide participants with an opportunity to contribute their insights and perspectives by suggesting any additional microservice smells that they believed were important but not included in the existing catalogue. By incorporating participants' suggestions, the goal was to ensure a more comprehensive and inclusive representation of microservice smells in the final catalogue, thereby enhancing its usefulness and relevance in real-world scenarios.

Out of the 31 participants, 7 responses were received. However, it is important to note that 2 of these responses were considered invalid, resulting in a final count of 5 responses on smells suggested by participants.

One of the participants talked about “Data distortion and duplication across persistencies” which refers to a common issue encountered in microservice architectures where data becomes distorted or duplicated when it is stored across multiple persistency layers or databases.

The respondent specifically notes that this issue can be particularly problematic when dealing with personally identifiable information (PII), which is subject to regulations like the General Data Protection Regulation (GDPR). This suggests that the mishandling of PII data, such as inaccurate or inconsistent storage across persistencies, can have severe consequences in terms of legal compliance and data privacy.

Regarding data management issues on microservices, there were another three respondents who raised a problem regarding boundaries on business logic. These responses address the importance of clearly defining boundaries and bounded contexts within the microservice architecture. Participants suggest that when there is failure to establish clear boundaries and ownership of each team's bounded context can result in ambiguity, overlapping responsibilities, and potential conflicts.

By determining whether a microservice is business logic bound or database query bound, developers gain valuable insights into the specific areas where performance improvements are needed. Addressing these performance-related smells is crucial for optimizing the overall efficiency and scalability of the microservice architecture.

Another participant mentioned that the deployment of a settings service that contains all the microservice settings would be important to avoid bigger deployment times (since the configuration settings are tightly coupled with the microservice, any modification to a single setting necessitates deploying the entire microservice again) and unnecessary redeployments.

It is also suggested that to address this smell and improve configuration management, the respondent suggests using a Helm<sup>1</sup> repository or a similar tool which is a package manager for Kubernetes that allows the separation of configuration settings from the microservice deployment.

### **3.4 Threats to validity**

There are, naturally, threats to validity and this section is intended to identify and discuss potential limitations and threats that may impact the validity of this research.

Starting with the research done using systematic mapping, the inclusion and exclusion criteria used to select studies may inadvertently exclude relevant research or include irrelevant studies, leading to a biased sample and the search strategy which if it is not comprehensive, may miss important studies. There can also be errors or subjectivity in data extraction and categorization of studies that can introduce bias if different researchers interpret and classify studies differently.

Regarding the survey, as it was done online there are a few different threats to its validity. As this was a long survey, respondents may have become fatigued and provided less thoughtful or consistent responses. Another threat is, as the survey response rate is low, and those who responded differ systematically from non-respondents in ways that can affect the study's outcomes.

---

<sup>1</sup> <https://helm.sh/>

## 4 State of the Art

To effectively tackle the problem at hand, it is imperative to have a clear understanding of the current knowledge and state of the field. This involves identifying what aspects can be improved or what has yet to be addressed. To achieve these objectives, this chapter is structured into two sections that delve into these topics in depth.

### 4.1 Microservice architectural smells

The objective of this section is to examine the literature and identify any existing compilations of microservice architecture smells for research purposes.

To conduct a thorough investigation of microservice architecture smells catalogues, it is necessary to begin by examining the topic from a broad perspective and then gradually move towards the specifics. Generally, the subject of architectural smells has not been extensively researched, which is why there are only a limited number of catalogues available for exploration. In this regard (Azadi et al., 2019) have referenced some of the work that has been done in this area as well as proposed a catalogue of architectural smells detected by tools. This catalogue includes a total of 12 architectural smells, each of which is characterized by a description, the violated principles, and the tools that can detect the particular architectural smell. This catalogue also provides a comparison of the detection capabilities of the different tools that can detect each architectural smell.

#### 4.1.1 Catalogues

##### **“On the Definition of Microservice Bad Smells” - (Taibi & Lenarduzzi, 2018)**

As we delve further into the topic of microservice architectural smells, it is worth noting that one of the earliest works in this area was conducted by (Taibi & Lenarduzzi, 2018). Their study highlighted the fact that no empirical research had been conducted on bad practices, antipatterns, or smells specifically related to microservices. The authors began their investigation by analysing a book titled "Microservices AntiPatterns and Pitfalls" (Richards, 2016) which identified three main pitfalls: Timeout, I Was Taught to Share, and Static Contract Pitfall. Subsequently, the authors reviewed other relevant works and compiled a table of additional pitfalls, which can be found in Appendix B of this document.

After reuniting the pitfalls mentioned, the authors conducted a survey among experienced developers, who were interviewed first to know if they were qualified to have a useful answer to the survey. The survey's objective was to determine which bad practices had the most effects on system development and what remedies were being used to correct them. To do this, it was necessary to ask the respondents to rate each detrimental practice on a scale of 0 to 10, where 0 denoted that the activity was not harmful and 10 denoted that it was exceedingly damaging. The rankings were used to identify which detrimental behaviours were more prevalent and to direct the creation of remedial measures. It was highlighted that the individual values themselves lacked importance and that only the rankings of the harmful behaviours did.

A total of 72 interviews were conducted with experienced developers, software architects, project managers, and agile coaches from 61 different organizations, no inexperienced participants were included in the study and all interviewees had at least five years of experience in software development. The participants belonged to different industries, including banks, companies that produce and sell their software as a service, consultancy companies specializing in migration to microservices, public administrations, and telecommunications companies. The practitioners reported a total of 265 different bad practices with their corresponding solutions, which were grouped based on open and selective coding, resulting in 11 microservice smells. The resulting smells and their descriptions are reported in Table 8. The full survey also includes the possible solutions for the smells, the adoption timeline for microservices by the organizations and the number of bad practices reported by each participant on average.

Microservice Smell	Description
API Versioning	APIs are not semantically versioned. Also proposed as Static Contract Pitfall.
Cyclic Dependency	A cyclic chain of calls between microservices exists.
ESB Usage	The microservices communicate via an enterprise service bus (ESB). An ESB is used for connecting microservices
Hard-Coded Endpoints	Hardcoded IP addresses and ports of the services between connected microservices exist. Also proposed as Hardcoded IPs and Ports.
Inappropriate Service Intimacy	The microservice keeps on connecting to private data from other services instead of dealing with its own data.
Microservice Greedy	Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages.
Not Having an API Gateway	Microservices communicate directly with each other. In the worst case, the service consumers also communicate directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance.
Shared Libraries	Shared libraries between different microservices are used.
Shared Persistence	Different microservices access the same relational database. In the worst case, different services access the same entities of the same relational database. Also proposed as Data Ownership.
Too Many Standards	Different development languages, protocols, frameworks, etc. are used. Also proposed as the Lust and Gluttony bad practices.
Wrong Cuts	Microservices are split based on technical layers (presentation, business, and data layers) instead of business capabilities.

Table 8 – Microservice smells catalogue proposed by (Taibi & Lenarduzzi, 2018).

#### **“Microservices Anti-Patterns: A Taxonomy” - (Taibi et al., 2019)**

The authors (Taibi et al., 2019) replicated and extended their work done on (Taibi & Lenarduzzi, 2018) using a mixed research method that combined an industrial survey, literature review, and interviews. They interviewed 27 experienced developers from 27 different organizations, who completed the same survey as in their previous study and were also asked if they had experienced any of the microservice smells presented in Table 8.

Upon concluding the study, a total of 20 microservice smells were collected, which is 9 more than what was gathered in the authors' previous study. This time, the microservice smells were categorized by the authors into two main groups: technical (including internal, communication, and other types) and organizational (including team-oriented and technology and tool-oriented types).

Technical microservice smells, as was said previously, can be categorized into three groups: internal, communication and others. Internal microservice smells impact the individual microservice and are listed in Table 9. Communication microservice smells are anti-patterns

that relate to the communication between microservices and are listed in Table 10. In addition, there are other types of technical microservice smells that do not fit into either the internal or communication categories and are listed in Table 11.

It should be noted that in each table, the microservice smells that are underlined indicate that they were newly added in the present study. While the study presents a way to detect each smell, as well as the issues it may cause and the solutions proposed by the interviewees, only the descriptions of the smells are provided.

Microservice Smell	Description
API Versioning	APIs are not semantically versioned. Also proposed as “Static Contract Pitfall”.
Hardcoded Endpoints	Hardcoded IP addresses and ports of the services between connected microservices. Also proposed as “Hardcoded IPs and Ports”.
Inappropriate Service Intimacy	The microservice keeps on connecting to private data from other services instead of dealing with its own data.
<u>Megaservice</u>	A service that does a lot of things. A monolith.
<u>Local Logging</u>	Logs are stored locally in each microservice, instead of using a distributed logging system.

Table 9 – Internal microservice smells (Taibi et al., 2019).

Microservice Smell	Description
Cyclic Dependency	A cyclic chain of calls between microservice
ESB Usage	The microservices communicate via an Enterprise Service Bus (ESB). Usage of ESB for connecting microservices
No API Gateway	Microservices communicate directly with each other. In the worst case, the service consumers also communicate directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance.
Shared Libraries	Usage of shared libraries between different microservices. Also named “I was taught to share”.

Table 10 – Communication microservice smells (Taibi et al., 2019).

Microservice Smell	Description
<u>Lack of Monitoring</u>	Lack of usage of monitoring systems, including systems to monitor if a service is alive or if it responds correctly.
Shared Persistence	Different microservices access the same relational database. In the worst case, different services access the same entities of the same relational database. Also proposed as “data ownership”.
Wrong Cuts	Microservices should be split based on business capabilities, not on technical layers (presentation, business, data layers).

Table 11 – Other technical microservice smells (Taibi et al., 2019).

Organizational microservice smells can be categorized into two groups: Team-Oriented and Technology and Tool Oriented. Team Oriented smells are anti-patterns that are related to the team's dynamics and are listed in Table 12. Technology and Tool Oriented are listed in Table 13.

<b>Microservice Smell</b>	<b>Description</b>
<u>Legacy Organization</u>	The company still work without changing its processes and policies. As example, with independent Dev and Ops teams, manual testing and scheduling common releases. Also proposed as "Red Flag".
<u>Non-homogeneous Adoption</u>	Only a few teams migrated to microservices, and the decision to migrate or not is delegated to the teams. Also defined as "Scattershot Adoption".
<u>Common Ownership</u>	One team own all the microservices
Microservice Greedy	Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages.

Table 12 – Team Oriented Microservice smells (Taibi et al., 2019).

<b>Microservice Smell</b>	<b>Description</b>
<u>Focus on the latest technologies</u>	The migration is focused on the adoption of the newest and coolest technologies, instead of based on real. The decomposition is based on the needs of the different technologies aimed to be adopted. Also proposed as "Focusing on Technology".
<u>Lack of Microservice Skeleton</u>	Each team develop microservices from scratch, without benefit of a shared skeleton that would speed up the connection to the shared infrastructure (e.g., connection to the API-Gateway).
<u>No DevOps Tools</u>	The company does not employ CD/CI tools and developers need to manually test and deploy the system.
Too Many Technologies	Usage of different technologies, including development languages, protocols, frameworks... Also proposed as "Lust" and "Gluttony".

Table 13 – Technology and Tool Oriented microservice smells (Taibi et al., 2019).

As previously mentioned, the authors utilized multiple research methods, including industrial surveys, interviews, and literature reviews. Through the literature review, additional microservice smells were identified but were not included in any of the existing categories as the interviewees did not consider them to be problematic. Table 14 presents these microservice smells that were previously missing.



Microservice Smell	Description	Group
Lack of Service Abstraction	Service interfaces are designed for generic purposes and not specifically designed for each service.	Technical Internal
Timeout	Management of remote process availability and responsiveness. It is recommended to use a timeout value for service responsiveness or sharing the availability and the unavailability of each service through a message bus, to avoid useless calls and potential timeout due to service unresponsiveness.	Technical Communication
Magic Pixie Dust	Believing a sprinkle of microservices will solve the development problems	Organizational Team-Oriented
Microservice as the goal	Migrating to microservices because everybody does it, and not because the company need it.	Organizational Team-Oriented
Pride	Testing in the world of transience.	Organizational Team-Oriented
Sloth	Creation of a distributed monolith due to the lack of independence of microservices.	Organizational Team-Oriented

Table 14 – Missing microservice from other sources of research methods found by (Taibi et al., 2019)

#### **“On the Study of Microservices Antipatterns: a Catalog Proposal” - (Tighilt et al., 2020)**

In this paper the authors (Tighilt et al., 2020) present a catalogue of microservice antipatterns that were discovered after a systematic literature review of papers on microservice architecture design and also after analysis of microservice-based systems.

The authors followed a systematic literature review process based on the guidelines proposed by (Barbara Kitchenham, 2004). They began by collecting research papers using relevant search queries related to microservices and antipatterns. The search was conducted in scientific search engines, resulting in a total of 1,195 unique references. The authors filtered these references based on title, abstract, and content, resulting in 21 papers specifically focused on the design of microservice-based systems. They then employed forward and backward snowballing techniques to identify additional relevant papers, iterating the process five times. In the end, a total of 27 papers describing microservice antipatterns were included (Tighilt et al., 2020).

To gain a deeper understanding of microservice antipatterns the authors conducted a manual analysis of 67 open-source microservice-based systems. This analysis aimed to identify potential violations of microservice design practices, which could indicate the presence of antipatterns. The implementation of each detected antipattern in the source code and documented the symptoms or hints associated with them were examined by the authors. This process helped them identify the specific refactoring solutions or practices that should be employed to address and eliminate these antipatterns (Tighilt et al., 2020).

Through discussions among the authors and considering previous studies on service-oriented architecture (SOA) patterns and antipatterns, the specifications and definitions of each microservice antipattern, along with their symptoms and possible refactoring solutions, were generalized. To describe the microservice antipatterns, the authors adapted a template from (Dudney et al., 2002), which includes the following elements:

- Antipattern: Name of the specific antipattern.
- Context: The circumstances in which the antipattern may occur.
- General form: How the antipattern is manifested.
- Symptoms: Indications or elements that indicate the presence of the antipattern.
- Consequences: The drawbacks associated with the presence of the antipattern.
- Refactored solution: The steps required to remove the antipattern and apply best practices.
- Advantages of refactoring: The benefits gained from eliminating the antipattern through the suggested refactoring solution.
- Trade-offs: The considerations and trade-offs involved in deciding whether to keep or remove the antipattern.

Similarly to the previous catalogue (Taibi et al., 2019), the authors organized their proposed antipattern catalogue into four categories, aligning with the development cycle of a microservice-based system. These categories are as follows (Tighilt et al., 2020):

- Design: This category encompasses antipatterns related to the specification of the architectural design of a microservice-based system.
- Implementation: Antipatterns in this category pertain to how the microservices are implemented within the system.
- Deployment: This category covers antipatterns associated with the packaging and deployment of microservice-based systems.
- Monitoring: Antipatterns within this category are concerned with the monitoring of microservice-based systems, including their behaviour and changes.

To assess the impact of each antipattern, the authors utilized the scale proposed by (Taibi et al., 2019). This scale assigns a level of impact (high, moderate, or low) to developers and end-users based on observations. The impact levels are determined as follows (Tighilt et al., 2020):

- High: Antipattern consequences directly affect end-users.
- Moderate: End-users may indirectly experience some impact, either in terms of performance or application evolution.
- Low: Antipattern consequences have minimal to no impact on end-users, primarily resulting in increased maintenance or deployment costs.

The resulting catalogue comprises a total of 16 antipatterns, which will be detailed in the subsequent tables, such as Table 15, Table 16, Table 17 and Table 18Table 17. In the

forthcoming sections, a similar approach will be followed as described earlier. Each table will highlight newly added microservice smells by underlining their names. However, in this case, the context and symptoms will be provided.

<b>Microservice Antipattern</b>	<b>Context</b>	<b>Symptoms</b>
Wrong Cut	A microservice should encapsulate a group of functionalities to allow the independent delivery of business capabilities. It should be owned, developed, and deployed by a single team. It should fulfil a single purpose.	Some of the following aspects can indicate the presence of the Wrong Cut antipattern in a microservice-based system: (1) high microservice coupling; (2) process calls; (3) front-end/ORM microservices; or (4) deployment dependencies.
Cyclic Dependencies	Microservices should be independent processing units that communicate through lightweight mechanisms (Fowler & Lewis, 2014) to avoid managing dependencies and the “distributed monolith” pitfall (Taibi & Lenarduzzi, 2018).	Cyclic dependencies manifest through (1) direct calls between microservices; (2) frequent communications between microservices; or, (3) the presence of HTTP requests in call-backs.
Mega Service	Microservices should be small and independent units, independently deployable and serving a single purpose (Fowler & Lewis, 2014).	A mega microservice is a microservice with a high number of lines of code, modules, or files, as well as a high fan-in.
<u>Nano Service</u>	Refactoring a monolith system into a microservices-based system is a complex problem. Microservices should fulfil single business capabilities, no more but also no less.	The nano microservice antipattern exists when (1) the system has a large number of microservices; (2) microservices exchange a lot of information; or (3) cyclic dependencies exist.

Table 15 – Design Antipatterns (Tighilt et al., 2020).

Microservice Antipattern	Context	Symptoms
Shared Libraries	Microservices should avoid sharing runtime libraries and code directly.	The presence of executable files or runtime libraries shared among multiple microservices, added at compile or packaging time, can indicate this antipattern.
Hardcoded Endpoints	Microservices must communicate with one another. They are independently deployed and usually communicate through REST APIs. Microservices can reach one another endpoints via IP addresses and port numbers.	Hardcoded endpoints antipattern show via the presence of IP addresses or fully qualified domain names in source code, configuration files, or environment variables.

Table 16 – Implementation Antipatterns (Tighilt et al., 2020).

Microservice Antipattern	Context	Symptoms
<u>Manual Configuration</u>	Microservices efficiency relies on automation and everything that can be automated should be automated.	Configuration files in every microservice and the reliance on environment variables can indicate the presence of this antipattern
No Continuous Integration / Continuous Delivery (CI/CD) (also known as No DevOps tools (Taibi et al., 2019))	The independent deployment of microservices allows relatively small teams -within a single enterprise- to easily apply iterative continuous development and delivery (DevOps) processes, and thereby increase system agility. The integration of Development and Operations, and the continuous delivery result in (1) reducing delivery time; (2) increasing delivery efficiency; (3) decreasing time between releases; and (4) maintaining software quality.	Some of the following symptoms can indicate the presence of the no CI/CD antipattern: (1) no version control repositories on microservices; (2) no unit/integration/functional tests; (3) no automated delivery tools; or (4) no staging environments.
No API Gateway	When building microservices-based systems, consumer applications need to communicate with a lot of microservices, and every consumer needs a very specific set of information.	Consumer applications sending multiple HTTP requests, or requests to multiple different URLs, and systems that have multiple front ends (Web, mobile, etc.) can be indicative of the presence of this antipattern

Microservice Antipattern	Context	Symptoms
Timeouts	Service availability refers to the possibility for a service consumer to connect and send a request to a service. Service responsiveness is the time taken by the service to respond to that request. It is common practice in distributed systems to have consumer applications/tasks use timeouts to handle service unavailability or unresponsiveness.	Request retrial and timeout values are good signs of the presence of this antipattern.
<u>Multiple Service Instances Per Host</u>	When microservices are built, multiple deployment strategies could be applied. We can choose to deploy either each microservice instance in its host or multiple microservices instances in a single host.	The hints of the presence of this antipattern could be (1) a single deployment platform; (2) a single version control repository; or (3) a global deployment script.
Shared Persistence	Microservices architecture is a way of building systems that decompose application code into small independent services. Each of these small services may need to persist and access data. However, to fully benefit from the microservices architecture, software architects need to handle data storage in a way where each microservice can store and access its data without affecting other microservices.	This antipattern is characterized by one or more of the following symptoms: (1) multiple microservices share the same configuration files and deployment environments; (2) database tables are prefixed; or (3) databases have a lot of schemas
No API Versioning	Sometimes, multiple versions of the exposed API of a given microservice must be supported. This is generally the case when a service API has undergone major changes and we need to support both the new and old versions for some period.	Some of the following are hints to the presence of this antipattern: (1) microservices endpoint URLs do not contain version numbers; (2) no custom header information is sent by the client; and (3) multiple microservices have similar names.

Table 17 – Deployment Antipatterns (Tighilt et al., 2020).

<b>Microservice Antipattern</b>	<b>Context</b>	<b>Symptoms</b>
<u>No Health Check</u>	The nature of microservices is volatile. A microservice can be deployed anywhere and can be unavailable for a particular amount of time or in a particular context.	No periodic HTTP request, no API gateway or no service discovery can be hints of the presence of this antipattern.
<u>Local Logging</u>	Each microservice produces a lot of information that is being logged in different file systems. This information is very useful in a monitoring context and should be easily accessed and stored.	Some indications of this antipattern are (1) the presence of log files inside microservices; (2) files being written by the microservice; (3) the usage of time-aware databases; and (4) logging frameworks and tools.
Insufficient Monitoring (also known as “Lack of Monitoring” (Taibi et al., 2019))	Because provided microservices are often subject to service level agreements (SLA), monitoring their behaviour and performance is crucial.	Some indications of this antipattern include the use of local logging for some microservices or the absence of health check endpoints.

Table 18 – Monitoring Antipatterns (Tighilt et al., 2020).

### **How Can We Cope with the Impact of Microservice Architecture Smells? (Ding & Zhang, 2022)**

This article presents a comprehensive Systematic Literature Review (SLR) that addresses the problems. The review involved an exploration of 13 white and 10 grey literature sources to gather relevant information. All the information was synthesized using the meta-ethnography qualitative method to answer specific questions.

The article investigates and provides an explicit definition of Microservice Architectures Smells based on their distinct characteristics, aiming to offer developers valuable insights into each Microservice Architecture Smells. It also defines five categories of Microservice Architectures, namely Design, Deployment, Monitor & Log, Communication, and Team & Tool. These categories were established by comparing the violated design principles and their influences on software systems. Additionally, the paper proposes a description template for defining Microservice Architectures. The paper delves into the issues caused by Microservice Architectures during the migration process from a monolithic system to Microservices. By aligning the identified issues and Microservice Architectures, the authors offer valuable solutions for developers and scholars involved in this migration process.

To achieve its objectives, the article puts forth two research questions which are: (1) How can one define and classify the existing Architectural Smells in the context of Microservice

architecture? (2) What issues arise in the migration process from Monolith to Microservice architecture, caused by architecture smells?

This study employed a combination of 13 white literature and 10 grey literature sources to gather information on Microservice Architecture Smells. To enhance the comprehension of specific smells, the researchers developed a template that included essential details such as the category, name of the smell, definition, design violation, interest, solution, and similar smells.

In total, the study identified and presented 22 distinct Microservice Architecture Smell, classifying them into five categories: Design, Deployment, Monitor & Log, Communication, and Team & Tool. These smells will be shown through the presented in Table 19, Table 20, Table 21, Table 22, and Table 23. In the forthcoming tables, a similar approach will be followed as described earlier. Each table will highlight newly added microservice smells by underlining their names. However, in this case, only name, definition and interest will be provided.

<b>Microservice Antipattern</b>	<b>Definition</b>	<b>Interest</b>
Use of business logic in communication among services	If the communication layer contains business logic means it has this smell such as data transfer in the communication layer.	There will be additional maintenance costs and changes in the service logic will cause the communication layer to change as well. And communication team must understand the details in the logic of services.
Mega Microservice	One microservices takes responsibility for any concerns.	It will cause difficulty in maintaining, testing and complexity of software.
Nano Microservice	The granularity of a monolith system divided into microservices is so fine that a single microservice does not fulfil one business capability.	The principle for monolith to microservice is based on business capability and suitable granularity. It will cause the coupling of the services.
Cyclic Dependencies	The dependencies or calls of microservices are like a cycle.	The microservices are not independent and if one fails will cause other microservices to fail too.
Wrong Cuts	The principle for dividing microservices should be based on business capability and microservices just focus on one single concern. If dividing microservice does not follow the principle will cause the smell.	It will cause the complexity and high coupling of microservice. It is harder to maintain the microservices

Table 19 – Design microservice smells (Ding & Zhang, 2022).

Microservice Antipattern	Definition	Interest
Wobbly service interactions	Microservice needs to be independent and clear boundaries of others. If the interaction of microservices is wobbly that means it is this smell.	If one microservice fails will cause another one to fail too and cause the coupling of different microservices.
Shared Libraries	Microservices share runtime libraries or execution files.	It causes the coupling of different microservices.
Shared Persistence	Multi microservices share one database.	It will cause the coupling of different microservices and harder to maintain them.

Table 20 – Deployment microservice smells (Ding & Zhang, 2022).

Microservice Antipattern	Definition	Interest
Insufficient message traceability	When messages contain insufficient data that causes difficult to find the source of the messages	It will make it harder to find dependencies of microservices
<u>Manual Configuration</u>	Configuration of instances, services and hosts is done manually by developers.	Manual configuration is time-consuming and error-prone
<u>Dismiss Documentation</u>	Inconsistent documents or outdated documents for software will cause this smell	With the development of microservices, if the documents of API are dismissed that will hinder the cooperation of different teams.
No health check	No endpoints for checking the health of the microservice	Consumers may wait a long time to get a response from microservices that are down.
Local Logging	The information or logs of microservices are stored in local storage	The logs locally are difficult to analyse and monitor.

Table 21 – Monitor & Log microservice smells (Ding & Zhang, 2022).



<b>Microservice Antipattern</b>	<b>Definition</b>	<b>Interest</b>
Single Layer Team	One team takes responsibility for many services. Each team has no explicit responsibility boundary for services.	It destroys the independence among services and causes external communication costs among teams
Inadequate techniques support	The techniques or tools for Microservice are not enough and dismiss a lot of key points of Microservice.	The automation of the service is broken and most of its advantages are not demonstrated.
Too many Standards	In one program, many developing languages or frameworks are used. Each team uses their techniques and no specific standard to limit them.	Add unnecessary complexity and maintenance problems.

Table 22 - Team & Tool microservice smells (Ding & Zhang, 2022).

<b>Microservice Antipattern</b>	<b>Definition</b>	<b>Interest</b>
<u>Lack of communication standards among microservices</u>	Lack of API or message format for microservices. Each team has its standards for communication	Need costs more to transform the messages and add the complexity of software.
Hardcoded Endpoints	The IP addresses, ports, and endpoints of microservices are explicitly/directly specified in the source code	It is difficult to track the URLs and endpoints and when the ports change the deploying of microservices also needs to change.
No API Gateway	Microservices are exposed and consumers communicate with them directly.	Consumers need to know each microservices in detail and harder to maintain the endpoints of microservices.
Timeouts	The unsuitable time set for sending messages or waiting for a response.	Too short timeout will cause not enough time to handle the request and too long time will waste the time to wait for unavailable microservices.
No API Versioning	No information is available on the microservice version. The microservice should support multi-API versions including the new and the old ones.	Changes to a microservice API will impact all consumers or consumers can not communicate microservice using different API versions.
ESB misuse	Central ESB is used for connecting microservices in an application.	ESB abuse may lead to undesired centralization of business logic and dumb services and coupling of microservices

Table 23 – Communication microservice smells (Ding & Zhang, 2022).

## 4.2 Detection Tools for Microservice Architecture Smells

To aid in the identification of software smells, researchers and practitioners have developed a range of software tools. These tools use automated analysis techniques to scan code and identify potential smells, making it easier for developers to detect and correct issues promptly.

Currently, several technologies, such as Arcan, Designite, and Structure 101, are available for detecting architecture smells. Despite their capability, none of these technologies can accurately detect microservice smells like the ones outlined in Section 4.1.1. As their detection and resolution may call for more specialised tools, these architectural smells provide a challenge for the development community. The creation of tools that can precisely identify and address these microservice architecture smells is thus urgently needed.

In this section, some of the microservice smell detection tools that are currently available will be explored, including their features and capabilities, as well as their future works. The focus will be on a few examples of existing tools, such as MSA-Nose (Walker et al., 2020),  $\mu$ TOSCA (Soldani et al., 2021), and MARS (Tighilt et al., 2023) and discuss how they can be used to improve microservice applications.

### 4.2.1 MSA-Nose

(Walker et al., 2020) build a solution as an open-source tool designed to identify a range of architectural smells. With its ability, this tool is a valuable addition to the arsenal of architecture analysis tools available to developers.

To fully detect architectural smells, MSA-Nose first analyses each microservice before integrating them into a bigger service mesh. Firstly, they make a graph that represents the relationships between the different microservices. This is accomplished via a scanning and matching-based, two-phase analytical technique. During the first phase, MSA-Nose identifies REST endpoints and collects metadata, including the endpoint's HTTP type, route, arguments, and return type. This is done by analysing the microservice's application configuration files to resolve IP addresses and paths, which define the fully qualified URLs for each endpoint. MSA-Nose then lists these endpoints and REST calls based on static code analysis using annotation-based REST API configuration commonly used in enterprise frameworks. In the second phase, MSA-Nose matches each endpoint with each REST call across different microservice modules based on the URL and metadata. URLs are generalized to address different naming of path variables across different microservice modules, and each resultant matching pair indicates inter-microservice communication (Walker et al., 2020).

Afterwards, MSA-Nose analyses the underlying dependency management configuration file for each microservice to find the dependencies and libraries used by each of the applications. Finally, the application configuration is analysed to determine information such as the port for the module, the databases it connects to, and other relevant environment variables for the application. The overall architecture of MSA-Nose is shown in Figure 12. The Resource Service

module takes the path of the source files and extracts metadata from those files. These metadata are then fed into the Entity Service and API Service modules, which produce descriptions of entities and definitions of API endpoints, respectively. The REST Discovery Service module takes the definitions of the API endpoints and resolves inter-microservice communications. Once the processing of each module is done, MSA-Nose begins the process of code-smell detection (Walker et al., 2020).

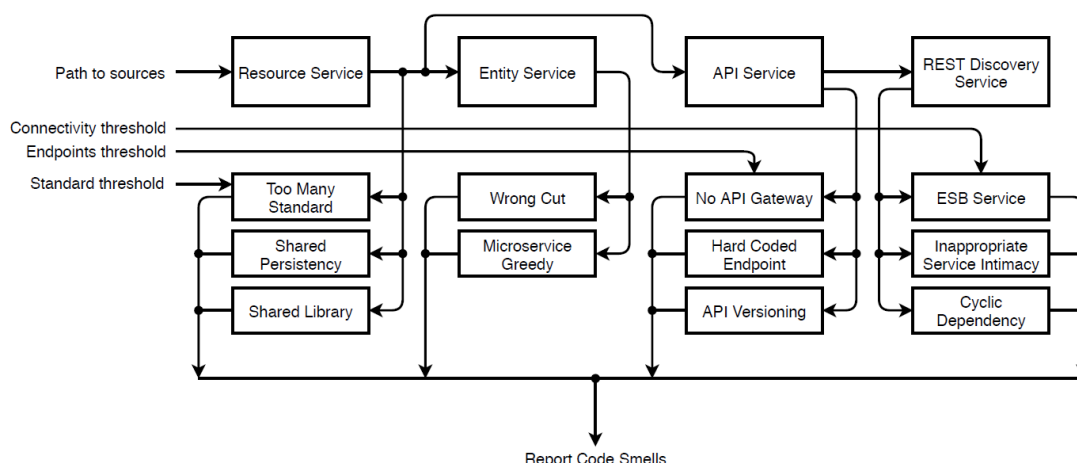


Figure 12 - MSANose architecture diagram (Walker et al., 2020).

#### 4.2.1.1 What and how microservice smells can be detected

For (Walker et al., 2020) paper's purpose, the catalogue proposed by (Taibi & Lenarduzzi, 2018) was used due to its recentness at the time and every microservice smell mentioned on it was used on this tool. These are ESB Usage, Too Many Standards, Wrong Cuts, Not Having an API Gateway, Hard-Coded Endpoints, API Versioning, Microservice Greedy, Shared Persistency, Inappropriate Service Intimacy, Shared Libraries, and Cyclic Dependency.

##### ESB Usage

To determine if an Enterprise Service Bus (ESB) is in use, the process involves counting connections between different system modules. An ESB module is recognized by having a notably high number of connections, a balanced proportion of incoming and outgoing connections, and it should connect to nearly all other modules. This method helps identify the central hub facilitating communication in complex systems (Walker et al., 2020).

##### Too Many Standards

Detecting an excessive use of standards in an application is complex because the definition of "too many" standards varies. Standards, in this context, refer to predefined guidelines or specifications that developers adhere to when designing software components. Developers often choose different standards for system modules based on factors like speed, features, and security. (Walker et al., 2020) keep track of the standards used in each application layer

(presentation, business, and data layer) and allow users to set their own "too many" threshold for each layer to adapt to specific requirements.

### Wrong Cuts

To detect wrong microservice cuts, MSA-Nose looks for an unbalanced distribution of artefacts within microservices across different layers of the application. For presentation microservices, it looks for an abnormally high number of front-end artefacts; for business microservices, it looks for an unbalanced number of service objects; and for data microservices, it looks for an unbalanced number of entity objects. Outliers in the number of these artefacts are identified and reported to the user. An outlier counts greater than two times the standard deviation away from the average count of artefacts in each microservice (as shown in the following equation), MSA-Nose reports the possibility of a wrongly cut microservice to the user (Walker et al., 2020).

$$2 * \sqrt{\frac{\sum_{i=0}^n (x_i - \bar{X})}{n - 1}}$$

### Not Having an API Gateway

If the scanned application has more than 50 distinct modules, MSA-Nose suggests using an API gateway, as it may not be possible to determine the absence of an API gateway from code analysis alone. This is a best practice suggestion and not an error because it is challenging to determine the absence of an API gateway, especially in cloud applications that rely on routing frameworks such as AWS API Gateway<sup>2</sup>, which uses an online configuration console and is not discoverable from code analysis (Walker et al., 2020).

### Shared Persistency

MSA-Nose detects shared persistency in an application by parsing its configuration files and comparing the persistence settings of each submodule to find shared data sources, such as those in a Spring Boot application's YAML file (Walker et al., 2020). A diagram explaining what this microservice smell is can be seen in Figure 13.

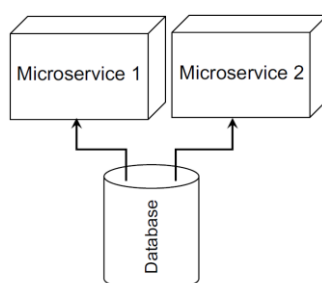


Figure 13 – Shared Persistency (Walker et al., 2020)

<sup>2</sup> <https://aws.amazon.com/api-gateway>

### **Inappropriate Service Intimacy**

Inappropriate service intimacy is when one microservice requests the private data of another. It can be detected by looking for a module directly accessing another's data source in addition to its own, or by looking for two modules with the same entities where one is only modifying or requesting the other's data (Walker et al., 2020).

### **Shared Libraries**

MSA-Nose detects shared libraries by scanning the dependency management files of each module to locate all shared libraries, with a focus on in-house libraries. If necessary, developers can extract them into a separate module to make the application more robust against changes in the libraries (Walker et al., 2020).

### **Cyclic Dependency**

To detect all cycles between modules, MSA-Nose utilizes a modified depth-first search (Tarjan, 1972). First, MSA-Nose extracts the REST communication graph for the microservice mesh. In the graph, each vertex represents a microservice, and each edge represents a REST API call. Then, MSA-Nose runs its cyclic dependency detection algorithm on the graph. MSA-Nose maintains a recursive stack of vertices while traversing the graph. Since the graph is unidirectional (client to server), MSA-Nose marks it as a cycle if a vertex already exists in the stack.

### **Hard-Coded Endpoints**

MSA-Nose detects hard-coded endpoints during the bytecode analysis phase by examining the parameters passed into function calls used to connect to other microservices. For example, in Spring Boot, MSA-Nose looks for any calls from RestTemplate and links the passed address back to any parameters passed to the function or any class fields to find the path parameters used. The system tests for both hardcoded port numbers and IP addresses, which should be avoided for easier scalability of the system in the future.

### **API Versioning**

MSA-Nose locates unversioned APIs in an application by identifying all fully qualified paths, and then matching each path against a regular expression pattern `"/v[0-9]+(?:[0-9]).*"` to detect unversioned paths. Any unversioned APIs are then reported to the user.

### **Microservice Greedy**

MSA-Nose finds superfluous microservices by analysing front-end files, service objects, and entity objects in the application, looking for outliers that could indicate potential greedy modules. They define outliers using the same equation as for wrongly cut microservices but focus only on those that are undersized instead of too large.

#### 4.2.1.2 Future trends

The area of microservice verification is a relatively new field, and there is still much to be explored. While there has been some examination of issues such as security, data constraints, and networking, there is still much work to be done in terms of verification. The pool of code smells for microservice-based applications has yet to be fully developed. However, this work shows that established code smells from industry advice and examination can be adapted for microservice-based applications. This can be achieved through an extensive survey among industry specialists, and the creation of a taxonomy of code smells exclusively for MSA.

The implementation of MSA-Nose, as described by (Walker et al., 2020), has a clear separation between metadata extraction and code-smell detection, making it easy to add new detection mechanisms without affecting the existing algorithms. This research could be also expanded into other languages and enterprise standards. Exploration for containerized microservices and rigorous deployment configuration analysis can be done for cloud-native applications.

The authors also propose that MSA-Nose can be integrated into the software development lifecycle, such as being added to the CI/CD pipeline to run an automatic screening test before performing the deployment. This can accelerate the code review process and reduce manual efforts and human errors of code reviewers and DevOps engineers, resulting in a shortened release and update cycle of microservice applications along with improved code quality.

#### 4.2.2 $\mu$ TOSCA toolchain

(Soldani et al., 2021) present a methodology for identifying and resolving architectural smells in microservice-based architectures. The authors build on a previous industry-oriented review and identify a set of architectural smells that could violate key design principles of microservices, along with corresponding architectural refactoring (Neri et al., 2020).

In this work, the authors propose using the Topology and Orchestration Specification for Cloud Applications (TOSCA) and introduce  $\mu$ TOSCA, a type system to specify microservice-based architectures as typed topology graphs. They formally define the conditions to identify the occurrence of the identified architectural smells and illustrate how to refactor the architecture to resolve them.

In the same work, they present  $\mu$ Freshener, a prototype tool that enables editing  $\mu$ TOSCA topology graphs and implements their methodology. However, manually representing the architecture of a complex microservice-based application in  $\mu$ TOSCA can be time-consuming and error-prone, so the authors propose a technique for automatically mining the architecture of a "black-box" microservice-based application. They present  $\mu$ Miner, a prototype tool that implements this technique to automatically derive a  $\mu$ TOSCA topology graph modelling the architecture of a microservice-based application starting from its deployment in Kubernetes.

The  $\mu$ TOSCA toolchain represented in Figure 14, consisting of  $\mu$ Miner and  $\mu$ Freshener, is a powerful combination for designing and analysing microservice-based applications. By utilizing the Kubernetes deployment of a microservice-based application,  $\mu$ Miner can automatically generate a  $\mu$ TOSCA file that describes the application's architecture. This file can then be fed into  $\mu$ Freshener, which uses automated analysis techniques to identify any architectural smells that may be present in the application's design. If any smells are detected,  $\mu$ Freshener provides suggestions for potential architectural refactorings that can be used to eliminate them. The  $\mu$ TOSCA toolchain enables architects to obtain "smell-free"  $\mu$ TOSCA specifications, improving the quality and maintainability of microservice-based applications.

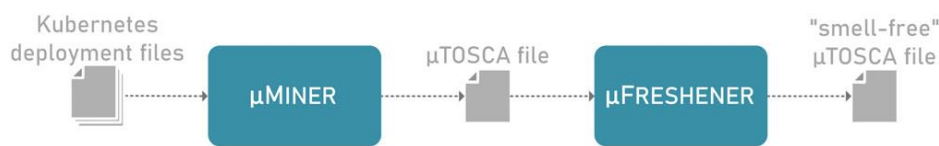


Figure 14 - The  $\mu$ TOSCA toolchain (Soldani et al., 2021).

#### 4.2.2.1 TOSCA

TOSCA is “an OASIS open standard that defines the interoperable description of services and applications hosted on the cloud and elsewhere; including their components, relationships, dependencies, requirements, and capabilities, thereby enabling portability and automated management across cloud providers regardless of underlying platform or infrastructure; thus, expanding customer choice, improving reliability, and reducing cost and time-to-value” (OASIS, n.d.).

The characteristics of TOSCA make them extremely portable and well-suited for DevOps environments by enabling the seamless, continuous delivery of applications across their full lifecycle. This results in greater agility and accuracy for businesses operating in the cloud, as they can easily match service and application requirements with the capabilities of cloud service providers. Automation of this process through TOSCA enables companies to take advantage of specialised expertise and promotes a competitive ecosystem for cloud platforms and service providers, allowing them to develop and better meet the demands of cloud-based companies. In summary, TOSCA helps companies overcome commoditization and maintain their leadership positions in the continually changing cloud world.

In Figure 15 a simple example of TOSCA is provided.

```

tosca_definitions_version: toska_simple_yaml_1_3tosca_simple_yaml_1_3

description: Template for deploying a single server with predefined properties.

topology_template:
  node_templates:
    db_server:
      type: toska.nodes.Compute
      capabilities:
        # Host container properties
        host:
          properties:
            num_cpus: 1
            disk_size: 10 GB
            mem_size: 4096 MB
        # Guest Operating System properties
        os:
          properties:
            # host Operating System image properties
            architecture: x86_64
            type: linux
            distribution: rhel
            version: 6.5

```

Figure 15 – TOSCA simple “Hello World” (OASIS, n.d.).

#### 4.2.2.2 What and how can microservice smells be detected

As mentioned previously, (Soldani et al., 2021) singled out the most recognized architectural smells violating key principles of microservices and the architectural refactorings enabling to resolution of the occurrence of such smells. Out of all the microservice smells that were collected, only four were selected by the researchers. These four were specifically chosen as they contradict three significant design principles, namely horizontal scalability, failure isolation, and decentralization.

The four microservice smells that can be identified and represented using  $\mu$ Freshener, along with the respective key design principle they go against, are:

- No API Gateway (horizontal scalability)
- Endpoint-based Service Interaction (horizontal scalability)
- Wobbly Service Interaction (isolation of failures)
- Shared Persistence (decentralisation)

For a better understanding of how this tool can detect the smells, the authors elaborated one visual representation of how the smells will be detected and how they will be solved. This visual representation can be seen in Figure 16.



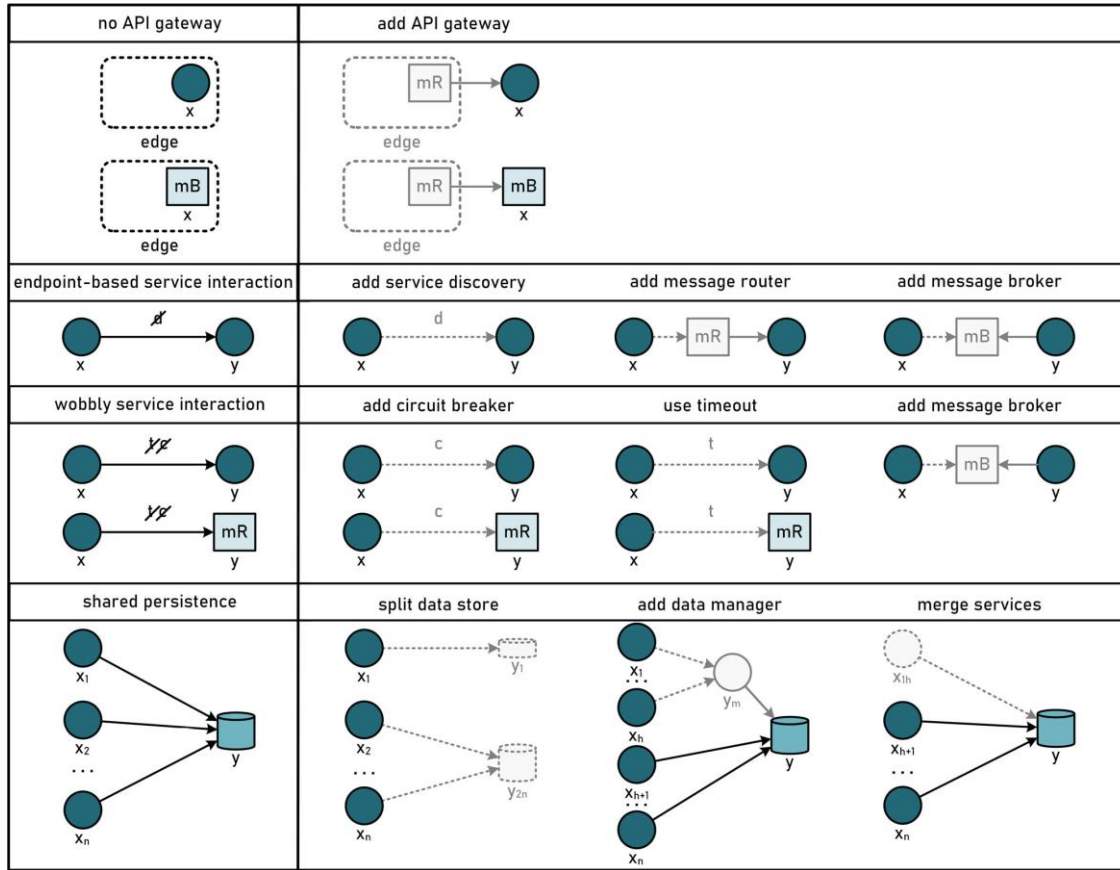


Figure 16 – Visual representation of the architectural smells and refactorings (Soldani et al., 2021).

### No API Gateway

The no API gateway smell in a microservice-based application occurs when external clients directly access internal components without passing through an API gateway, which violates the horizontal scalability of microservices. When a component is scaled out by adding replicated instances, external clients may continue to invoke the original instance and ignore the newly added replicas. To check for this smell in a microservice-based application modelled by a  $\mu$ TOSCA topology graph, it is necessary to verify whether any edges in the architecture do not contain a message router (Soldani et al., 2021).

Figure 16 visually depicts occurrences of the no API gateway smell in a  $\mu$ TOSCA topology graph when a component (either a service or an asynchronous message broker) is placed at the edge of the architecture. Also, it shows two architectural refactorings that can solve this issue, which involve introducing a message router acting as an API gateway or reusing an existing one in the application. These refactorings prevent the component from being directly accessed from outside the application (Soldani et al., 2021).

### **Endpoint-based service interaction**

An endpoint-based service interaction smell occurs in a microservice-based application when a service directly calls another service without using a message router or dynamically discovering the actual endpoint of the service being called. This smell also violates the horizontal scalability of microservices because new instances of the called service cannot be reached by the invoker. This can happen when the location of the instance of the invoked service is hardcoded in the source code of the invoker (Soldani et al., 2021).

Figure 16 shows how to address endpoint-based service interaction smell in a  $\mu$ TOSCA topology graph of a microservice-based app by introducing an intermediate integration pattern like a message router or a service discovery mechanism. The refactoring aims to decouple the interaction between the invoking service and the invoked service. It's important to update the outgoing interaction and reuse an existing message router/broker if available (Soldani et al., 2021).

### **Wobbly service interaction**

In a service interaction, when a failure in the invoked service can cause a failure in the invoker and start a chain reaction of failures, the interaction is considered "wobbly". This occurs when the invoker consumes the functionality of the invoked service without handling potential failures through mechanisms such as circuit breakers or timeouts (Soldani et al., 2021).

Figure 16 visually depicts the occurrences of wobbly service interactions in  $\mu$ TOSCA topology graphs, where one service invokes another without any failure-handling mechanisms like circuit breakers or timeouts. Architectural refactoring to resolve this issue is also shown, including using circuit breakers or timeouts, replacing the interaction with an asynchronous message broker, and decoupling interactions between services. These refactorings can avoid failures or prevent services from getting stuck waiting for a response, and can also resolve endpoint-based service interaction smells if present (Soldani et al., 2021).

### **Shared Persistence**

A shared persistence smell affects a microservice-based architecture when multiple services interact with the same database, directly or through intermediate message routers.

Figure 16 shows the shared persistence smell in a microservice architecture where multiple services interact with the same database. Three architectural refactorings are shown to reduce the number of services accessing the database. These refactorings are diverse and apply to different situations depending on the services accessing the database. The solutions include splitting the database, using a data manager to proxy the access, or merging the services into one.

#### 4.2.2.3 Future works

Regarding architecture smells, (Soldani et al., 2021) plan to expand their capabilities to detect and resolve more architectural smells, including those identified by industry-driven reviews and other researchers. This can be achieved by extending the existing  $\mu$ TOSCA types to model additional entities, detecting new smells based on these entities, and adapting  $\mu$ Freshener accordingly. For instance,  $\mu$ TOSCA has already added a type for grouping nodes to represent team assignments and plans to formalize team-related architectural smells described in industry-oriented reviews and extend  $\mu$ Freshener to address these smells.

### 4.2.3 MARS

(Tighilt et al., 2023) present MARS, a tool-based approach designed for the specification and detection of microservice antipatterns, described in Figure 17. The approach relies on a comprehensive metamodel that encompasses the essential data required for specifying and applying detection rules to the source code of microservice-based systems (Tighilt et al., 2023).

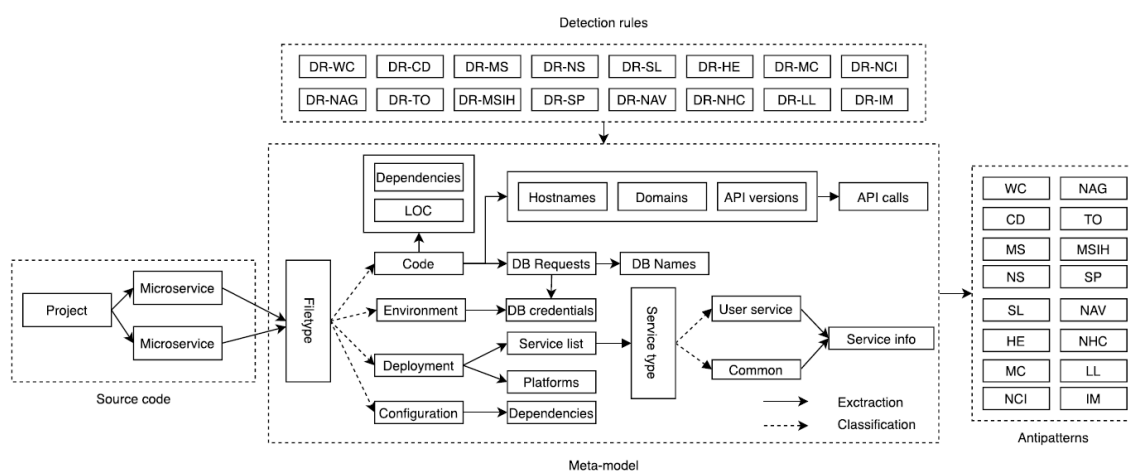


Figure 17 - Microservice Antipatterns Research Software (Tighilt et al., 2023).

(Tighilt et al., 2023) present significant novel contributions to the field of microservice antipattern detection. Originally, MARS, a highly automated tool, is introduced, equipped with a novel metamodel specifically designed for detecting 16 distinct microservice antipatterns. This collection of antipatterns was curated through a multifaceted approach that draws upon the outcomes of a comprehensive and diverse literature review, coupled with a meticulous manual analysis of 64 microservice-based systems, conducted in previous research (Tighilt et al., 2020).

#### 4.2.3.1 What and How Microservice Smells Can Be Detected

Utilizing the MARS toolset, 16 specific antipatterns were meticulously specified and their occurrences were successfully detected within a dataset comprising 24 microservice-based

systems. Subsequently, a manual validation process was employed to assess the precision and recall of the detected instances (Tighilt et al., 2023). In the next subsections will be presented an explanation of the microservice antipatterns detected as well as the detection rules with a textual and pseudo-code description.

### Wrong Cuts (WC)

Microservices are organised around technical layers (business, presentation, and data) instead of functional capabilities, which causes strong coupling among microservices and impedes the delivery of new business functions (Tighilt et al., 2023).

In microservices, a single file type is utilized within the source code, exemplified by a microservice consisting exclusively of presentation code interfacing with another microservice dedicated solely to business logic, and the identification of this antipattern is based on file extensions, content, and programming languages (Tighilt et al., 2023). In Figure 18 it is possible to check the pseudo-code description of this antipattern.

```
frontend_languages: a list that contains frontend
extension languages
threshold_frontend_files: a threshold for the
allowed percentage of frontend_languages
in a microservice (80% in our study)
1 def WrongCut(Microservice MicroS):
2     exist = false
3     cpt = 0
4     for extension in frontend_languages :
5         for file in MicroS.Code.source_files:
6             if extension in file:
7                 cpt += 1
8     if cpt > threshold_files *
        MicroS.Code.source_files.size:
9         exist = true
10    return exist
```

Figure 18 – Wrong Cut pseudo-code description (Tighilt et al., 2023).

### Cyclic Dependencies (CD)

Multiple microservices are circularly co-dependent and thus no longer independent, which goes against the very definition of microservices (Tighilt et al., 2023).

The authors employ the call graph of the microservice-based system, which is analysed to identify circular dependencies among microservices (Tighilt et al., 2023). In Figure 19 it is possible to check the pseudo-code description of this antipattern.

isConnectedTo(): a function that verifies the existence of a direct dependency between two microservices.

```
1 def CyclicDependencies( Microservice MicroSA,
                           Microservice MicroSB ):
2   return isConnectedTo( MicroSA, MicroSB )
   AND isConnectedTo( MicroSB, MicroSA )
```

Figure 19 – Cyclic Dependencies pseudo-code description (Tighilt et al., 2023).

### Mega Service (MS)

A microservice provides multiple business functions. A microservice should be manageable by a single team and should pertain to a single business function (Tighilt et al., 2023).

A mega service, distinguished by its support for multiple business functionalities and potential size, is compared to microservices lacking this antipattern. The identification of mega services by MARS involves assessing both the lines of code and the number of files within a microservice, with criteria established by an expert specifying certain threshold values (Tighilt et al., 2023). In Figure 20 it is possible to check the pseudo-code description of this antipattern.

```
1 def MegaService( Microservice microS ):
2   exist = false
3   if LOCs( microS.Code ) > threshold_LOCs
4     exist = true
5   return exist
```

Figure 20 – Mega Service pseudo-code description (Tighilt et al., 2023).

### Nano Service (NS)

Results from a fine-grained decomposition of a system, i.e., when one business function requires many microservices to work together (Tighilt et al., 2023).

A nano service, an excessively fine-grained microservice offering only a fragment of a business function within a microservice-based system, often arises from an overly detailed system decomposition. MARS identifies nano services by examining the microservice's lines of code and file count, which should not surpass predetermined expert-defined thresholds (Tighilt et al., 2023). In Figure 21 it is possible to check the pseudo-code description of this antipattern.

```
1 def NanoService( Microservice MicroS ):
2   exist = false
3   if LOCs( MicroS.Code ) < threshold_LOCs AND
4     NumFiles( MicroS.Code ) < threshold_files :
5     exist = true
6   return exist
```

Figure 21 – Nano Service pseudo-code description (Tighilt et al., 2023).

## Shared Libraries (SL)

This relates to the sharing of libraries and files (e.g., binaries) by multiple microservices, which breaks their independence as they rely on a single source to fulfil their business function (Tighilt et al., 2023).

Source files, libraries, or other artefacts from one microservice are shared and utilized by other microservices (Tighilt et al., 2023). In Figure 22 it is possible to check the pseudo-code description of this antipattern.

```
1 def SharedLibs(Microservice[] microservices):
2     shared_libs = []
3     libs = []
4     for each ms in microservices:
5         for each dep in ms.dependencies:
6             if libs.contains(dep) AND
7                 NOT shared_libs.contains(dep):
8                 shared_libs.add(dep)
9             else:
10                libs.add(dep)
11    return shared_libs.length > 1
```

Figure 22 – Shared Libraries pseudo-code description (Tighilt et al., 2023).

## Hardcoded Endpoints (HE)

URLs, IP addresses, ports, and other endpoints are hardcoded in the source code of microservices and/or configuration files, which interferes with load balancing and deployment (Tighilt et al., 2023).

Within certain source code, deployment files, configuration files, or environment files, REST API calls are found to contain statically defined IP addresses, port numbers, and/or URLs, with potential instances of hard-coded endpoints even in the absence of a discovery service (Tighilt et al., 2023). In Figure 23 it is possible to check the pseudo-code description of this antipattern.

```
service_discovery_tools: a list that contains
names of existing service discovery tools

1 def Hard-codedEndpoints(System aSystem):
2     if !intersect(aSystem.dependencies,
3                 service_discovery_tools):
4         potential_hard-coded = true
5     if potential_hard-coded:
6         for each ms in aSystem.microservices:
7             if has_urls(ms.Configuration)
8                 OR has_urls(ms.Code)
9                 OR has_urls(ms.Environment)
10                OR has_urls(ms.Deployment):
11                list_urls.append(ms.Code.HTTP)
12    return list_urls
```

Figure 23 – Hardcoded Endpoints pseudo-code description (Tighilt et al., 2023).

## Manual Configuration (MC)

Refers to configurations that must be manually pushed in some microservices and, since microservice-based systems evolve rapidly, their management should be automated, including their configuration (Tighilt et al., 2023).

Each microservice possesses individual configuration files, with no microservice taking on the role of configuration management, and the system's dependencies do not include any configuration management tools (Tighilt et al., 2023). In Figure 24 it is possible to check the pseudo-code description of this antipattern.

```
defined_config_libs: a list that contains names  
of existing service configuration libraries  
  
1 def ManualConfiguration(System aSystem):  
2     exist = false  
3     for each cl in defined_config_libs:  
4         if NOT aSystem.dependencies.contain(cl)  
            AND length(aSystem.Configuration) > 0:  
5             exist = true  
6         for each ms in aSystem.microServices:  
7             if NOT ms.dependencies.contain(cl)  
                AND length(ms.Configuration) > 0:  
8                 exist = true  
9     return exist
```

Figure 24 – Manual Configuration pseudo-code description (Tighilt et al., 2023).

## No Continuous Integration/Continuous Delivery (NCI)

Not using CI/CD, which is important for microservices to automate repetitive steps during testing and deployment, undermines the microservice architectural style, which encourages automation wherever possible (Tighilt et al., 2023).

The absence of continuous integration/delivery-related data in configuration files and version control repositories is noted, and the analysis is based on an adaptable roster of CI/CD tools (Tighilt et al., 2023). In Figure 25 it is possible to check the pseudo-code description of this antipattern.

```

defined_ci_libs: a list of names of
existing CI/CD tools
defined_ci_folders: a list of names of
existing CI/CD configuration folders
(e.g., .circleci, .travisci)

1 def NoCICD(System aSystem):
2     exist = true
3     for each ms in aSystem.microservices:
4         if intersect(ms.dependencies,
                        defined_ci_libs):
5             exist = false
6     if exist:
7         if Regex_match(defined_ci_folders,
                        aSystem.GitRepository):
8             exist = false
9     return exist

```

Figure 25 - No Continuous Integration/Continuous Delivery pseudo-code description (Tighilt et al., 2023)

### No API Gateway (NAG)

Consumer applications communicate directly with microservices and must know how the whole system is decomposed, managing endpoints and URLs for each microservice (Tighilt et al., 2023).

The absence of common API gateway implementation signatures is observed in the source code, and there are no frameworks or tools about API gateways within the microservices' dependencies (Tighilt et al., 2023). In Figure 26 it is possible to check the pseudo-code description of this antipattern.

```

api_gateway_libs: a list of names of API gateways

1 def NoApiGateway(System aSystem):
2     exist = true
3     for each agl in api_gateway_libs:
4         if aSystem.dependencies.contain(agl):
5             exist = false
6     for each ms in aSystem.microServices:
7         if ms.dependencies.contain(agl):
8             exist = false
9     return exist

```

Figure 26 – No API Gateway pseudo-code description (Tighilt et al., 2023).

### Timeouts (TO)

Timeout values are set and hardcoded in HTTP requests, which leads to unnecessary disconnections or delays (Tighilt et al., 2023).

Timeout values are included in REST API calls, while there are no indicators of common circuit breaker implementations in the source code, and the microservices' dependencies do not



include any circuit breakers (Tighilt et al., 2023). In Figure 27 it is possible to check the pseudo-code description of this antipattern.

```
list_circuit_breakers: a list that contains circuit
breakers libraries

1 def Timeouts ( Microservice MicroS):
2   return (NOT MicroS.dependencies
           .contain( list_circuit_breakers )
           AND intersect( Fallback , MicroS.Code))
           OR intersect(Timeout , MicroS.Code)
```

Figure 27 – Timeouts pseudo-code description (Tighilt et al., 2023).

### Multiple Service Instances Per Host (MSIPH)

Multiple microservices are deployed on a single host, which prevents their independent scaling and may cause technological conflicts inside the host (Tighilt et al., 2023).

The utilization of deployment technologies, such as Docker Compose, is absent in the system's configuration. Instead, a single deployment file within the source code is responsible for deploying the entire system (Tighilt et al., 2023). In Figure 28 it is possible to check the pseudo-code description of this antipattern.

```
1 def MultipleServiceInstancePerHost( System aSystem):
2   no_docker_file = 0
3   system_has_docker = false
4   if ( conf_file in aSystem.config_files )
5     .contain( docker-compose.yml):
6     system_has_docker=true
7   for each ms in aSystem.microservices:
8     if length(ms.Deployment.docker_files)<1:
9       no_docker_file+=1
9   return NOT system_has_docker AND
           no_docker_file >=
           length(aSystem.microservices)
```

Figure 28 – Multiple Service Instances Per Host pseudo-code description (Tighilt et al., 2023).

### Shared Persistence (SP)

Multiple microservices share a single database meaning that they no longer own their data and cannot use the most suitable database technology for their business function (Tighilt et al., 2023).

Data-source URLs are shared among microservices, resulting in the creation of a single database that multiple microservices access within the system (Tighilt et al., 2023). In Figure 29 it is possible to check the pseudo-code description of this antipattern.

```

1 def SharedPersistence(Microservice[] microservices):
2     shared_databases = []
3     databases = []
4     for each ms in microservices:
5         for each db in ms.Code.DataBase:
6             if data_bases.contain(db) AND
               NOT shared_data_bases.contain(db):
7                 shared_data_bases.add(db)
8             else:
9                 data_bases.add(db)
10    return length(shared_data_bases) > 1

```

Figure 29 – Shared Persistence pseudo-code description (Tighilt et al., 2023).

### No API Versioning (NAV)

Happens when no information is available about a microservice version, which can break changes and force backward compatibility when deploying updates (Tighilt et al., 2023).

Endpoints and URLs do not contain version numbers and no version information is present in the configuration files (Tighilt et al., 2023). In Figure 30 it is possible to check the pseudo-code description of this antipattern.

```

1 def HasNoApiVersioning(System aSystem):
2     no_api_versioning = 0
3     has_api_versioning = false
4     for each ms in aSystem.microservices:
5         if NOT ms.Configuration
               .contain('apiVersion'):
6             no_api_versioning +=1
7     for each file in aSystem.config_files:
8         if file.contain('apiVersion'):
9             has_api_versioning = true
10    return NOT has_api_versioning AND
               no_api_versioning
               >= length(aSystem.microservices)

```

Figure 30 – No API Versioning pseudo-code description (Tighilt et al., 2023).

### No Health Check (NHC)

This relates to microservices that lack regular health checks, which can result in undetected unavailability, potentially causing timeouts and other errors (Tighilt et al., 2023).

No “health check” or “health” endpoint exists and no common implementation of health checks is used (Tighilt et al., 2023). In Figure 31 it is possible to check the pseudo-code description of this antipattern.

health\_check\_tools: a **list** of health-check libraries

```
1 def HasNoHealthCheck(System aSystem):
2     no_health_check=true
3     number_ms_without_hc = 0
4     for each dp in aSystem.dependencies:
5         if health_check_tools
            .contain(dp):
6             no_health_check=False
7     for each ms in aSystem.microservices:
8         for each dp in ms.dependencies:
9             if NOT health_check_tools
                .contain(dp):
10                 number_ms_without_hc += 1
11 return no_health_check AND
    length(number_ms_without_hc) > 0
```

Figure 31 – No Health Check pseudo-code description (Tighilt et al., 2023).

### Local Logging (LL)

Results from microservices have their logging mechanism, which prevents aggregation and analyses of their logs and the monitoring and recovery of systems (Tighilt et al., 2023).

The author detects this antipattern by examining whether (1) there is a lack of distributed logging in the dependencies, and/or (2) there is no common logging microservice, with each microservice maintaining its log file paths (Tighilt et al., 2023). In Figure 32 it is possible to check the pseudo-code description of this antipattern.

list\_logging\_libs: **list** of logging libraries

```
1 def LocalLogging (System aSystem):
2     exist = true
3     for each ll in list_logging_libs:
4         if aSystem.dependencies.contain(ll):
5             exist = false
6     for each ms in aSystem.microServices:
7         if ms.dependencies.contain(ll):
8             exit = false
9     return exit
```

Figure 32 – Local Logging pseudo-code description (Tighilt et al., 2023).

### Insufficient Monitoring (IM)

Describes neglecting to record data on performance levels and failures of microservice-based systems that would be useful for maintenance purposes (Tighilt et al., 2023).

The author detects this antipattern by searching for a monitoring framework or library, such as Prometheus, in the microservice dependencies (Tighilt et al., 2023). In Figure 33 it is possible to check the pseudo-code description of this antipattern.

```

list_monitor_libs: list of monitoring libraries

1 def InsufficientMonitoring ( System aSystem ):
2   exist = true
3   for each ml in list_monitor_libs:
4     if aSystem.dependencies.contain(ml):
5       exist = false
6     for each ms in aSystem.microServices:
7       if ms.dependencies.contain(ml):
8         exist = false
9   return exist

```

Figure 33 – Insufficient Monitoring pseudo-code description (Tighilt et al., 2023).

#### 4.2.3.2 Results and Comparisons

After applying MARS on 24 microservice-based systems written in Java, the results obtained demonstrate that MARS enables the specification and detection of microservice antipatterns with an impressive average precision rate of 82% and a commendable recall rate of 89%. This outcome underscores the potential of this highly automated approach, which was further substantiated by its application in a large-scale study. Specifically, MARS accurately identified prevalent antipatterns such as Shared Libraries, Multiple Service Instances Per Host, and Cyclic Dependencies, showcasing its effectiveness in detecting common issues within microservice architectures. The tool also achieved impressive precision and recall scores in identifying issues related to Wrong Cuts, Manual Configurations, No CI/CD, No API gateways, Timeouts, and Shared Persistence antipatterns. (Tighilt et al., 2023) points out that while these results are encouraging, it's worth noting that MARS generated a higher number of false positives when detecting the remaining seven antipatterns.

In this document, (Tighilt et al., 2023) ran a comparison with MSA-Nose, a tool that was already covered in the previous sections that also focuses on Java-based microservice systems and encompasses 11 antipatterns, eight of which overlap with MARS's detection scope. The comparison was limited to these common antipatterns, with an attempt to replicate MSA-Nose's results using the Ticket-Train system from the dataset. MSA-Nose primarily identified Shared Libraries while missing occurrences of the No API Versioning antipattern. For Shared Libraries, the tool exhibited an average detection precision of 1.5% and a recall of 100%, attributed to its reliance on comparing library names without accounting for duplicated local libraries in microservice repositories. Regarding No API Versioning, the tool achieved an average detection precision of 57% and a recall of 47%, primarily due to its examination of microservice files rather than the system's complete configuration files (Tighilt et al., 2023).

(Tighilt et al., 2023) concludes that MARS significantly outperforms MSA-Nose in microservice antipattern detection, excelling not only in shared libraries and no API versioning but also covering a broader range of microservice antipatterns.

The authors anticipate that this research will lay the groundwork for future practical and research applications, ultimately contributing to the enhancement of microservices' design and implementation.

#### 4.2.3.3 Future works

(Tighilt et al., 2023) intends to enhance the detection capabilities for specific antipatterns, notably circular dependencies, while also extending the analysis to identify new antipatterns and assess their prevalence in a wider array of established microservice-based systems. There is a plan to conduct empirical and quantitative investigations into the prevalence of microservice antipatterns within a larger dataset, with a focus on examining their repercussions on maintenance practices. These efforts are poised to furnish valuable insights for developers and researchers, offering guidance on best practices and cautionary considerations when developing microservice-based systems.

## 5 Solution

In this chapter, the intention is to provide an in-depth description of the process involved in enriching a microservice architecture smell catalogue with additional microservice architectural smells. The focus will be on elucidating the steps and methodologies employed to expand the repository of architectural observations, thereby enhancing its comprehensiveness and utility.

In addition, this chapter describes the developments made to improve the MSA-Nose tool (Walker et al., 2020).

### 5.1 Proposed Catalogue

#### 5.1.1 Analysis

As detailed in Section 5.2, the process of gathering and analysing documents related to microservice architecture smells and anti-patterns was conducted during the screening, analysis, and reporting phases. This section now aims to assess the remaining unexplored aspects that warrant attention in updating the microservice smells catalogue proposed by (Taibi et al., 2019), which aligns closely with the proposed update.

Moreover, the intention is to leverage the open responses from users to augment and refine this update further. By considering both unexamined areas and user input, a more comprehensive and improved microservice smells catalogue can be developed.

### **5.1.2 Catalogue Improvements**

Following the thorough analysis conducted in Section 4.1 concerning the catalogue by (Taibi et al., 2019) and the subsequent analysis of documents in Section 5.2, opportunities for enhancement have become evident. After the final analysis is done the updates to the catalogue will be shown throughout this section, which means that only the smells to add to the (Taibi et al., 2019) catalogue will be shown.

One improvement is the inclusion of security-related smells within the realm of microservices, as described in the study by (Ponce et al., 2022). Integrating these security-related smells is deemed valuable for fortifying the integrity of the microservice system. The smells that were selected will be presented below.

#### **Insufficient Access Control**

This smell arises when access control measures are lacking within one or more microservices. This deficiency can potentially compromise the confidentiality of data and business functions in those microservices. The presence of this smell can expose microservices to security vulnerabilities, such as the "confused deputy problem," where attackers can manipulate services to access unauthorized data (Ponce et al., 2022).

In the context of microservices, traditional identity control models are inadequate. They require client details and permissions to be dynamically validated with each request. Microservices necessitate an automated decision-making process for permitting or denying calls between services. In addition, development teams must manage user identities without introducing excessive latency or contention through frequent interactions with a centralized service (Ponce et al., 2022).

This smell was also reported in the study done by (Waseem et al., 2021) and by (Söylemez et al., 2022).

#### **Unnecessary Privileges to Microservices**

The "Unnecessary Privileges to Microservices" smell occurs when microservices are granted access levels, permissions, or functionalities that exceed what is required for their business functions (Ponce et al., 2022).

This situation arises when a microservice is granted access to databases or message queues, even when these resources are not essential for the microservice's intended business function. Consequently, unnecessary exposure of resources increases the attack surface, posing risks to confidentiality and integrity. In the event of an intruder gaining control of a service, they can potentially access and modify all data and messages accessible to that service (Ponce et al., 2022).

### **Using Own Crypto Code**

Using proprietary encryption solutions and algorithms poses significant risks to the confidentiality, integrity, and authenticity of data in software applications. Unless extensively tested, these custom encryption solutions can lead to security vulnerabilities. When development teams implement their encryption solutions, they may inadvertently introduce inadequate security measures, potentially resulting in confidentiality, integrity, and authenticity issues (Ponce et al., 2022).

Interestingly, the usage of Own Crypto Code can be more detrimental than having no encryption solution at all. This is because it can create a false sense of security, leading organizations to believe their data is adequately protected when it remains vulnerable to threats (Ponce et al., 2022).

This smell was also pointed out as an issue in the study done by (Waseem et al., 2021), as developers do not use encryption/decryption tools properly.

### **Non-secured service-to-service communications**

This smell occurs when two microservices in an application interact without establishing a secure communication channel, even if they reside within the same network. Given the highly distributed nature of microservice-based applications, the proliferation of communication interfaces and channels increases the overall attack surface of the application. Each exposed API and communication channel represents a potential attack vector that malicious intruders could exploit (Ponce et al., 2022).

Microservices frequently rely on intercommunication to perform their business functions, and if these channels lack security measures, the transmitted data becomes susceptible to man-in-the-middle, eavesdropping, and tampering attacks. This vulnerability not only jeopardizes the confidentiality of service-to-service communications but also compromises their integrity and authenticity. Intruders could intercept and manipulate data in transit, potentially leading to security breaches (Ponce et al., 2022).

### **Multiple User Authentication**

This smell manifests when a microservice-based application offers multiple access points for user authentication. Each of these access points represents a potential vulnerability that could be exploited by an intruder to gain unauthorized access as an end-user. This approach increases the attack surface and poses a risk to the authenticity of the microservice-based application.

Utilizing multiple access points for user authentication introduces challenges in terms of maintainability and usability. The need to develop, maintain, and utilize user login functionality in various parts of the application can lead to complexity and potential usability issues.



(Zhong et al., 2022) also mentioned that the fact of repeating development means a lack of consideration for modularity, thus reflecting poor architectural practices.

(Ponce et al., 2022) identified additional microservice smells, some of which propose solutions involving the adoption of essential microservice development best practices. One of these smells aligns with an existing microservice architectural smell related to the existence of an API Gateway, exemplified by the "Publicly Accessible Microservices" smell.

Another example is the "Hardcoded Secrets" smell, also highlighted by (Ponce et al., 2022). Detecting such smells can be facilitated by implementing or utilizing CI/CD tools like SonarQube. Notably, this approach corresponds to an already recognized microservice issue included in the catalogue proposed by (Taibi et al., 2019).

Regarding the industrial inquiry conducted by (Zhou et al., 2023) developers and architects from various organizational domains reported challenges in developing with microservice architecture. These challenges were categorized into pairs of practices and associated pains, as outlined in the initial analysis presented in Section 3.2.6.2.

One of the smells that is raised from pain 3 (Complexity of API Management) is **Inconsistent API Management and Understanding**. This smell encompasses challenges such as the difficulty in ensuring that APIs adhere to their contracts, repetitive implementation of interfaces, different teams having varying levels of understanding regarding APIs, difficulties in identifying and resolving problems due to independent service releases, a lack of effective methods for maintaining consistent API understanding, and the adoption of complex internal regulations, including naming rules, and manual verification to enforce API development standards (Zhou et al., 2023).

It's worth noting that while this study has the potential to uncover additional smells, some of them have already been defined. For instance, pains akin to "Excessive Technology Diversity" (corresponding to the "Focus on Latest Technologies" smell) and "Unsatisfying Monitoring and Logging" (related to "Lack of Monitoring") were previously identified.

Nonetheless, certain pains may arise when applying the prescribed patterns intended to address smells in the catalogue provided by (Taibi et al., 2019). These specific pains encompass "Inadequate Automation" and "Chaotic Independence" (arising from the application of "No DevOps tools"), "Data Inconsistency" (stemming from the application of "Shared Persistence"), and "Ad-hoc organizational transformation" (like the disadvantages of having the smell "Legacy Organization").

## 5.2 MSA Nose Improvement

In this section, it is intended to describe the improvements made to the MSA-Nose tool developed by (Walker et al., 2020).

Different repositories were created to fit and separate all the developments. The developments done on the MSA-Nose tool can be found on GitHub (<https://github.com/JSamoes/msa-nose>) and this contains all the extensive work done in this research project. This repository provides as a comprehensive resource, carefully documenting how to use the project. Not only does it provide the codes and implementations, but it also includes thorough documentation.

The extension and UI can be found also on GitHub (<https://github.com/JSamoes/msa-nose-extension>).

### 5.2.1 Analysis

In this section, all the requirements for the final solution will be analysed so that this work can contribute to an improvement of the MSA-Nose (Walker et al., 2020) tool. Before any update, a representation of the components of this tool can be found in Figure 34.

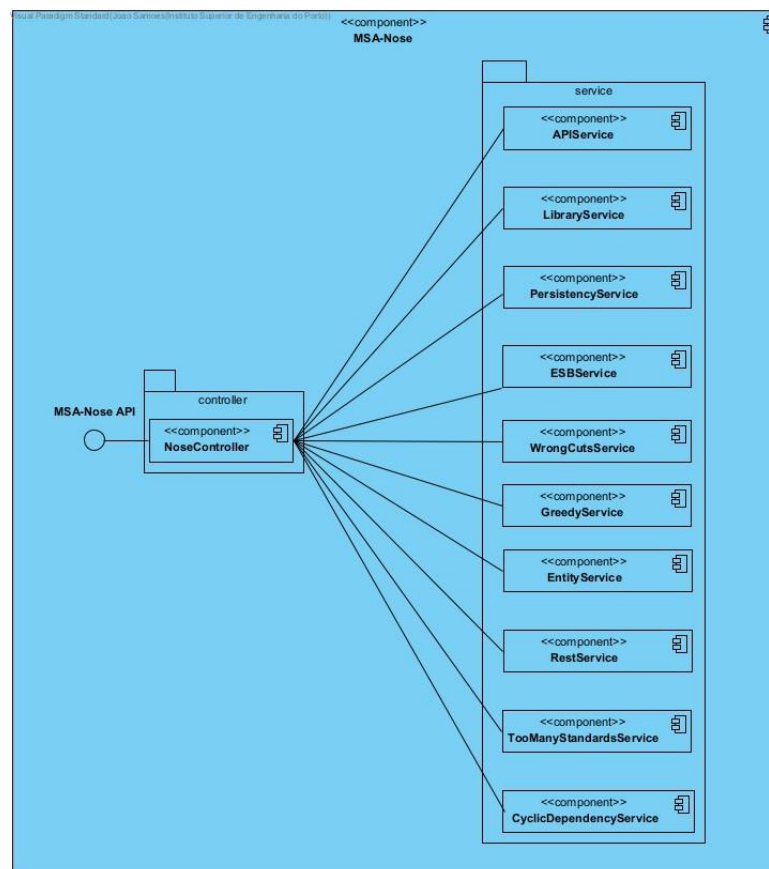


Figure 34 – Components Diagram.

In Section 4.2, the analysis encompassed various microservice smell detection tools. After this evaluation, it was determined that MSA-Nose stood out as the tool most relevant to the objectives of this document, primarily due to its static analysis capabilities, as it can help identify issues before runtime. The fact that MSA-Nose is open-source, developed in Java, and designed for use with Java microservices also factored into its choice. Furthermore, it aligns with one of the original catalogues of microservice smells (also discussed in Section 4.1). Given that this catalogue has been updated by other authors, integrating these updates into the tool represents a significant improvement.

This enhancement aims to elevate the overall experience of developers working with microservice architecture. The tool will effectively highlight common problems, although it's worth noting that not all the latest microservice smells can be incorporated, as some may require specific organizational knowledge. Nonetheless, diligent efforts will be made to analyse, design, and develop those that can be identified through code analysis.

To further enhance the developer's experience, an extension to Visual Studio Code editor will be included in the solution. This extension is designed to streamline the tool's usability, making it more accessible and user-friendly.

### 5.2.2 Requirements

The identified requirements (non-functional and functional) will be described throughout this section.

This work's core purpose is to enhance the MSA-Nose tool, with a primary focus on expanding its functionality and refining its usability. To achieve these objectives, strict adherence to established microservice architecture patterns and guidelines is imperative, ensuring the accurate verification of detected smells. This rigorous adherence serves to not only improve the tool but also facilitates the effective implementation of microservice architecture, thereby mitigating architectural issues and promoting a smoother adoption of this architectural style within projects. The summarized view of the non-functional requirements can be seen in Table 24.

Requirement number	Description
1	Improve MSA-Nose usability and functionality attributes
2	Implement microservices with fewer architectural issues
3	Microservice architecture patterns must be followed

Table 24 – MSA-Nose improvement non-functional requirements.

The functional requirements of the work can be seen in the use case diagram shown in Figure 35.

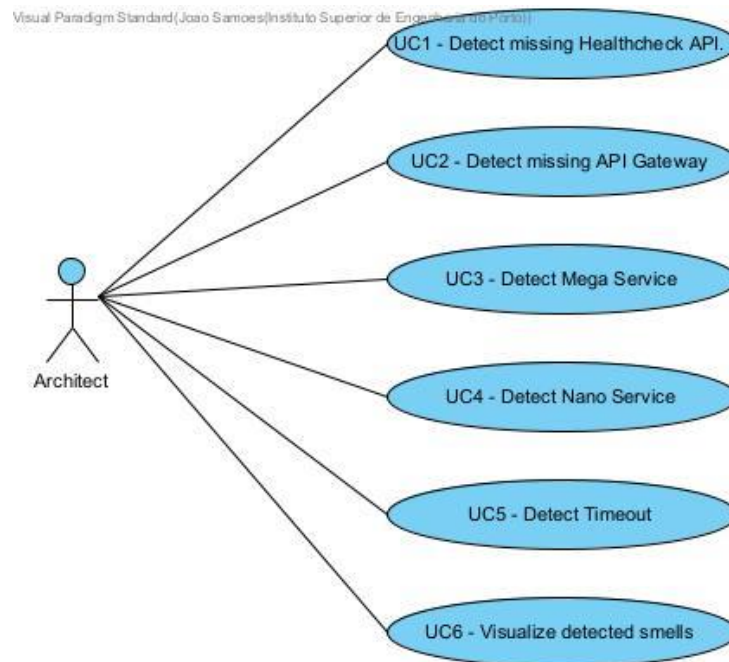


Figure 35 – Functional requirements.

All the use cases have as actors the developer as it is the one that needs to have the smells detected to improve its experience.

As previously mentioned, the use cases will be around the implementation of new microservice smells that were added to the newest catalogues. There's, however, a use case that has as its goal the update of the implementation of detection of missing API Gateway (UC2).

UC6 pertains to usability and enables users/developers to visualize the microservice architecture smells that have been detected. This use case encompasses both the implementation of the front end and the extension.

### 5.2.3 Design and Implementation

Logical and process views following the 4+1 view model of architecture (Kruchten, 1995) are presented in the following sections.

#### 5.2.3.1 Logical view

Figure 36 represents the component diagram of the MSA-Nose system created to enhance the tool created by (Walker et al., 2020).

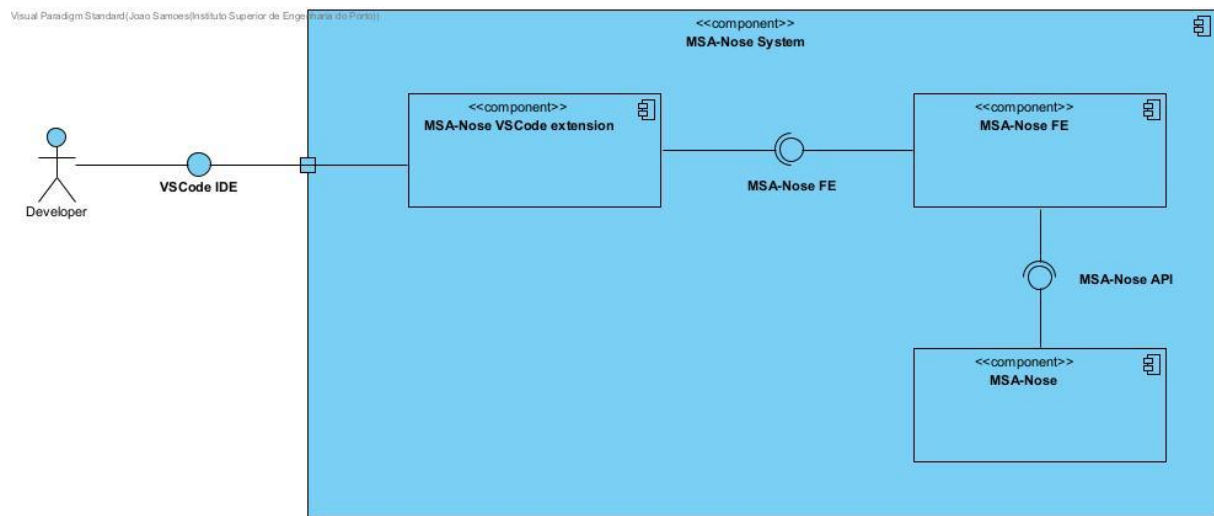


Figure 36 – Logical view of a components' diagram with a level 2 abstraction.

As described in the section before, the user will have contact with “Visual Studio Code”, which is a code editor, via an extension that contains a webview interface and sends all the required information that is made available by the front end. This last one sends all the required information that came from the extension and is inputted by the user to the backend which is the MSA-Nose tool.

After all the process is done in the backend, the user will be shown the detected smells in its IDE.

#### 5.2.3.2 Process View

In this section, the process view for each use case (see Figure 35) is detailed.

#### UC1 – Detect missing Healthcheck API

At a higher level of abstraction, the developer initiates a request to the MSA-Nose system via the NoseController, providing details about the microservice to be examined, including the path to its root. Upon receiving this information, the NoseController forwards it to the APIService, where a new function is created for retrieving a map of the available APIs within the specified microservice. This map enables the system to collect all the endpoint paths and subsequently filter for any that do not conform to the provided health check regular expression.

If no endpoints are found to match the regular expression, the microservice is labeled as "missing" and added to a list. This list is then returned to the user in the form of a Data Transfer Object (DTO). This process is further illustrated in Figure 37.

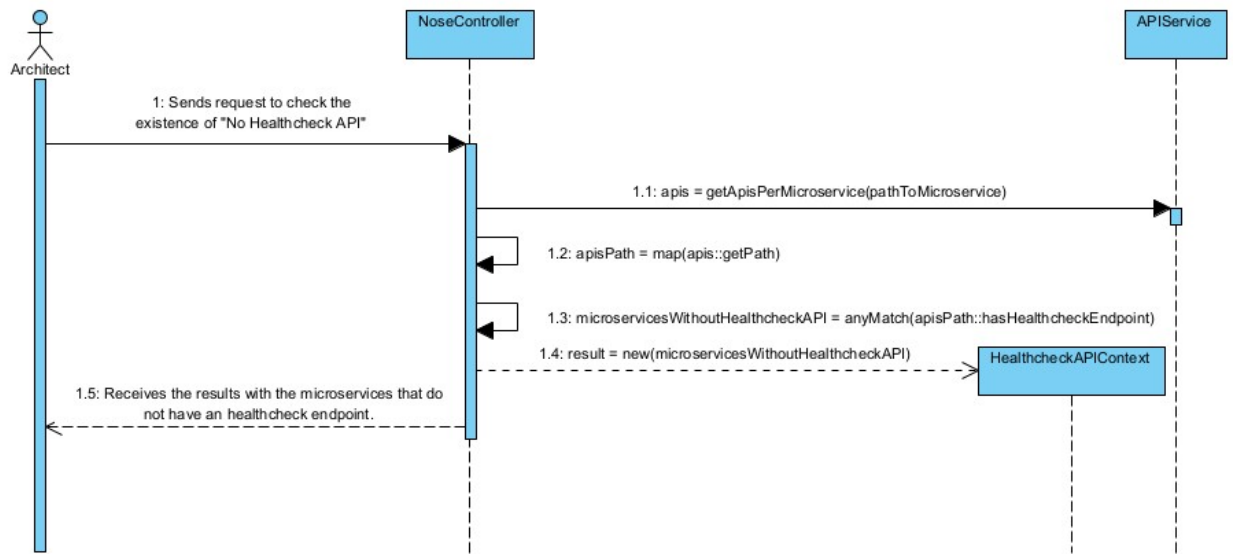


Figure 37 – Process view (UC1).

## UC2 – Detect missing API Gateway

As previously mentioned, this use case was initiated to enhance an existing rule. The initial implementation verified whether the number of microservices exceeded 50.

To improve this rule, a new function was introduced that calls the LibraryService. Within this function, it assesses whether any dependencies across all the microservices are associated with a gateway dependency, such as Spring Cloud Gateway. Additionally, it examines other files for any indications of APIGateway usage, for example, the presence of a "nginx.conf" file, which suggests the utilization of Nginx.

For a visual representation of the interactions between the various components of the MSA-Nose tool, please refer to Figure 38, which presents a sequence diagram illustrating the process.

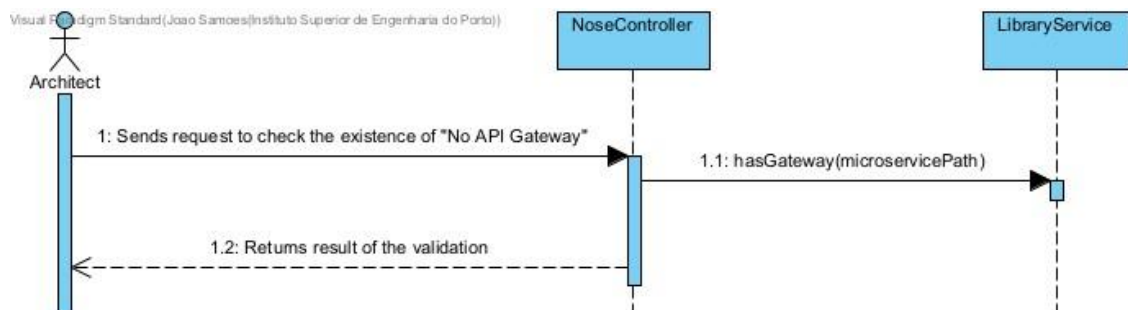


Figure 38 – Process view (UC2).

## UC3 – Detect "Mega Service"

In the pursuit of identifying this microservice antipattern, a pivotal constraint considered was the collective size of all the files within the microservice. If the total number of lines across all files surpassed 2000, the microservice was categorized as a "Mega service," and the name of its controller was included in an array. The flow of this process is visually depicted in Figure 39.

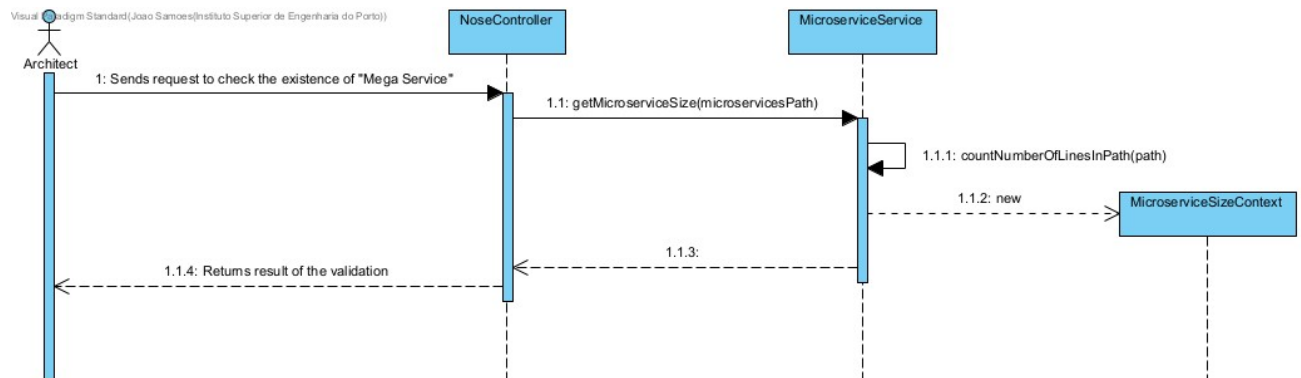


Figure 39 – Process View (UC3).

#### UC4 – Detect “Nano Service”

This rule was put into practice by utilizing the same approach as the previous use case. However, in this instance, the assessment is based on the total number of lines within the microservice, scrutinizing if it falls below the threshold of 200 lines. Microservices that meet this criterion are designated as "Nano Services".

#### UC5 – Detect “Timeout”

Implementing this rule involved conducting two distinct checks. The first check was performed on the properties file located within the resources folder, while the second check was applied to all classes annotated with "@Configuration." In the latter case, the examination focused on methods that monitored alterations to read and connect timeout values. For a visual representation of the sequence of operations in this use case, please refer to Figure 40.

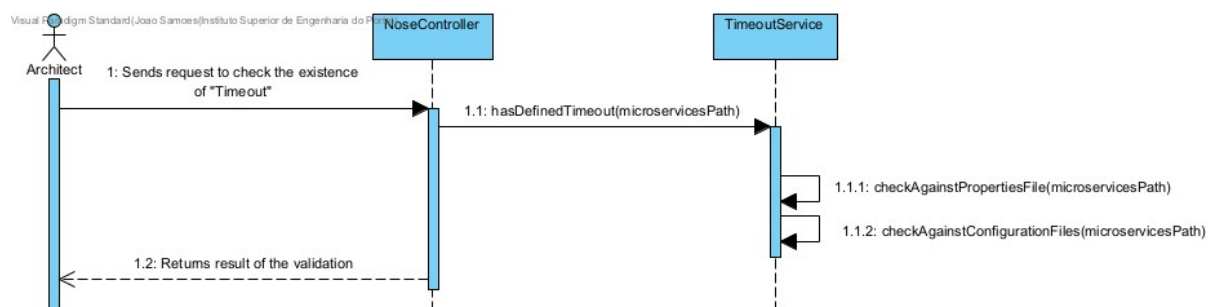


Figure 40 – Process View (UC5).

#### UC6 – Visualize detected smells

In this use case, the objective is to enhance the understanding of the detected smells. To achieve this, the idea is to develop a Visual Studio Code extension that features a webview

containing comprehensive information regarding the validation results. It's important to note that this webview necessitates a frontend service for its presentation. To illustrate the functioning of this approach, a sequence diagram for UC6 is provided, as depicted in Figure 41.

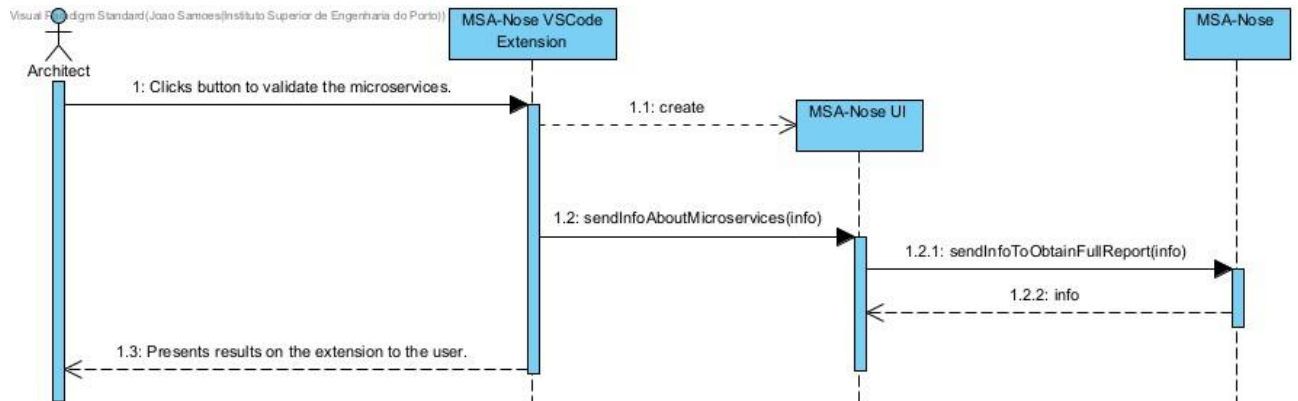


Figure 41 – Process View (UC6).





## 6 Evaluation

This chapter has the goal of explaining how the solution will be implemented be evaluated, as well as the metrics, the hypotheses, the test methodology and the result.

### 6.1 Methodology

Evaluation methodologies are systematic frameworks and techniques used to assess the effectiveness, performance, and impact of programs, projects, policies, products, or any other initiative. These methodologies are essential tools for collecting data, analysing outcomes, and making informed decisions based on evidence.

Evaluation methodologies encompass a wide range of approaches, each tailored to specific contexts and objectives. One of these is by conducting numerous assessment tests on diverse microservices apps, which is imperative to gain a precise understanding of the performance and suitability of the implemented solution. To gauge the effectiveness and accuracy of the tool, it is essential to subject it to testing across a diverse range of scenarios. An important aspect of evaluating the advancements made in the MSA-Nose tool is to compare its performance against the same microservice applications utilized in the case study conducted by (Walker et al., 2020) in their Case Study to have a full comparison not only because of performance but also to have the same results in what wasn't touched. This comparative analysis not only allows for the assessment of performance but also ensures that any untouched aspects yield consistent results.

However, the testing efforts will not be limited to just these applications; a broader scope is intended. Table 25 provides a comprehensive list of microservices, along with their respective repository URLs, sourced from the Microservices Project List (M. Rahman et al., 2019) (available at [https://github.com/daviddetaibi/Microservices\\_Project\\_List](https://github.com/daviddetaibi/Microservices_Project_List)). This expanded

testing scope provides a more holistic perspective on the tool's performance and applicability, encompassing a wider range of microservices and scenarios.

Microservice name	Repository URL
Train Ticket	<a href="https://github.com/FudanSELab/train-ticket/">https://github.com/FudanSELab/train-ticket/</a>
Teacher Management System	<a href="https://github.com/cloudhubs/tms2020">https://github.com/cloudhubs/tms2020</a>
Piggy Metrics	<a href="https://github.com/sqshq/PiggyMetrics">https://github.com/sqshq/PiggyMetrics</a>

Table 25 – Microservices information.

## 6.2 Case Studies

In this section the tests are described per microservice, mirroring the approach taken by (Walker et al., 2020) study, the analysis will commence with a meticulous manual analysis of the microservices to determine the presence of any microservice-related issue. Subsequently, a rigorous evaluation of the MSA-Nose application, along with any implemented enhancements, is carried out through a series of comprehensive tests. It's worth emphasizing that these tests were conducted on a system equipped with an Intel i7-8750H processor and 16GB of RAM.

### 6.2.1 Train Ticket

The selection of this microservice benchmark stemmed from its moderate size, making it an ideal candidate to comprehensively assess all tool conditions. Designed to mirror real-world interactions among microservices within an industrial context, this benchmark was considered one of the largest available at the time according to (Walker et al., 2020) study. It's noteworthy, however, that the number of microservices has since evolved, increasing from 41 to 47 microservices.

Table 26 shows the results of the analysis, comparing the results before and after the improvements, comparing manual and MSA-Nose analyses and time spent for each of the code smells.

Smell	Manual (B-Imp)	Manual (A-Imp)	MSA-Nose (B-Imp)	MSA-Nose (A-Imp)	Time (ms) (B-Imp)	Time (ms) (A-Imp)
ESB Usage	No	No	No	No	1	1
Too Many Standards	No	No	No	No	213	263
Wrong Cuts	0	0	2	1	1487	1456
Not Having an API Gateway	No	Yes	No	Yes	1	77

Smell	Manual (B-Imp)	Manual (A-Imp)	MSA-Nose (B-Imp)	MSA-Nose (A-Imp)	Time (ms) (B-Imp)	Time (ms) (A-Imp)
Hard-Coded Endpoints	28	0	28	0	1	1
API Versioning	76	90	76	90	1981	2318
Microservice Greedy	0	0	0	0	2093	2246
Shared Persistency	0	0	0	0	123	106
Inappropriate Service Intimacy	1	0	1	0	1617	1864
Shared Libraries	4	7	4	7	237	461
Cyclic Dependency	No	No	No	No	1	1
Mega Service	NA	0	NA	0	NA	2219
Nano Service	NA	0	NA	0	NA	
Timeout	NA	No	NA	No	NA	2141
Health check API	NA	40	NA	80	NA	2038
Total					7755	15192

Table 26 – Train Ticket case study results (“B-Imp” means Before Improvement and “A-Imp” means After Improvement).

This microservice, being notably substantial in size, served as an excellent litmus test for gauging the effectiveness of the recent enhancements. Notably, following the implementation of these improvements, the time invested in detecting these microservice-related issues increased by nearly twofold compared to the pre-implementation period. This shift in timing can be attributed to the fact that the API Gateway now performs comprehensive checks across all the "pom.xml" files of each microservice. This process assesses whether they contain any dependencies related to API Gateways or if there are any configuration files of alternative API Gateways.

Concerning the outcomes related to the newly identified microservice smells, specifically for the Mega and Nano Service categories, we considered any microservice containing Java files with a total line count exceeding 2000 lines as a Mega Service and those with fewer than 200 lines as a Nano Service. Fortunately, none of the microservices fell within these extreme size ranges, indicating that this test was successfully passed.

Concerning Timeout, multiple methods exist for ascertaining how to establish read or connect timeouts in an API. To determine if a microservice had defined timeouts, the analysis involved examining the "application.properties" or "application.yml" files for any indications of properties such as "spring.httpclient.read-timeout" or "spring.httpclient.connection-timeout." All Java files annotated with "@Configuration" were inspected to locate methods returning a

"RestTemplate" object. If such methods were found, a verification process was carried out to check if the "setReadTimeout" and "setConnectTimeout" functions were being used or not. Notably, the manual analysis yielded a "false" result, mirroring the outcome of the MSA-Nose tool improvement, which means that this test was also successful.

In the context of the Health Check API, existing resources utilized in the collection of Unversioned APIs smell were leveraged. This approach not only streamlined the process but also prevented code and functionality duplication. To detect this smell, each microservice controller was scrutinized to identify and map all endpoints. Subsequently, this mapping was subjected to analysis, wherein each key (corresponding to a microservice controller name) had its values (APIs) examined against a predefined regular expression. In cases where none of the APIs adhered to this regular expression, the key was retained, and an array was generated containing the names of all microservice controllers that did not comply with the specified pattern. On this smell's behalf, there is a one-to-one relationship between the microservice controller name and the microservice, implying that each microservice should ideally have just one controller. However, this ideal scenario does not always hold.

The manual analysis revealed that none of the 40 controllers contained a single health check endpoint. However, the MSA-Nose analysis identified the absence of a health check endpoint in any of the 80 controllers gathered. This discrepancy in the number of controllers identified suggests that the detection of these issues was not entirely successful, given that the number of controllers had doubled compared to the initial manual analysis.

## 6.2.2 Teacher Management System

The Teacher Management System (TMS) is an enterprise application designed by Baylor University to support the Texas Educator Certification training program, focusing on Computational Thinking, Coding, and Tinkering. TMS comprises four microservices: User Management System (UMS), Question Management System (QMS), Exam Management System (EMS), and Configuration Management System (CMS). These microservices are developed using the Spring Boot framework, following a structured architecture with controller, service, and repository layers. The application is packaged with Docker, deployed using Docker-compose, and employs NGINX for routing (Walker et al., 2020).

Just like in the initial case study, Table 27 displays the outcomes of MSA-Nose both before and after enhancements. It presents a comparison of manual analysis and MSA-Nose analysis, along with the time taken for each detection.

Smell	Manual (B-Imp)	Manual (A-Imp)	MSA-Nose (B-Imp)	MSA-Nose (A-Imp)	Time (ms) (B-Imp)	Time (ms) (A-Imp)
ESB Usage	No	No	No	No	1	1
Too Many Standards	No	No	No	No	66	58
Wrong Cuts	0	0	0	0	279	19
Smell	Manual	Manual	MSA-Nose	MSA-Nose	Time (ms)	Time (ms)

	(B-Imp)	(A-Imp)	(B-Imp)	(A-Imp)	(B-Imp)	(A-Imp)
<b>Not Having an API Gateway</b>	Yes	Yes	No	Yes	1	190
<b>Hard-Coded Endpoints</b>	2	3	2	0	1	1
<b>API Versioning</b>	62	69	62	65	546	492
<b>Microservice Greedy</b>	0	0	0	0	271	499
<b>Shared Persistency</b>	0	0	0	0	60	17
<b>Inappropriate Service Intimacy</b>	0	0	0	0	1	1
<b>Shared Libraries</b>	2	2	2	2	47	22
<b>Cyclic Dependency</b>	No	No	No	No	1	1
<b>Mega Service</b>	NA	1	NA	1	NA	489
<b>Nano Service</b>	NA	0	NA	0	NA	
<b>Timeout</b>	NA	No	NA	No	NA	448
<b>Health check API</b>	NA	12	NA	17	NA	521
				<b>Total</b>	<b>1074</b>	<b>2759</b>

Table 27 – Teacher Management System case study results (“B-Imp” means Before Improvement and “A-Imp” means After Improvement).

In the realm of performance, a phenomenon like the one observed in the previous case study occurred. The time required to identify the microservice smells was significantly influenced by the introduction of the new features, more than doubling the execution time in the case of the TMS microservice.

Concerning the outcomes related to the new smells, the manual analysis correctly classified the QMS microservice as a Mega Service, aligning with the result obtained from the MSA-Nose analysis. Likewise, both the Nano Service and Timeout smells were correctly identified through manual analysis and confirmed by the MSA-Nose tool.

However, when it comes to the Health Check API, a situation analogous to the previous case study emerged. In this instance, more controllers were identified than the actual number in existence, resulting in the erroneous identification of this microservice smell.

### 6.2.3 Piggy Metrics

Piggy Metrics represents a straightforward financial advisory application designed to showcase the Microservice Architecture Pattern, utilizing technologies like Spring Boot, Spring Cloud, and Docker. This application is divided into three fundamental microservices, specifically the Account service, Statistics service, and Notification service. Each of these microservices can be independently deployed, and they are structured around distinct business domains, providing modularity and flexibility to the application. In line with the

methodology employed in previous case studies, the outcomes will be presented in a tabular format (see Table 28). This table will encompass the results of both manual and MSA-Nose analysis, along with the corresponding time taken to detect each microservice smell. Given that (Walker et al., 2020) did not analyse this particular microservice benchmark, there was no preceding manual assessment or time measurements available for the fifth column (representing the pre-improvement period). These time measurements were obtained using the system properties described earlier in this section.

Smell	Manual	MSA-Nose (B-Imp)	MSA-Nose (A-Imp)	Time (ms) (B-Imp)	Time (ms) (A-Imp)
ESB Usage	No	No	No	1	1
Too Many Standards	No	No	No	33	39
Wrong Cuts	0	0	0	12	11
Not Having an API Gateway	Yes	No	No	1	130
Hard-Coded Endpoints	0	0	0	1	1
API Versioning	3	9	4	349	299
Microservice Greedy	0	0	0	398	248
Shared Persistency	0	0	0	13	8
Inappropriate Service Intimacy	0	0	0	1	1
Shared Libraries	4	4	4	37	32
Cyclic Dependency	No	No	No	1	1
Mega Service	1	NA	1	NA	236
Nano Service	0	NA	0	NA	
Timeout	No	NA	No	NA	259
Health check API	3	NA	3	NA	225
Total				847	1491

Table 28 - Piggy Metrics case study results (“B-Imp” means Before Improvement and “A-Imp” means After Improvement).

As mentioned earlier, this microservice was initially characterized as comprising only three microservices. However, it's important to note that MSA-Nose lacks the capability to distinguish between business-specific microservices and others. This distinction becomes evident when examining the results of the API versioning smell, which included endpoints from microservices that do not pertain to those with core business logic. It's worth adding that the method employed to retrieve APIs from a microservice effectively eliminates duplicates. Consequently, in cases where different controllers within a microservice share the same path, only one of those duplicates will appear in the results.

The method used to retrieve APIs also had an impact on the way controllers with the Health Check API smell were collected, causing it to not identify their names accurately. Nevertheless, it's important to note that the count of controllers or microservices with this smell is accurate.

Therefore, while it is technically a false positive in terms of the method used to identify them, the overall count is correct. The issue lies in the method, not the outcome.

Concerning the API Gateway smell, following the implemented improvements, it failed to detect the utilization of the gateway within this microservice. This was attributed to the microservice's reliance on a dependency that had been without any updates for over four years, however with no reported vulnerabilities on their dependencies since May 23, 2019.

### **6.3 Threats to validity**

To mitigate potential threats to validity, a new microservice was introduced into the test dataset. This was done to address the possibility that the initial dataset was biased due to its origin from specific projects.

However, despite this effort, certain threats to validity persist. Reproducibility may not align with (Walker et al., 2020)'s original setup, as variations in settings can introduce discrepancies that impact result validity.

Furthermore, alterations in the microservices employed and the absence of continuous improvements since its release in the tool can also influence the outcome.





## 7 Conclusions

This chapter describes the contributions arising from the conducted work. Subsequently, attention turns to the primary limitations and constraints encountered during the development and writing process. Finally, a section presents information regarding potential improvements that could be explored in future work.

### 7.1 Contributions

The objectives of this study were first created, as detailed in Section 1.3, and Table 29 subsequently details these goals and their related successes.

Number	Goal	Success
1	Explore and evaluate the comprehensiveness of existing catalogues of microservice smells, also gauging the acceptance of the included smells.	Achieved
2	Improve a microservice smell detector that incorporates and improves upon existing applications. This detector aids in identifying smells in a microservice architecture, thus helping to improve the design and implementation of microservices.	Achieved

Table 29 – Goals achievement.

Beginning with goal number 1, which involved exploring, evaluating, and contributing to a microservice catalogue, significant progress was made. This achievement was made possible through the completion of a systematic mapping study, which involved the analysis of various articles published since 2021, focusing on issues related to the implementation of microservice smells. Additionally, we conducted an industrial survey, which garnered

participation from 31 practitioners experienced in microservices. This survey played a pivotal role in identifying the challenges and concerns that developers encounter in microservice implementation, thus providing valuable insights for future research endeavours.

As for goal number 2, which pertained to enhancing the MSA-Nose detection tool, progress was made in expanding the repertoire of detectable microservice smells. While there is certainly room for further refinement, this solution is now accessible as an open-source project, conveniently located at the beginning of Section 5.2. for reference. Detailed results stemming from this implementation can be in Chapter 6.

## **7.2 Difficulties along the way**

Throughout the course of developing this document, several challenges were encountered, which influenced the outcomes of this work.

The initial challenge, as previously noted in Section 3.4, was the length of the industry survey. Its extensive nature dissuaded some potential participants from completing it, despite the majority of the questions being close-ended. This ultimately resulted in a smaller number of survey respondents.

In addition, there were constraints experienced during the implementation phase. Certain microservice smells could not be feasibly detected through code analysis or static analysis methods. Consequently, the number of implemented smells was limited to those for which detection methods could be successfully devised, resulting in a smaller set of implemented smells.

Furthermore, the initial concept revolved around selecting a variety of tools capable of detecting microservice-related smells and consolidating them into a single extension. The goal was to provide users with a comprehensive toolkit for assessing their microservice applications. However, it became evident that many of these detection tools were identifying the same set of microservice smells. Consequently, the decision was made to opt for the tool that detected the largest number of these smells, ensuring a more efficient and focused approach to microservice smell detection.

## 7.3 Future Work

There is room for enhancements in the implementation of new microservice smells, incorporating both static and dynamic analysis approaches. Dynamic analysis, for instance, can aid in identifying additional security-related issues. Moreover, the existing microservice smells discussed in this document could benefit from further refinement by subjecting them to testing across a wider range of microservices to identify additional issues.

Additionally, an envisaged improvement, initially considered, is the creation of a repository containing a microservice replete with anti-patterns. Such a repository would serve as a valuable resource, providing developers with a tangible example of what to avoid when implementing microservices.

Lastly, concerning the new microservice smells and the development of the smells catalogue, there is ample opportunity for further exploration. As evidenced by the research conducted, there are numerous areas yet to be explored. Consequently, there is the potential for ongoing updates and expansions to the catalogue.



# References

- Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016*, 44–51. <https://doi.org/10.1109/SOCA.2016.15>
- Arcelli Fontana, F., Lenarduzzi, V., Roveda, R., & Taibi, D. (2019). Are architectural smells independent from code smells? An empirical study. *Journal of Systems and Software*, 154, 139–156. <https://doi.org/10.1016/j.jss.2019.04.066>
- Azadi, U., Fontana, F. A., & Taibi, D. (2019). Architectural smells detected by tools: A catalogue proposal. *Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019, May*, 88–97. <https://doi.org/10.1109/TechDebt.2019.00027>
- Barbara Kitchenham. (2004). Procedures for Performing Systematic Reviews. *Keele University Technical Report*, 33(2004), 1–26.
- CCSU. (n.d.). *CS 410/510 - Software Engineering Resilience Engineering*. <https://cs.ccsu.edu/~stan/classes/CS410/Notes16/14-ResilienceEngineering.html>
- Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150, 77–97. <https://doi.org/10.1016/j.jss.2019.01.001>
- Ding, X., & Zhang, C. (2022). How Can We Cope with the Impact of Microservice Architecture Smells? *ACM International Conference Proceeding Series*, 8–14. <https://doi.org/10.1145/3524304.3524306>
- Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, 195–216. [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12)
- Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., & Osborne, D. (2002). *J2EE Antipatterns*.
- Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure it? Manage it? Ignore it? Software practitioners and technical debt. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, 50–60. <https://doi.org/10.1145/2786805.2786848>
- Foundation, E. (2020). *2020 Jakarta EE Developer Survey Report* (Vol. 0, Issue C).
- Foundation, E. (2021). *2021 Jakarta EE Developer Survey Report* (Vol. 0, Issue C).
- Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Vol. 857 LNCS*. <https://doi.org/10.1007/bfb0020422>

- Fowler, M. (2014, March 25). *Microservices*.  
<https://martinfowler.com/articles/microservices.html>
- Fowler, M., & Lewis, J. (2014, March 25). *Microservices - a definition of this new architectural term*. <https://martinfowler.com/articles/microservices.html#footnote-monolith>
- Garcia, J., Daniel, P., Edwards, G., & Medvidovic, N. (2009). Identifying Architectural Bad Smells. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 255–258. <https://doi.org/10.1109/CSMR.2009.59>
- GitLab. (n.d.). *What are microservices?* | GitLab. Retrieved January 21, 2023, from <https://about.gitlab.com/topics/microservices/>
- GitLab. (2022, September 29). *What are the benefits of a microservices architecture?* | GitLab. <https://about.gitlab.com/blog/2022/09/29/what-are-the-benefits-of-a-microservices-architecture/>
- Kitchenham, Barbara Charters, S., Budgen, D., Brereton, P., Turner, M., Linkman, S., Jørgensen, M., Mendes, E., & Visaggio, G. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*.
- Koen, P. A., Ajamian, G. M., Boyce, S., Clamen, A., Fisher, E., Fountoulakis, S., Johnson, A., Puri, P., & Seibert, R. (2002). Fuzzy Front End : and Techniques. *Industrial Research*, pp, 5–35. [http://www.stevens.edu/cce/NEW/PDFs/FuzzyFrontEnd\\_Old.pdfNEW/PDFs/FuzzyFrontEnd\\_Old.pdf](http://www.stevens.edu/cce/NEW/PDFs/FuzzyFrontEnd_Old.pdfNEW/PDFs/FuzzyFrontEnd_Old.pdf)
- Kruchten, P. B. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6), 42–50. <https://doi.org/10.1109/52.469759>
- Larrucea, X., Santamaria, I., Colomo-palacios, R., & Ebert, C. (2018). *Microservices*.
- Loukides, M., & Swoyer, S. (2020, July 15). *Microservices Adoption in 2020 – O'Reilly*. <https://www.oreilly.com/radar/microservices-adoption-in-2020/>
- Lu, N., Glatz, G., & Peuser, D. (2019). *Moving mountains – practical approaches for moving monolithic applications to Microservices*.
- Neri, D., Soldani, J., Zimmermann, O., & Brogi, A. (2020). Design principles, architectural smells and refactorings for microservices: a multivocal review. *Software-Intensive Cyber-Physical Systems*, 35(1–2), 3–15. <https://doi.org/10.1007/s00450-019-00407-8>
- OASIS. (n.d.). *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC / FAQ*. Retrieved February 25, 2023, from <https://www.oasis-open.org/committees/tosca/faq.php>
- Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008). Systematic mapping studies in software engineering. *12th International Conference on Evaluation and Assessment in Software Engineering, EASE 2008, June*. <https://doi.org/10.14236/ewic/ease2008.8>
- Pigazzini, I., Fontana, F. A., Lenarduzzi, V., & Taibi, D. (2020). Towards microservice smells detection. *Proceedings - 2020 IEEE/ACM International Conference on Technical Debt*,

- TechDebt 2020*, May, 92–97. <https://doi.org/10.1145/3387906.3388625>
- Ponce, F., Marquez, G., & Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC, 2019-Novem*(September). <https://doi.org/10.1109/SCCC49216.2019.8966423>
- Ponce, F., Soldani, J., Astudillo, H., & Brogi, A. (2022). Smells and refactorings for microservices security: A multivocal literature review. *Journal of Systems and Software*, 192, 111393. <https://doi.org/10.1016/J.JSS.2022.111393>
- Rahman, A., Parnin, C., & Williams, L. (2019). The Seven Sins: Security Smells in Infrastructure as Code Scripts. *Proceedings - International Conference on Software Engineering, 2019-May*, 164–175. <https://doi.org/10.1109/ICSE.2019.00033>
- Rahman, M., Panichella, S., & Taibi, D. (2019). *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS.
- Refactoring Guru. (n.d.). *Shotgun Surgery*. Retrieved February 26, 2023, from <https://refactoring.guru/smells/shotgun-surgery>
- Rich, N., & Holweg, M. (2000). VALUE ANALYSIS. *INNOREGIO: Dissemination of Innovation and Knowledge Management Techniques*, 0–31.
- Richards, M. (2016). *Microservices AntiPatterns and Pitfalls*.
- Richardson, C. (2018). *Microservices Patterns: With Examples in Java*.
- Sampaio, A. (2015). Improving Systematic Mapping Reviews. *ACM SIGSOFT Software Engineering Notes*, 40(6), 1–8. <https://doi.org/10.1145/2830719.2830732>
- Sharma, T., Singh, P., & Spinellis, D. (2020). An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 25(5), 4020–4068. <https://doi.org/10.1007/s10664-020-09847-2>
- Soldani, J., Muntoni, G., Neri, D., & Brogi, A. (2021). The  $\mu$ TOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software - Practice and Experience*, 51(7), 1591–1621. <https://doi.org/10.1002/spe.2974>
- Söylemez, M., Tekinerdogan, B., & Tarhan, A. K. (2022). Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. *Applied Sciences (Switzerland)*, 12(11). <https://doi.org/10.3390/app12115507>
- STRIMBEI, C., DOSPINESCU, O., STRAINU, R.-M., & NISTOR, A. (2015). Software Architectures – Present and Visions. *Informatica Economica*, 19(4/2015), 13–27. <https://doi.org/10.12948/issn14531305/19.4.2015.02>
- Suryanarayana, G., Samarthayam, G., & Sharma, T. (2014). *Refactoring for software design smells : managing technical debt*.
- Taibi, D., & Lenarduzzi, V. (2018). On the Definition of Microservice Bad Smells. *IEEE Software*,



- 35(3), 56–62. <https://doi.org/10.1109/MS.2018.2141031>
- Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5), 22–32. <https://doi.org/10.1109/MCC.2017.4250931>
- Taibi, D., Lenarduzzi, V., & Pahl, C. (2019). *Microservices Anti-Patterns : A Taxonomy*.
- Tarjan, R. (1972). Depth-first search and linear graph algorithms Connected Compon... *SIAM Journal on Computing*, 1(2), 146–160.  
/citations?view\_op=view\_citation&continue=/scholar%3Fhl%3Den%26as\_sdt%3D0,5%26scilib%3D1025&citilm=1&citation\_for\_view=pfCBJt8AAAAJ:0EnyYjriUFMC&hl=en&oi=p
- Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G. El, Privat, J., & Guéhéneuc, Y. G. (2020). On the Study of Microservices Antipatterns: A Catalog Proposal. *ACM International Conference Proceeding Series*, 1.  
<https://doi.org/10.1145/3424771.3424812>
- Tighilt, R., Abdellatif, M., Trabelsi, I., Madern, L., Moha, N., & Guéhéneuc, Y. G. (2023). On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. *Journal of Systems and Software*, 204, 111755.  
<https://doi.org/10.1016/j.jss.2023.111755>
- Viggiato, M., Terra, R., Rocha, H., Valente, M. T., & Figueiredo, E. (2018). *Microservices in Practice: A Survey Study*. <http://arxiv.org/abs/1808.04836>
- Walker, A., Das, D., & Cerny, T. (2020). Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences (Switzerland)*, 10(21), 1–20.  
<https://doi.org/10.3390/app10217800>
- Waseem, M., Liang, P., Shahin, M., Ahmad, A., & Nassab, A. R. (2021). On the nature of issues in five open source microservices systems: An empirical study. *ACM International Conference Proceeding Series*, 201–210. <https://doi.org/10.1145/3463274.3463337>
- Wu, M., Zhang, Y., Liu, J., Wang, S., Zhang, Z., Xia, X., & Mao, X. (2022). On the Way to Microservices: Exploring Problems and Solutions from Online Q&A Community. *Proceedings - 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022*, 432–443. <https://doi.org/10.1109/SANER53432.2022.00058>
- Zhong, C., Huang, H., Zhang, H., & Li, S. (2022). Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation. *Software - Practice and Experience*, July, 2574–2597. <https://doi.org/10.1002/spe.3138>
- Zhou, X., Li, S., Cao, L., Zhang, H., Jia, Z., Zhong, C., Shan, Z., & Babar, M. A. (2023). Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. *Journal of Systems and Software*, 195, 111521.  
<https://doi.org/10.1016/j.jss.2022.111521>

# Appendix A (Value Analysis)

## Value analysis

This chapter describes the value analysis of the dissertation, which intends to give a contextualization of the problem, the proposal emergence, and the value intended to be provided to the end user.

“Value Analysis can be defined as a process of systematic review that is applied to existing product designs in order to compare the function of the product required by a customer to meet their requirements at the lowest cost consistent with the specified performance and reliability needed” (Rich & Holweg, 2000). This analysis is therefore necessary so the end user can be aware of the value of the proposed work.

To contextualize and support this value analysis this chapter will have an explanation of the New Concept Development Model (NCD) which is defined as the innovation process where new products or ideas are generated for the market.

## New Concept Development Model

In the business world, innovation is a process that, beyond introducing a new idea or concept, requires analysis and implementation to generate value a customer feels he can afford to pay for. To guarantee that the product created/developed has the biggest value possible to the end-user, a method represented in Figure 42 was created to characterize the innovation process.

This innovation process may be divided into three areas: the fuzzy front end (FFE), the new product development (NPD) process, and commercialization (Koen et al., 2002).

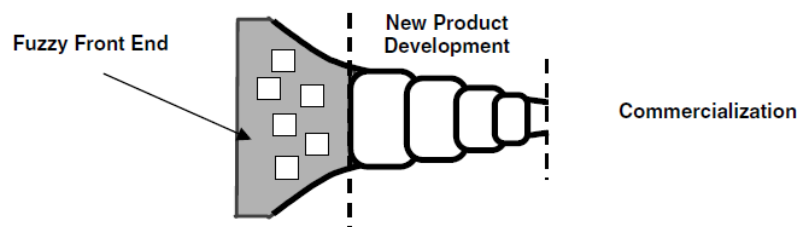


Figure 42 – The innovation process (Koen et al., 2002)

The first two steps are where an opportunity is found and where the product is created and discussed. These steps are distinct from each other in terms of the Nature of Work, Commercialization Date, Funding, Revenue Expectations, Activity, and Measures of Progress (Koen et al., 2002). However, both aim to create a new product, whereas commercialization aims to sell the created product.

However, the first part of the innovation process (FFE) had room for improvement due to a lack of common terms and definitions between companies, and so, to address this shortcoming the New Concept Development (NCD) Model was created (Koen et al., 2002).

This model, as shown in Figure 43, is composed of three parts:

- Five controllable activity elements (opportunity identification, opportunity analysis, idea generation, enrichment, idea selection, and concept definition).
- The engine that drives the five key elements with leadership, culture, and business strategy.
- Influencing factors that may affect the entire innovation process; consist of the outside world and/or the enabling sciences that are uncontrollable by the corporation.

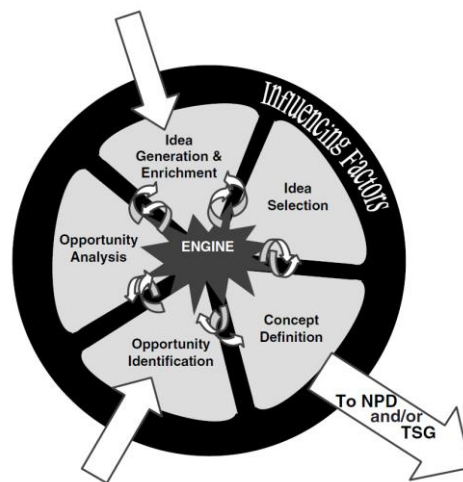


Figure 43 – Relationship diagram representing the NCD model (Koen et al., 2002).

## Opportunity Identification

This element is where the organization identifies opportunities it might want to pursue and the market or technology arena in the company may want to participate. The essence of this element is the sources and methods used to identify opportunities to pursue. Typically this element is driven by the business goals (Koen et al., 2002).

There are a few ways to be more efficient in identifying new opportunities. One of the techniques is technology trend analysis, which is a technique that consists of “collecting, analysing, and communicating the best available information on competitive trends” (Koen et al., 2002) to better define opportunities and product improvement.

### **The popularity of microservice architecture and the impact of incorrect implementation**

Microservice architecture is a software architecture that is still in consolidation. However, its popularity has been increasing at a fast pace since 2015 (Di Francesco et al., 2019) due to its characteristics and benefits.

To follow the practices that bring a plethora of advantages to software development, companies like Amazon, Deutsche Telekom, LinkedIn, Netflix, SoundCloud, The Guardian, Uber, and Verizon are quickly adopting microservice-based approaches (Larrucea et al., 2018).

This need to quickly implement microservice-based applications can lead practitioners to experience challenges about microservice boundaries that will negatively impact quality attributes (e.g., reusability, testability, and maintainability) of the developed application/service. The major effect of these would be not having the many advantages that the microservice architecture has to offer. Microservices affected by architecture smells are more frequently to be changed than clean ones and the more architecture smells a microservice is affected the more likely it is to be altered (Zhong et al., 2022).

## Opportunity Analysis

An opportunity is analysed and assessed during the opportunity analysis to confirm that it is worth pursuing. For that to be possible, additional information is needed for translating opportunity identification into specific business and technology opportunities. The tools and methods used in opportunity analysis to determine if an opportunity existed may be used again in this element however more resources will be expended, providing more detail on the appropriateness and attractiveness of the selected opportunity (Koen et al., 2002).

As shown in the opportunity identification, a survey was given to subscribers of a well-known IT media company O'Reilly where they were asked to what extent, and how were they using the microservice architecture. Of the 1502 respondents, most have an IT role in their company (more than 75%) (Loukides & Swoyer, 2020). As shown in Figure 44, the number of users that are using microservices in their organization for the first time in the past 3 years was almost half of the respondent's total.

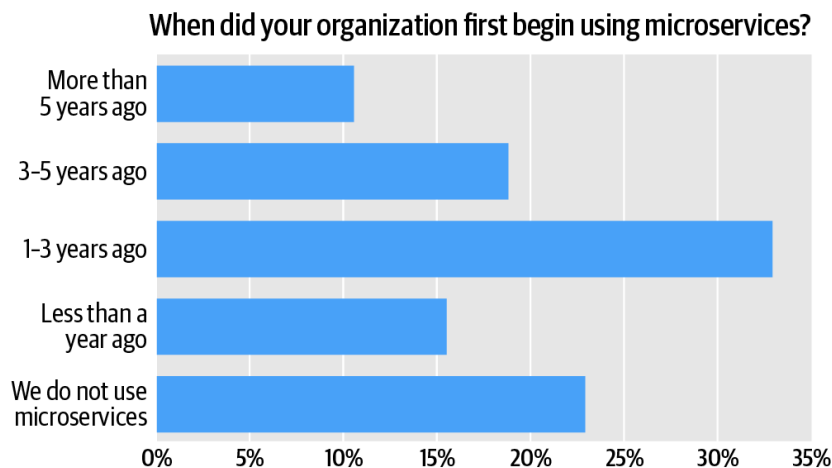


Figure 44 - Duration of use of microservice from the survey respondents (Loukides & Swoyer, 2020)

Many authors refer to the microservice as the most used architecture model used by different companies, being this approach used by leading software consultancy firms and product design companies due to its appealing architecture that allows teams and software organizations to be more productive in general, and build frequently more successful products (Alshuqayran et al., 2016). The survey conducted by the Eclipse Foundation in 2021, Jakarta EE concludes that the popularity of microservices had a nominal increase from 2020 to 2021, from 39% to 43%, respectively, of usage of the microservice architecture for implementing Java systems in the cloud (Foundation, 2020, 2021).

However, the adoption of the microservice architecture is not a simple process. As it has a wide and in-discussion definition, with many points that need to be reached, this adoption tends to be complex, because requires managing distributed architecture and its challenges, which include network latency and unreliability, fault tolerance, complex services' orchestration, data consistency and transaction management, and load balancing (Di Francesco et al., 2019).

These new challenges imposed on the developer trigger architectural smells on the application or, in other words, symptoms of bad code or design that can cause different quality problems, such as faults, technical debt, or difficulties with maintenance and evolution (Arcelli Fontana et al., 2019).

A study conducted by (Ernst et al., 2015) concluded that one of the main sources of technical debt in an application is architectural smells, as seen in Figure 45, which is also the only consensual source of it.

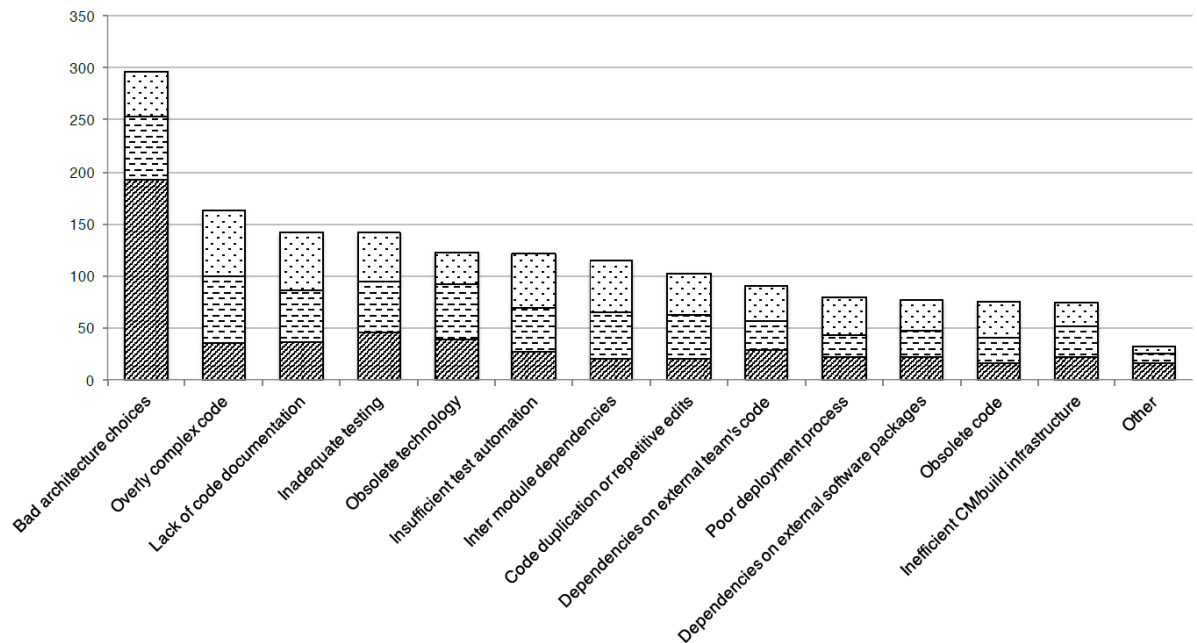


Figure 45 - Ranking sources of technical debt. Choice 1 is represented by hatches; Choice 2, dashes; and Choice 3, dots (Ernst et al., 2015).

Any type of software smell will negatively affect the application; however, architectural smells can have a bigger impact when present than code smells because they are at a bigger level of software systems.

To be able to check how architectural smells affect applications, (Zhong et al., 2022) conducted research with industrial collaboration that checked what impacts do the architectural smells have on the maintainability (which is one of the quality attributes of software) of the MSA-based system. To measure maintainability, (Zhong et al., 2022) adopted five measures that could be extracted from the revision history. These measures include the changes made to a system, that indicate the difficulty to modify the system, namely, Commit Count per File (CCF) and Commit Line Count per File (CLCF). These also include the independence committers have in changes, that is, the degree to which the system can be changed independently by committers, namely Commit Overlap Ratio (COR), Commit Fileset Overlap Ratio (CFOR), Pairwise Committer Overlap (PCO). The higher the values on these measures, the lower the maintainability.

For this study to be successful, (Zhong et al., 2022) used 118 microservices that contained 44,334 files and 14,070 commits. Also, to correlate the maintainability of the microservice with the existence of architecture smells, (Zhong et al., 2022) carefully chose a list of six architecture smells, which can be identified precisely at different levels in a software system. The chosen ones were Cyclic Dependency (CD) – a circular chain of dependencies among a set of abstractions, Hub-like Dependency (HD) – an abstraction that has excessive ingoing and outgoing dependencies with other abstractions, Unstable Dependency (UD) – which occurs when an abstraction depends on another abstraction less stable than it, Concern Overload

(CO) – A component implements an excessive number of concerns, Scattered Functionality (SF) – multiple components are responsible for realizing the same concern, and some of those components are responsible for orthogonal concerns, Modularity Violation (MV) – Files of two structurally independent components are shown to change together frequently in the revision history (Zhong et al., 2022).

After the assessment was done, a table was created that compared architecture smells and maintainability measures, as shown in Table 30.

		CCF	CLCF	COR	CFOR	PCO
Percent difference between smelly and clean microservices	pd	248.17%	52.13%	83.89%	44.57%	223.03%
	p-value	<0.001	0.615	<0.001	<0.001	<0.001
Correlation between AS number and the measures	pc	0.238	-0.058	0.475	0.508	0.497
	p-value	0.010		<0.001	<0.001	<0.001

Table 30 – Architecture Smells versus maintainability measures (Zhong et al., 2022).

The study used a 2-sample t-test to compare the maintainability measures of "smelly" and "clean" microservices. The greatest difference in maintainability measures was found with CCF and the smallest was with CFOR. Results showed that the p-value was less than 0.05 for four of the maintainability measures, which verified the hypothesis that smelly microservices are more difficult to maintain. However, there was no statistical significance for CLCF values. The study also found positive correlations between the number of architectural smells a microservice has and most of the maintainability measures using Pearson Correlation Analysis, which suggests that microservices affected by architectural smells are generally more difficult to maintain (Zhong et al., 2022).

Avoiding architectural issues in a software system requires early identification and mitigation of potential problems. One way to achieve this is by using tools that can detect and report code-level smells and other issues iteratively. Tools like SonarQube<sup>3</sup> can analyse the codebase and provide feedback on issues such as duplicate code, long methods, and large classes, among others. However, when it comes to architectural issues, there is not yet an effective tool to automatically detect and report on these issues.

Developing such a tool could be beneficial for software teams, as it could provide insights into potential problems with the architecture of a system and help teams proactively address these issues. This tool could analyse the dependencies between services and detect cases of cyclic dependencies or feature envy, which could indicate architectural issues.

<sup>3</sup> <https://www.sonarsource.com/products/sonarqube/>

## Appendix B (Additional Pitfalls)

<i>Bad practice</i>	<i>Description</i>
Timeout <sup>11</sup> (also named Dogpiles <sup>12</sup> )	The service consumer cannot connect to the microservice. Mark Richards recommends using a time-out value for service responsiveness or sharing the availability and the unavailability of each service through a message bus, so as to avoid useless calls and potential time-outs due to service unresponsiveness. <sup>11</sup>
I Was Taught to Share <sup>11</sup>	Sharing modules and custom libraries between microservices.
Static Contract Pitfall <sup>11,12</sup>	Microservice APIs that aren't versioned, possibly causing service consumers to connect to older versions of the services.
Mega-Service <sup>13</sup>	A service that is responsible for many functionalities and should be decomposed into separated microservices.
Shared Persistence <sup>13,14</sup>	Using shared data among services that access the same database.
Data Ownership <sup>14</sup>	Data should not be directly shared among different services. Microservices should own only the data they need and possibly share it via APIs.
Leak of Service Abstraction <sup>13</sup>	Designing service interfaces for generic purposes and not specifically for each service.
Hardcoded IPs and Ports <sup>12</sup>	Hard-coding the IP address and ports of communicating services, therefore making it harder to change the service location afterward.
Not Having an API Gateway <sup>15</sup>	Services directly exposed to the outside and connected to each other. Services should not be exposed through an API gateway layer and should not be connected directly, so as to simplify the connection and support monitoring, and authorization issues should be delegated to the API gateway. Moreover, changes to the API contract can be easily managed by the API gateway, which is responsible for serving the content to different consumers, providing only the data they need.
Lust <sup>16</sup>	Using the latest technologies.
Gluttony <sup>16</sup>	Using too many different communication protocols such as HTTP, protocol buffers, Thrift, etc.
Greed <sup>16</sup>	Services all belonging to the same team.
Sloth <sup>16</sup>	Creating a distributed monolith due to the lack of independence of microservices.
Wrath <sup>16</sup>	Blowing up when bad things happen.
Envy <sup>16</sup>	The shared-single-domain fallacy.
Pride <sup>16</sup>	Testing in the world of transience.

Table 31 – The main pitfalls proposed in non-peer-reviewed literature and practitioner talks (Taibi & Lenarduzzi, 2018).





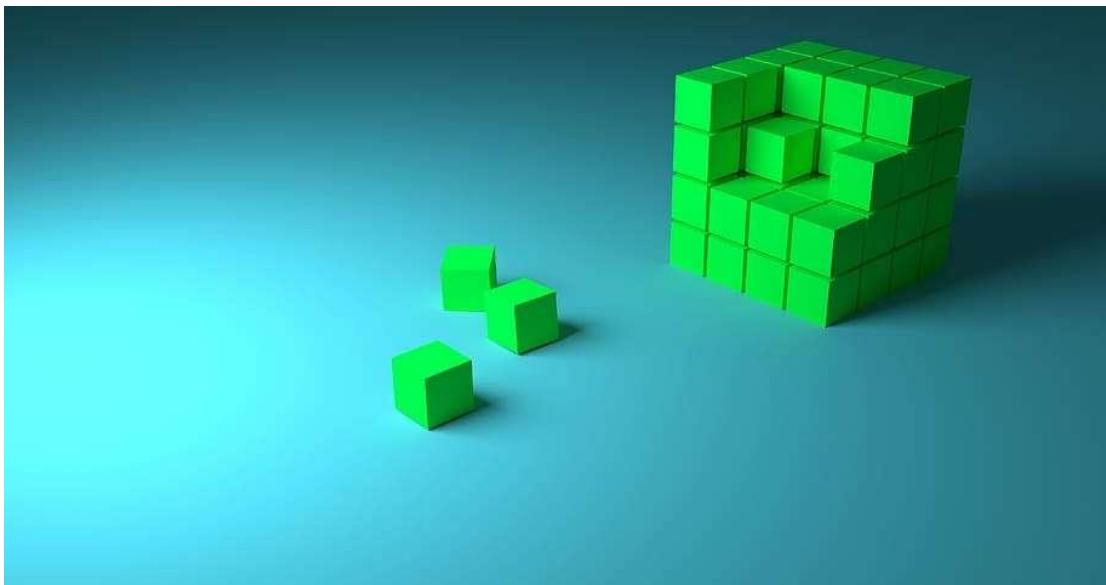
## Appendix C (Survey)

### Microservice Smells - Catalogue Analysis

Hello, my name is João Samões. I am enrolled in the Master in Informatic Engineering, Software Engineering branch, at Instituto Superior de Engenharia do Porto. My thesis topic is "Detection of microservice smells".

This survey aims to collect feedback from developers on their thoughts about a microservice **smell** catalogue built in previous studies. Your goal is to identify and prioritize microservice smells.

The survey should take no more than 10 minutes to complete and your responses will remain anonymous. At the end of this survey, you can leave your e-mail so that you can receive the final result of this study.



The aim of this section is to collect information about your background and experience.

1. In what area of software engineering do you work/have more experience with? \*

*Marcar apenas uma oval.*

☐ Software Architecture

☐ Software Development

☐ Software Management

☐ Outra: \_\_\_\_\_

2. What is your company?

\_\_\_\_\_

3. How many years of experience do you have in the area of software engineering? \*

*Marcar apenas uma oval.*

☐ <1 year

☐ 1-2 years

☐ 3-5 years

☐ >5 years

4. How many years of experience do you have specifically with microservices? \*

*Marcar apenas uma oval.*

☐ <1 year

☐ 1-2 years

☐ 3-5 years

☐ >5 years

## Start of the smells catalogue analysis

If you reach here it means that you have a valuable input to this study.  
The following sections will be divided by two major groups (Technical and Organizational) and then will be divided by a few more restrictive groups that were gathered in a the study done by Davide Taibi, Valentina Lenarduzzi and Claus Pahl in "Microservices Anti-Patterns : A Taxonomy".  
Each smell will have an explanation and you will be asked a simple question about it.

### Technical Microservice Smells - Internal

These microservice smells are anti-patterns that impact the individual microservice.

#### API Versioning

This smell occurs when APIs are not semantically versioned. In the case of new versions of non-semantically versioned APIs, API consumers may face connection issues. For example, the returning data might be different or might need to be called differently.

5. API Versioning: What is the possible effect of this smell on the quality of microservice architecture?

\*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

#### Hardcoded Endpoints

Hardcoded IP addresses and ports of the services between connected microservices.  
Microservices connected with hardcoded endpoints lead to problems when their locations need to be changed.

6. Hardcoded Endpoints: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Inappropriate Service Intimacy

The microservice keeps on connecting to private data from other services instead of dealing with its own data. Connecting to private data of other microservices increases coupling between microservices. The problem could be related to a mistake made while modeling the data.

7. Inappropriate Service Intimacy: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Megaservice

A service that does a lot of things. A monolith.

8. Megaservice: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Local Logging

Logs are stored locally in each microservice, instead of using a distributed logging system. Errors and microservices information are hidden inside each microservice container. The adoption of a distributed logging system eases the monitoring of the overall system.

9. Local Logging: What is the possible effect of this smell on the quality of microservice architecture?

\*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Lack of Service Abstraction

The absence of service abstraction results in the interface reflecting the provider's model of interaction, rather than the consumer's model. The consumer's model is typically more aligned with the domain, simpler, and more abstract. When the provider's model is leaked into the interface, it constrains the evolution of the implementation

10. Lack of Service Abstraction: What is the possible effect of this smell on the quality of microservice architecture?

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Technical Microservice smells - Communications

Anti-patterns that are related to the communication between microservices.

### Cyclic Dependency

A cyclic chain of calls between microservices. E.g., A calls B, B calls C, and C calls back A. Microservices involved in a cyclic dependency can be hard to maintain or reuse in isolation.

11. Cyclic Dependency: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### ESB Usage

The microservices communicate via an Enterprise Service Bus (ESB). Usage of ESB for connecting microservices.

12. ESB Usage: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### No API-Gateway

Microservices communicate directly with each other. In the worst case, the service consumers also communicate directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance.

13. No API-Gateway: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Shared Libraries

Usage of shared libraries between different microservices, which tightly couples microservices together, leading to a loss of independence between them. Moreover, teams need to coordinate with each other when they need to modify the shared library.

14. Shared Libraries: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Timeout

This smell refers to the challenge of managing remote process availability and responsiveness in distributed architectures. Service availability is the ability to connect and send requests, while service responsiveness is the time taken to respond. Relying solely on timeout values for responsiveness can lead to the timeout antipattern, causing delays and potential failures. Proper management of service responsiveness and availability is crucial in distributed architectures.

15. Timeout: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Other Technical microservice smells

Microservice smells that do not fit in neither the previous groups (internal or communication).

### Lack of Monitoring

Lack of usage of monitoring systems, including systems to monitor if a service is alive or if it



16. Lack of Monitoring: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Shared Persistence

Different microservices access the same relational database. In the worst case, different services access the same entities of the same relational database.

17. Shared Persistence: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Wrong Cuts

Microservices should be split based on business capabilities, not on technical layers (presentation, business, data layers).

18. Wrong Cuts: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Organizational Microservice smells - Team-oriented

Anti-patterns that are related to team's dynamic.

19. Legacy Organization: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Non-homogeneous adoption

Only few teams migrated to microservices, and the decision if migrate or not is delegated to the teams. Duplication of effort. E.g. effort for building the infrastructure, deployment pipelines.

20. Non-homogeneous adoption: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Common Ownership

One team own all the microservices. Each microservice will be developed in pipeline, and the company is not benefiting of the development independency

21. Common Ownership: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Microservice Greedy

Teams tend to create of new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two staticHTML pages. This anti-pattern can generate an explosion of the number of microservices composing a system, resulting in a useless huge system that will easily become unmaintainable because of its size.

22. Microservice Greedy: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Magic Pixie Dust

Many enterprises are considering microservices architecture as a solution to their software delivery challenges. However, simply adopting microservices without addressing underlying root causes such as development process, organisation, and code quality can lead to anti-patterns. Microservices alone won't solve all development problems, and it's important to understand the prerequisites and problems that microservices architecture addresses.

23. Magic Pixie Dust: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Microservice as the goal

Forcing teams to adopt microservices without addressing inefficient processes, siloed organisation, or poor software quality can lead to negative consequences. Microservices adoption should not be seen as a one-size-fits-all solution and may not benefit an organisation that has underlying issues. Imposing microservices architecture without considering the application's specific needs may be counterproductive.

24. Microservice as the goal: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Pride

Testing in a microservice environment is challenging due to interactions between microservices. Adopting disposable, transient microservices can make testing difficult, but it can be mitigated with techniques such as service virtualisation and API simulation.

25. Pride: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Sloth

Neglecting Non-Functional Requirements and delaying decisions in microservices architecture can lead to issues. Identifying and addressing technical requirements early in the design phase is crucial. Security considerations, especially in containerised environments, should be prioritised and integrated into the build pipeline for thorough testing, including performance and load testing.

26. Sloth: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Organizational Microservice smells - Technology and Tool Oriented

Anti-patterns related to the technologies and tools used on the application.

### Focus on latest technologies

The migration is focused on the adoption of the newest and coolest technologies, instead of based on real. The decomposition is based on the needs of the different technologies aimed to adopt.

27. Focus on latest technologies: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Lack of Microservice Skeleton

Each team develop microservices from scratch, without benefit of a shared skeleton that would speed-up the connection to the shared infrastructure (e.g. connection to the API-Gateway).

28. Lack of Microservice Skeleton: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### No DevOps tools

The company does not employ CI/CD tools and developers need to manually test and deploy the system.

29. No DevOps tools: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

### Too Many Technologies

Usage of different technologies, including development languages, protocols, frameworks...

30. Too Many Technologies: What is the possible effect of this smell on the quality of microservice architecture? \*

*Marcar apenas uma oval.*

	0	1	2	3	4	5	6	7	8	9	10	
Not	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Extremely important

## Open-Ended questions about microservices

This section invites your input to brainstorm and identify potential new microservice smells, or issues in microservices architecture. Your contributions can help in expanding the understanding of possible warning signs or red flags in microservices development.

31. Are there any microservice smells that you would add to this catalogue?

---

---

---

---

---

## E-mail submission

If you would like to receive the final study via e-mail, please provide your e-mail address in the form below. We will ensure that you receive the study in your inbox as soon as it is available. Thank you for your interest!

32. Insert your e-mail

---