UiO : **University of Oslo**

Saulo Soares de Toledo

Managing Architectural Technical Debt in Microservices

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

2022

© Saulo Soares de Toledo, 2022

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2538

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without
permission.

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

To my wife, Emanuele, and my family.

Abstract

Background: Software systems must continuously evolve to meet new business requirements. A modular software architecture is key to facilitating the evolution of the system. Many software development organizations also require their software to be deployed on the cloud due to demands on scalability and availability. Microservices is an architectural style that allows the implementation of all these requirements. However, software architecture is prone to sub-optimal solutions, because of several factors such as time constraints, uncertainty, miscommunication, and the growing complexity of software systems. Such factors may lead to architectural technical debt (ATD). There are only a few studies about ATD in the context of microservices (MS-ATDs).

Objective: This work aims to understand what MS-ATDs are, including their costs and solutions, and investigate methods to support their quantification, prioritization, and management.

Method: The reported studies combined qualitative and quantitative research methods. We started with a multiple case study in seven large software development organizations to identify MS-ATDs, costs, and solutions. We then proceeded with an in-depth multiple case study in four large software development organizations on how a specific MS-ATD, the misuse of shared libraries, affects development agility. Next, we performed another multiple case study in three large software development organizations in the early stages of migration to microservices to understand how MS-ATDs occur during migration and how they can be prioritized. Finally, we conducted a quantitative case study in a large company before and after refactoring some MS-ATDs to understand how the refactoring affected the occurrence of incidents. We used incidents as a proxy of MS-ATD costs.

Results: Our results include a catalog of MS-ATDs, their causes, their solutions, a quantification of the debts' interest based on the number of incidents resulting from the debts, and an approach to prioritize MS-ATDs. Examples of MS-ATDs are the lack of communication standards among microservices, the misuse of shared libraries, and an excessive number of small products. We report negative effects of misusing shared libraries on development agility. We also presented suggestions of how software development organizations could deal

with this problem. Finally, we proposed a systematic prioritization approach for MS-ATDs based on factors such as the likelihood of their occurrence, the difficulty of their resolution, and their importance for the practitioners.

Conclusion: Software development organizations are still learning how to use microservices and pay a high interest due to the lack of experience with this architectural style. Such organizations can use our catalog to identify 16 different MS-ATDs, their costs, and solutions. They can also integrate our lightweight MS-ATD prioritization approach into their agile development processes. Researchers can find our results valuable in understanding MS-ATDs and contributing to reducing the current knowledge gap that leads to MS-ATD high costs. Finally, we quantified part of the MS-ATD interest using incidents. Such an approach might be helpful for companies and researchers when seeking new ways of measuring the debt's interest.

Acknowledgments

First and foremost, thanks to Almighty God for all His blessings in uncountable situations. Words like “coincidence” or “luck” cannot describe the numerous circumstances in which things worked out when I needed them, especially in the most challenging moments.

I also want to express my gratitude to the many people that made this work possible. I will not be able to put all their names here, but I would like to thank them all from this first beginning. This thesis will permanently engrave their contributions.

I want to express deep and sincere gratitude to my primary supervisor, Antonio Martini, for believing in me from the first day we met, in an online interview, until today. He has been a unique supervisor and always seeks novel and ground-breaking contributions to Software Engineering research. His effort made it possible to conduct this research as an interaction between academy and industry. This Ph.D. would never be possible without his support.

I want to thank Dag Sjøberg, my second supervisor. Dag has an impressive knowledge of research in software engineering. I am still trying to learn more about that knowledge from his papers, emails, and discussions. He has also been supportive from day one of our conversations, not only in my research and papers but far more in several other aspects of my life in Norway. I am deeply grateful for Dag’s support.

I also want to thank my wife, Emanuele Montenegro Sales, for her patience and support. She left many things behind to support me in this Ph.D. Nothing would have been possible without her help. She finished her Ph.D. during my first months in Norway, and I could not be present in her defense. However, she shared many of her experiences with her Ph.D., which made things easier for me.

A special thank you to Francisco Neto from the Chalmers University of Technology, Sweden, a person I am glad to have as a friend. I am not sure anymore when it was the first time we met. Still, I recall an interview for a project to work for a company through one of the Federal University of Campina Grande’s laboratories in Brazil. From that point forward, he helped with my master’s degree, which was in progress, and later sent me the link for applying to this Ph.D. position. We then had several long phone calls about

the Nordics and how it would be to have a Ph.D. here.

I also thank my parents and siblings for their support. First to my father, Severino, who wanted me to finish this Ph.D., but unfortunately passed away in 2020. I had the opportunity to meet him on his 93rd anniversary, yet in good health, and stayed with him until his last day. I thank him for his support and the many conversations we had. Then, I thank my mother, Zélia, for all her dedication and help making things work. She helped me from the most simple to the most complicated things before and during my stay in Norway, no matter how tired she was. And finally, to my siblings, Zeneide, Sérgio, Sílvio, Sidney, and Simões. They supported me in so many ways I can never repay.

Many thanks to all the professors and colleagues from the 10th floor in the Department of Informatics. Our many lunch days together made me feel comfortable in a new country. To mention a few: Victoria Stray, which was also part of the evaluation committee that interviewed me, Dag Langmyhr, and Stein Michael, who both brightened up my lunch hours with amazing conversations, Stein Krogdahl, passed away, which left me with memories of a knowledgeable and smiling person in every conversation we had, Yngve Lindsjørn, who I kept trying to listen to understand Norwegian, Paulo Ferreira, who shared many valuable ideas and information in my native language, Eric Jul, of whom I could never tell the difference in the Norwegian accent as highlighted by his colleagues, Siri Jensen, who had great patience in repeating Norwegian words so that I could understand them, Herman Jervell, a very attentive and knowledgeable person, Arne Maus, Geir Horn, Henrik Løvold, Gunnar Bergersen, Dino Karabeg, Eyvind Axelsen, Marthe Berntzen, Raluca-Madalina, Oleks Shturmov, Mahdieh Kamalian, and Lucas Paruch.

Also, thanks to Jan Bosch, director of the Software Center in Sweden, for allowing me to participate in several Software Center events and for his many insights during my mid-term evaluation. Thanks to Knut-Helge Rolland, passed away, the second reviewer of my mid-term evaluation, who also gave me valuable insights for my thesis. And thanks to Helena Holmström for the valuable feedback on the final version of this thesis.

Thanks to my other primary papers coauthors, Agata Przybyszewska, Johannes Skov Frandsen, and Phu Nguyen, for their support and contributions. And thanks to the many researchers who invited me to external collaborations. Starting with Wilhelm Hasselbring and Henning Schnoor from the University of Kiel, which invited me to a visit to that university. The many circumstances, including the unexpected Covid-19 outbreak, prevented us from working on our original plans. However, the feedback I received during that visit was helpful to my research efforts. And then to the professors and colleagues from the GroWDebt workshop, in which we coauthored an IEEE paper: Paris

Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Sas, and Angeliki Tsintzira.

Finally, thanks to all the contributors to this thesis who participated directly with interviews or indirectly by setting up contacts and meetings. And thanks to the many friends who helped make my life enjoyable outside work.

List of Publications

Included papers

Chapter 5

de Toledo, S. S. and Martini, A. and Sjøberg, D. I. K. 'Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study'. In: *Journal of Systems and Software* **177** (2021). DOI: 10.1016/j.jss.2021.110968.

Chapter 6

de Toledo, S. S. and Martini, A. and Sjøberg, D. I. K. 'Improving Agility by Managing Shared Libraries in Microservices'. In: Paasivaara, Maria and Kruchten, Philippe (Ed.), *Lecture Notes in Business Information Processing*. Springer Berlin/Heidelberg. **396** (2020), pp. 195–202. DOI: 10.1007/978-3-030-58858-8_20.

Chapter 7

de Toledo, S. S. and Martini, A. and Nguyen, P. H.
and Sjøberg, D. I. K. 'Accumulation and prioritization of Architectural Debt in three companies migrating to microservices'. In: *IEEE Access* **10** (2022), pp. 37422–37445. DOI: 10.1109/ACCESS.2022.3158648.

Chapter 8

de Toledo, S. S. and Martini, A. and Sjøberg, D. I. K. and Przybyszewska, A. and Frandsen, J. S. 'Reducing Incidents in Microservices by Repaying Architectural Technical Debt'. In: *Proceedings of the 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. (2021), pp. 196–205. DOI: 10.1109/SEAA53835.2021.00033.

Other papers

Avgeriou, P. C. and Taibi, D. and Ampatzoglou, A. and Fontana, F. A. and Besker, T. and Chatzigeorgiou, A. and Lenarduzzi, V. and Martini, A. and Moschou, A. and Pigazzini, I. and Saarimaki, N. and Sas, D. D. and de Toledo, S. S. and Tsintzira, A. A. (2021). An 'An Overview and Comparison of Technical Debt Measurement Tools'. In: *IEEE Software.* **38(3)**, (2021), pp. 61–71. DOI: 10.1109/MS.2020.3024958.

de Toledo, S. S. and Martini, A. and Sjøberg, D. I. K. 'Summary: Identifying architectural technical debt, principal and interest in microservices – A multiple-case study (short paper)'. In: *Companion Proceedings of the 15th European Conference on Software Architecture (ECSA).* Journal First Track. **2978** (2021). http://ceur-ws.org/Vol-2978/.

de Toledo, S. S. and Martini, A. and Przybyszewska, A. and Sjøberg, D. I. K. 'Architectural Technical Debt in Microservices: A Case Study in a Large Company'. In: Avgeriou, Paris and Schmid, Klaus (Ed.) *Proceedings of the Second International Conference on Technical Debt.* IEEE. (2019), pp. 78–87. DOI: 10.1109/TechDebt.2019.00026.

Contents

Abstract **iii**

Acknowledgments **v**

List of Publications **ix**

Contents **xi**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

2 Background and related work **3**
2.1 Technical Debt 4
2.2 Microservice Architecture 7
2.3 Technical Debt and the development with microservices . 12

3 Research Questions and the Thesis Studies **15**
3.1 Research studies addressing the definition of MS-ATDs (RQ1) 16
3.2 Research studies addressing the occurrence of MS-ATDs (RQ2) 20
3.3 Research studies addressing the prioritization of MS-ATDs (RQ3) 21
3.4 Research studies addressing the repayment of MS-ATDs (RQ4) 24

4 Research Methodology **27**
4.1 Research context 27
4.2 Case study research 30
4.3 Interviews 32
4.4 Document analysis 36

4.5 Validity and reliability 36

Papers **40**

5 Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study **41**
5.1 Introduction 42
5.2 Background 43
5.3 Methodology 54
5.4 Results 69
5.5 Discussion 83
5.6 Related Work 90
5.7 Conclusions and Future Work 92

6 Improving agility by managing shared libraries in microservices **95**
6.1 Introduction 96
6.2 Background 97
6.3 Methodology 97
6.4 Results 98
6.5 Discussion and Threats to Validity 103
6.6 Conclusions and Future Work 104

7 Accumulation and prioritization of Architectural Debt in three companies migrating to microservices **105**
7.1 Introduction 106
7.2 Background 110
7.3 Research design 117
7.4 Results and discussion 125
7.5 Limitations 160
7.6 Related Work 161
7.7 Conclusion 162

8 Reducing Incidents in Microservices by Repaying Architectural Technical Debt **165**
8.1 Introduction 166
8.2 Background 167
8.3 Methodology 175
8.4 Results 182
8.5 Discussion 188

8.6 Related work 190
8.7 Conclusions and future work 190

9 Discussion and Conclusions 193
9.1 Research questions and contributions of the thesis 193
9.2 Future work 198
9.3 Conclusion 199

Bibliography 201

Appendices 211

A Study 1 Interview guide 213

List of Figures

1.1 The topics discussed in this thesis. 2

2.1 The relationship between TD and ATD. 3
2.2 Microservice characteristics and migration topics. 9
2.3 The most expressive references among the ones mentioned in Sections 2.1 and 2.2. 14

3.1 The MS-ATD research gap and the research questions 16
3.2 Relationship between the RQs and the studies. 18
3.3 The MS-ATDs definition 19
3.4 The occurrence of MS-ATDs 22
3.5 MS-ATDs prioritization 23
3.6 The benefits of the MS-ATDs repayment 25

5.1 Monolithic and microservice architectures 44
5.2 Microservices synchronous communication 46
5.3 The API gateway 46
5.4 Microservices asynchronous communication 47
5.5 Service discovery, registry and services instances 48
5.6 The circuit breaker 49
5.7 ATD and the related concepts 52
5.8 Methodology overview 55
5.9 Transforming quotations into codes through open coding . . . 58
5.10 Identifying the relationship among debt, interest and principal . 61
5.11 A message carrying metadata 70
5.12 The benefits of tracking dependencies among services 71
5.13 Common interests and principals among the debts. 87

6.1 Shared libraries example 99
6.2 How to handle shared functionality 102

7.1 New ATD after migration to microservices 106
7.2 Research question and in the studied context 108

7.3 An overview of the research process. 119
7.4 Percentage of practitioners who voted for each answer regarding the debts found on each company 129
7.5 Values for the calculation of the ranking of MS-ATDs found for each company. 135
7.6 Percentage of practitioners who voted for each answer regarding the debts foreseen on each company 136
7.7 Values for the calculation of the ranking of MS-ATDs foreseen for each company. 142
7.8 Percentage of practitioners who voted for each answer regarding the debts practitioners know how to avoid on each company . 143
7.9 Values for the calculation of the ranking of MS-ATDs the practitioners know how to avoid for each company. 148
7.10 Importance of the MS-ATDs as perceived by the practitioners 149
7.11 Priority ranking normalized between 1 and 10 153

8.1 Reduction of incidents reducing the total interest 170
8.2 Example of solution to solve ATDs 173
8.3 Data collection overview. 177
8.4 Categories of incidents. 179
8.5 Incidents by category 185
8.6 Incidents per type distributed over time for both architectures. 186

List of Tables

2.1 The types of TD 6

3.1 Research questions. 17

4.1 Overview of the included publications, the companies involved, and the research methodologies. 27
4.2 Summary of interviews in this thesis by company. 33

5.1 Companies context 57
5.2 Type and number of interviews and interviewees by company . 60
5.3 Architectural Technical Debt identified on each company. . . . 62
5.4 Catalog of Architectural Technical Debts, interest and principal. 65

6.1 Issues reported by companies as the result of using shared libraries 98

7.1 The MS-ATDs selected for this study. 113
7.2 Attendees for the first presentation. 121
7.3 The practitioners' raw answers 126
7.4 Ranking of the most encountered MS-ATDs 130
7.5 Ranking of MS-ATDs foreseen in each company 137
7.6 Ranking of MS-ATDs that companies do not know how to avoid 141
7.7 Ranking of the most important MS-ATDs according to the practitioners 150
7.8 Priority rankings of the proposed MS-ATDs 154
7.9 The changes in the rankings when using different probabilities 158

8.1 Incidents by group and priority 183

A.1 Interview guide for Study 1. 214

Chapter 1

Introduction

Architectural technical debt (ATD) is present when architectural sub-optimal solutions lead to a benefit in the short term but increase the overall costs in the long run [MBC15]. Every software is prone to ATD: software development organizations may accumulate ATD when trying to accelerate the development of a software architecture or when an architecture is degraded due to architectural choices that were optimal in the past but are currently hindering architecture development [Ver+21].

Over the past years, we have seen a growth in the popularity of the microservices architectural style [Dra+17]. A microservice architecture consists of a suite of small and independent services working together and communicating through lightweight mechanisms [LF14]. This architectural style facilitates software scalability and reduces the length of testing, build and release phases, among other advantages [Fow15]. Such advantages made several software development organizations use microservice architectures in their solutions.

Still, architecting with microservices is not easy [DLM19]. Software development organizations are still learning how to use them properly [TLP20]. Many of those organizations are developing microservices from scratch, while others are migrating from other architectural styles. It is common to see success stories in workshops, conferences, and other events worldwide for both cases[1].

Like any architectural style, microservices are prone to ATD. However, the current literature on the topic does not define ATD in microservices, their causes, solutions, or how to avoid them (see the systematic mapping study about architecting with microservices by Di Francesco et al. [DLM19] and the systematic literature review about the management of ATDs by Besker et al. [BMB18]). Software development organizations also do not know how to deal with these debts.

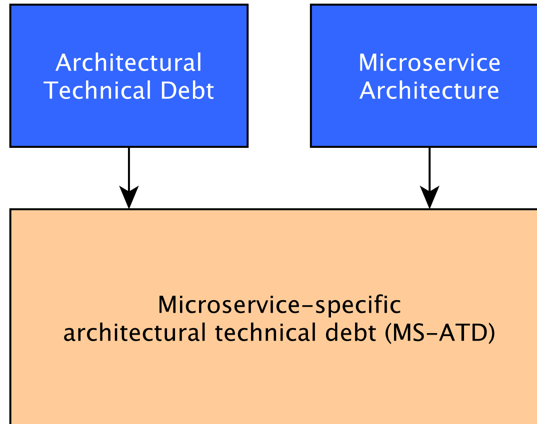
Moreover, microservices are distinct from other architectural styles. Specific characteristics lead to microservice-specific ATDs (MS-ATDs). Like any ATDs, identifying and managing MS-ATDs are essential to prevent software failure.

[1] Some examples are the presentations "Mastering Chaos" by Josh Evans (Netflix, 2016, available at https://youtu.be/CZ3wIuvmHeM), "Amazon and the Lean Cloud" by Werner Vogels (Amazon, 2011, available at https://vimeo.com/29719577), and "What We Got Wrong: Lessons from the Birth of Microservices" by Ben Sigelman (Google, 2018, available at https://youtu.be/-pDyNsB9Zr0).

This work investigates what MS-ATDs are, how they occur, and how to prioritize and repay them.

Figure 1.1 1 shows the main topics discussed in this thesis. The following chapters will expand each of these topics.

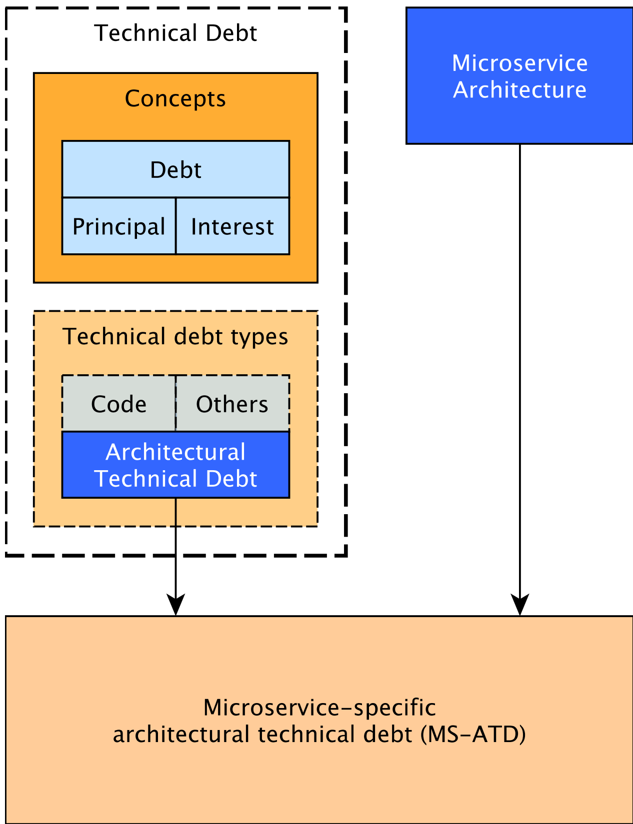
Figure 1.1: The topics discussed in this thesis.



Chapter 2

Background and related work

Figure 2.1: The relationship between TD and ATD.



Architectural technical debt (ATD) is a specific type of technical debt (TD). Figure 2.1 presents the relationship between TD and ATD, as well as related concepts.

2.1 Technical Debt

The first mention of "technical debt," or TD in short, dates back to 1992, in a report written and presented by Ward Cunningham at the *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA)* [Cun92]. Cunningham described TD as a metaphor to explain the fragile balance between making decisions with short-term benefits and their consequences: an increase in long-term software development costs. Since then, researchers and industry practitioners worldwide have used the metaphor. The first research studies on TD emerged in the early 2000s and the more significant ones after 2010 [CLM21].

Several years after the original definition of TD, in 2007, Steve McConnell wrote the first most noticeable refinement of the metaphor. He defined two types of TD: the *unintentional*, which happens due to low-quality work and the *intentional*, caused by strategic decisions. Later, in 2009, Martin Fowler defined the "Technical Debt Quadrant," classifying TD as *deliberate* or *inadvertent* and *reckless* or *prudent*. The definition by McConnell and the Technical Debt Quadrant by Fowler expanded the original interpretation given by Cunningham, given space to interpret (*intentional* or *unintentional*, *deliberate* or *inadvertent*, and *reckless* or *prudent*) TD also as part of an overall investment strategy to speed up software development.

Several years later, in 2016, the Dagstuhl Seminar 16162 "Managing Technical Debt in Software Engineering," came up with the most recent and well-used definition of TD [Avg+16]:

> *In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.*

Essentially, the TD metaphor is composed of three main components:

- The *debt* is a contingent technical issue or sub-optimal implementation that might or not incur intentionally. Practitioners might intentionally take the debt to accelerate the development process. Alternatively, the

debt might emerge due to technology degradation when an acceptable solution in the past is not satisfactory anymore. One example of debt is skipping the support for scalability in a web development platform due to time-to-market constraints.

- The *interest* is the cost of taking the debt. Following the previous example of debt, part of the interest could be the cost for handling maintenance issues and potential loss of clients due to current platform limitations, which cannot scale to support a higher number of clients.
- The *principal* is the cost of developing the optimal architecture from the beginning or the cost of repaying the debt. In the example above, the principal is the cost of rewriting the software to be scalable later, such as costs due to effort spent on extra development and learning technologies.

Many have contributed to the state of the art of technical debt since the definition of the term [CLM21]. Examples are the TD landscape by Kruchten et al. [KNO12], which separates the concept of TD from other issues such as code smells and coding style violations, the TD categorization by Li et al. [LAL15], and the accumulation model of architectural TD by Martini et al. [MBC15].

2.1.1 Types of Technical Debt

Over the years, the research on TD classified the debts into distinct types. Each type applies to different software artifacts, such as code, architecture, and requirements. Alves et al. [Alv+14] and Li et al. [LAL15] are two secondary studies that systematically identified TD types in the research literature. They identified 13 and 10 distinct types of TD, respectively, as shown in Table 2.1.

Some TDs were studied more than others over the years. Li et al. [LAL15] shows that code TD is the most cited type in the research literature (40% of the studies assessed), followed by ATD (27% of the studies assessed). The other types of debt are spread throughout the remaining 33% of the studies assessed). Alves et al. [Alv+16] inferred that a plethora of tools for source code analysis might explain the high amount of studies on code TD. Other types of TD do not have such an abundance of tool support. ATD is the second most studied TD type. However, answering how to identify, monitor, and manage ATD is still an open research question [Ver+21]. There is increasing evidence that ATD is the most challenging type of debt to handle [KNO12]. This thesis assesses the management of ATDs in microservices, as described in Section 3.

On the other hand, Service Debt is the least studied type of TD, with a single study on the topic by Alzaghoul and Bahsoon [AB13] as identified by

Table 2.1: The TD types according to Alves et al. [Alv+14] and Li et al. [LAL15].

Debt	**Identified by**		**Refers to**
	Alves et al.	**Li et al.**	
Architectural Debt	X	X	Sub-optimal architectural decisions.
Build Debt	X	X	Difficulties during the software building phase.
Code Debt	X	X	Issues found in the software source code.
Defect Debt	X	X	Defects found in the source code, usually identified by test activities.
Design Debt	X	X	Issues at the design level, such as complex classes and methods or code smells.
Documentation Debt	X	X	Lack of system documentation.
Infrastructure Debt	X	X	Suboptimal infrastructure decisions.
People Debt	X		People and socio-technical decisions.
Process Debt	X		Inefficient processes.
Requirements Debt	X	X	Inconsistencies between the actual implementation and its optimal requirements.
Service Debt	X		Issues regarding web service selection and composition in cloud-based architectures.
Test Automation Debt	X		Issues with the test automation functionalities.
Test Debt	X	X	Poor testing practices.
Versioning Debt		X	Incorrect source code versioning management.

Alves et al. [Alv+14]. At first hand, this type of TD seems related to the topic of this thesis. However, the authors studied non-microservice SOA applications. They investigated a scenario in which architects want to replace pieces of their software with web services available in the cloud market. Different vendors may have different but similar web services, and the architects have to decide which one to use. According to the authors, selecting one of the web services might incur TD. They propose an approach to support practitioners to decide what web service to use. In this scenario, their applications might be monoliths consuming web services. Furthermore, the term "web services" itself was strictly defined as using specific technologies not commonly used in microservices (see

the definition of web services by the W3C Working Group[1]).

2.1.2 Architectural Technical Debt

Software architecture plays a significant role in the development of large systems [KNO12]. Therefore, it is important to manage ATDs. Examples of ATDs are incorrect or not well-defined data structures, unexpected dependencies among components, pieces of the architecture that do not satisfy the product requirements (e.g., scalability), and misplaced functionality (e.g., business logic in the software UI) [MB17].

Several tools can help with TD, and some of them expose architecture-related issues such as coupling and circular dependencies [Avg+21]. However, such tools primarily rely on analyzing the software source code [BMB16]. Therefore, they cannot capture whether architectural decisions, for example, concerning the source-code framework used to build the application or database technology in a particular setup lead to debts.

ATD might emerge from decisions made in previous phases of software development. For example, frameworks chosen several years earlier might not support newer scalability requirements or migration to the cloud. Modifications in the business context, time pressure, or lack of understanding about a system's future business requirements might be causes for consequential changes in the system requirements and, thus, new ATD. Regardless of the causes, ATDs may be costly and require management to prevent the system from being unmaintainable [MBC15].

2.2 Microservice Architecture

The microservice architectural style requires breaking down applications into smaller, independent pieces. Each piece, called a microservice, provides independence among teams, improved maintainability, and scalability. Over the years, several authors proposed definitions for microservices. Di Francesco et al. [DLM19] identified at least 27 authors using their own or informal definitions of the term. However, the most used definition is the one provided by Lewis and Fowler [LF14] in 2014: "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."

Microservices are born in the industry. They were in use years before that definition in 2014. According to Newman [New19], Lewis spotted the originally

[1] https://www.w3.org/TR/ws-arch/wsa.pdf

called “micro-apps” in a few companies to which he had access yet in 2011. After that, “micro-apps” became a topic under discussion in a few circles. Later, they were renamed “micro-services” (as one can still find them named in some publications), and finally “microservices” [New19]. Sometime later, the most well-accepted definition of the term was born.

In the subsequent years, microservices were better developed and adopted and became a matter of interest by many companies worldwide. Many of these companies gradually moved out from SOA-specific solutions in favor of microservices. Not all these companies managed to have pure microservice architecture, so it is somewhat common to see architectures that mix microservices and other SOA-oriented solutions. According to Di Francesco et al. [DLM19], the number of research papers on microservices increased considerably after Lewis and Fowler’s [LF14] definition of the term.

Microservices are becoming increasingly popular and are frequently described as an alternative to monolithic applications, which are built and deployed as a single unit. However, microservices also bring some challenges, such as the risk of increased data inconsistency and operational complexity [Fow15]. Figure 2.2 highlights the topics discussed in the remainder of this section.

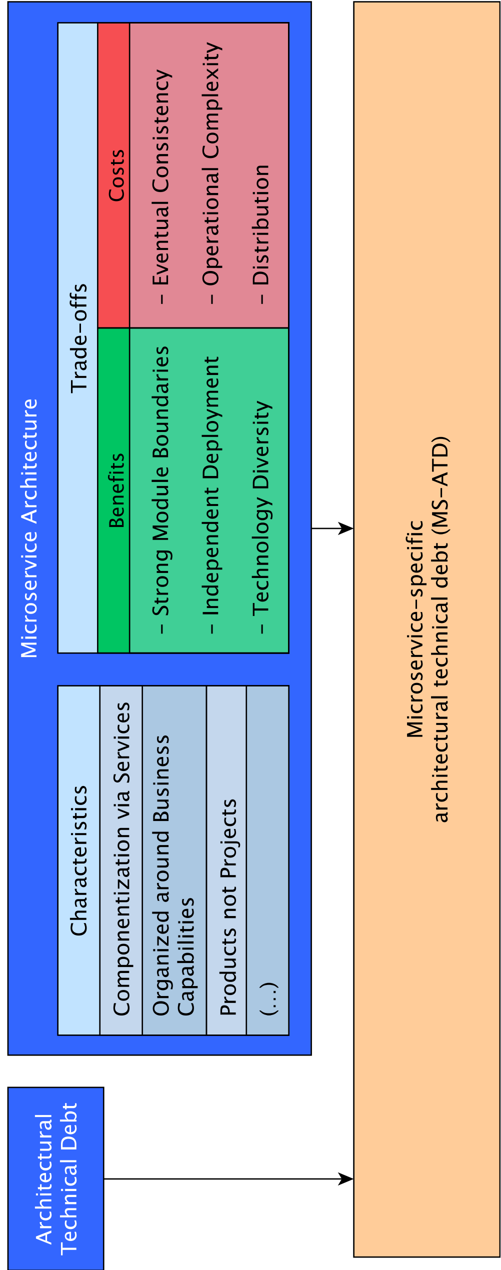


Figure 2.2: Microservice characteristics and migration topics.

2.2.1 Microservices characteristics

Lewis and Fowler [LF14] proposed a set of characteristics to describe microservices. According to them, not all microservice architectures have all the proposed characteristics, but they should exhibit most. We present those characteristics below:

- **Componentization via Services:** A component is a unit of software that can be updated or replaced independently. Examples are libraries that can be linked to the running software. Although microservices architectures can use libraries, their primary way of componentizing is through services. These services are independently deployable, and usually do not require changes in other services.
- **Organized around Business Capabilities:** The microservice architecture is organized around atomic business functions, reducing dependencies among components. Since each development team is independent, they can develop their software at their own speed.
- **Products not Projects:** Many software development organizations create software using a project model, in which a team works on the project until its delivery. After that, the development team hands it over to a maintenance team. In contrast, microservices shift the focus from the project to the product. Teams working with microservices are responsible for the whole lifetime of the product, including its maintenance and evolution. Other software architectures can benefit from this approach, but microservices are smaller, making it easier for teams to take ownership of the services.
- **Smart endpoints and dumb pipes:** The mechanism used for communication among services should not contain logic, acting as "dumb pipes." All the logic should be moved to the endpoints, i.e., the microservices themselves.
- **Decentralized Governance:** It is common on centralized governance models that everything is standardized to use a single technology platform. On the other hand, Microservice development teams prefer to take advantage of different standards and technologies.
- **Decentralized Data Management:** Microservices prefer letting each service manage its own database, which facilitates that conceptual models

of the world differ between systems: "users" in one department may be called "vendors" and "authors" in another department. Such decentralized data management facilitates independence among services and teams, but it also introduces some challenges, such as difficulties to manage distributed transactions when needed.

- **Infrastructure Automation:** Microservices benefit significantly from the use of infrastructure automation techniques. As the number of services grows, it is increasingly necessary to have good infrastructure automation.
- **Design for failure:** Microservices should be designed for failure because it is increasingly likely that one or more services become unavailable for a number of reasons as the number of services grows.
- **Evolutionary Design:** Microservices can be created or extinguished due to business needs. The microservice architecture should allow the replacement, upgrade, or removal of services.

2.2.2 Microservices trade-offs

Microservices provide several benefits, but they have trade-offs. Fowler [Fow15] proposed the following benefits:

- **Strong Module Boundaries:** If well-defined, microservices have strong module boundaries and reinforce the modular structure of the product, facilitating the management of microservice projects, especially the larger ones.
- **Independent Deployment:** It is easy to deploy independent services. They are autonomous and usually do not require deep knowledge of other services to be deployed. If the architecture is well-defined, services that go wrong after deployment should not affect the remainder of the system.
- **Technology Diversity:** Microservices allow developers to mix multiple programming languages, development frameworks, communication methods and protocols, databases, and other involved technologies.

However, those benefits have costs [Fow15]:

- **Operational Complexity:** The overall architecture is complex and must be adequately managed. The company must have a mature operations team to help with such management.

- **Eventual Consistency:** Distributed systems frequently rely on decentralized data. It is common to have different databases with data that is related, so the databases need to be synchronized. Synchronization is needed when one of the databases receives the data before the others. Eventually, all databases will be updated, and there will be consistency in the data. Eventual consistency might be beneficial to some projects. However, in many cases, such as when migrating from systems with strong consistency, maintaining the desired consistency in the data may be too costly.

- **Distribution:** Microservices are distributed systems, which brings many challenges such as handling failures on remote calls and additional latency.

2.3 Technical Debt and the development with microservices

Software development organizations adopt microservices aiming to work more efficiently and gain all the benefits this architectural style provides. However, architecting with microservices is not easy. It requires dealing with many challenges, such as the management of network latency, data consistency, and fault tolerance [DLM19]. There are trade-offs to be considered by practitioners (see Section 2.2.2).

Moreover, it is clear from research and practice that every software architecture is prone to ATD, and that is not different for microservices. Some ATDs are specific to microservices due to this architectural style's unique characteristics. Therefore, managers, architects, developers, and other roles must understand and manage the repayment of microservice-specific ATDs (MS-ATDs) to develop their software architecture. Consequently, ATD management is vital to increase the software life expectancy since ATD accumulation may lead to a development crisis [MBC15].

Usually, Product Owners (POs)[2] and architects[3] (or equivalent roles, depending on the software development method adopted by the company)

[2] The Product Owner (PO) is the customer's representative in Scrum [SS20], an agile framework. There are equivalent roles in other agile frameworks, such as the on-site customer in eXtreme Programming (XP) [UKS19]. Among their functions, POs should prioritize the work to be done by the development teams.

[3] Software architects are responsible for the technical solutions that satisfy the business requirements and specific system qualities [McB07]. Examples of such qualities are scalability, reusability, and responsiveness. As such, their architectural design may lead to more or less ATD [LLA15].

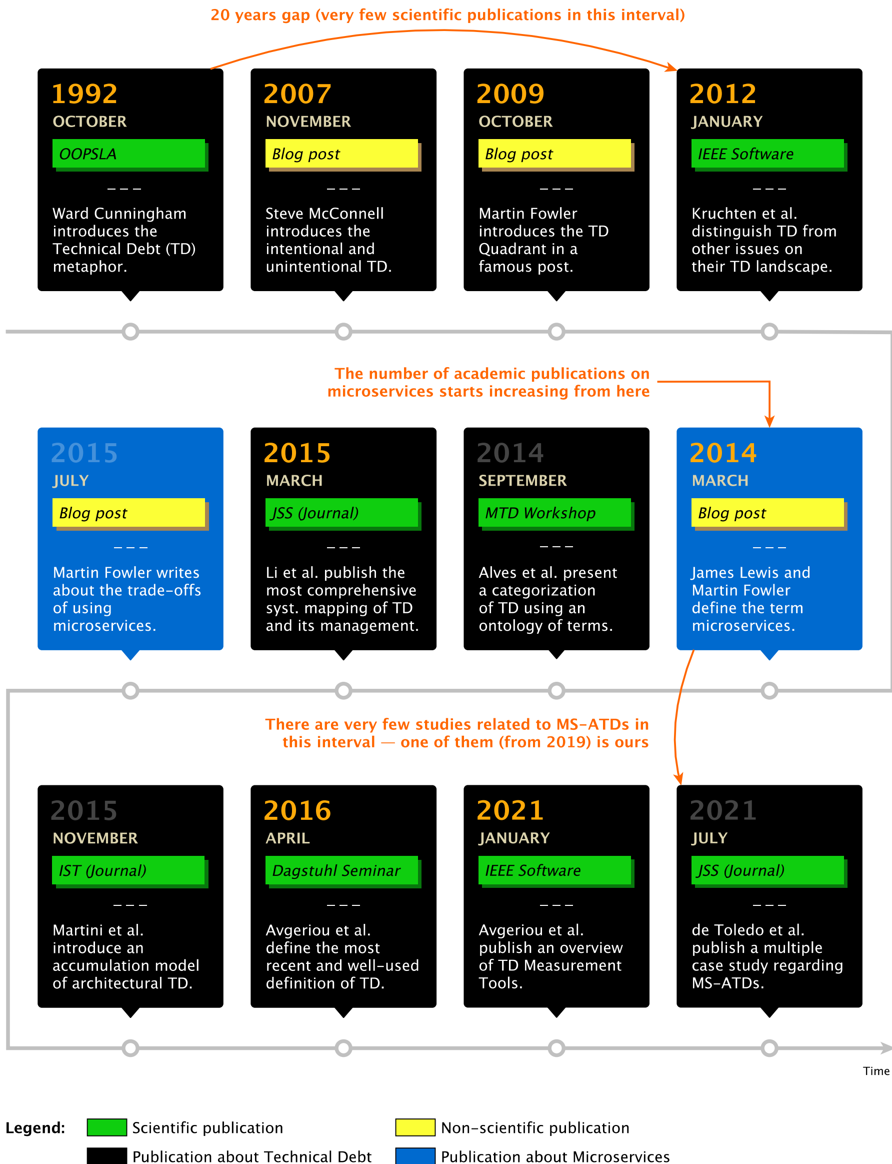
are the ones responsible for prioritizing feature development and ATD repayment [MB15]. Although knowing that ATD must be repaid and requires management, such prioritization-responsible practitioners frequently down-prioritize ATD repayment because of feature prioritization [MB15]. One reason for such down-prioritization is that such roles use individual experience or intuitive judgment to decide about ATD repayment, which makes it hard to justify the decision for the management [MB17]. Another reason is that they do not know the costs of the debts in the long run or that the debts exist. Time pressure, reuse of legacy software, and lack of documentation might contribute to this accumulation of debts [MBC15]. Therefore, POs, architects, and equivalent roles need more information for practical prioritization of the ATDs [MB15]. Nevertheless, as we will discuss next, the lack of knowledge about MS-ATDs in the research literature indicates a research gap that requires investigation.

Software developers are less involved in the prioritization than roles such as POs and architects. However, they are responsible for the implementations that may cause or repay ATD. Therefore, they also need more information about ATDs and related costs to avoid or repay them properly.

Both ATD and microservices are reasonably recent areas of study; see Figure 2.3 for reference. Despite being introduced in 1992, blog posts and non-scientific discussions were the most relevant remarks regarding ATD in the first 20 years. Only after that did TD become a topic under discussion in pertinent research circles [CLM21]. Concerning ATD, one specific type of TD, the lack of tools to identify it further narrowed the research efforts during these years [CLM21]. On the other hand, microservices as a research topic started to become popular a few years later, in 2014 [DLM19].

Due to the recent nature of both areas, previous research has not addressed MS-ATDs satisfactorily. Researchers on microservices addressed complexity, low flexibility, security needs, and other concerns [DLM19], but not from an ATD point of view. Still, they lack in-depth discussions regarding when such concerns are debts, when they are acceptable, and when they require repayment. For example, two microservices might be highly dependent on each other. The high coupling among those might be against the microservice definition as independent services. However, this coupling might never be debt and never cause problems or undesired costs to the software. On the other hand, researchers on ATD have not yet started investigating MS-ATDs in recent studies [BMB18]. Therefore, there is a gap in research revealing missing knowledge regarding MS-ATDs that would benefit practitioners and researchers.

Figure 2.3: The most expressive references among the ones mentioned in Sections 2.1 and 2.2.



Chapter 3

Research Questions and the Thesis Studies

As stated in Section 2.3, architecting with microservices is not easy and is prone to MS-ATDs. If not managed, MS-ATDs may accumulate and lead to a development crisis. Therefore, managing MS-ATDs is vital to increasing the software life expectancy. In their systematic mapping study, Li et al. [LAL15] described a set of activities for TD management found in the research literature. Those activities include identifying and prioritizing debts.

Despite the importance of managing MS-ATDs, the previous research literature does not describe what MS-ATDs are, how they occur, or how to prioritize or repay them (see Figure 3.1 and Section 2.3). Practitioners trust their individual experience or intuitive judgment to decide on debt repayment. However, complex architectures involving many microservices might make this task difficult.

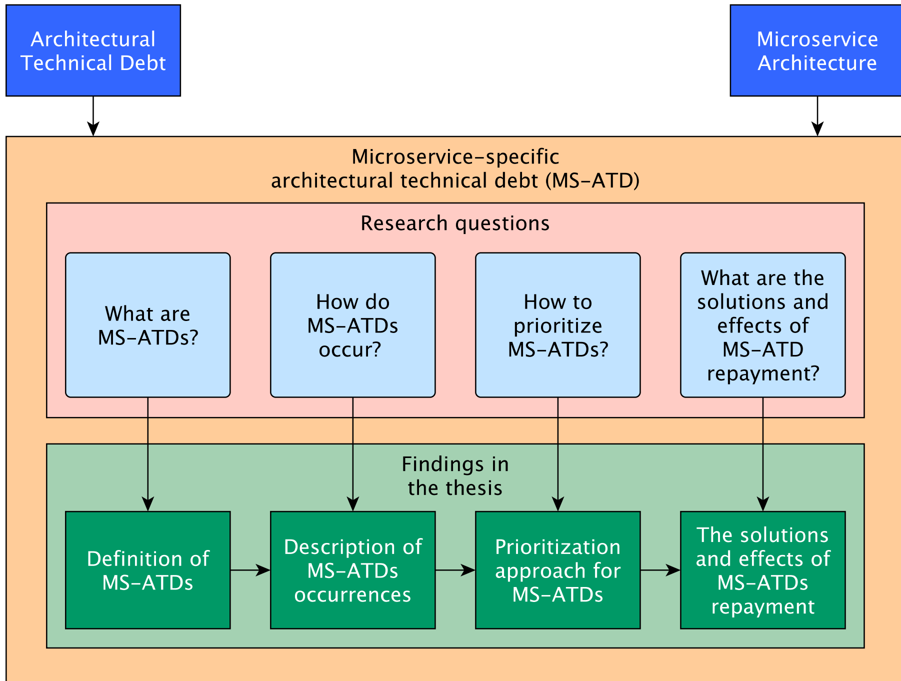
We started searching for previous research regarding MS-ATDs in the systematic mapping study by Di Francesco et al. [DLM19] and the systematic literature review by Besker et al. [BMB18]. Later, we performed a lightweight literature review for the years not covered by those studies. As we did not find any relevant contributions to this research topic, we proposed four main research questions (RQs) and eight subquestions, presented in Table 3.1.

Microservices introduced new ways of thinking about how to implement the software architecture. Unique characteristics such as the focus of each microservice on single business functionalities and the need for lightweight communication mechanisms lead to specific ATDs not found in other architectures. We call them MS-ATDs.

As shown in Figure 3.1, previous research literature on MS-ATD does not describe what MS-ATDs are, how they occur, or how to prioritize or repay them. A recent systematic mapping study about architecting with microservices [DLM19] did not find research on ATD. Another recent systematic literature review regarding the management of ATDs [BMB18] also did not find contributions related to microservices. We performed a lightweight literature review for the years not covered by those studies looking for research papers on the area and did not find any relevant contributions. Therefore, we proposed

four main research questions (RQs) and eight subquestions, presented in Table 3.1.

Figure 3.1: The MS-ATD research gap and the questions answered in this thesis.



This Ph.D. thesis presents four research studies named 1 to 4. Each research study addresses fully or partially one or more RQs. Figure 3.2 provides an overview of each RQ and the studies. We describe the general findings for each RQ in Chapter 10. The remaining of this section describes how each study answers each RQ in more detail.

3.1 Research studies addressing the definition of MS-ATDs (RQ1)

The first main research question (RQ1) aims to understand what MS-ATDs are, including their consequences and solutions, from distinct perspectives in

Table 3.1: Research questions.

Main Research Question (RQ) ID	**Sub-question ID**	**RQ description**
1	What are MS-ATDs?	
	1.1	What are the most critical MS-ATDs?
	1.2	What is the negative impact of such MS-ATDs?
2	How do MS-ATDs occur?	
	2.1	How do MS-ATDs occur in early-stage microservices?
	2.2	How do MS-ATDs occur in mature microservice systems?
3	How to prioritize MS-ATDs?	
	3.1	Which MS-ATDs do practitioners consider risky?
	3.2	Which methods can companies use to prioritize the avoidance or repayment of MS-ATDs?
4	What are the solutions and effects of repaying or avoiding MS-ATDs?	
	4.1	What are possible solutions to repay or avoid MS-ATDs?
	4.2	What are the effects of repaying MS-ATDs on their interest?

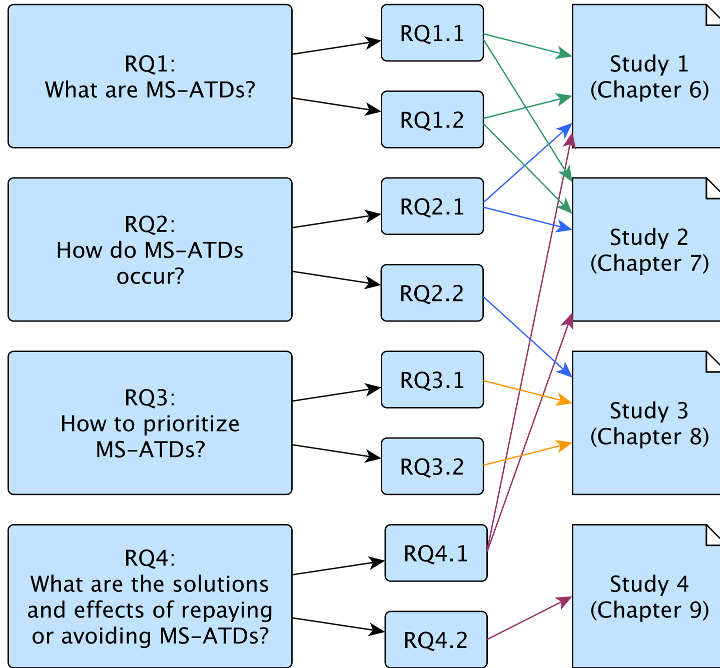
the software industry. Figure 3.3 6 illustrates how we answer RQ1 by defining MS-ATDs and is explained in the remainder of this section.

MS-ATDs are a subset of ATDs specific to the microservice architecture. As any TD, MS-ATDs have interest and principal. There are no differences between ATDs and MS-ATDs apart from the fact that the latter are instances of ATDs that are not found on other architectures or, at least, not precisely in the same way as seen in microservices. To define MS-ATDs concretely, we created a catalog of them with their description, including their interest and principal.

The following examples illustrate how MS-ATDs are different from debts in other architectural styles:

Coupling among services may emerge on other Service-Oriented Architectures (SOAs). However, microservices are smaller because they focus on a single task each. There is no such limitation in SOA, in which a service can easily do

Figure 3.2: Relationship between the RQs and the studies.



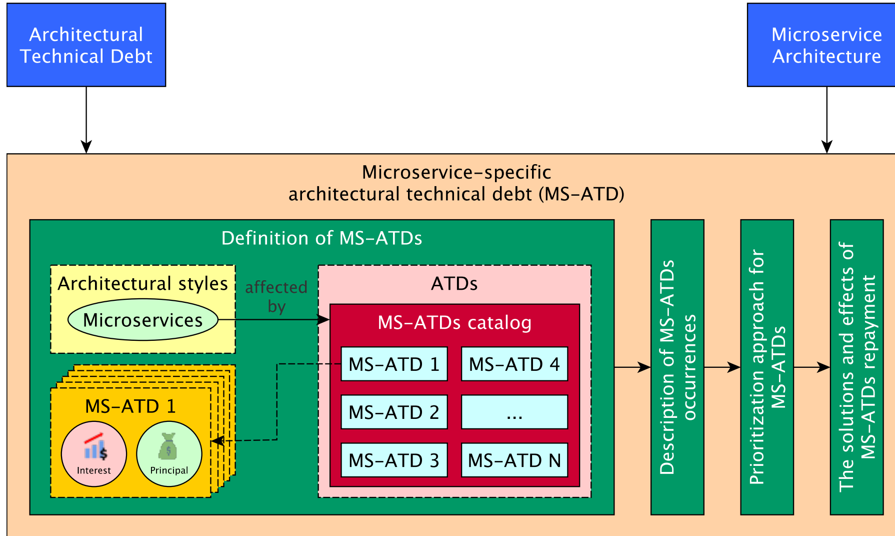
multiple jobs. The equivalent in microservices of the functionality provided by a single SOA service is usually composed of many smaller services working together. Therefore, more services are involved in a microservice architecture than in equivalent SOA architectures. A more significant number of independent services also raises the probability and costs of coupling. Therefore, the costs with coupling are different for each architecture.

The communication among microservices should use dumb pipes, i.e., no business logic in the communication layer. On the other hand, other SOA approaches accept and even promote logic in the communication layer for implementing data transformation capabilities, for example.

Both monolithic and microservice architectures may use shared libraries. However, a monolith is a single application, while microservices are many smaller independent applications. It is easier to upgrade the version of a library in a single application than in many. Microservice applications are easily composed of dozens to several hundreds of services, increasing the costs of such

upgrades compared to monoliths.

Figure 3.3: Answering RQ1 by defining MS-ATDs, a subset of ATDs specific to the microservices architecture and identifying their interest and principal.



The first main research question (RQ1) aims to understand what MS-ATDs are, including their consequences and solutions, from distinct perspectives in the software industry. We address this research question in Studies 1 and 2:

- *Study 1 (Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study)*: In this study, we survey MS-ATDs in seven large companies running microservices. We create a catalog of the debts, the costs for taking them (interest), and solutions (principal).

- *Study 2 (Improving agility by managing shared libraries in microservices)*: In this study, we dive deep into one of the debts identified in Study 1 to understand the impact on development agility.

3.2 Research studies addressing the occurrence of MS-ATDs (RQ2)

Starting a microservice architecture from the beginning is different from migrating from an existing architecture. When migrating from existing architectures, software development organizations frequently need to maintain both the original and the new ones running until the migration is complete, and sometimes they even keep part of the old architecture. On the other hand, starting the implementation directly with microservices does not require this effort. Therefore, there are differences in how MS-ATDs occur in each situation.

Yoder and Merson [YM20] investigated migrating approaches from monolithic architectures to microservices. They stated that one of the first decisions companies running such migrations have to make is completely rewriting the monolith or gradually migrating functionalities to microservices. In the last case, some functionalities remain in the monolith until migrated. Solutions that practitioners would consider MS-ATDs, e.g., sharing databases with the original implementation, may be a necessary step for the migration. Therefore, the MS-ATDs during migration may differ from those in a complete rewrite of the software architecture.

Furda et al. [Fur+18] named a gradual migration to microservices as "incremental modernization." This kind of migration poses a unique situation where both the original and the new architectures coexist and work together. The microservices may have to communicate with the legacy software using suboptimal approaches because the legacy application may not have the proper support for using the right approach. This situation raises a discussion about whether the suboptimal communication is an MS-ATD because it is also a necessary step to the migration, especially considering that the practitioners should remove the suboptimal communication approach once they conclude the migration. Sometimes, having a temporary debt is acceptable or the right choice for the project.

As presented in Figure 3.4, the final microservices architecture accumulates MS-ATDs already in early stages, either from a gradual migration or from a complete rewrite of the architecture. However, MS-ATDs may accumulate differently in different contexts. We, therefore, investigated how MS-ATDs accumulate in these different contexts in Studies 1, 2, and 3 as follows:

- *Study 1 (Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study)*: This study investigates how the MS-ATDs occur in seven large companies running microservices. We

describe the context of the companies and discuss the debts for each company.

- *Study 2 (Improving agility by managing shared libraries in microservices)*: The definition of microservices does not discuss the use of shared libraries. However, large companies make extensive use of them. Study 1 found that misusing shared libraries may be an MS-ATD related to issues such as breaking changes and dependencies among teams. The companies from Study 1 that discussed the problem also reported how this debt affected the development agility. Study 2 dives deep into the situation to examine the occurrence of the misuse of shared libraries by the involved companies.

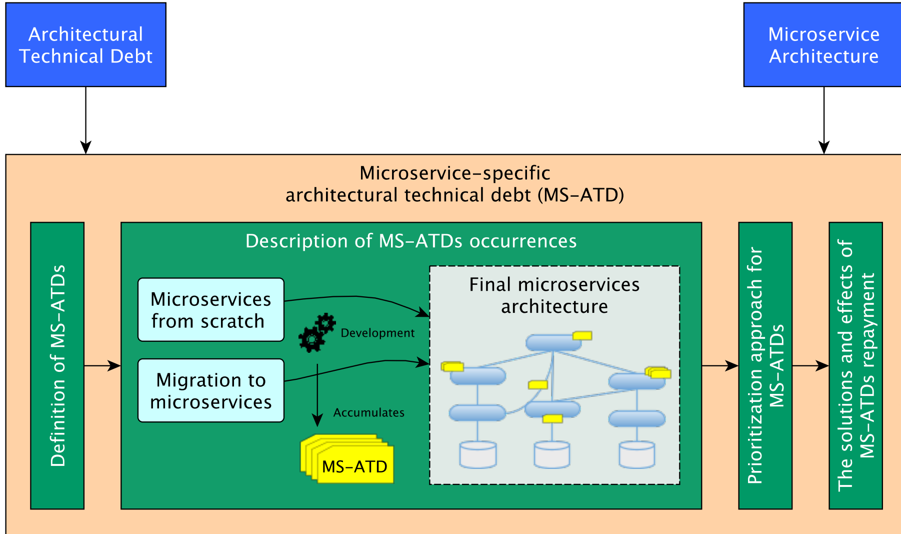
- *Study 3 (Accumulation and prioritization of Architectural Debt in three companies migrating to microservices)*: Studies 1 and 2 only investigate companies with mature microservice architectures. Study 3 complements them by presenting the occurrence of MS-ATDs in companies incrementally migrating to microservices.

3.3 Research studies addressing the prioritization of MS-ATDs (RQ3)

Companies remain competitive by constantly delivering value to their customers. Software products provide value by meeting their consumer's demands. Still, the quantity of customer requirements that a software development organization could implement in their software is frequently more extensive than the personnel available to do the job. Additionally, several non-functional requirements are also critical to the software, even though many stakeholders ignore or down prioritize them at some point. Thus, software development team members must correctly prioritize functional and non-functional requirements if they want to deliver value to customers continuously.

MS-ATDs are not requirements, but they need management. But they are invisible and thus hard to control. Fortunately, researchers invested time in finding better ways to manage TD over the years. In 2011, for example, Guo and Seaman [GS11] proposed an initial portfolio approach in which they identify TDs as *items*, allowing visualization and tracking. Following this idea, MS-ATDs can be prioritized together with other requirements. POs, architects, and related roles have *what* to prioritize, but now they lack information on *how* to do it.

Figure 3.4: How MS-ATDs occur, either when developing microservices from scratch or migrating from previous architectures. The final microservices architecture contains the debts accumulated during the development of the microservices.

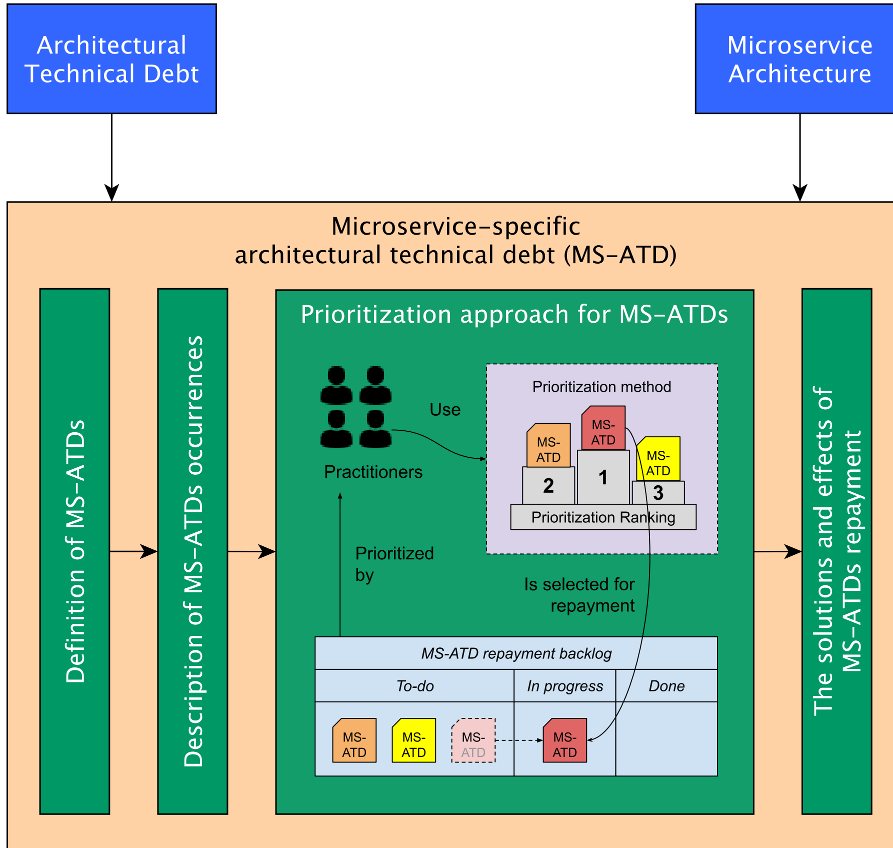


MS-ATD prioritization is complex because different MS-ATDs have different impacts in distinct contexts. For example, one project might depend much more on asynchronous communication approaches than another. MS-ATDs affecting asynchronous communication approaches are more impactful in the former context than in the latter. Therefore, MS-ATDs demand prioritization according to their specific context before repayment.

Practitioners involved in the MS-ATDs prioritization must know the ATDs' impact to be able to prioritize them [Mar+18]. It is however a hard problem to quantify the impact of debts. Practitioners frequently resort to informal approaches to determine such impact [Ver+21].

Researchers have been trying to discover techniques to prioritize ATDs. Martini et al. [Mar+18] proposed a systematic approach for prioritizing ATDs through architectural smells. However, the tool used (Arcan) can only analyze a small subset of ATDs identified through architectural smells in the source code, and the debts analyzed are not related to microservices. As described in Studies 1 and 2, we identify most MS-ATDs qualitatively, and tools such as

Figure 3.5: MS-ATDs prioritization. Having an appropriate prioritization method and a backlog of debts, practitioners may prioritize the repayment of MS-ATDs.



Arcan cannot identify nor help to prioritize them. There is a need to create prioritization methods for MS-ATDs.

Our third research question (RQ3) aims to investigate new ways of prioritizing MS-ATDs. Study 3, *Accumulation and prioritization of Architectural Debt in three companies migrating to microservices*, addresses RQ3 by proposing an approach for prioritizing MS-ATDs that is not dependent on tools. As presented in Figure 3.5, practitioners may use our prioritization approach to

select the MS-ATDs to prioritize repayment.

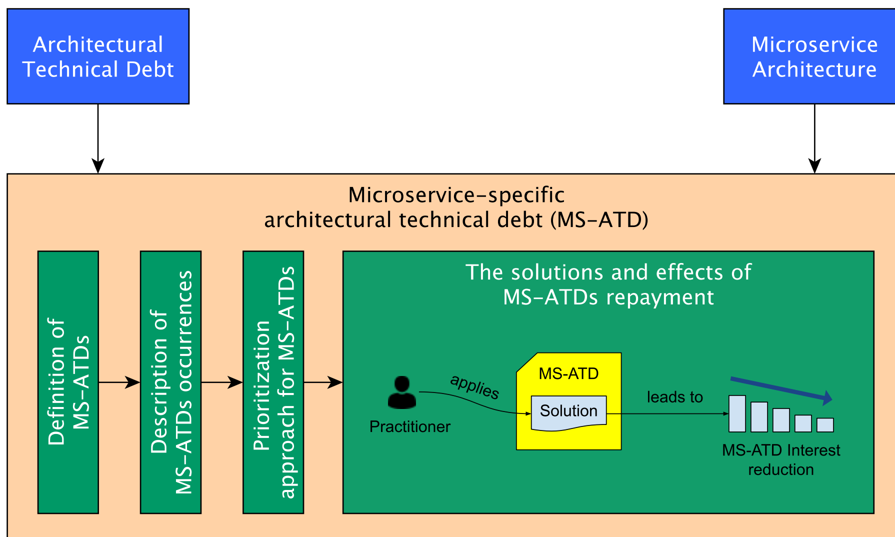
3.4 Research studies addressing the repayment of MS-ATDs (RQ4)

ATD repayment concerns eliminating or mitigating the negative costs of a specific ATD [LLA14]. An ATD may not be repaid at once because of the costs involved in such repayment. Therefore, there are cases in which an ATD is partially repaid.

As shown in Figure 3.6, repaying MS-ATDs involves identifying solutions for the debt. The solution should lead to a reduction of the debt's interest, and the costs saved by avoiding interest should usually justify the costs of repayment [MB16b]. We address this research question in Studies 1, 2, and 3:

- *Study 1 (Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study)*: This study identifies solutions for MS-ATDs repayment from seven large companies running microservices.
- *Study 2 (Improving agility by managing shared libraries in microservices)*: This study dives deep into the use of shared libraries in microservices. It addresses how to repay the related MS-ATD to improve agility in developing microservices with proper debt management.
- *Study 4 (Reducing Incidents in Microservices by Repaying Architectural Technical Debt)*: This study investigates the impact and benefits of MS-ATD repayment in a case study in a large financial services company.

Figure 3.6: The benefits of the MS-ATDs repayment. Practitioners may apply solutions to repay MS-ATDs. That leads to benefits that reduce the total interest in the project.



Chapter 4

Research Methodology

This thesis aims to understand MS-ATDs, that is, how they occur and how they can be prioritized and repaid. The thesis includes four studies in ten large European companies named from A to J, summarized in Table 4.1 together with the research methods and techniques used.

Table 4.1: Overview of the included publications, the companies involved, and the research methodologies.

Study	Paper Title (chapter)	Companies involved	Research Method	Research Technique
1	Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study (chapter 6)	A, B, C, D, E, F, G	Case Study - Qualitative	Semi-structured Interviews (n=25)
2	Improving Agility by Managing Shared Libraries in Microservices (chapter 7)	A, D, E, F	Case Study - Qualitative	Semi-structured Interviews (n=6)
3	Accumulation and prioritization of Architectural Debt in three companies migrating to microservices (chapter 8)	H, I, J	Case Study - Mixed methods	Structured interviews (n=47) + Validation Semi-structured group interviews (n=16)
4	Reducing Incidents in Microservices by Repaying Architectural Technical Debt (chapter 9)	A	Case Study - Mixed methods (mostly quantitative)	Semi-structured Interviews (n=10) + Quantitative analysis (8330 incidents) + Member checking (n=2)

The remainder of this section describes the research context, the companies involved, and the factors that led to selecting the research approaches.

4.1 Research context

This thesis is based on collaborations with software companies located in Europe working with microservices. For confidentiality reasons, we limit the

description of the companies to the information approved by them. All those companies are architecting with or migrating to microservices. Some of them started microservices-like architectures before the official definition of the term in 2014 by James Lewis and Martin Fowler. Examples of such companies are A, E, and G in this study. Due to the old age of their systems, it is still possible to find SOA-oriented solutions that are inadequate for microservices. Frequently, these implementations are a source of MS-ATDs, as we can see later in this thesis. Other companies started their architecture development shortly after Lewis and Fowler [LF14] defined microservices. At that time, microservices were a relatively new subject, and such companies had difficulties dealing with them, accumulating ATD. Company F is an example of such in our study.

Yet other companies worked with many internal projects related to different products. Some of their projects used monolithic architectures. Others used old SOA approaches due to requirements such as scalability. Finally, especially after 2014, when microservices became a trending topic, they started to develop new products using this architectural style. As such, these companies acquired valuable experience with microservices. Companies B, C, and D are examples.

Finally, some companies have monoliths and just started migrating to microservices because they need to scale to attend to greater demand or other reasons. These companies have their first experiences with microservices. Examples of these companies are H, I, and J.

Companies A to J have different experiences with microservices and operate in different contexts. By looking into them, we can further understand how MS-ATDs affect software development from a broader perspective. Below you will find a more detailed summary of each of those companies:

- **Company A** is a financial services institution that develops software to assist users with financial operations, money management, payments, insurance, and investments. The company employs more than 30 thousand employees, mainly in Europe, with more than two thousand working in IT-related positions. The project under investigation in the company had approximately 1000 unique microservices, of which about 150 were business-critical. This project has more than ten years on the market.

- **Company B** develops software as services available in the cloud to third-party consumers. These consumers buy licenses for using the services available. The company employs more than seven thousand employees in Europe, but only about 200 employees worked in the project under evaluation. This project consisted of 50 unique microservices and started

about two years before our research. The company, however, is several years older.

- **Company C** develops software in the Internet of Things (IoT) domain. Their software aims to manage IoT devices. The company also collects and processes the information provided by the devices. Company C was as big as Company A, with more than 30 thousand employees, mainly in Europe. Approximately two-thirds of their employees work in IT-related positions. About 500 employees worked on the project under investigation. The project under investigation had about 80 unique microservices and started two years before the research collaboration.
- **Company D** provides health services. Its portfolio includes software for managing medical devices and medical data processing and storage. About 250 people were involved in the project under investigation, which had about 40 unique microservices and was started one and a half years before the research collaboration. Company D also had more than 30 thousand employees and about 1200 of those work in IT-related positions.
- **Company E** develops software used in public services, such as payslip management and taxes administration. The company is a significant player in the sector, with more than 30 thousand employees. However, they have a relatively small IT department, with only 250 employees, 150 of which were involved in their main project, which was investigated in our research. They had about 400 unique microservices in a more than ten years old project.
- **Company F** develops software used in trains, metros, and other transport solutions mainly for people and goods. Its project was four years old and had about 600 unique microservices. They had about 300 employees, and about half worked in IT-related positions and were all involved in the project under investigation.
- **Company G** is a software-oriented company with the more extensive software among our case studies, counting approximately 3000 unique microservices. The company develops software solutions to transport people, goods, and services. They had more than 20 thousand employees, and an unknown but undoubtedly large number of them work in IT-related positions. Their solution is more than ten years old.
- **Company H** provides business software and IT-related development and consultancy. It employs nearly a dozen thousand employees and

has hundreds of thousands of customers, mainly in Northern Europe. The project under investigation was a dynamic ERP system for large companies, which was one of the company's flagship products and was under migration to microservices.

- **Company I** has more than 20 thousand employees and provides IT and product engineering services. It serves thousands of customers in more than 90 countries. We conducted our study in one of the company's branches located in a Nordic country developing financial services, such as banking solutions.

- **Company J** is one of the largest financial services groups in the Nordic region (mainly banking). The company has more than 9000 employees and serves several millions of customers. We conducted our study with software teams working at the core of their in-house software department.

4.2 Case study research

The overall research strategy adopted in this thesis is a mix of qualitative and quantitative research, but primarily qualitative, based on case studies. The remainder of this section explains the rationale behind selecting these methods.

Our primary research objective is exploratory, empirically identifying and understanding MS-ATDs in the context of large companies. Such a context is intensely industrial and requires a flexible research method adapted to distinct companies and processes to observe the occurrence of MS-ATDs, their costs and solutions, without disrupting the work environment.

Case studies investigate phenomena in their context [RH08; Yin18]. They cannot identify causal relationships but provide a deeper understanding of the phenomena under consideration [RH08]. Case studies were initially designed to be used in social science research [Yin18] but started to be used for many kinds of software engineering research over the years [RH08].

Changes in the software architecture depend on the projects' context. The software architecture for the internet of things (IoT) differs from the architecture developed for a banking solution, which is in turn different from software developed for an e-commerce platform, but all three examples may use microservices. Therefore, the context of the project is important to identify MS-ATDs. Understanding the phenomenon of MS-ATDs requires studying the perception of individuals, which in turn requires qualitative methods. According to Yin [Yin18], case studies are effective to investigate context-dependent phenomena, including the perception of distinct individuals.

Runeson and Höst [RH08] present three other major empirical research strategies used in software engineering research: survey, experiment, and action research. Each of those strategies has its advantages and disadvantages. We briefly summarize them below and explain why we preferred case studies instead.

- **Case study versus survey:** A survey is a collection of information standardized for a specific population frequently done through questionnaires or interviews [Rob02]. Surveys provide an overview of the studied field, not in-depth studies [RH08]. They are primarily quantitative and aim to describe the population under consideration [RH08]. Our study, however, has unclear boundaries between phenomena and context. Given that surveys are designed beforehand, and we do not have prior knowledge regarding MS-ATDs, we would have required running interviews or pilots before finalizing the survey design to avoid missing critical factors for understanding the context. We wanted to deeply understand the companies' context, but surveys only provide a more in-breadth overview. Therefore, we found case studies to be a better option in this case.
- **Case study versus action research:** Action research is closely related to case studies, but they aim to influence or change the objects in focus in the research [RH08]. Action research requires mutual learning and involvement of researchers and practitioners [SDJ07]. Case studies, on the other hand, are solely observational [Yin18]. Our research aimed to understand the phenomenon of MS-ATDs rather than being mutually involved with practitioners in specific cases. In addition, action research is a feasible alternative if the researchers and the companies involved are long-term committed to each other because the time required for evaluating the changes might be considerable. In our case, the companies were involved in this thesis for a relatively short period. We focused on obtaining evidence from multiple companies in order to capture as many distinct contexts as possible, rather than focusing on a deep commitment with one of them.
- **Case study versus experiment:** An experiment is an empirical inquiry in which we manipulate one or more variables in the study while keeping other variables under control in order to understand the effects of the manipulation. Experiments require a certain level of control of the environment under study. Generally, it collects valuable statistics through many observations. There were not many projects available in our

cases from which we could make valid statistical inferences, and the understanding of MS-ATD is not mature enough for us to run experiments. Case studies are observational and study phenomena in depth in a given context. Therefore, they were a better option for our investigations.

4.3 Interviews

According to Yin [Yin18], interviews are one of the most important sources of case study information. They are particularly useful for exploratory research [RH08]. Interviews are also the primary source of information for this thesis because our studies were mainly exploratory. They enabled us to follow up on topics the practitioners commented on but we had not envisioned beforehand. Table 4.2 summarizes all the interviews performed in this thesis. We interviewed a total of 69 subjects in ten large European companies. We reached out to several other companies, but not all decided to contribute to our research. The remainder of this section details the information presented in Table 4.2.

4.3.1 Types of Interviews

Interviews can be unstructured, semi-structured, and fully structured [Rob02]. Unstructured interviews develop the dialog based on the mutual interest of the interviewee and the researcher. Fully structured interviews require all questions prepared beforehand and asked in the same order as planned during the interview. Semi-structured interviews lie between those two other options: the method requires prior planning, but the researcher may ask new questions during the interview and not follow the same strict order.

Interviews can also take place in a group context. Group interviews allow the participants to interact with each other and increase the amount of data the researchers may get by collecting from several people simultaneously. Therefore, they are usually flexible and carry out the characteristics of both a discussion and an interview, because the traditional format of question and answer may eliminate the group interaction, one of the strengths of group interviews [Rob02]. In a group interview, participants may see the differences in perceiving the problem. However, there might be conflicts of personalities, and some participants might agree with the stronger opinions instead of exposing their own.

Table 4.2: Summary of interviews in this thesis by company.

Comp.	First interviews		Follow-up interviews		Mean # weeks between interview rounds	TOTAL
	# interviewees	Type	# interviewees	Type		
A	10	Individual	3	Individual	30	**13**
B	2	Individual	1	Individual	26	**3**
C	3	Individual	1	Individual	29	**4**
D	3	Individual	-	-	-	**3**
E	2	Individual	-	-	-	**2**
F	1	Individual	-	-	-	**1**
G	1	Individual	-	-	-	**1**
H	9	Group	3	Group	12	**12**
I	19	Group	7	Group	6	**26**
J	19	Group	6	Group	4	**25**
TOTAL	**69**		**21**			**90**

4.3.2 The main interviews

We selected our interviewees by convenience sampling, i.e., from the collaboration network we had access to. The interviewees directly or indirectly worked with the object of investigation, the MS-ATDs. We only interviewed participants with an active role in their respective projects and relevant to our studies.

The interviews performed were partially face-to-face and partially remote. Face-to-face meetings allow the interpretation of non-verbal actions and support the communication flow. Therefore, we preferred in-person meetings and traveled to the interviewee locations when possible. However, in some cases, the distance between researchers and the interviewees, and the influence of external factors (such as incompatibility of travel schedule) required audio or video interviews. We mostly had semi-structured interviews.

The main interviews for Study 3 differ from the other studies. They contained structured, closed questions to obtain quantitative data to rank the MS-ATDs. We used the data collected from those interviews to propose an approach to prioritize the MS-ATDs.

4.3.3 Case selection

A total of 10 companies named from A to J, as presented in Table 4.2, participated in this study. We selected all the companies by convenience. The first seven companies, A to G, had substantial experience with microservices, a requirement for our first studies. Therefore, we just accepted companies working with microservices.

Companies H, I, and J started migrating to microservices and have less experience with this architectural style. Therefore, they participated in a different kind of study in which we investigated the migration to microservices.

4.3.4 Data analysis

Qualitative data collection and respective analysis are frequently carried out in parallel because the analysis can reveal the need for additional data. Commonly, new insights arrive in later stages of the research, so new data must be collected and analyzed. Therefore, the data analysis process must follow a systematic approach, allowing researchers to navigate back and forth between data collection and analysis and to be able to communicate their findings in a way readers can follow the results and conclusions back to the data [Run+12].

We transcribed most of the interviews throughout the studies for posterior analysis. Then, we used open coding, an approach that is part of grounded theory, for our data analysis. Grounded theory is a systematic inductive methodology that builds a theory grounded in the data to explain the topic under investigation [CS15]. Open coding is usually the first step of coding in exploratory studies and aims to produce a set of concepts that fit the data [CS15]. We used it for its rigor as a systematic approach.

Open coding is an interpretive process by which one breaks the data and labels them as codes. The codes enable researchers to continuously compare concepts in the data [CS15]. The codes' comparison already allowed us to understand the data and draw our results and conclusions. We used the computer-assisted qualitative data analysis software tool NVivo[1] to process the codes and categorize the data.

[1] https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software

The structured interviews from Study 3 resulted in quantitative data. They consisted of votes related to specific contextual information on MS-ATDs, such as whether a specific debt is found in the context of the project in which the practitioner is involved, or if the project participants consider the debt important to be prioritized. We analyzed the data using descriptive statistics and proposed an approach to prioritize debt repayment.

4.3.5 Follow-up interviews

We conducted follow-up interviews in all our studies with distinct purposes. Studies 1, 2, and 4 (with Companies A to F) required a longer interval between the main and follow-up interviews due to the amount of qualitative data to process. Study 3 (with Companies H, I, and J) contained a dataset that was easier to evaluate and required less time between the interview rounds. Table 4.2 shows the mean weeks between the interview rounds for each company in this thesis.

In Studies 1 and 2, the follow-up interviews were necessary to clarify aspects not covered in the original interviews or to investigate missing details. For example, some companies distinguished the impact of libraries developed by the company from libraries from external parties, such as open-source software. This distinction was not clear for certain companies interviewed before. Thus, we conducted additional interviews to ask about this subject.

Study 3 developed a prioritization approach for MS-ATDs based on the main (structured) group interviews. We presented the results from that group interviews to a subset of the study's original participants. After that, we conducted follow-up semi-structured group interviews to validate the usefulness of our approach. In that case, semi-structured interviews allowed us the freedom to follow up on topics presented by the interviewees when answering our questions.

Study 4 required a cleanup of the original dataset because, according to previous interviews, there was duplicated or invalid data in the dataset. After receiving the dataset, we performed a cleanup based on the information from the earlier interviews. For example, some automatic tools reported the same incident several times until fixed. However, many of our assumptions during this cleanup were initially open to interpretation. Considering that changes in those assumptions would affect the interpretations of the results, we contacted two former interviewees with extensive knowledge about the solution to check such assumptions. We conducted two separate interviews. We made the proper adjustments in the dataset according to feedback from those interviewees. Runeson and Höst [RH08] describe this approach as *member checking*.

4.4 Document analysis

Initial interviews with participants in Study 4 raised concerns about incidents as an effect of MS-ATDs. Therefore, we collected information about numerous incidents for analysis. Company A assigned an architect to collect data internally and send them to the researchers. This architect was also responsible for removing sensitive data from the dataset before sharing it. We received the following documents:

- Two spreadsheets, one containing details about the incidents with the previous architecture and the other containing details about the incidents in the new architecture.
- One set of slides from an internal training about how the incidents should be registered by practitioners.
- Two text documents containing lists of critical incidents in the previous and the refactored architecture, respectively.

We used the presentation and the list of incidents to clarify information given during the initial interviews. The spreadsheets together contained more than eight thousand entries of incidents, including a short description of the incident, the date and time it was registered, the urgency, impact, and priority of the incident according to internal guidelines, the type of the incident, and the submitter (a human or an automatic software).

We organized the information in the spreadsheets as a dataset and used graphs and descriptive statistics to understand them. We compared the number of incidents before and after the repayment of the debts.

Measuring the actual interest of the debts is difficult due to the lack of reliable metrics and data for such calculations. However, the incidents have been reported as directly related to many MS-ATDs found in Company A. They were one of the main drivers for the architectural refactoring. Therefore, we used them as a proxy for part of the debts' interest.

Study 4 was exploratory, so we did not conduct hypothesis testing. Future research may be able to run hypothesis testing when the understanding of MS-ATDs matures.

4.5 Validity and reliability

We designed and carried out all research activities according to well-known guidelines for case studies [RH08; Yin18]. We reduced potential sources of bias

as discussed in the remainder of this section. Even so, potential threats to the validity and reliability of the research remain and are impossible to avoid. The remainder of this research discusses the potential threats to the validity and reliability of this work and how we mitigated them when possible. We use the criteria for judging the quality of research designs presented by Yin [Yin18]: construct validity, internal validity, external validity, and reliability.

4.5.1 Construct validity

Construct validity concerns the definition of a concept and how it is measured by a set of indicators [SB21]. Studies 1, 2, and 3 have threats to construct validity because participants could have distinct understandings of the debts. Therefore, if two different definitions of the same debt appeared in Studies 1 and 2, we enforced one single definition in the follow-up interviews. When the participant discussed something with a different meaning, we considered both distinct debts. For example, we separated the definition of *shared libraries* from what we called *external dependencies*. Although the participants originally discussed both as *shared libraries*, we separated the concepts, because they have different meanings.

In Study 3, the votes of the participants could have different meanings, invalidating our results. We mitigated this threat by presenting all the MS-ATDs to the participants beforehand and asking if they understood the explanations. We also collected data regarding that debt right after the explanation.

4.5.2 Internal validity

Internal validity threats concern the cause-effect phenomena [Yin18]. Most of our studies were descriptive and exploratory. We did not try to prove causality. Although we have some hypotheses that might indicate causality in some cases, we do not claim to have any evidence on that. Therefore, internal validity threats are not applicable for the studies in this thesis.

4.5.3 External validity

External validity threats concern to what extent it is possible to generalize the study's findings beyond the immediate study [Yin18].

This thesis is explorative in the sense that studies on MS-ATDs have not been reported in the research literature before. It is too early to discuss the generalization of the results. As explained by Yin [Yin18], researchers should think of case studies as opportunities to shed empirical light on some theoretical

concepts or principles. Therefore, analytic generalization, i.e., based on a theory, is more relevant to case studies than statistical generalization. However, we find it too early to propose a theory, even an initial one.

Still, regarding generalization, it is more likely to find the same MS-ATDs in other companies if they were found in several of our investigated companies.

In this study, our interviewees formed only a fraction of all the employees in the respective companies, so the debts we found may not represent the whole MS-ATD spectrum of the projects under investigation. Consequently, even though some MS-ATDs may be rare in our studies, they might have been more common in the investigated companies but might have been undetected. In any case, we recommend other researchers to replicate our studies to make the results more generalizable.

4.5.4 Reliability

Reliability threats concern the extent to which the data and the respective analysis depend on the involved researchers [RH08]. There might be factors that the researchers were not aware of because the interviewees themselves were also not aware of or did not express in the interviews, such as the quality of the implementations and management issues. We mitigated this threat by asking participants from the companies to confirm the results through member checking in some cases.

Additionally, there is a threat in which the researchers may have interpreted what the practitioners said in the interviews based on the researchers' background. For example, the practitioners mentioned how they used a particular service mesh technology to improve the visibility of microservices, but they did not mention such a reason explicitly. Therefore, the researchers assumed that the new service mesh was used to solve network issues, as reported by a company in a previous study. We mitigated this threat by addressing our inter-reliability by having at least two researchers present in most interviews. We also discussed every result and interpretation among the researchers involved in the research.

Furthermore, we designed some of our studies to contain validation phases with member checking to prevent wrong interpretations of the results.

Papers

Chapter 5

Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study

Saulo S. de Toledo, Antonio Martini, Dag I. K. Sjøberg

Published in *Journal of Systems and Software*, April 2021, volume 177. DOI: 10.1016/j.jss.2021.110968.

Abstract

Background: Using a microservices architecture is a popular strategy for software organizations to deliver value to their customers fast and continuously. However, scientific knowledge on how to manage architectural debt in microservices is scarce.

Objectives: In the context of microservices applications, this paper aims to identify architectural technical debts (ATDs), their costs, and their most common solutions.

Method: We conducted an exploratory multiple case study by conducting 25 interviews with practitioners working with microservices in seven large companies.

Results: We found 16 ATD issues, their negative impact (interest), and common solutions to repay each debt together with the related costs (principal). Two examples of critical ATD issues found were the use of shared databases that, if not properly planned, leads to potential breaks on services every time the database schema changes and bad API designs, which leads to coupling among teams. We identified ATDs occurring in different domains and stages of development and created a map of the relationships among those debts.

Conclusion: The findings may guide organizations in developing microservices systems that better manage and avoid architectural debts.

Contents

5.1	Introduction	42
5.2	Background	43
5.3	Methodology	54
5.4	Results	69
5.5	Discussion	83
5.6	Related Work	90
5.7	Conclusions and Future Work	92

5.1 Introduction

Microservice architecture is a relatively new architectural style that is becoming increasingly popular in the industry. A microservice is a small component that can be developed and deployed independently, is easy to scale, and has a single responsibility [Dra+17]. Such characteristics make microservices particularly convenient for continuous delivery [Thö15]. Microservices are built around business capabilities and provide an architectural style capable of organizing cross-functional teams around services [Dra+17]. Microservices are supposed to support companies to deliver value to their customers in a fast and continuous fashion.

Despite their advantages, microservices are still an emerging technology. There are still drawbacks of using such an architectural style, such as data inconsistency among various services [Fur+18]. As a simple example, suppose an online bookstore has three services: one for managing the book catalog, another to manage orders, and a third for deliveries to the customers. Each of those services has its own database. When a client finishes an order, the book must be removed from the stock by the book service, and the delivery must be triggered. When the orders database is updated, the product remains in an inconsistent state until the other services finish updating their databases. Meanwhile, the client and the store own the book since it is still available in the stock and orders databases, and there is a chance the company will sell more books than it has, causing problems for the company.

Companies are still learning how to properly migrate from old monolithic software to systems that use microservices. There are still several challenges in implementing microservices from scratch to make them easy to maintain and evolve [Bog+19a], which leads to a situation in which practitioners make architectural sub-optimal decisions that lead to a benefit in the short term,

but increase the overall costs in the long run, i.e., a situation described by a metaphor known as Architectural Technical Debt (ATD) [VML18].

Di Francesco et al. [DLM19; DML17] state that fundamental principles, claimed benefits, and quality (including maintainability) of microservices still must be proven by research and envisioned further qualitative studies with practitioners. Studies on managing ATD in microservices, which directly affects software maintainability, would be part of such a request.

There is a body of grey literature and books that concern microservices, migrations, and related practices. Still, such literature does not focus specifically on ATD. In particular, it does not explore the diversity of challenges regarding ATD and microservices across companies.

As a contribution to meeting the needs described above, we conducted a multiple-case study in seven international Europe-based companies to investigate ATD in microservices through the following research questions:

- **RQ1:** What are the most critical ATD issues in microservices?
- **RQ2:** What are the negative impacts of such ATD issues?
- **RQ3:** What are possible solutions to repay or avoid such ATD issues?

For each of the identified ATDs, we outlined how to determine interest and principal, which is needed to develop metrics for quantifying the costs of the ATDs. Our contributions may also support practitioners' decision-making in projects involving microservices.

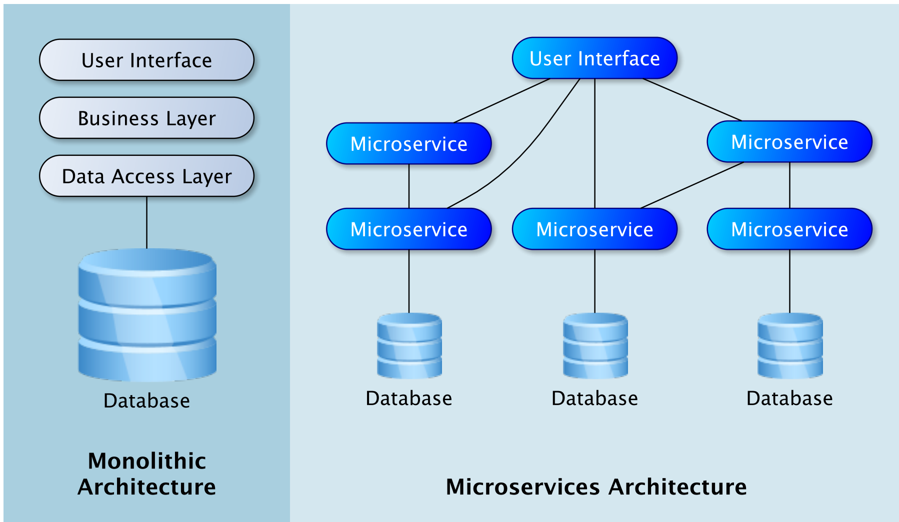
5.2 Background

This section describes the concepts of microservices and architectural technical debt.

5.2.1 Microservices

Lewis and Fowler [LF14] provide the most accepted definition of the microservices architectural style: "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". Microservices are frequently described as an alternative to monolithic applications, built and deployed as a single unit (see Figure 5.1), since well-known companies, such as Amazon and Netflix, have been using microservices

Figure 5.1: Monolithic and microservice architectures



to overcome difficulties with their previous monolithic architectures [LF14]. Applications that use microservices are easier to scale, have shorter cycles for testing, build and release, and are less frequently affected by downtime than monolithic applications [Fow15]. These and other characteristics make microservice applications particularly desirable for continuous delivery. In fact, new features can be independently and continuously tested and delivered by updating specific microservices without changing the whole product, which drastically reduces lead time. Still, there are challenges, such as the risk of increased data inconsistency and operational complexity [Fow15].

Microservice architecture may also be described as one way of implementing Service Oriented Architecture (SOA), although there are different views on this claim [Zim17]. Certainly, SOA describes a set of applications that cannot be considered microservices. For example, many SOA applications are implemented using an Enterprise Service Bus (ESB), an infrastructure that mediates requests among services, intercepts communications, and provides transformation capabilities, among other functions [NG05]. ESBs can be a single monolithic artifact that can be deployed together with the services at the same place [MW16]. In contrast, microservices employ what is called a *dump pipe* or a communication layer without business logic. Other characteristics can also describe SOA but not microservice architectures. Rademacher et

al. [RSZ17] provide a list of such characteristics including the following: (i) there is no guidance about the service granularity in SOA, while a microservice architecture suggests that each service represents one capability only; (ii) SOA may support transport protocol transformation, while microservices usually apply REST over HTTP or a protocol supported by a message bus; (iii) there are several service types in SOA (e.g., business, enterprise, application), while there are only two types of microservices—that is, they are functional (representing business capabilities) or infrastructure (providing technical capabilities like authentication and authorization) services.

Despite such differences, there are several concepts and techniques in the area of microservices that were borrowed from SOA, such as the approaches for communication detailed in Section 5.2.1.1; the concepts of scalability, service discovery, and service registry detailed in Section 5.2.1.2; and the concepts of service availability and responsiveness detailed in Section 5.2.1.3. There are some adaptations of those concepts and techniques in a microservice architecture, such as a limited set of communication protocols. Other concepts such as Service Mesh, explained in Section 5.2.1.4, emerged to support microservice architectures [Li+19].

In summary, while there is an overlap, there are certainly many differences in techniques and concepts between SOA and microservice architecture. In this paper, we focus on microservices.

5.2.1.1 Microservices communication

In a microservice architecture, clients and services may communicate directly with each other synchronously [New17] (Figure 5.2) or through an API gateway [MW16] (Figure 5.3). They can also communicate asynchronously through a message bus [New17], which holds the message in a queue until one or more services consume(s) it, as shown in Figure 5.4. Such communication may also be mixed: for example, by using a synchronous request with an asynchronous response.

5.2.1.2 Scalability and service discovery

In the microservices context, scalability is the service's ability to cope and perform under high demand. A scalable microservice adapts itself to the needs of its consumers [MVA18]. It is possible to scale those services by running multiple instances of them; each instance of the same service should be able to replace any other so that if one fails or already has high traffic, another working instance may be used in its place (see service E in Figure 5.5). An available

Figure 5.2: Microservices synchronous communication

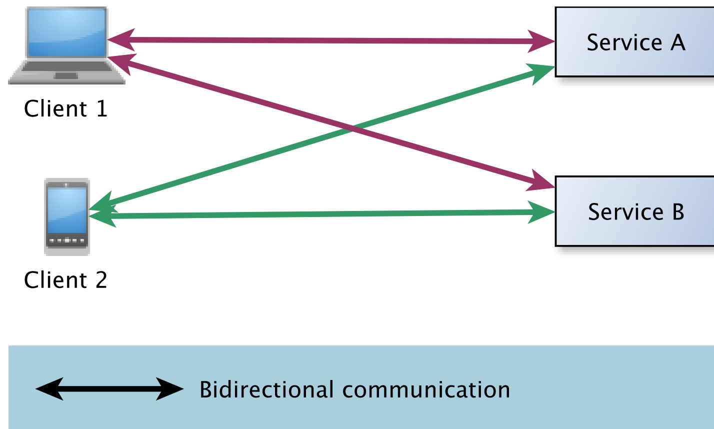


Figure 5.3: The API gateway

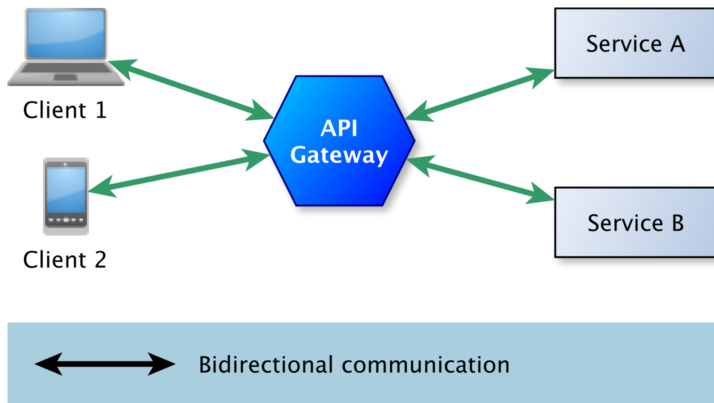
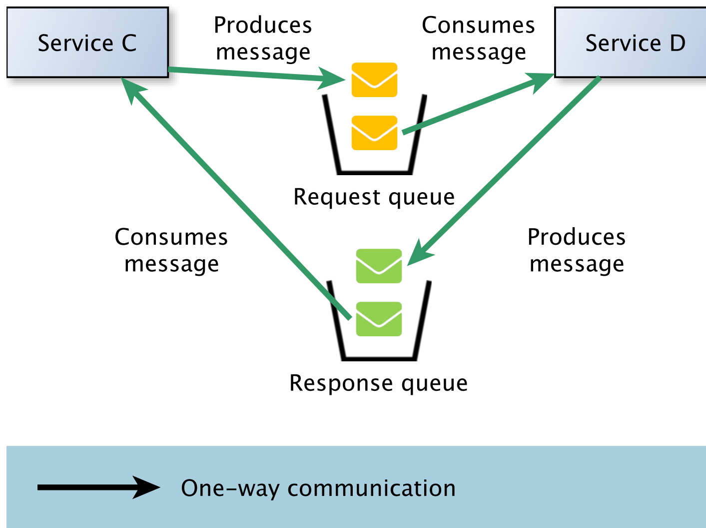


Figure 5.4: Microservices asynchronous communication

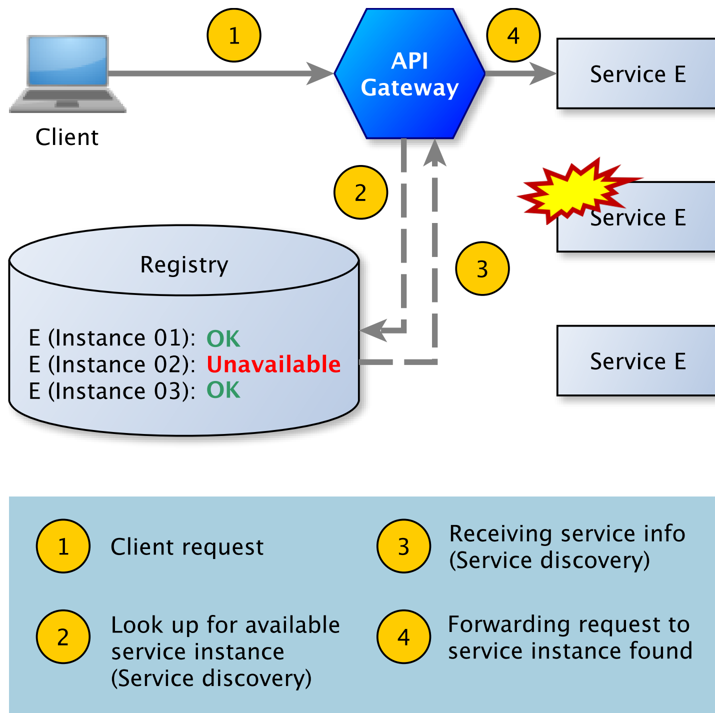


instance of the service may be found by a service discovery mechanism in such situations. The service discovery (i.e., the act of finding a running instance of a service) may be performed by consulting a service registry (i.e., a service that stores information about other services and their available and unavailable instances; for example, service E instances 01–03 in Figure 5.5) [MW16]. One way of performing the service discovery is exemplified in Figure 5.5. The API gateway can query the registry to find a running instance of service E that the client requires. The service discovery method exemplified before is called server-side discovery, as opposed to the client-side discovery, in which the client is responsible for querying the service registry [MW16].

5.2.1.3 Service availability and responsiveness

In the setting where one service (consumer) requests information from a remote service (producer), *service availability* is the producer's ability to accept the request in a timely manner [Ric16]. Inversely, *service responsiveness* is the consumer's ability to receive a timely response [Ric16]. When a consumer makes a request to a producer, the consumer does not know whether they will receive a response. Since they cannot wait indefinitely, it is common for the consumer to wait a specific period of time (the timeout) until they give up and consider

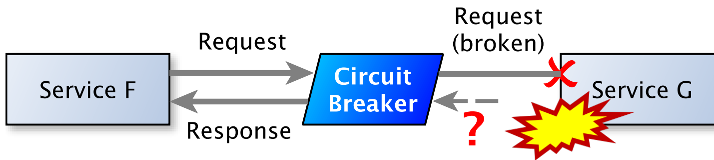
Figure 5.5: Service discovery, registry and services instances



the request as a failure. At this point, the consumer may try the request again. When the producer has a very high load or is entirely inaccessible, consumers may keep (i) repeatedly try to connect until the producer is available or (ii) wait the whole timeout period until they can take some other action [MW16]. Both cases waste resources.

A better technique is to use *circuit breakers*. When services are down or demonstrate high latency and are mostly unusable, a circuit breaker takes the lead and immediately responds to the consumer (Figure 5.6). The consumer uses fewer resources waiting for services that failed (i.e., they will not keep running and waiting for an unavailable service that may never send a response). Such a solution prevents network or service failure from cascading to other services since some may depend on the consumer in our example [MW16].

Figure 5.6: The circuit breaker



5.2.1.4 Service Mesh

Finally, *service meshes* have emerged with the popularization of microservice architectures. Service meshes are dedicated infrastructure layers acting on the service-to-service communication, designed to make the services safe, reliable, and more observable. They usually implement several of the mechanisms introduced in previous sections, as well as others such as service discovery mechanisms, load balancing, encryption, circuit breaking, and service observability [Li+19]. These mechanisms are not strictly required when implementing microservices, but they might be beneficial, especially when there are many microservices.

5.2.2 Architectural Technical Debt

This section gives an overview of ATD, its management, and its relationship to related concepts.

5.2.2.1 An overview of ATD

ATD is a type of technical debt (TD) consisting of suboptimal architectural solutions, which deliver benefits in the short-term but increase overall costs in the long run. Identifying ATD is particularly important since problems in the architecture may slow down new functionalities and raise the related costs. Several authors [BMB17a; Ern+15; KNO12] describe ATD as the most challenging type of TD to be unveiled and managed due to the lack of research and practical tool support.

The three main concepts of TD are the debt itself and its interest and principal [Avg+16]:

- **Debt:** A sub-optimal solution that has short-term benefits but will generate future interest payment is called a debt. For example, suppose a

functionality is implemented using different components. If the developers create the components without carefully planning their interfaces in order to develop them faster, the solution may end up with tightly coupled components. Such a solution may be easy and fast to develop, but the product's maintenance may be costly. Due to tightly coupled components, changes in one of the components may cause consequential changes in other components. When the product is updated, it may take a long time to change and test all the components, slowing down the delivery of new functionalities.

- **Interest:** The extra cost that must be paid because of a debt, or the amount that will be saved if there is no such debt, is called interest. In the previous example, the interest is the cost of the additional effort needed to test and update all the dependent components each time a component is changed.

- **Principal:** The cost of developing a solution that avoids the debt, or the cost of refactoring a solution to avoid the debt, is called the principal. In the previous example, the principal is the cost of the effort (e.g., time and resources) required to develop the involved components' interfaces so that they are not tightly coupled and can be changed and tested independently from each other.

It can be profitable in some particular circumstances to accumulate the debt [Bes+18]. In theory, deciding whether to accumulate the debt is supported by a simple calculation: If the interest is less than the principal, it is better to accumulate the debt. If the interest is more significant than the principal, the debt should be avoided [MB16b; Sch13]. However, it is not easy to know—or measure—the actual costs in practice regarding either the principal or the interest. It is still important for the involved stakeholders to be conscious of a debt's principal and interest. Practitioners need to make decisions on their ATD.

It is difficult to avoid the accumulation of some of the ATDs during the software's life cycle [MBC15]. Thus, it is important to know when these debts should be repaid and when to avoid their accumulation. Areas like microservices still lack ways of identifying and measuring ATD [de +19], which motivates this study.

5.2.2.2 ATD Management

Managing ATD is difficult [BMB18] but it is important to repay the debt [LLA14]. Li et al. [LLA14] describe the ATD management process through the following activities:

- **ATD identification:** In this phase, the ATD items (including their interest and principal) are detected and described.
- **ATD measurement:** In this phase, the debts' costs and benefits are analyzed and estimated.
- **ATD prioritization:** In this phase, the items are sorted by some criteria (e.g., importance) to decide which ATD item must be repaid first or if ATD should be repaid instead of investing in other activities, such as developing new features.
- **ATD repayment:** In this phase, architectural decisions are made to repay the debt, even if partially.
- **ATD monitoring:** In this phase, ATD items are monitored over time regarding their costs and benefits.

In this study, we start this process by identifying ATD in microservices. We then indicate what should be measured and contribute with information helpful for ATD prioritization and monitoring. We also present solutions for ATD repayment.

5.2.2.3 ATD versus related concepts

Figure 5.7 presents the relationship between ATD and other concepts such as architectural patterns, anti-patterns, erosion, drift, and smells. All these concepts have been associated with ATD. A few others, such as defects and degraded system qualities, are also discussed in the TD literature and are briefly discussed in this section. Although various studies exist on these different concepts, there is no comprehensive work clarifying their relationships. Therefore, we report our interpretation of such concepts based on the available literature.

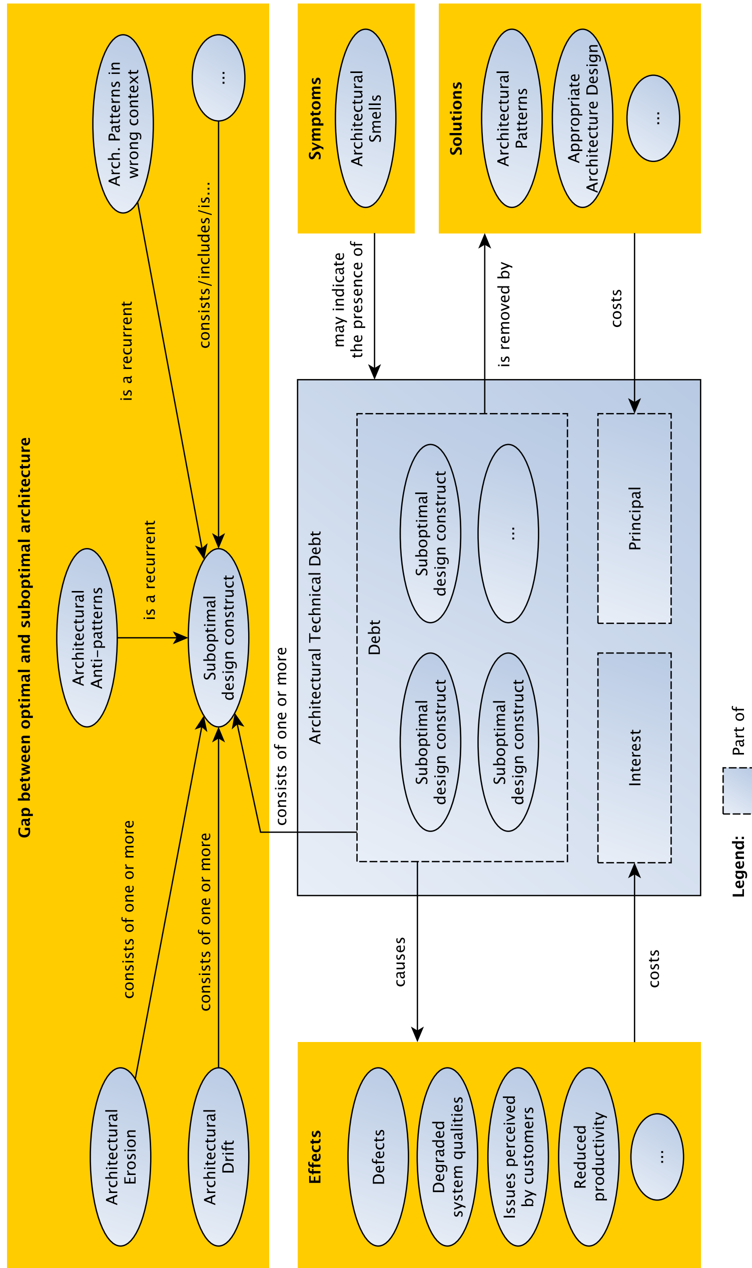


Figure 5.7: ATD and the related concepts

The most up-to-date definition of TD, available in the Dagstuhl seminar report 16162 [Avg+16], states that "technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible." When discussing architecture, we focus on design constructs. Figure 5.7 shows the relationship between what we call sub-optimal design constructs and some well-known terms we discuss next.

Architectural smells are indicators of design problems; as such, they may be symptoms of the presence of ATD [Mar+18] (see Figure 5.7). However, architectural smells might not point to ATD in certain contexts. Martini et al. [Mar+18] reported a situation in which a set of cyclic architectural dependencies, found in a particular graphical user interface (GUI) component, was not considered suboptimal. In that particular example, such a smell was fairly common as a normal (good) pattern. The reported smells do not represent ATD, as there is no interest and principal in such cases.

Architectural patterns are general, reusable architectural solutions [MA18]. When used correctly and with other context-defined needs, such as an appropriate architecture design, they may be solutions to existing ATD. However, many solutions are context-specific and should be discussed in the context of the respective debts; for example, the design of good APIs (see, for example, Mosqueira-Rey et al. [Mos+18]). An architectural pattern may or may not be a proper solution to a known issue in such contexts. Each solution has a cost, which in turn represents the principal of the debt it is removing (Figure 5.7).

Architectural anti-patterns are repeatable suboptimal design constructs that violate design principles and increase the likelihood of having bugs and changes [Mo+19]. An anti-pattern might represent a debt if it generates interest, but there might be cases in which the anti-pattern does not. Besides, the suffered interest of ATD can consist of something else than bugs or changes, for example, a loss of development speed or the degradation of other software qualities [Mar+18]. In the example by Martini et al. [Mar+18] that we reported when discussing architectural smells above, an anti-pattern could be causing the cyclic dependencies disclosed, but no interest was reported. In fact, the practitioners said the opposite: it was a solution (a pattern). The research on anti-patterns in microservices is still in its infancy; for example, there is no well-defined taxonomy [Bog+19b]. Bogner et al. [Bog+19b] identified 14 studies on SOA patterns. Only one book and one paper focused on microservices, although many of the SOA patterns presented in the studies seem relevant to microservices.

Architectural erosion describes design changes in a system's architecture that

violate the original architecture [VB02]. An example of architectural erosion is the addition of workarounds to bypass the decided architecture. *Architectural drift* describes the situation in which the rules implied by the architecture are unclear, leading to divergences between parts of the architecture and making it easier to have violations (i.e., erosion) [VB02]. Both terms represent how the amount of suboptimal design constructs increases over time, but they do not consider interest and principal, as in TD. Also, they become worse over time, accumulating more debts. The debt, however, remains the same. Still, its costs (interest and principal) may vary depending on the context (e.g., a debt leading to a data leak in a bank system is far worse than the same debt in a newsletter system in which, only the email addresses are compromised). The notions of architectural erosion and drift are discussed in the literature through different terms, such as architectural degeneration, software or design erosion, and architectural or design decay [DB12].

Defects are conditions in a software product that need to be fixed because they cause the software's malfunction or produce unexpected results. As explained by Kruchten et al. [KNO12], defects are visible for the customers and are, therefore, different from any kind of TD, such as ATD. Defects, as well as degraded system qualities and other issues perceived by customers, or even internal issues such as reduced productivity, might be an effect caused by the existence of some kind of TD. All these effects have a cost—the interest of the debt which caused them (Figure 5.7).

Other concepts apart from those we discussed may also be used to perceive the gap between the suboptimal and optimal constructs or solutions. For example, misused architectural patterns (i.e., their use in a context they are not suitable for) may also be responsible for such a gap. However, describing such a gap between optimal and suboptimal design is not enough to formulate the problem as ATD, which is focused on the financial variables related to its costs (principal and interest) and dependent on the contexts.

5.3 Methodology

This study aims to identify the most common and critical ATD issues, interests, and principals in products using microservice architectures. We investigated which circumstances led to ATD and identified solutions and insights related to its occurrence. We conducted an *exploratory multiple-case study*, where each analyzed product represents a case. The remainder of this section presents the cases and how the data was collected and analyzed. Figure 5.8 outlines our methodology.

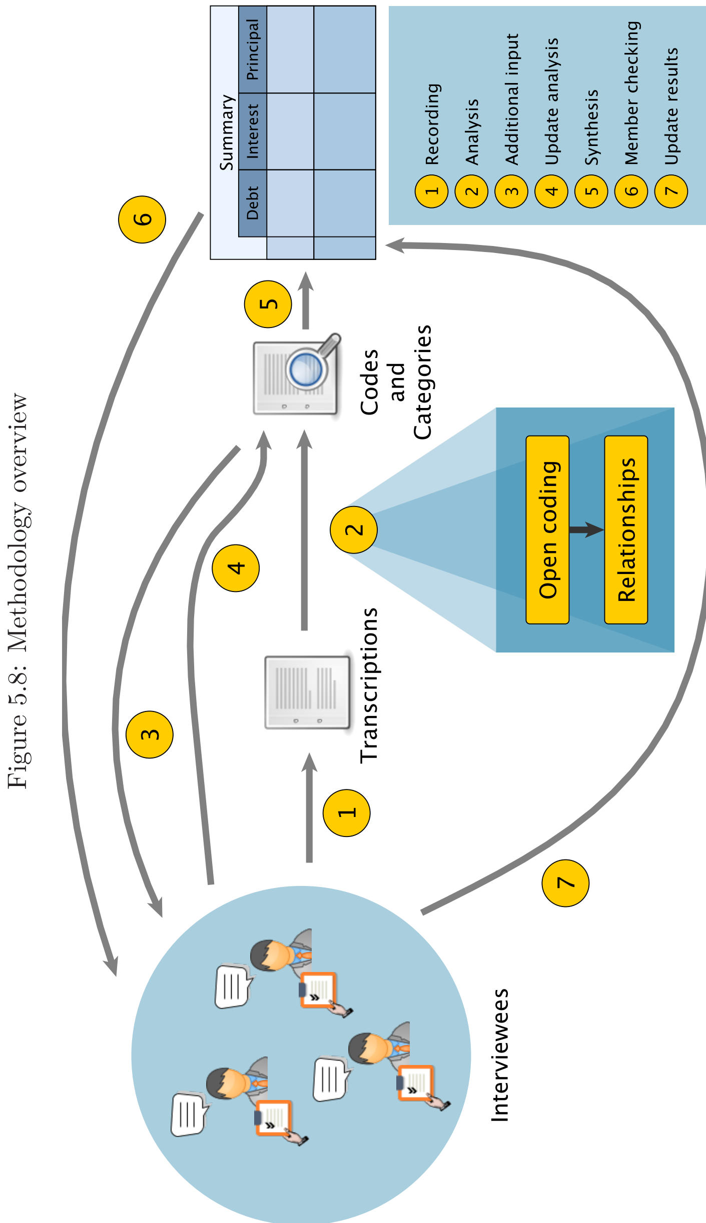


Figure 5.8: Methodology overview

5.3.1 Case Selection

We studied seven different software products in seven large international companies. Two products were provided by different sub-companies within the same multinational conglomerate. All the products had a microservice architecture. Although in some of the products, minor parts were previously developed using a monolith design or SOA approaches, the overall architecture was considered a microservice architecture, and such parts were in the process of being migrated.

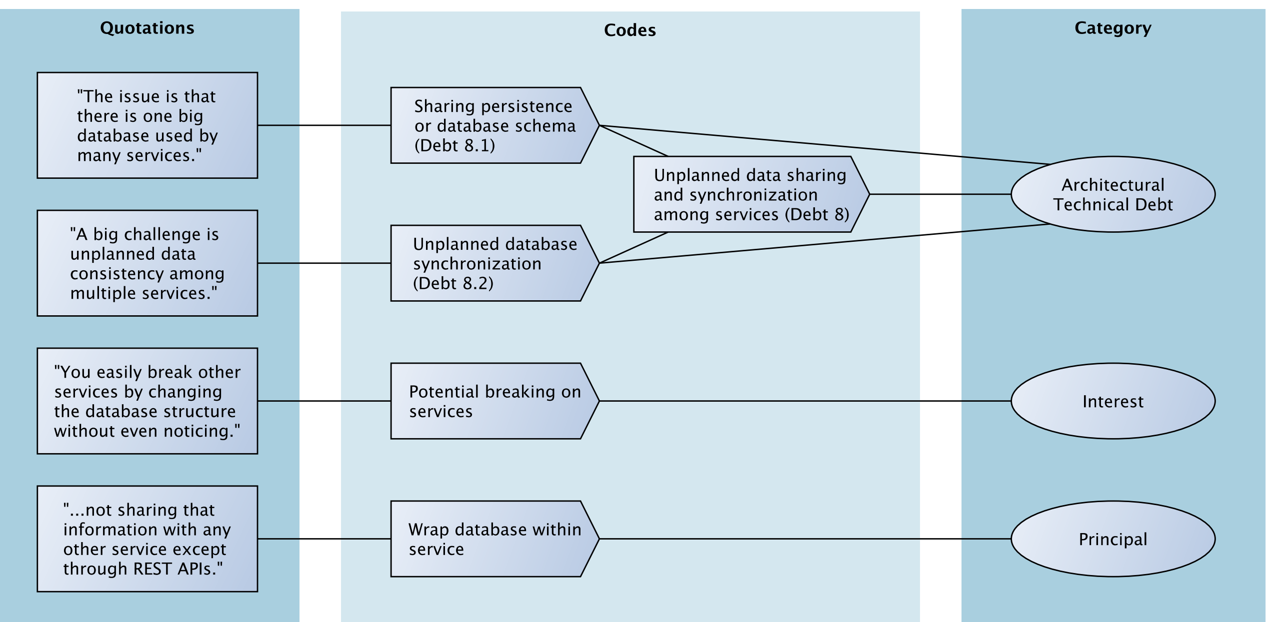
The various application domains of the products gave us diversity in investigated contexts, which helped to understand whether the found problems and solutions were widespread or domain-specific. Table 5.1 shows a summary of the studied companies and products. For confidentiality reasons, we named the companies *A, B, C, D, E, F*, and *G*, respectively. The application domains of the microservices projects we investigated are as follows:

- **Finance:** The software was used to assist users with financial operations, money management, payments, insurance, and investments.
- **Cloud:** The software provided a set of services to be used in the Cloud by third-party consumers.
- **IoT:** A software in the Internet of Things domain that was used to control, share information, and/or gather data from devices connected to the internet.
- **Health:** The software was used to provide health services, such as user profiles and medical information.
- **Public services:** The software was used to provide public services, such as payslip management and taxes.
- **Transport:** The software was used to assist users of public and private transport of both passengers and goods.

Table 5.1: Companies context

Context	Company						
	A	**B**	**C**	**D**	**E**	**F**	**G**
Application domain	Finance	Cloud	IoT	Health	Public services	Transport	Transport
Approx. number of employees	>30000	>7000	>30000	>30000	>30000	320	>20000
Approx. number of employees on IT	>2000		20000	1200	250	150	
Approx. number of employees in the case	2000	200	500	250	150	150	
Approx. number of unique microservices in the case	1000	50	80	40	400	600	3000
Age of the product	>10 years	>2 years	>2 years	>1.5 years	>10 years	>4 years	>10 years
Stage of development	Migration	Initial development	Initial development	Initial development	Evolving	Evolving	Evolving

Figure 5.9: Transforming quotations from practitioners into codes through open coding, and classifying them into categories. This is an example extracted from our analysis, so the number of the debts emphasized in the figure are references to our results in Section 5.4. The rest of the analysis is done in the same way.



We also present the stage of each software project according to the following classifications:

- **Initial development:** The software was developed using microservices from the beginning.
- **Migration:** The software is migrating from an old solution, such as a monolith or other service approach, to microservices.
- **Evolving:** The system is consolidated as a microservice approach and is currently being maintained and evolved.

In this multiple-case study, a case is a given company's specific product. For simplicity reasons, cases are referred to by their company's name.

5.3.2 Data Collection

We performed 25 interviews with 22 employees in different roles, as detailed in Table 5.2. We selected the interviewees and companies through convenience sampling (i.e., selecting from the collaboration network we had access to). All the interviewees had several years of experience in their roles. They all gave consent for the interviews to be recorded and transcribed (Step 1 in Figure 5.8). We used the semi-structured interview guide presented in Appendix A. The interviews lasted between one and two hours.

The study started with Company A, where we could access several employees. This helped us have a solid understanding and a rich amount of details about the initial set of existing ATDs. We then continued the study with additional companies to investigate whether the results were general or differed across contexts.

As we progressed with the interviews in different contexts, new aspects emerged, such as additional ATD instances or further details for specific instances. We updated the interview guide along the course of interviews regarding Debts 8, 10.2, 11, and 12 and added Questions 7, 9, 10, and 11 for the ensuing interviews. When all the cases were investigated, we returned to the previous companies to interview additional subjects. If they were not available, we asked for shorter complementary interviews (20–30 minutes) with the subjects we had met before. We covered newly discovered aspects in these interviews, as represented by Step 3 in Figure 5.8. For example, a later interviewee clearly distinguished between internal shared libraries (produced by the team) and external dependencies (produced by external parties), such

Table 5.2: Type and number of interviews and interviewees by company

C.	Num. Inter-views	Interviewees	Interviewers	Type of interview
A	11	3 product owners/managers 2 architects 5 developers	1st and 2nd authors *(main interviews)* 1st author *(returning interview)*	Face-to-face
B	3	1 product leader 1 architect	1st and 2nd authors *(main interviews)* 1st author *(returning interview)*	Audioconference
C	4	2 architects 1 software engineer	1st and 2nd authors *(first 2 interviews)* 1st author *(third and returning interviews)*	Audioconference
D	3	1 product manager 2 architects	1st and 2nd authors *(first 2 interviews)* 1st author *(third interview)*	Audioconference
E	2	2 architects	1st and 2nd authors	Face-to-face
F	1	1 architect	1st and 2nd authors	Face-to-face
G	1	1 software engineer	1st and 2nd authors	Videoconference
TOTAL:				
7	25	22		

as frameworks and open-source software. Since the previous interviews did not clearly make this distinction, we returned to previous interviewees to ask for additional clarifications and details.

We also returned to previous interviewees to ask about newly discovered ATD items identified in later interviews. For example, we asked all the interviewees whether they perceived the heterogeneity of approaches caused by the services' implementation neutrality nature as harmful.

During all the initial interviews, two researchers were present. For four companies, where the distance did not allow us to have face-to-face interviews, we conducted the interviews using audio or video conferencing tools, as detailed in Table 5.2.

We did not follow Steps 3 and 4 of our methodology (Figure 5.8 going back to the interviewees for additional input) for Companies F and G. We did not find information missing from the other contexts that required additional interviews.

The final interview guide is presented in Appendix A.

5.3.3 Data Analysis

Steps 2 and 4 in Figure 5.8 show our data analysis, which was mainly performed using *open coding*, an approach that is part of *grounded theory* [CS15]. Grounded theory is a rich systematic methodology that involves several other steps not followed in this study.

Open coding is usually the first step of coding in exploratory studies and aims to produce a set of concepts that fit the data [CS15]. Figure 5.9 presents a fragment of this analysis step: selected quotations in the transcriptions or audio recordings are flagged with a label (a code). Later, we found that, despite different wording, some findings were related to a more general topic coded at a higher level category, as in the example in Figure 5.9. This last step allowed us to identify related ATD issues. Finally, we associated these codes to categories such as "ATD," "Interest," and "Principal" in a deductive manner.

We performed the open coding phase by finding relationships between the codes in the categories above. Figure 5.10 shows an example. The open coding provided us with three different codes: one for an instance of debt, another for an instance of interest, and the last for an instance of principal. After the open coding phase, we found that the interest and the principal were, respectively, the consequence and the solution for the debt (e.g., there were costs with multiple API versions (interest) because the APIs were poorly designed (debt), so they were grouped together). The set of codes and their relationships were used in our report phase that synthesized our results (Step 5, Figure 5.8).

Figure 5.10: Identifying the relationship among debt, interest and principal

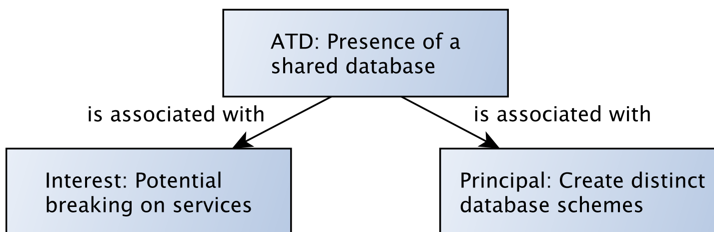


Table 5.3: Architectural Technical Debt identified on each company.

ID	Debt	Comp. A			Comp. B			Comp. C			Comp. D			Comp. E			Comp. F			Comp. G		
		D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P
1.	Insufficient metadata in the messages																					
1.1.	Insufficient message traceability	X	X	X										X	X	X						
1.2.	Poor dead letter queue growth management	X	X	X										X	X	X						
2.	Microservice coupling	X	X	X	X	X		X	X	X				X	X	X	X	X	X			
3.	Lack of communication standards among microservices	X	X	X																		
4.	Inadequate use of APIs																					
4.1.	Poor RESTful API design				X	X	X	X	X	X	X	X	X									
4.2.	Use of complex API calls when messaging is a simpler solution										X	X	X									

Continued on next page

Table 5.3 – continued from previous page

ID	Debt	Comp. A			Comp. B			Comp. C			Comp. D			Comp. E			Comp. F			Comp. G		
		D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P
5.	Use of inadequate technologies to support the microservices architecture	X	X	X				X	X	X												
6.	Excessive diversity or heterogeneity in the technologies chosen across the system	X	X	X										X	X		X	X	X	X	X	X
7.	Manual per service handling of network failures when target services are unavailable				X	X	X	X		X												
8.	Unplanned data sharing and synchronization among services																					
8.1.	Sharing persistence or database schema							X		X	X	X	X							X	X	X
8.2.	Unplanned database synchronization							X	X	X												
9.	Use of business logic in communication among services	X	X	X																		

Continued on next page

Table 5.3 – continued from previous page

ID	Debt	Comp. A			Comp. B			Comp. C			Comp. D			Comp. E			Comp. F			Comp. G		
		D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P
10.	Reusing third-party implementations																					
10.1.	Many services using different versions of the same internal shared libraries	X	X	X							X	X	X	X	X	X	X	X	X			
10.2.	External dependencies with various licenses requiring approval				X	X	X															
11.	Overwhelming amount of unnecessary settings in the services	X	X	X	X	X		X	X	X	X	X		X	X		X	X	X	X	X	X
12.	Excessive number of small products				X	X								X	X							

Table 5.4: Catalog of Architectural Technical Debts, interest and principal.

ID	Architectural Debt	Consequences (Interest)	Solutions (Principal)
1.	Insufficient metadata in the messages		
1.1.	Insufficient message traceability	**A, E:** impossibility to identify and deactivate services that are not necessary anymore **A:** impossibility to track the source of messages, incurring costs with, for example, data tracing regulations	**A, E:** add services ownership metadata to the messages, allowing identification of their source **A:** implementation of a Canonical Data Model that ensure compliance
1.2.	Poor dead letter queue growth management	**A, E:** impossibility to identify the source of messages and determine the causes of the message loss in the dead letter queue	**A:** removal of the dead letter queue and to move the responsibility of the message deliveries to the endpoints **A, E:** add metadata to identify the source of the messages **A, E:** splitting the dead letter queue into smaller queues, managed by different teams
2.	Microservice coupling	**A:** increasing amount of unnecessary services **A, B, C, E, F:** too many dependencies among teams, creating coordination overhead; cascading changes in service consumers when producers are updated; eventual breaking on services **B, C, E, F:** eventual incidents in not updated services	**A:** use some time to design generic and independent services **C:** internal training about API development **E:** use of an API-first approach while designing services **F:** considering slot for continuous API improvement during development

Continued on next page

Table 5.4 – continued from previous page

ID	Architectural Debt	Consequences (Interest)	Solutions (Principal)
3.	Lack of communication standards among microservices	**A:** cost with translations among services; overwhelming amount of message formats for developers	**A:** ensure standardization with a Canonical Data Model
4.	Inadequate use of APIs		
4.1.	Poor RESTful API design	**C, D:** APIs are not stable, with frequent breaking changes, hard to use and frequently not backwards compatible **B:** instability demands the creation and maintenance of multiple API versions	**B, C:** additional effort to stabilize the API and avoid changes in the future **C:** management of API versions; tracking of internal and external consumers; definition of clear deprecation strategy **D:** definition of a standard for the APIs
4.2.	Use of complex API calls when messaging is a simpler solution	**D:** additional coupling among services; tests are inherently complex	**D:** redesign of services using a messaging approach
5.	Use of inadequate technologies to support the microservices architecture	**A:** big latency in the services communication; need of a dedicated team to maintain the third-party tool **C:** impossibility to provide some functionalities	**A, C:** proper planning about the technology and migration as soon as possible
6.	Excessive diversity or heterogeneity in the technologies chosen across the system	**A, E, F:** some services cannot communicate each other **E, F:** developers cannot easily migrate to other teams **A, F:** resistance to change technologies later **G:** developer velocity slows down, need to maintain distinct tools and additional source code repositories	**A, F:** limiting the set of technologies used by the teams **G:** use of language specific monorepositories and incentive their use for related projects: related software written in the same programming language are more likely to use the same tooling

Continued on next page

Table 5.4 – continued from previous page

ID	Architectural Debt	Consequences (Interest)	Solutions (Principal)
7.	Manual per service handling of network failures when target services are unavailable	**B:** extra cost on maintaining additional complexity in the architecture	**B, C:** use of third-party products (e.g., circuit breakers) that provide such mechanisms **B:** use of a service mesh
8.	Unplanned data sharing and synchronization among services		
8.1.	Sharing persistence or database schema	**D, G:** potential breaking on services **D:** complex database schema and difficulty to track services using the data	**C, D:** having separated databases for each service **C:** creation of distinct database schemes for each service inside the same database **G:** wrapping of the database within a service, preventing direct access
8.2.	Unplanned database synchronization	**C:** synchronization issues may be visible to users	**C:** the solution is context dependent, depending on the problem, a shared database might be needed, or a more complex transaction mechanism must be implemented
9.	Use of business logic in communication among services	**A:** unnecessary cost to maintain business logic in the communication layer	**A:** moving such business logic to the services, keeping the communication layer as thin as possible
10.	Reusing third-party implementations		

Continued on next page

Table 5.4 – continued from previous page

ID	Architectural Debt	Consequences (Interest)	Solutions (Principal)
10.1.	Many services using different versions of the same internal shared libraries	**A, E:** costs to plan and update all related services; several services may not be updated and multiple versions of the library should be maintained **A:** cost of issues generated by refusal to adopt by early adopters **D:** cost to handle breaking changes **F:** dependency between service and library developer teams	**A, D, E, F:** reduction in the use of shared libraries **D:** replication of simple code, creation of services to perform complex code functionalities
10.2.	External dependencies with various licenses requiring approval	**B:** delays while waiting for approval of new libraries	**B:** investing in a process to evaluate and approve external dependencies as fast as possible
11.	Overwhelming amount of unnecessary settings in the services	**A, B, C, E, F:** complex environment **D, G:** unexpected issues after deploy	**A, G:** creation of repository for configuration settings **A, F:** reducing of the amount of configuration settings on services **G:** requirement of peer approval before accepting changes on settings **C:** creation of a configuration server to automate deploy of configuration settings
12.	Excessive number of small products	**B, E:** governance on multiple projects instead of one (if a monolith)	Our study reveals no solution to this problem

We performed the qualitative analysis with NVivo[1], which tracks the links between codes created on top of the data and to the original quotations to which they were grounded.

Finally, we performed *member checking*, in which study participants could review the findings [Run+12] to increase reliability (Step 6 in Figure 5.8). We sent out a summary of our findings to at least one interviewee in each company and asked for review and feedback. We updated our descriptions according to the comments we received (Step 7 in Figure 5.8).

5.4 Results

Table 5.3 shows the companies' most critical ATDs and how they are distributed. For each company, an "X" in columns *D (debt)*, *I (interest)*, or *P (principal)* indicates, respectively, that the company has accumulated the debt and has identified (or paid/is paying) its interest and/or its principal. An empty cell in the table means that the interviewees did not report the related debt, interest, or principal for that company.

Table 5.4 shows the negative impact and the solutions for each ATD according to each company. As our respondents could not provide an actual numerical cost for the interest (i.e, the negative impact cost) and the principal (i.e., the solution's cost), we present the closest possible qualitative description of these costs that we could extract from our data. All the proposed solutions were applied successfully by practitioners in their projects.

The remainder of this section describes the identified debts and their interest and principals.

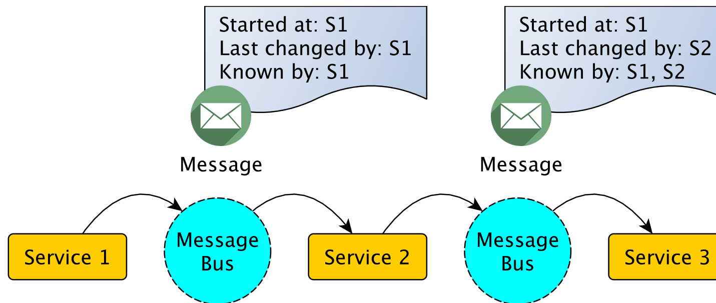
Debt 1: Insufficient metadata in the messages

An ATD is present when messages contain insufficient metadata. A data packet used in the communication through APIs, such as REST, may be considered messages. However, all data packets were sent through a message bus in our study. In such cases, the metadata is typically used to track messages and add other useful information at the cost of increasing overall message size.

This debt may be accumulated because developers want to keep the message lighter for performance reasons (Company A) or due to poor service planning (Company E).

[1] https://www.qsrinternational.com/nvivo/home

Figure 5.11: A message carrying metadata that is updated every time it is consumed and forwarded by a different service.



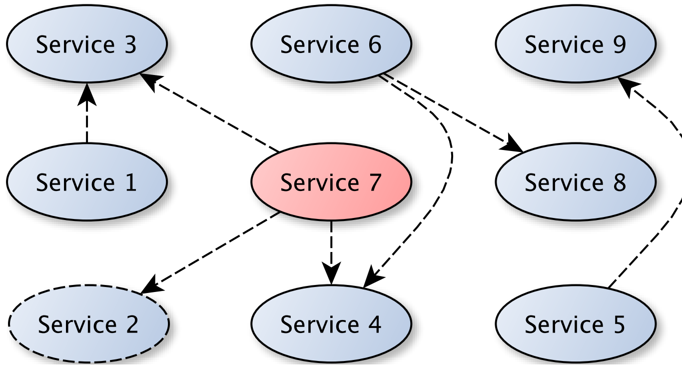
Debt 1.1: Insufficient message traceability

When messages contain insufficient metadata, developers might find it difficult to track the messages' source. Figure 5.11 shows services that deliver messages through a message bus. If no traceability metadata is available, Service 2 consumes a message from the message bus, but the service does not know which service produced the message.

Interest. The primary issue is the impossibility of tracking dependencies among services. One interviewee exemplified this debt by saying that "there is a regulatory requirement to document the traceability of data to the data source, and the lack of metadata and a data dictionary make it difficult to fulfill this requirement." Figure 5.12 presents a set of services and their dependencies. If Service 7 is no longer needed, Service 2 can also be deactivated because no other services use it, but Service 4 cannot be deactivated because it is used by Service 6. If it is impossible to track dependencies, it is not safe to deactivate dependent services, as in the examples mentioned above (Companies A and E). The number of services in the product will grow, and possible unused services such as Service 2 in Figure 5.12 will remain deployed and consume resources (Companies A and E). Besides, it is impossible to track the messages' sources when necessary. This incurs costs related to, for example, data-tracing regulations (Company A). In such a case, a financing company that, by law, must track financial operations might find this impossible to do because no available metadata indicates the sources.

Principal. Some interviewees described the primary solution as adding service ownership metadata to the messages as the (Companies A and E). Figure 5.11 gives an example of hypothetical metadata information attached

Figure 5.12: In the absence of a tracking dependencies mechanism, it is impossible to know that, if Service 7 is no longer necessary, Service 2 can also be deactivated, but Service 4 cannot.



to the messages. The metadata allows Service 3 to know (i) the service from which the message flow originated, (ii) the entire list of services that used that information, and (iii) the last service that changed the message. Company A went further and proposed defining such requirements with "the implementation of the canonical [data] model design pattern" to "ensure compliance with data traceability." A canonical data model is a design pattern in which there is agreement on and standardization of data definitions in different business systems [HW12] to ensure that the services contain the required information.

Debt 1.2: Poor dead letter queue growth management

A *dead letter queue* receives messages sent to a nonexistent or full queue and messages rejected for other reasons. A dead letter queue accumulates such messages to facilitate their inspection. The lack of mechanisms to control the dead letter queue's growth represents an ATD issue because the queue becomes "one place with lots of messages and no ownership."

Interest. Dead letter queues grow so quickly that inspecting the messages becomes impossible, so the queue consumes more and more resources. Cleaning up the queue without losing important information may be impossible due to the high number of dead messages. Besides, it may be difficult to predict the causes that lead such messages to be sent to the dead letter queue instead of their original destinations (Companies A and E).

Principal. The primary solution is to remove the dead letter queue, shifting

responsibility for message deliveries to the services (Company A). That would help solve any issues that cause the queue to grow, such as services not caring about the accumulation of messages. If that is impossible, Companies A and E agree that there are still two possible solutions, as articulated by one of our interviewees: "You should distribute this [the dead letter queue] either by having ownership metadata on the message or having distributed queues." Therefore, adding metadata that identifies the messages' sources would make it easy to handle lost messages because it would be easier to track their creators. Splitting single dead letter queues into smaller queues managed by different teams helps "divide and conquer" the problem.

Debt 2: Microservice coupling

Microservices should be designed to be mostly independent of each other and have well-defined boundaries and interfaces. A lack of knowledge or negligence about good design practices may lead a company to accumulate ATD related to tightly coupled microservices. One of our interviewees said, "How do you decouple the dependencies is always something you need to work out." We found this debt more often in the products that use APIs.

Interest. The accumulation of this debt creates too many dependencies among teams, so delays from one team may affect the other teams' development plans by, for example, delaying deliveries. Besides, someone with access to all involved teams must handle the unnecessary coordination overhead. This was exemplified by one case in which three teams were developing "three very tight microservices," and there was a "need to tightly coordinate the work between them" (Companies A, B, C, E, and F). Another cost incurred by this debt is the cascading effect of spending time and effort updating and deploying many dependent services due to changes made to one service (Companies A, B, C, E, and F). If accumulating this debt is a common practice in a company, the number of services easily increases. Various services end up not being useful in diverse situations because they are too solution-specific (Company A). Finally, the debt may cause incidents into dependent services that are not updated as required (Companies B, C, E, and F).

Principal. Designing services to be generic and independent usually incurs a cost, which is the principal, as also mentioned by Company A. Company E reduced the accumulation of the aforementioned debt by "being API first," which requires teams to consider what the service is instead of focusing on the code. According to the interviewee who made the suggestion, "It is more about orchestrating APIs," and "what's behind an API is no longer relevant." Company C considered using internal training about developing good APIs

to mitigate the problem. Company F proposed setting aside some time slots during development to continuously clean, refactor, and improve the APIs.

Debt 3: Lack of communication standards among microservices

When autonomous teams do not have proper guidelines or standard models for creating APIs or message formats (depending on how their services communicate with other services), the company may accumulate a debt in which many APIs or message formats emerge from the various teams because, as stated by an interviewee from Company D, “each message producer of messages is left to define the format of the data themselves.” While this debt might not be a problem in small systems, especially because microservices allow teams to decide their standards, having many standards will incur additional costs and become an issue. This is the “Tower of Babel problem” in our previous work [de +19]. We found evidence of this problem in Company A’s use of messages.

Interest. Developers often exert unnecessary effort when translating messages among distinct formats to allow different services to communicate. According to our sources, this debt leads to “data duplication, lack of consistency, and unwanted complexity.” The solution becomes overwhelmingly complex due to too many API or message formats. Each time one service must interact with another, the team that develops the first service must learn a new message or API format to define the proper translations (Company A).

Principal. According to Company A, “this problem is typically solved in organizations by using the canonical [data] model design pattern.” The implementation must be properly policed, or this model may also become complex and costly.

Debt 4: Inadequate use of APIs

Poor API design (i.e., failing to properly plan the API interface, error codes, etc.) may be the easiest way to have working code initially, but this has negative effects. In some cases, APIs are used in situations where other solutions, such as messaging, would be preferable. Such situations constitute the debt of inadequate API use. We present details below on each of those situations.

Debt 4.1: Poor RESTful API design

When using RESTful APIs, several conventions address their readability and use. For example, the REST Uniform Interface standardizes implementing *create*, *retrieve*, *update*, and *delete* operations in a resource. HTTP also includes

a list of status codes[2] that should be used in the API's responses. In our study, some developers tried designing RESTful APIs without following the proper conventions because they "were still focusing on the functional part of the job," resulting in a poor design (Companies B, C, and D). An example of this problem was using operations that should have performed a resource update but instead retrieved a collection of items (Companies B, C, and D)..

Interest. Poor API design causes several issues: (i) the API is difficult to use because its results may not follow expected conventions (Companies C and D); (ii) APIs are not stable, as one interviewee emphasized: "Should I make any changes, I need to make a new revision of the API" (Companies C and D); (iii) the API's instability requires the creation of new API versions and the need to maintain the old and deprecated versions (Company B); (iv) such changes also make it difficult to maintain backward compatibility in new versions of the API (Companies C and D); as a final result, (v) intentional and unintentional breaking changes[3] are common (Companies C and D).

Principal. Our interviewees suggested that APIs be standardized (Company D), versioned (Company C), and kept as stable as possible so that changes in the services do not affect their APIs (Companies B and C). Our interviewees also suggested using an explicit deprecation strategy. In other words, after the deprecation announcement, a previously defined period of support should not be extended (Company C). It is also useful to track internal and external consumers. Hence, it is possible to contact them directly and request migration to a new and more stable version of the API (Company C).

Debt 4.2: Use of complex API calls when messaging is a simpler solution

There are situations in which an asynchronous communication approach is particularly appropriate, such as executing long-running jobs. In other situations, such as when one service needs an immediate response from another after updating a user's address, synchronous communication is a better choice. A REST call is synchronous, whereas the messaging approach is asynchronous. Using REST when messaging is more appropriate constitutes another architectural debt because there are costs associated with using an

[2] https://tools.ietf.org/html/rfc2616#section-6.1.1

[3] A breaking change is a change in one part of a software (e.g., in a microservice's API) that potentially causes incompatibility with other components, causing failure. Examples of breaking changes for an API are changes in the response codes (e.g., *200 OK* to *201 Created* in HTTP) and renaming the location of the resource (e.g., renaming */user* to */users*, so previous clients are not able to find the related resource anymore).

improper solution. We found a specific instance of this problem in Company D, which described “rather complex service calls back and forth where messaging would have been a much better solution and allowed for much better testing.”

Interest. Instead of preparing services to respond to events (e.g., when a message arrives), API endpoints were created for each instance of communication among the services (Company D). Such a situation increases coupling among the services due to “a relatively complex handshake between two different services.” In other words, all the involved services depend directly on each other’s API endpoints instead of simply triggering an event (message). Besides, the services are harder to test due to the aforementioned complexity. Thus, the costs of maintaining such services increase.

Principal. According to Company D, the primary solution is “moving completely, for these particular cases, to a message passing” approach. The messages should be generic enough to be used by all the involved services without complex processing.

Debt 5: Use of inadequate technologies to support the microservices architecture

Technology choices may positively or negatively affect software architecture. Technologies used in microservices are different from those used in other architectural styles (for example, those ones used for service discovery and circuit breaking, as well as others discussed in Section 5.2.1). There are certainly technological similarities with other SOA approaches, but however, there are also differences (e.g., ESBs should not be used in microservices, as discussed in Section 5.2.1). For example, different cloud providers support different sets of tools and technologies, such as operating systems and storage software; some architectural choices simply do not work with them or face limitations. One interviewee said, “The technological base that we built a platform upon was not the best choice for what we wanted to offer.” Therefore, selecting an inadequate set of technologies, such as a Platform-as-a-Service or Infrastructure-as-a-Service provider that does not support the technology required for the software architecture incurs a debt.

Interest. This debt’s interest is context-dependent. In our findings, choosing the wrong technology as the message bus responsible for transferring messages among services caused considerable latency in such communication, with the consequent costs of requiring a team to maintain the third-party tool instead of working on other priorities (Company A). Company C could not provide certain new features because the previously selected platform did not provide enough sufficiently managed services. The company had to deal with

the costs of implementing the required functionalities without the availability of proper technologies.

Principal. Companies A and C reported that the primary solution should be planning the architecture before selecting the technologies because it is harder to change later on. Since that was impossible, they migrated to more appropriate technologies as soon as possible to prevent the interest from growing.

Debt 6: Excessive diversity or heterogeneity in the technologies chosen across the system

Selecting programming languages and related technologies are architectural choices that must be made in any project. Microservices give developers the freedom to choose different tools, programming languages, communication technologies, API standards, messaging technologies, and other technologies for each service. Although this is an advantage because some languages, frameworks, and technologies are more appropriate to specific tasks, such freedom can also lead to the interest described below. Such freedom may lead to debt, causing the company to have an excessively diversified environment.

Interest. We found the following set of issues in four companies: (i) Services that use one technology cannot communicate with services that use another, such as cases with REST APIs on one side and messaging technologies on the other (Companies A, E, and F). When services that use distinct technologies must communicate, a workaround must be developed. (ii) Individuals from one team cannot easily migrate to another because the necessary skills are quite different (Companies E and F). (iii) It is difficult to change the technology choices later because the services run for a long time and the teams become accustomed to their own choices (Companies A and F). Finally, (iv) developer performance decreases when teams must maintain distinct tools with potentially divergent setups (e.g., languages or environments) and when there is a need to maintain additional tooling source code repositories (Company G).

Principal. The primary solution reported by the companies was limiting the set of technologies available to the teams, especially those technologies used by multiple services and teams (Companies A and F). When talking about limiting the set of available technologies, one of the interviewed practitioners said, "I don't want too much alignment on that, but we need to keep the complexity under control and work to minimize the complexity of all parts." Besides, using programming language-specific monolithic repositories[4] allows

[4] This is also known as monorepos; it is a single source code repository for storing many projects.

related software (e.g., different microservices for distinct payment methods such as cash and credit cards) written in the same programming language to share tools from the same repository. Such repositories encourage using the same deployment tools, which may prevent a surge in different setups and support the merging of services because they use the same technologies (Company G).

Debt 7: Manual per service handling of network failures when target services are unavailable

When distinct microservices communicate synchronously (Section 5.2.1.1), they usually contact each other through the network. Unfortunately, several issues can occur during such communication. The communication channel may be overloaded, or the target service may be unavailable for many reasons, such as a lack of resources, crashes, or timeouts. One well-known approach is to create a mechanism within the service to retry contacting the target a fixed number of times. However, this approach has proven to be a debt because "every single developer has to think about how to handle that case themselves," leading to the costs described below.

Interest. Our interviewees mentioned that implementing the retry mechanisms manually on each service increased the code's complexity because developers must decide how the service handles the situation: "Should it retry, give up, or send a signal error? What should it do?" This approach increases the services complexity (and related maintenance cost) and increases the architecture's overall complexity (Company B).

Principal. The companies that reported this problem suggested reducing the complexity by using third-party products that provide features with retry mechanisms, such as circuit breakers (Section 5.2.1.3) (Companies B and C). One interviewee from Company B declared, "That is why we introduce the service mesh: to simplify that [the complexity created by developers when they must think about handling the retries]." Service meshes are introduced in Section 5.2.1.4 and usually contain circuit breaking mechanisms. Still, they should be used carefully and only when needed because they may cause an overhead, especially if the team does not have experience with them.

Debt 8: Unplanned data sharing and synchronization among services

Microservices may have their own databases, but they can also share or synchronize data with other services. In such situations, the company may incur the debt of not planning data sharing or synchronization among services,

leading to various costs. This debt could be interpreted as a way of causing coupling among services (Debt 2). Still, we consider it another because fixing it does not necessarily solve the coupling issue, nor do the fixes we present for solving Debt 2 solve the databases' issues.

Debt 8.1: Sharing persistence or database schema

Sharing the same persistent storage or database schema with multiple microservices is a critical architectural debt that can easily lead to high costs. For example, one company recognized the following situation: "We still have a common database today that is a technical debt that we are aware of, and we will have to get rid of this common database."

Interest. A service may require changes in the database schema or the data stored in the database. Such modifications may potentially break other services that use the same schema: "You easily break other services by changing the structure without even noticing it or noticing it too late" (Company D and G). If using the same database schema for different services is common in the company, an unknown number of services may use the same database schema. Therefore, it is difficult for a development team to know whether other services use the schema due to the lack of tracking for such information. This increases the odds of breaking other services (Company D). In such cases, the database design is complex and contains information from multiple services (Company D).

Principal. The ideal solution is to use separate databases for each service: "We are trying to split up that common database so that each service is responsible for its own database" (Companies C and D). It is also possible to wrap the database in a service, exposing it through an API instead of requiring direct access (Company G). Although the last solution does not solve the database complexity problem (i.e., the database still contains data from multiple services, which is more complex than having separate databases), it does reduce direct database schema manipulation. If wrapping the databases in this way is impossible for any reason (e.g., the services are business-critical and the migration cannot be done at once), our interviewees suggested using different database schemas for each service to enable the services to be changed independently (Company C).

Debt 8.2: Unplanned database synchronization

Microservices increase the likelihood of having distributed databases, but they may require synchronization. However, one interviewee said, "A big challenge is

data consistency in use cases involving multiple services and multiple databases, which must be somehow consistent or aligned to fulfill the use case successfully." In such a situation, the company may incur the debt of improperly planned synchronization.

Interest. The software composed of multiple microservices will remain inconsistent until all related databases are updated. This can lead to bugs. An interviewee from Company C said, "There were cases in which we had features that required us to basically align three databases to show the right information on a [user's] dashboard." Because we cannot present the real use case due to confidentiality restrictions, we explain the problem using a fictitious example: an online bookstore has 10 copies of a particular book. The store is developed using microservices and contains a microservice for purchases and a microservice for managing its inventory. Two users purchase the same book simultaneously, one for a single copy and the other for all 10 copies. Both users pay for their orders simultaneously, but the one who is buying all 10 copies finishes first. When the user buying a single copy finishes the payment, there are no books left because there is no mechanism to synchronize the inventory management and purchase services in this example. Company C encountered a similar problem. The costs vary depending on the business criticality of the affected product features.

Principal. Company C reported that distinct solutions could be considered depending on feature criticality. In some cases, a database must be shared by the services (see Debt 8.1), or a complex transaction mechanism must be planned, but no approach to implementing such a transaction was reported. We found no similar problems and solutions in other companies.

Debt 9: Use of business logic in communication among services

Microservices encourage the use of dumb pipes (i.e., simple message routers) for communication. The communication layer used by microservices should not include business logic. However, in projects like one developed by Company A, "the data transported changed within the communication channel itself." The changes are made by the services communication channel using business logic. Using business logic in the services communication layer constitutes an architectural debt because such a logic is not supposed to exist.

Interest. Maintaining additional business logic apart from the services is costly, as any changes to the services may also require changes to the communication layer where the business logic is located. Besides, "each time a new system is on-boarded, you need to set up the communication flow, requiring the communication channel team to provide the flow and possibly set up some

business logic." In other words, an external team—the communication channel maintainers—must understand details about how the related services work to implement the business logic (Company A).

Principal. Our interviewees suggested moving all business logic to the services themselves, thus allowing the communication layer to act as a dumb pipe. The costs involved are related to implementing such logic on each service: each team must understand and implement the changes independently (Company A).

Debt 10: Reusing third-party implementations

Reusing code can reduce resources while programming. In software development, reuse may occur by developing libraries used in various microservices. We call them *shared libraries* in the context of microservices because various services usually share them. On the other hand, reusable code—such as frameworks, language extensions, and libraries—may also be developed by external parties. We call codes from external parties *external dependencies*. For a single microservice, shared libraries and external dependencies can both be considered third-party implementations. We found evidence that using such third-party implementations may be an architectural debt and that the interest differs depending on whether they are shared libraries or external dependencies.

Debt 10.1: Many services using different versions of the same internal shared libraries

Companies may develop their own libraries to reuse code. Such libraries can act as black boxes for complex operations, and other reasons exist for using such libraries. One company states, "You could use REST, but if you want to be efficient, you want a native binding because it is faster." However, these libraries may constitute architectural debt if many services use such libraries and cause the interest described next.

Interest. Several negative impacts must be considered: (i) New releases of the libraries may require updates on every service using them. A roadmap must be established to handle such changes (Companies A and E). (ii) Several versions of the libraries must be maintained, because replacing old versions in all running services might be impossible for reasons like development priorities: "Sometimes the clients are business-critical and, in their roadmap, upgrading to a new version of a library it is not the top priority" (Companies A and E). (iii) Early adopters may refuse to implement new versions, especially if breaking changes exist (Company A). (iv) If library use cases are frequently unknown,

breaks may occur due to unexpected situations. Such breaks lead to fixes that may lead to breaking changes when libraries are updated (Company D). (v) The service's developers using the library depend explicitly on the team that is developing the library, so delays in releasing library versions with some required functionality will likely affect the service's developer team (Company F).

Principal. All the companies that mentioned the problem agree that they should avoid and discourage using shared libraries as much as possible (Companies A, D, E, and F). Company D suggested that a complex shared code should be transformed into services and that more straightforward codes should be duplicated by the different teams. Several practitioners suggested considering exceptions only when no better alternative exists "to keep the amount of shared libraries as minimal as possible" (Companies A, D, E, and F).

Debt 10.2: External dependencies with various licenses requiring approval

External dependencies are any libraries, frameworks, or similar software developed by external parties. We found evidence that their use might lead to architectural debt when the types of licenses allowed to be used by the company are strictly limited. Many products depend on some externally developed software; not accumulating this debt is almost impossible, but steep interest can be avoided.

Interest. All third-party codes' licensing limitations must be documented: "We need to document whether they are exportable in order to be able to perhaps include them in the main application and send them to trial, or even in order to be able to run them in a public cloud because that is also an export from one country to another." Eventually, some dependencies must be replaced due to non-compliance with regulations. Licenses may limit business models (e.g., it may not be possible to sell the product or service) and even prevent the software's distribution to some countries. Due to the high risk of regulation issues, approving such dependencies may be time-consuming and cause delays (Company B).

Principal. No current approach exists to handle this issue other than investing in a process to evaluate and approve such dependencies as fast as possible. Company B suggests that teams not use external dependencies whose licenses were not approved in advance to avoid this issue.

Debt 11: Overwhelming amount of unnecessary settings in the services

Microservices can be reused and deployed in various settings by tweaking parameters. One company explained the situation: “Microservices tend to expose some configuration settings that can basically be overridden. So, you can set a default value, and then whoever is using or deploying your microservice can override it at deployment time. When we add many microservices together and aggregate them into big settings trees, handling these kinds of parameters becomes very overwhelming.” For instance, allowing parameters to be overridden in different environments adds some control over resource usage. However, the company may incur debt caused by many unnecessary configuration settings among the services, which leads to higher maintenance costs.

Interest. Managing many parameters takes time and leads to overhead while deploying products (Companies A, B, C, E, and F). Such debt also increases the likelihood of unexpected issues caused by wrong setting values: “It was a very frequent problem that deployments would go wrong because somebody changed something in deployment scripts, and it would take a while to figure out why it was going wrong.” The greater the number of values, the greater the likelihood of mistakes. More value combinations will also need to be tested. Mistakes require time to fix, and more tests will require more development time (Companies D and G).

Principal. Some interviewees suggested creating a repository to keep the configuration settings, usually managed by a control version system (Companies A and G). Besides, peer approval for every change in the settings repository before production helps prevent issues (Company G). Interviewees also suggested reducing the number of configurable settings on each service to the minimum necessary (Companies A and F). A configuration server to automatically apply changes from the settings repository to the deployments also simplifies managing the settings (Company C).

Debt 12: Excessive number of small products

By definition, a microservice is a small product with a single capability and its own life cycle from development to deployment. It includes documentation and any governance required for software products. For example, one company reported that “each of the microservices effectively became a very small product, and we have a full process for handling products.” This may lead to debt via an excessive number of small products.

Interest. Each microservice must be governed separately—which requires overhead with dedicated management, development, deployment, and maintenance. This does not exist with monoliths because they require only one deployment (Companies B and E).

Principal. Our study reveals no solution to this problem. The companies reported that they "haven't really found a good way to change our standard ways of working and documenting to handle that yet," and "these kinds of aspects are a bit difficult when you have very small units of software flowing all over the place."

5.5 Discussion

This section discusses our ATDs in different contexts, how they relate overall, how to avoid them, and the limitations of our study.

5.5.1 Debts in different contexts

We discuss the various ATDs in relation to the company application domain, the project stage, and other specific context factors. We aim to help practitioners understand, adapt, and apply our results to their specific contexts.

Difficulties with message traceability (Debt 1.1) affect financial systems more than other product types. Poor management of dead letter queues (Debt 1.2) was more common in companies that migrated from older approaches. Problems with coupling (Debt 2) affect most companies in this study; only Company G did not report critical coupling issues among services, but we have only one interviewee from this company.

The lack of communication standards among services (Debt 3) occurred more frequently in applications with many services; smaller products seem to avoid this problem (Table 5.1 shows each project's number of microservices). However, this conclusion requires more investigation because Companies F and G, which have many services, did not report it as a problem.

The inadequate use of APIs (Debt 4) is a debt that requires attention every time a new API is created or updated. Not all the companies reported it, but it affected most companies using APIs in our investigation (some other companies used primarily messaging approaches or a balance between APIs and messaging). Our interviewees did not discuss techniques to design good APIs. We focused on reporting our findings from the interviews. However, we believe that the API design plays an important role in fixing this debt. See, for example, the work of Mosqueira-Rey et al. [Mos+18], in which they

present a systematic approach for developing usable APIs. There might be a relation between this debt and Debt 3 (lack of communication standards among microservices), although it is not apparent from our data. Inadequate technology use (Debt 5) is usually harder to fix because, for example, changing an entire platform is hard. Many parts of a project may depend on chosen technologies, so changing them may require an entirely new project. Therefore, Debt 5 requires more attention at the project's beginning, when technologies are being chosen.

One solution proposed for solving the overwhelming diversity of technologies (Debt 6) is limiting the number of technologies used. This might contradict the definition of microservices because they are supposed to be independently deployable units that give practitioners the freedom to choose the technologies to use. However, the diversity should be reasonably limited in practice because it may lead to various problems, such as difficulties migrating developers to other teams in a company and a decrease in a team's performance because of the need to maintain potentially divergent setups. Additionally, despite monolithic repositories being suggested as a solution for some cases of this debt, there is a risk that using them for many microservices might lead to overloaded repositories. Therefore, a good practice may be using monolithic repositories with smaller subsets of microservices and only when such microservices share the same technology stack.

Manually handling retries while trying to communicate a temporarily unavailable service (Debt 7) may also require attention. This debt was only found while using APIs because the messaging technologies do not require direct access to another service—only to the message bus.

Issues with shared data (Debt 8) were rare, but their effects were some of the most dangerous ones (e.g., breaking other services). They require attention, especially while designing new services. Note that Debts 8.1 (Sharing persistence or database schema) and 8.2 (Unplanned database synchronization) are highly interconnected: Debt 8.2 might result from splitting a database, for example, from solving Debt 8.1. On the other hand, sharing a database among services, which is a solution proposed by some of the interviewees for Debt 8.2, may incur Debt 8.1. The last case can be avoided by using different database schemas.

Business logic in the communication layer (Debt 9) is dangerous, especially for legacy systems, and companies developing new products seem to be aware of this issue and are avoiding it successfully.

Misusing shared libraries (Debt 10.1) may generate high costs. Therefore, we argue that they should be used carefully and only when needed. Our interviewees did not report the same issues for shared libraries while discussing

external dependencies, such as frameworks. This study does not identify the reasons for differences among shared libraries and external dependencies. Issues with external dependencies (Debt 10.2) seem to relate to licensing and are restricted to companies with multiple deploys of the same software in various regions worldwide.

A common problem for every project using microservices involves many configurable settings in microservices (Debt 11). All the companies reported that the services' deployment is complex and error-prone because of the number of settings to define.

The excessive number of products (Debt 12) is hard to avoid. It may relate to the services' granularity or other unknown factors. It might simply be a drawback of using microservices. Thus, this problem must be investigated further.

Many of the debts described here may be found in other architectural styles. For example, poor RESTful API design (Debt 4.1) may be found in monoliths. However, they may be different in microservices because each service is a separate application, which adds to the costs. A monolith, for example, may expose a single API, whereas the same functionality might require the existence of multiple APIs in separate applications, with multiple points of failure instead of a single one. Also, microservices use several additional technologies not applied in other architectural styles. We introduced a general description of those technologies in Section 5.2.1. Many of those technologies were originally developed or adapted to support SOA, but others were created specifically to support microservices, as discussed in Section 5.2.1.

Other approaches that have been proposed in grey literature can help manage ATD. For example, regarding Debt 8.1, Newman [New19] suggests using database views and creating a wrapping service. Newman [New19] also discusses other patterns that we could not match with our findings, including migration patterns, user interface composition, and database synchronization. Although such patterns might have been used, they were not mentioned by our respondents.

5.5.2 Common interests and principals among the debts

Understanding how debts are related is essential to better plan and analyze the consequences of refactoring. Figure 5.13 illustrates the debts, their interests and their principals. It shows which interests and principals are common among the debts. It was built by grouping the interests described in Table 5.4 into higher-level categories. Then we connected all the debts to these high-level interest categories to show which debts generated similar interests. For

example, the extra work for handling cascading changes and a higher number of unnecessary services from Debt 2 and the overwhelming amount of message formats for developers to handle from Debt 3 were both categorized into "development overhead." Some interviewees explicitly stated some relationships; the researchers inferred others through the cross-case analysis.

Finally, we repeated the same procedure for the principals, but we found only two principals shared between debts: the metadata standardization and the design of generic services.

Development overhead is caused by nine out of the twelve debts we reported (or eleven of sixteen, considering the sub-debts). Thus, it is not possible to reduce such development overhead without investing in repaying many different debts. The developers are the ones most affected by such overhead.

Potential breaks within the services are caused by four debts (Debts 2, 4, 8, and 10). Investing in paying those debts might reduce the probability of having to deal with instability and cascading failures.

Dependencies among teams are caused by three debts (Debts 2, 4, and 10). Such dependencies contribute to delays in the project as well as productivity loss because the developers on one team must wait for another team to start their work. Similarly, team velocity reduction is caused by three debts (Debts 4, 6, and 10). It may be advisable to pay extra attention to Debts 6 and 10 because they are responsible for many different interests.

Figure 5.13 also shows other specific interests and principals that are related to only one or two debts.

We expect a mapping like the one shown in Figure 5.13 to help practitioners to focus on the debts they want to manage according to the most costly issues they perceive in their projects.

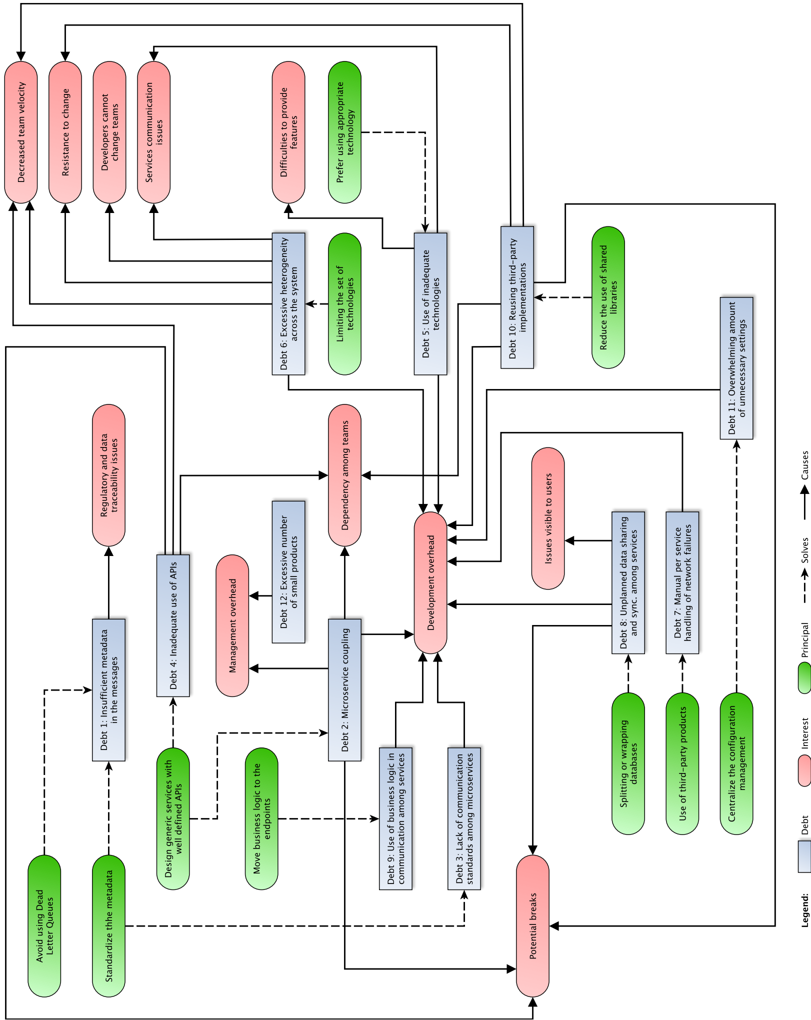


Figure 5.13: Common interests and principals among the debts.

5.5.3 Microservices coupling

We have found evidence that four debts (or six, considering the sub-debts) were indirectly increasing the probability of microservices coupling (Debt 2) and its consequences. Debt 3 (Lack of communication standards among microservices) leads to an excessive number of APIs or message formats, increasing the likelihood of having formats designed as coupled by default. As a result, the involved services cannot communicate with ones other than those originally developed for such interaction.

The main consequence of Debt 4 (Inadequate use of APIs) is having a set of services whose boundaries are not well defined, leading to the creation of functionalities in the wrong services. Thus, some services that should work independently must work together (i.e., they are coupled because some functionality required by one of them is in another service).

Debt 8 (Unplanned data sharing and synchronization among services) may cause an indirect coupling through the database. Changes in the database triggered by some change or additional functionality may eventually cause breaks in the other services. The services that share the database depend on each other's database structure.

Finally, Debt 9 (Use of business logic in communication among services) might lead to coupling because the logic is usually coupled to the services it is designed for.

In our study, coupling seems to be a common issue to be handled by microservices developers. It was mentioned several times by developers not only as a debt itself as described in Debt 2 but also as an indirect consequence of other debts. Investing in repaying the four debts (in addition to Debt 2) above might reduce the probability of having coupling among services.

5.5.4 Approaches for avoiding ATDs in microservices

Some techniques described in the non-peer-reviewed literature may help handle some ATDs reported in our research. Two techniques that may connect to two ATDs reported by the companies in this work are Domain-Driven Design (DDD) [Eva04] and Sagas [GS87].

One of the most significant challenges of using microservices relates to boundaries between them. The wrong boundaries may increase coupling among microservices (Debt 2.1). DDD is a powerful approach for defining microservices boundaries, but using DDD off the shelf may be hard in practice. It deserves more research, however.

To remove Debt 8.1, some of the studied companies chose to separate databases. None of the interviewees shared details about approaches they might have been using to avoid inconsistencies in the data. However, three approaches are: using the *eventual consistency* model (optimistic replication) [Vog08], *sagas* [GS87], and *two-phase commits* [RK98].

Vogels [Vog08] explains that a system using an eventual consistency model does not guarantee that subsequent accesses will return the last updated value right away, but eventually all accesses to the data will return the last updated value. The delay in the consistency is a trade-off versus high availability in distributed systems.

Garcia-Molina and Salem [GS87] proposed sagas in 1987. Today, the technique is adapted to microservices. Sagas involve implementing transactions among the services via a sequence of local transactions in several services (a saga). If one transaction fails, a set of compensating transactions may restore the databases' previous state.

Two-phase commits are discussed in distributed database research (e.g., [RK98]). This technique consists of preparing the commit (Phase 1) and storing it permanently (Phase 2) after an agreement from all involved parties. Newman [New19] argues against the use of two-phase commits mainly because of the existence of a window of inconsistency that might lead to problems. When the data must be in two different places, Newman [New19] suggests using sagas instead. Regarding solutions for ATD, both approaches require further investigation.

5.5.5 Microservice architecture maturity

Microservice architecture is still maturing. We notice that practitioners' different understandings about what comprises a microservice lead to different practical decisions. On the other hand, SOA is a more mature concept supported by several standards, such as the family of WS-* standards for web services. Microservices are a way of implementing SOA, but there are no standards such as the WS-* to guide microservices' implementation, making practitioners less supported with best practices.

Large companies tend to have heterogeneous and experienced staff members with distinct backgrounds. Such a variation among staff members may generate different opinions about solving the same problems. Because emerging architecture's definitions and related technologies are still maturing, the different opinions increase practitioners' likelihood of struggling with ATD in such environments. Consequently, knowing about frequent and costly ATD in microservices is important.

5.5.6 Limitations

The sample size of respondents from companies is limited, particularly from Companies F and G, with only one interviewee each (see Table 5.2). This might create bias because we could not triangulate the data via additional perspectives from other participants from the same company (source triangulation).

Furthermore, the fewer people we interviewed in a company, the fewer smells may have been found. Note that not finding a particular debt does not mean the debt does not exist—only that we were unable to find it.

Also, we selected our interviewees through convenience sampling. Thus, the debts we found may not be representative of the respective companies.

Returning to the interviewees multiple times may make them biased during the process, either in favor of the study or in favor of (or against) their system. Still, the number of return interviews with the same interviewees was low, and we primarily asked about problems other than those discussed before.

Questions 7, 9, 10, and 11 emerged during our interviews and were directly related to the debts identified in the preceding interviews, so they may introduce some bias to our results. We mitigated this issue by asking those questions at the end of the interview after asking open questions about their existing debt. By doing so, we avoided anchoring the subjects' answers to specific debts. Also, interviewees may be uncomfortable stating their thoughts for several reasons and may not tell the whole truth. We mitigated the problem via source triangulation. In cases with only one person (Companies F and G), it was impossible to achieve source triangulation.

The purpose of a multiple-case study is to investigate a set of cases in depth, not to generalize findings statistically. Yin [Yin18] states, "rather than thinking about your case(s) as a sample, you should think of your case study as the opportunity to shed empirical light on some theoretical concepts or principles." Practitioners may judge to what extent our findings apply to their particular context based on similarities and differences between their company and the investigated companies. A survey does not provide a rich context-related insight but enables reporting results statistically. A natural step further in our research is to conduct a survey where we collect opinions from more people and companies regarding the identified ATDs, interests, and principals.

5.6 Related Work

We identified a set of ATDs in microservices that could hinder the adoption of microservices. A deeper study about barriers (and drivers) for adopting

microservices comes from Knoche and Hasselbring [KH19], who identify issues related to compliance, regulations, and licenses as barriers. We identified that licensing and regulations might become an ATD (and a barrier) for microservice architecture.

This article extends our previous work [de +19] by investigating six additional companies. The current results confirm the found debts in our previous single case study apart from business logic in the communication among services. The current study resulted in cross-company insights and, specifically, an update of our catalog of ATDs. Moreover, the originally-proposed debts in de Toledo et al. [de +19] were changed as follows:

- "Too many point-to-point connections among services" is better explained as a coupling issue (Debt 2) and is also related to Debt 3;
- "Business logic implemented in the communication layer" is now described in Debt 9;
- "There is no approach to standardize the communication model among services" is described in Debt 3;
- "Weak source code and knowledge management for different services" is removed from our catalog because we found that it is better classified as a distinct, non-architectural type of debt;
- "Unnecessary presence of different middleware technologies in the communication among services" is merged into Debt 6, which is a more general debt description.

Taibi et al. [TL18] interviewed 72 developers and built a catalog of bad smells on microservices. More recently, Taibi et al. [TLP20] provided a taxonomy of architectural and organizational anti-patterns in microservices and their possible solutions. There is a close relationship among ATDs, architectural smells, and architectural anti-patterns, but they are different concepts; not all bad smells and anti-patterns are ATD: a code duplication may be considered a bad smell, but not TD if there is no interest. Still, some of the ATDs we have identified can be considered anti-patterns. However, describing them as ATDs enables us to evaluate them in terms of interest and principal. For example, some smells identified by Taibi et al. [TL18] overlap with ours: (i) shared persistence, which overlaps with Debt 8.1; (ii) too many standards, which overlaps with Debt 6; (iii) shared libraries, which overlaps with Debt 10.1; and (iv) Microservice Greedy, which overlaps with Debt 12. However, no

solution is offered other than careful consideration of services to create. They present the same problems and solutions we found in our interviews, reinforcing the importance of these problems. No other overlaps exist. Our study also presents more details about the debt and its interest and principal.

Hasselbring and Steinacker [HS17] argue that transforming internal libraries into open-source software may reduce issues with shared libraries (i.e., it may solve Debt 10.1). Despite insufficient evidence to confirm their suggestion, we believe more in-depth studies could confirm or refute such findings.

Bogner et al. [Bog+19a] performed a qualitative study with 10 companies via 17 interviews to explore evolvability assurance processes for 14 microservice-based systems. Many of the issues reported were related to ATD. Our work differs in the research focus: they investigated evolvability assurance processes and came across ATD issues, but we systematically investigated the debt, interest, and principal in microservices' architecture. Their work partially overlaps with the following ATDs we found in our study: (i) technological heterogeneity, in which they discuss that their participants are divided about the use of several different technologies in microservices, which relates to Debt 6; (ii) inter-service dependencies and the ripple effect, which refers to Debt 2; (iii) breaking API changes, which is an interest of Debt 4.1; and (iv) distributed code repositories, in which they argue that it may complicate the access to the source-code and relates to Debt 6 as well.

In summary, some related studies overlap with ours in a limited way, but none is as extensive and comprehensive as ours concerning ATD in microservices.

5.7 Conclusions and Future Work

During software development, it is vitally important to manage ATD to avoid extra costs in the long term. We provided a cross-company analysis to create a catalog of ATDs in microservices, their consequences (interest), and their solutions (principal). Moreover, we created a map of relationships among ATDs, their interest and principal. Such a map may support practitioners in identifying and avoiding ATDs and planning refactorings to remove them.

Regarding RQ1, we found ATDs that included business logic among services, shared databases, lack of data-traceability mechanisms, poorly designed APIs, and shared libraries. As for RQ2, we observed that such debts caused substantial interest, such as unexpected breaks due to changes in the database schema or other dependencies, unnecessary API complexity, coupling among services, and dependencies among teams. Finally, for RQ3, we identified how companies handle such ATDs and the ATD costs.

Future work includes running a survey to increase our results' generalizability and collect additional information on repayment prioritization. Furthermore, based on the insights reported in this article, we propose a new study that investigates metrics for measuring debt, principal, and interest in microservice architecture to quantify costs and benefits and support prioritization and decision-making.

Acknowledgements. We are grateful to all the interviewees of this study. We thank the anonymous referees for their comments.

Authors' addresses

Saulo S. de Toledo University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, saulos@ifi.uio.no

Antonio Martini University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, antonima@ifi.uio.no

Dag I. K. Sjøberg University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, dagsj@ifi.uio.no

Chapter 6

Improving agility by managing shared libraries in microservices

Saulo S. de Toledo, Antonio Martini, Dag I. K. Sjøberg

Published in *Lecture Notes in Business Information Processing (LNBIP)*, June 2020, volume 396. DOI: 10.1007/978-3-030-58858-8_20.

2

Abstract

Using microservices is a way of supporting an agile architecture. However, if the microservices development is not properly managed, the teams' development velocity may be affected, reducing agility and increasing architectural technical debt. This paper investigates how to manage the use of shared libraries in microservices to improve agility during development. We interviewed practitioners from four large international companies involved in microservices projects to identify problems when using shared libraries. Our results show that the participating companies had issues with shared libraries as follows: coupling among teams, delays on fixes due to overhead on libraries development teams, and need to maintain many versions of the libraries. Our results highlight that the use of shared libraries may hinder agility on microservices. Thus, their use should be restricted to situations where shared libraries cannot be replaced by a microservice and the costs of replicating the code on each service is very high.

Contents

6.1 Introduction 96
6.2 Background 97
6.3 Methodology 97
6.4 Results 98
6.5 Discussion and Threats to Validity 103
6.6 Conclusions and Future Work 104

6.1 Introduction

A microservices architecture may be considered a kind of agile architecture. Over the years, large companies such as Amazon and Netflix shared their success histories with microservices on dozens of presentations[1], always highlighting how such architectural style helped them to be agile and surpass many of the limitations and impediments they had in their previous monolithic software solutions. Since then, many other companies and practitioners tried to learn about microservices and adopted them in their projects.

However, systems that use microservices may become more complex than monolith systems [STV18]. Practitioners are still struggling with the adoption of this architectural style in their projects, and there is not much knowledge about Architectural Technical Debt (ATD) in microservices [de +19].

ATD is a metaphor used to describe architectural suboptimal decisions that, in exchange of benefits in the short term, incurs future additional costs for the software. There are many studies on ATD in general but few on ATD in microservices and no discussion about agility. Our previous study [de +19] investigates what is ATD in microservices through a qualitative case study in a single company, Lenarduzzi and Taibi [LT18] presents a position paper about code debt on microservices, also in a case study in a single company, Bogner et al. [Bog+19a] performed a qualitative case study in 10 companies to explore evolvability assurance processes for microservice-based systems. The three studies have distinct scopes.

In this study, we investigate the practice of using shared libraries in companies that use microservices, and how do these companies manage such libraries in order to improve agility. We define a *shared library* as a piece of software developed in-house containing a collection of resources used by several components. Externally developed components such as frameworks and language support extensions are not considered shared libraries in this study. Shared libraries are used as a black box by the different components, have their own version management and are copied and bundled together with the components.

Taibi and Lenarduzzi [TL18] have shown that the use of shared libraries may be a microservice bad smell and have proposed solutions for removing the smell. We extend that work by presenting an expanded list of issues and solutions, and do it in the context of different companies.

We pose the research questions as follows:

[1]Examples of presentations are "Mastering Chaos" by Josh Evans (Netflix, 2016), "Amazon and the Lean Cloud" by Werner Vogels (Amazon, 2011) and "What We Got Wrong: Lessons from the Birth of Microservices" by Ben Sigelma (Google, 2018).

RQ1: Which practical issues when using shared libraries in microservices hinder agility in organizations?

RQ2: Which solutions do developers apply to solve such issues?

In order to answer these questions, we conducted a multiple-case study in four large international companies that use microservices. The remainder of this paper is structured as follows: Section 6.2 presents our background, Section 6.3 our methodology, Section 6.4 our results, Section 6.5 our discussion and threats to validity. Section 6.6 concludes and outlines future work.

6.2 Background

Using microservices architecture is an approach that decomposes a single application into a collection of small and loosely coupled services; such services are autonomous, independent of each other and run on separate processes [LF14]. A few other characteristics are also taken in consideration while defining microservices, such as loose coupling, organization around business capabilities and ownership by small teams.

Microservices may improve agility by allowing teams to focus on small pieces of software, facilitating aspects like change, scalability and testing. As it raises new ways of developing software, it also raises new kinds of ATD [de +19]. If properly managed, the accumulation of ATD may be beneficial to the software development, but it is necessary to know when the debt should be avoided and how to prevent its accumulation [MB16b].

ATD is based on financial terms and has three main concepts [BMB18]: *debt*, which describes a sub-optimal solution that yields short-term benefits, but recurring to the later payment of some interest; *interest*, which is the additional cost that has to be paid because of the accumulation of debt; and *principal*, which is the cost of refactoring in order to remove the debt.

6.3 Methodology

We conducted a multiple-case study in four large international companies, with more than 1000 employees. For confidentiality reasons, the companies are named *A*, *D*, *E* and *F*, respectively. The studied projects operate in the domains as follows, respectively: financial systems, healthcare systems, city management and transport mobility.

We interviewed six architects: one from Company A, two from each of Companies D and E, and one from Company F. We conducted semi-structured interviews that lasted from 30 minutes to one and a half hours. We discussed

Table 6.1: Issues reported by companies as the result of using shared libraries

ID	Issue	Company			
		A	D	E	F
1	Impossibility to update library in service due to priorities	X		X	
2	Need to maintain too many versions of the library	X		X	
3	Impossibility to update library in service due to breaking changes			X	
4	Delays while waiting for fixes		X	X	X
5	Early adopters refusing to migrate	X			
6	Failures due to unknown use cases		X	X	
7	Failures after library upgrades		X	X	
8	Overhead to library maintainers		X	X	X
9	Dependent agile teams	X	X	X	X

several aspects of architecture beyond the scope of this investigation, such as architectural issues and solutions while using microservices. The questions in the interview guide relevant to this study are available at https://bit.ly/ImprAgilitySL. Three of the interviews were conducted face-to-face. The three other ones were conducted through remote audio calls due to the physical distance between the parts.

6.4 Results

6.4.1 The issues caused by using shared libraries

Table 6.1 shows which issues related to shared libraries were found in which companies. We refer to the those issues by using their IDs between parenthesis in the following paragraphs. The context related to the issues discussed below is illustrated in Figure 6.1, an example reported by Company D: A team is assigned to create and maintain a library for authentication and authorization. Versions of the library are regularly released with fixes or new functionalities. Other teams are assigned to develop microservices. Eventually and due to several reasons, several microservices end up using distinct versions of the library. We present below the causes and implications of such circumstances for each company in the context of the projects we investigated.

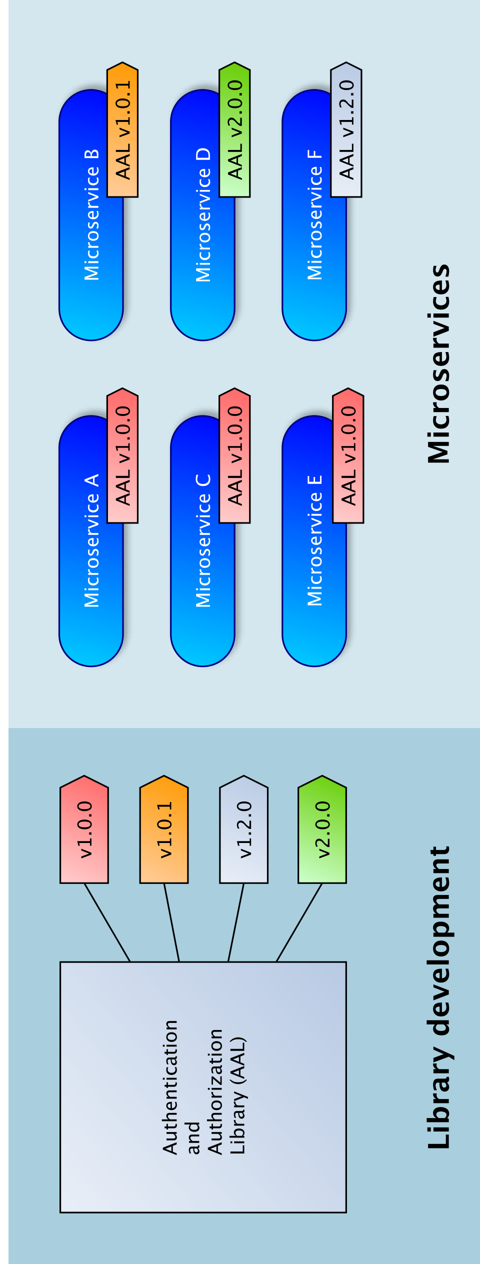


Figure 6.1: Shared libraries example

Company A could not migrate all the clients to a newer version of a library right after its release. Distinct teams have different priorities: some services are critical, some are secondary, some have more urgent updates (1). Such a scenario required libraries maintainers to be active in supporting previous versions of their libraries that were still being used in production (2). Even in situations where the library was supposed to be updated soon, the company experienced delays in the process due to other priorities (1). In addition, the company also identified situations where early adopters resisted to migrate (5), since a new version of the library was released right after they finished the integration of the previous version in their project.

In Company D, the developers experienced a number of system breaks. Later they identified that part of the breaks were caused by the use of libraries in many unforeseen and untested situations (6). In addition, Company D also noticed an overhead on library maintainers (8) and consequent delays. Since the functionality was provided by the libraries, the teams using them had to wait for the fixes, which caused delays in new microservices releases (4). In some situations, the new versions of the libraries caused new issues that prevented the microservices to be released in production right away (7).

Company E, similarly to Company A, found itself in a situation where it was not possible to migrate all the clients, which required teams to support many deprecated versions of libraries (2). Breaking changes and internal roadmap priorities were some of the factors that prevented developers to use new versions of the libraries (3 and 1). The use of shared libraries became a bottleneck, causing failures on microservices (6 and 7), delays while waiting for fixes (4) and an unexpected amount of extra work for library developers (8).

Company F reported delays in delivering new functionalities as the most damaging issue connected to the use of shared libraries (4). The library developers had to handle an extensive amount of change requests, including requests for additional features and fixes (8). The microservices developers were frequently blocked while waiting for the arrival of the new versions of the libraries.

In all four companies, there was a clear dependency (coupling) among the microservices developers and the library teams (9).

6.4.2 How to manage issues regarding the use of shared libraries

All the companies reported that the use of shared libraries should be reduced as much as possible. Company D reported that many libraries implemented

trivial functionality that could be implemented by the microservices themselves, and the fixes could be implemented by the teams, reducing the delays caused by third-party developers. Company F suggested that well-defined and well-documented interfaces of their own implementations were important for guiding practitioners when they did not use shared libraries to provide required functionality.

Figure 6.2 shows solutions proposed by the companies for the issues caused by the use of shared libraries. Considering the example presented in Figure 6.1, simple functionalities, such as extracting an ID or user name from a token, could be implemented by the services themselves. Such a functionality is easy to implement, usually by using a well-known technique that can be learned by the developers, and that does not require the use of an entire library. On the other hand, some functionalities are complex and could involve, as in our example, many security steps. In such circumstances, an external microservice with a well-defined interface, good documentation and a versioning policy should be maintained by a separate team. Well-defined interfaces should not be changed unless in exceptional cases, meaning that internal bug fixes may be conducted without the other services noticing it, and new functionality may be added without breaking previous behavior unless a breaking change is strictly necessary. Such a scenario reduces the need for changes in the other microservices that are using the aforementioned interfaces. Finally, if there are important reasons for not using one of the approaches above, the use of shared libraries may be acceptable. Similar approaches may be found in other migration reports. Balalaie et al [BHJ16], for example, moved common libraries to microservices when they migrated to such an architecture style. Hasselbring et al. [HS17] argue that code should not be shared among microservices because teams and applications should be as independent and loosely coupled as possible.

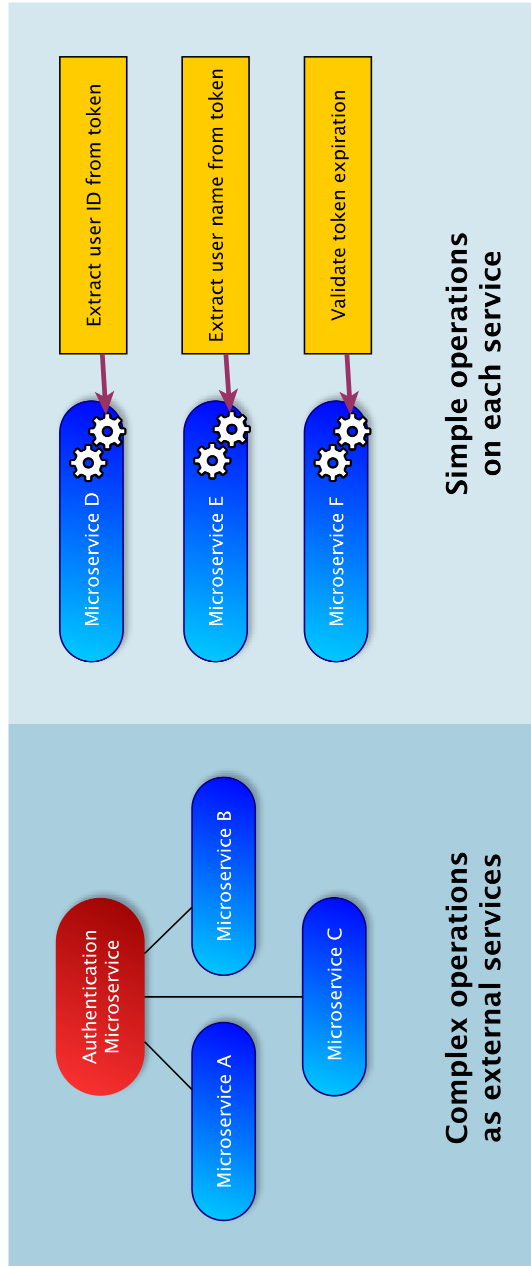


Figure 6.2: How to handle shared functionality

6.5 Discussion and Threats to Validity

Our results suggest that using shared libraries in some contexts impacts on the development flow, causing delays, reducing development velocity and hindering agility. In such cases, shared libraries are an ATD that may lead to costly interest if not managed properly. By sharing the experience from other practitioners on issues and solutions, we can prevent others from having to pay high software maintenance costs later.

We answer the research questions introduced in Section 6.1 by listing the issues (RQ1) raised by the use of shared libraries and by presenting corresponding solutions (RQ2). The issues we identified do not seem connected to any specific application domain; the practitioners from the different companies complained about similar issues and solutions. We do not claim that shared libraries should never be used. However, their use should be controlled to prevent high costs. There are also drawbacks of such an approach. For example, it may incur additional latency; performance may decrease due to network as opposed to in-memory invocations; reliability may decrease since the service might not be reachable; and complex functionality may not be possible to be implemented in a distributed system. Such drawbacks should be carefully considered in practical situations.

Companies should also consider the reasons for replacing their shared libraries. There may be alternative solutions, such as improving processes for development, testing and quality assurance, which should be considered when the drawbacks of moving to services may be more costly than using shared libraries.

Regarding the validity of this study, we consider the following threats: (i) The interviewees may have interpreted the concept of shared libraries differently. We mitigated this threat by asking the interviewees to clarify if they were talking about libraries developed internally or about external dependencies; (ii) Our sample of interviewees was small from each company, we do not know how representative the opinions in this study were for the investigated companies. Still, the sample was heterogeneous and the practitioners were located in three different countries, with projects from four different companies; (iii) There might be factors that the interviewees were not aware of or did not express in the interviews, such as the quality of the implementations and management issues.

6.6 Conclusions and Future Work

In four Europe-based companies, we identified a set of issues that reduce development velocity and hinder team agility while using shared libraries in microservices. We highlighted two solutions: creating additional microservices or implementing the code in the microservices themselves. Although these solutions have been reported by Taibi and Lenarduzzi [TL18], we went beyond their work by presenting and discussing a more comprehensive list of issues, and relating them all to the different companies. Our results suggest that the use of shared libraries may increase the complexity of the system, which in turn decreases development agility, cause delays and raises maintainability costs. Our results do not indicate that shared libraries should not be used at all, but if there are no acceptable alternatives, they should be used rather carefully as they often generate costly interest. As an alternative to the use of shared libraries, simple functionalities should be implemented by each microservice, whereas complex functionalities should be implemented by external microservices with well defined interfaces, good documentation and adequate versioning policies.

As future work, we propose a further investigation of the problem, increasing the size of the sample and looking for practitioners with different experiences. As part of this investigation, we propose to look for a decision process supported by the factors that influence the trade-off between using a shared library and a microservice. We would also like to investigate the problem and their solutions with other architectural styles, like Service Oriented Architecture, in order to identify whether there are other solutions proposed by practitioners that could be used in microservices. In addition, we would like to investigate the external dependencies and how moving to them could affect our results.

Authors' addresses

Saulo S. de Toledo University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, saulos@ifi.uio.no

Antonio Martini University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, antonima@ifi.uio.no

Dag I. K. Sjøberg University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, dagsj@ifi.uio.no

Chapter 7

Accumulation and prioritization of Architectural Debt in three companies migrating to microservices

Saulo S. de Toledo, Antonio Martini, Phu H. Nguyen, Dag I. K. Sjøberg

Published in *IEEE Access*, March 2022. DOI: 10.1109/ACCESS.2022.3158648.

Abstract

Many companies migrate to microservices because they help deliver value to customers quickly and continuously. However, like any architectural style, microservices are prone to architectural technical debt (ATD), which can be costly if the debts are not timely identified, avoided, or removed. During the early stages of migration, microservice-specific ATDs (MS-ATDs) may accumulate. For example, practitioners may decide to continue using poorly defined APIs in microservices while attempting to maintain compatibility with old functionalities. The riskiest MS-ATDs must be prioritized. Nevertheless, there is limited research regarding the prioritization of MS-ATDs in companies migrating to microservices. This study aims to identify, during migration, which MS-ATDs occur, are the most severe, and are the most challenging to solve. In addition, we propose a way to prioritize these debts. We conducted a multiple exploratory case study of three large companies that were early in the migration process to microservices. We interviewed 47 practitioners with several roles to identify the debts in their contexts. We report the MS-ATDs detected during migration, the MS-ATDs that practitioners estimate to occur in the future, and the MS-ATDs that practitioners report as difficult to solve. We discuss the results in the

3

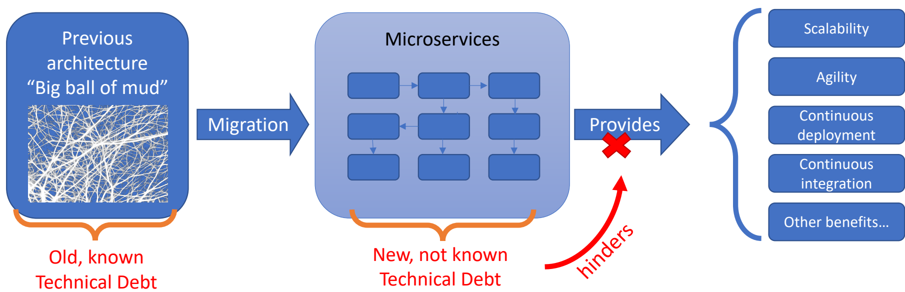
context of the companies involved in this study. In addition, we used a risk assessment approach to propose a way for prioritizing MS-ATDs. Practitioners from other organizations and researchers may use this approach to provide rankings to help identify and prioritize which MS-ATDs should be avoided or solved in their contexts.

Contents

7.1	Introduction	106
7.2	Background	110
7.3	Research design	117
7.4	Results and discussion	125
7.5	Limitations	160
7.6	Related Work	161
7.7	Conclusion	162

7.1 Introduction

Figure 7.1: New ATD found after the migration to microservices may hinder the benefits of the new architecture.



When companies migrate their software towards a microservice architecture, the software is split into a small set of independent services. Many of the practical difficulties encountered in previous architectures can be mitigated by using microservices. They support small and frequent releases, improve scalability, and promote independence among teams. However, microservices

also bring new management and technical demands, such as the need to be business-domain driven and understand distributed systems [Fow15].

Like any architectural style, the microservice architecture is prone to architectural technical debt (ATD), which may incur high costs [dMS21]. ATD is a type of technical debt (TD) consisting of sub-optimal architectural solutions, which deliver benefits in the short-term but increase overall costs in the long run [BMB16]. Some ATDs are specific to microservices [dMS21] and might not be considered as problems in other architectures. For example, microservices should communicate through a "dumb pipe" (i.e., there should not exist any transformation logic between the services), while in previous Service Oriented Architectures (SOA), a common approach is to have some logic between services to transform data. In this paper, we consider ATDs only in the context of microservices applications and thus name them MS-ATDs.

One of the reasons for companies to migrate to microservices is repaying known ATDs from their previous architectures while, at the same time, obtain the benefits of this new architectural style. Figure 7.1 exemplifies such a migration: a company has a "Big Ball of Mud" architecture [FY97] and repays ATDs in a migration to microservices. As the company believes that most ATDs from before were paid, the new microservice architecture can be prone to new, unknown MS-ATDs. The new microservice architecture is expected to have better scalability and agility, allow enhanced continuous integration and delivery pipelines, and provide many other benefits, such as allowing independent teams to work in parallel, having more testable code, and better control of costs in the cloud [Fow15; LF14]. However, the new MS-ATDs reduce these benefits and can be more costly than previous debts.

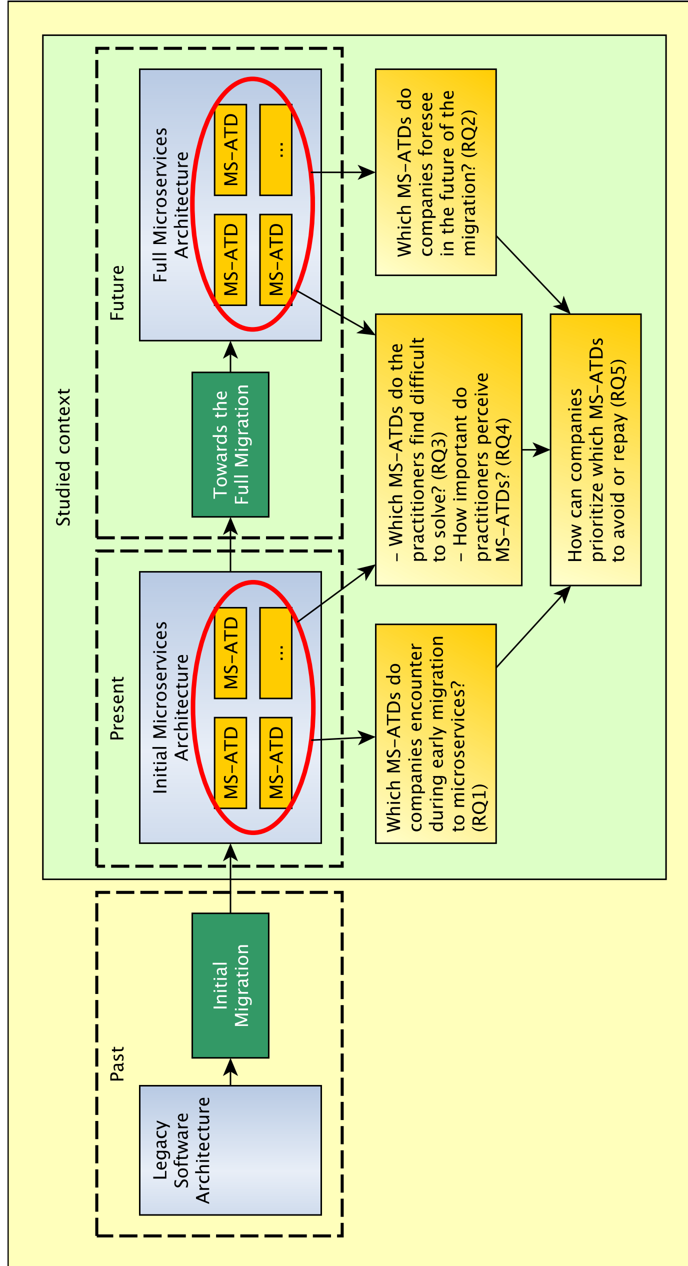


Figure 7.2: Relationship between the research questions, the ongoing architecture (with the migration in progress), and the final microservice architecture (as envisioned by the practitioners).

Migration to microservices has been investigated in previous studies from different perspectives. Several authors have proposed tools and approaches for assisting migration to microservices; see the systematic mapping by Bushong et al. [Bus+21]. A few other studies have investigated TD and related concepts in this new architectural style [dMS21; Len+20; TLP20]. However, none of these studies have covered how ATD has accumulated during migration from a prior architecture to microservices. A lightweight survey of the current literature looking for the terms "microservices," "micro-services," "prioritization," and "technical debt" in some of the major research databases (ACM Digital Library[1], IEEE Xplore[2], Scopus[3]) highlighted that there are no relevant papers on how to prioritize MS-ATDs. Existing secondary studies on microservices [Bog+19b; DLM19; DML17] do not address prioritizing MS-ATDs. Companies must address the costs of such debts at a later stage of migration. Furthermore, after observing ATDs from the old architecture being repaid during migration, practitioners might have a false impression that the project is going well without noticing new MS-ATDs.

During migration to a microservice architecture, practitioners have the opportunity to identify MS-ATDs in a timely manner before they become widespread in the entire architecture. Knowing the risky and costly MS-ATDs facilitates practitioners in deciding which MS-ATDs to avoid, remove, and prioritize repayment. This study investigates the following research questions (RQs) in companies that have started their migration to microservices:

- **RQ1:** Which MS-ATDs do companies encounter during early migration to microservices?
- **RQ2:** Which MS-ATDs do companies foresee in the future of the migration?
- **RQ3:** Which MS-ATDs do the practitioners find difficult to solve?
- **RQ4:** How important do practitioners perceive MS-ATDs?
- **RQ5:** How can companies prioritize which MS-ATDs to avoid or repay?

To answer these questions, we conducted a multiple case study of three companies in the early stages of migration to microservices. As shown in

[1] https://dl.acm.org/
[2] https://ieeexplore.ieee.org/
[3] https://www.scopus.com/

Figure 7.2, we investigated the present and future stages of migration to microservices in these companies. RQ1 aims to identify the debts that occur in the early stages of migration (*Present* in Figure 7.2) to microservices: TD is often introduced early and persists throughout the software life cycle [BMB17a]. RQ2 investigates the debts estimated to occur in the future, helping practitioners to avoid or mitigate them (*Future* in Figure 7.2). Answering RQ3 highlights MS-ATDs that are difficult to remove and thus may either require more effort to be repaid or remain in the system for a long time. Answering RQ4 highlights which MS-ATDs practitioners consider risky (important to them) and, thus, should be prioritized for removal or mitigation. The difficulty and the risk of the MS-ATDs affect how practitioners prioritize them. Finally, RQ5 investigates how companies can prioritize the removal or mitigation of MS-ATDs identified and discussed in the previous questions (see Figure 7.2). Prioritizing MS-ATDs is an important management activity [LAL15].

The remainder of this paper is organized as follows. Section 7.2 provides the background for this study. Section 7.3 describes our research design. Section 7.4 presents our results and discussion as well as implications for research and practice. Section 7.5 discusses the limitations of this study. Section 7.6 presents the related work. Section 7.7 concludes the paper and highlights future work.

7.2 Background

7.2.1 Migration to Microservices Architecture

Lewis and Fowler [LF14] defined the microservice architecture 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." In a microservice architecture, each microservice is autonomous, allowing developers to select the most appropriate set of tools and programming languages to be used. Small services tend to reduce code complexity and increase code maintainability. Moreover, because microservices are deployed independently, each microservice has its own delivery pipeline, can be tested independently, and can be scaled individually [LF14].

The microservice architecture is an alternative to monolithic applications, which are developed as a single unit [New19]. Compared with monolithic applications, microservices are easier to scale, have shorter cycles for testing, building and release, and are frequently less affected by downtime [Fow15]. However, the microservice architecture also has drawbacks and challenges. Having each service deployed separately introduces latency in communication,

requires the management of network failures, increases operational complexity, and demands the management of eventual consistency [Fow15]. During migration to microservices, practitioners reported extended time to release features, high coupling, and deficiencies in communication and knowledge sharing, among others [DLM18].

Microservices may be considered as a way of implementing Service-Oriented Architecture (SOA), although there are different opinions about whether microservices are an instance of SOA [Zim17]. There is a clear overlap between the characteristics of SOA and the microservice architecture. Many concepts and techniques in microservices have been borrowed from SOA, such as service discovery, service registries, API gateways, and circuit breakers [MW16]. Even so, SOA describes applications that cannot be considered microservices. For example, many SOA applications are still implemented using an Enterprise Service Bus (ESB), a centralized software component providing infrastructure to the services composing the application and mediating communication. An ESB may intercept and modify the data, among other functions [NG05]. On the other hand, microservices require a *dump pipe* for communication (i.e., a communication layer used simply to transfer data) without any modifications or transforming capabilities. Other characteristics apply for SOA but not for microservices, such as: there is no such guidance about the service granularity in SOA, while for microservices, each microservice should represent only one capability, and SOA may support transport protocol transformations, while microservices usually rely on REST over HTTP or a protocol supported by a message bus [RSZ17].

Well-known companies, such as Amazon and Netflix, have been using microservices to overcome difficulties with their previous monolithic architectures [LF14]. The success of microservices in these companies made other companies embrace this architectural style and migrate to it from their previous monolithic architectures. There are many reports on such migrations in the industry and academia [DLM19].

There are several approaches for migrating monolithic architectures to microservices, ranging from decomposition strategies to data-driven approaches [Len+20]. Many patterns have also been discussed in academia and industry to guide migration [New19; YM20]. Frequently, the migration is advised to be incremental in that microservices gradually replace the functionality in the monolith or new microservices are created to implement new features [YM20]. An incremental migration results in both the original and new architectures coexisting and working together.

7.2.2 Prioritization of Architectural Debt (ATD)

Technical debt (TD) denotes a suboptimal solution that delivers short-term benefits at the expense of increased overall costs in the long run [Avg+16]. An ATD is a type of TD related to a product's architecture [BMB16]. Findings from previous surveys identify ATD as one of the most challenging types of TD to unveil and manage [BMB17a; Ern+15; KNO12].

Microservice architecture is prone to ATD. De Toledo et al. [dMS21] listed 16 ATDs specific to microservices (i.e., MS-ATDs), which were organized into 12 more general ATDs. Despite the possibility of finding these ATDs in other architectural styles (e.g., APIs might be inadequately used in any architectural style), their causes and consequences are different from those in microservices.

Table 7.1 shows seven MS-ATDs, an example for each, and a brief explanation of what is specific to microservices in each debt. These MS-ATDs are a subset of those initially reported by de Toledo et al. [dMS21], and were selected according to the criteria described in Section 7.3.1. Table 7.1 also lists the number of each MS-ATD in de Toledo et al. [dMS21] for correspondence.

MS-ATDs must be prioritized before repayment [LAL15]. One way of prioritizing is through risk assessment [Guo+11; MB16b]. Risk represents the possibility of loss (suffering the impact of the debt). The MS-ATDs that pose the highest risk should be addressed first. Through risk assessment, we developed a systematic approach to identify, analyze, and evaluate the risk of each MS-ATD. We define the risk of a debt as the probability of the debt to occur multiplied by its impact, as shown in Equation 7.1. We use this definition in Section 7.3.4 to address the risk of the debts.

$$risk(debt) = probability(debt) \times impact(debt) \tag{7.1}$$

Table 7.1: The MS-ATDs selected for this study.

#	Name and description	Example	What is new in microservices?
1	*Insufficient metadata* (#1 in [dMS21]): Many microservices communicate through messaging. However, these messages could have additional metadata to identify their producers, consumers, targets, and others in some cases. Insufficient metadata may make it challenging to track dependencies among services and find the producers for debugging purposes, as well as other issues.	H system comprises many microservices processing data available through a message bus. In this example, a message can only be consumed by one service at a time, but there is no guaranteed order. If the message gets malformed, the reason might be related to a specific combination of modifications made by previous services. If there is no metadata for tracking the changes, it might be challenging to identify the causes of the issue.	Different SOA approaches might use messages in their communication, but microservices are more fine-grained than those, which increases the number of services and, consequently, the impact of the debt. Monoliths are self-contained and usually do not need messaging approaches to establish communication among their modules. On the other hand, Microservices might critically depend on messages to communicate with other services because they form a distributed system.
2	*Microservice coupling* (#2 in [dMS21]): Microservice coupling is about how changes in one service require changes in another service. The coupling might be done intentionally to save development time, and it frequently increases team dependency. There are different types of coupling [New19], but we focus on the services' implementation, including their contracts and interfaces (e.g., API endpoints and message formats).	H *files* service provides access to a set of files for authorized users. The authorization is currently performed at the *users* service. For every request received by the *files* service, another request is made to the *users* service to verify access to the files. Changes to the *users* service's API might affect the *files* service, creating a dependency among the development teams. The files access authorization should possibly be moved to the *files* service instead.	Joupling among microservices frequently causes dependency among teams, reducing teams' velocity and agility. While companies have a false impression that they have decoupled teams and well-defined agile practices, they might be silently blocked by coupling among the microservices.

Continued on next page

Table 7.1 – continued from previous page

#	Name and description	Example	What is new in microservices?
3	*Inadequate use of APIs* (#4 in [dMS21]): Many services communicate with the other services through APIs. When these APIs are not well defined or misused, they lead to issues.	Hn API exposed through HTTP that is not following required standards. For example, removing an item from a shopping cart is done through an HTTP method called GET, but it should use DELETE instead. This API's users have difficulties understanding it.	Each microservice exposing an API might be developed by a different team, increasing the probability of having many different API standards. The inadequate use of APIs impacts the functioning of other services and development teams.
4	*Excessive diversity* (#6 in [dMS21]): Microservices allow mixing multiple programming languages, data-storage technologies, supporting tools, and others. However, having too many different technologies across the system may create difficulties with standardization and management.	Using containers is a common technique in environments running microservices. Developers can easily find start containers running almost any GNU/Linux-based system to use in their projects. However, there are several hundreds of containers setups, and each one uses its own distinct tools. Using too many distinct containers for solving the exact same problem requires team members to learn a different setup every time they change teams.	H monolith is usually developed using a limited set of programming languages and tools. On the other hand, Microservices can be developed with completely distinct languages and setups, increasing the likelihood of excessive diversity. Other SOA approaches might also suffer from this problem, but the number of services in a microservice architecture is usually higher, making the problem more costly.

Continued on next page

Table 7.1 – continued from previous page

#	Name and description	Example	What is new in microservices?
5	*Unplanned data sharing/synchronization* (#8 in [dMS21]): Microservices should have their own databases. When different microservices share the same database, unexpected issues such as cascading breakings may occur. On the other hand, when microservices have distinct databases, there might be problems with data synchronization.	Two microservices share a database and use a table of users. There is a field in the database storing the users' full names. The developers in the first service decide to split the *full name* column into *first name* and *last name*. The second service might stop working because it does not find the original field for the full name.	Microservices should have their own databases, which is not required for other architectural styles, including other SOA approaches. The way the databases are designed for microservices is also different: they should reflect the business domain, which leads to distinct domain-related issues. Sharing databases or synchronizing them might lead to blocking and other issues among teams.
6	*Misusing shared libraries* (#10.1 in [dMS21]): Many companies encapsulate code into libraries and distribute them to be used by many services. Suppose such a distribution is not properly managed. In that case, many libraries may lead to difficulties, including breaking changes, dependencies among teams, delays, and additional costs to update all the services using the libraries on every new release.	H data encryption library is developed by a separate team in the company and used by dozens of services throughout the project. A high-security issue is found in the library, and the library developers release a new version with the fix. Every service should update the library to the last version. Due to other priorities and feature development, it is not possible to update the library in the entire organization, and many services will remain with the issue.	Libraries have different consequences in distinct architectural styles. Monoliths, for example, bundle them in a single deployment package, while microservices do the same for each deployment. A system with hundreds of services may have hundreds of deployments of the same library. Thus, issues found in a single library immediately affect dozens or hundreds of services and teams.

Continued on next page

Table 7.1 – continued from previous page

#	Name and description	Example	What is new in microservices?
7	*Unnecessary settings* (#11 in [dMS21]): Each microservice usually has a set of settings to be defined in the environment, such as the database address, memory limits, and others. However, if there are many unnecessary settings, the probability of misconfiguration is more significant, potentially leading to crashes or other issues.	One of the configuration settings for accessing databases is to inform a "port number." Databases run in a default port if not changed. For example, it is unnecessary to have a configuration setting in an application to inform the default port of a database management service. The application could have an optional configuration setting that, if not present, automatically falls back to the default value.	There are several times more settings in microservices than in monolithic architectures, increasing the likelihood of unnecessary settings. The impact of those settings in a distributed setup is higher than in a single deployment setup because of the more significant amount of settings and the possibility of causing cascading failures.

7.2.3 MS-ATD during migration to microservices

After deciding to migrate to microservices, a company must consider completely rewriting the software in the new architectural style or proceeding with an incremental migration. A company must make this decision by considering its context. However, a complete rewrite is often very costly. In that case, there are approaches that companies may use to proceed with an incremental migration, such as those proposed by Yoder and Merson [YM20] and Newman [New19]. The companies in our study executed an incremental migration.

MS-ATDs can occur during migration to microservices. For example, unplanned data sharing may arise when practitioners share databases among several microservices and the previous architecture, or microservice APIs may be malformed because practitioners maintain compatibility with the previous architecture. Knowing which MS-ATDs occur at different times during migration might help practitioners overcome their difficulties before they become too harmful, reduce costs, and speed up migration.

7.3 Research design

This section describes the process of our exploratory multiple-case study [Yin18], which is summarized in Figure 7.3, and a detailed protocol is available online[4].

Our study was conducted in three companies during the early stages of migration to microservices. We scheduled three 45-minute presentations, one for each participating company, with practitioners involved in microservice projects in the respective companies. The goals of the presentations were to raise practitioners' awareness of MS-ATDs, ensure they had the same understanding as the researchers regarding the debts, and collect initial data.

During the scheduled presentations, we introduced the practitioners to a list of MS-ATDs based on a previous study (Step 2 of Figure 7.3). The identification and selection (Step 1) of those MS-ATDs are described in Section 7.3.1. Next, we asked interviewees to answer a set of predefined questions (Step 3). The results from the interviews were analyzed, summarized, and presented to a subset of the original participants in another round of three 45-minute presentations, one for each company (Step 4). During the second interaction with the companies, we collected additional information through semi-structured interviews (Step 5). We recorded all interactions with the participants for posterior analysis.

[4] https://bit.ly/3qVwFJq

7.3.1 Identification and selection of the MS-ATDs used in this study

Practitioners from the participating companies granted us a limited amount of time enough to prioritize seven MS-ATDs. We selected those MS-ATDs from a list found in de Toledo et al. [dMS21], one of the most comprehensive studies covering MS-ATDs in large companies running mature microservice projects. The list from de Toledo et al. [dMS21] includes 12 debts, and we selected seven debts according to the following criteria:

(i) We selected the debts reported by at least three companies in the previous study, resulting in six out of the original 12 debts. We considered that frequent MS-ATDs found by companies running microservices for several years are likely to be found in other companies. The original study divided the debt "reusing third-party implementations" into two distinct sub-debts. For simplicity, we focused only on the sub-debt "misuse of internal shared libraries" reported by multiple companies.

(ii) We previously knew that the companies involved in this study extensively used asynchronous communication among services and were interested in prioritizing any related MS-ATDs. Thus, we included an additional debt in our list: "Insufficient metadata," the only of the remaining debts related to asynchronous communication among services, resulting in a total of seven debts, as detailed in Table 7.1.

Other debts we did not consider in this study might be relevant to the companies, including (but not limited to) the debts identified by de Toledo et al. [dMS21]. However, additional debts might be considered in future prioritization by practitioners and future research studies.

7.3.2 Studied companies

We studied three large software companies that had just started modernizing their (large) legacy monolithic systems using a microservice architecture. Figure 7.2 presents the relationship between the research questions, the ongoing architecture (with the migration in progress), and the final microservice architecture as presently visualized by the practitioners.

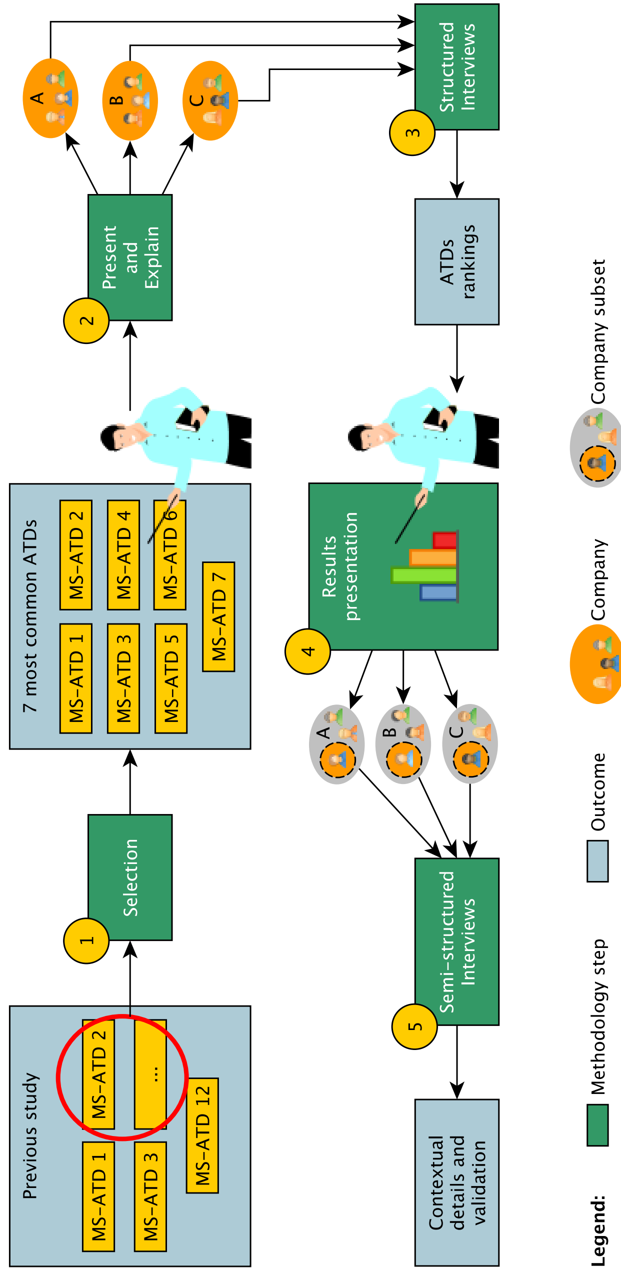


Figure 7.3: An overview of the research process.

Company H provides business software and IT-related development and consultancy, employs nearly a dozen thousand employees, and has hundreds of thousands of customers, mainly in northern Europe. The participants in our study are from the core teams that have been developing a dynamic ERP system for large companies, which is one of the company's flagship products. It is a large monolithic system in which customers can buy licenses and install them through Company H or its certified partners. Software teams have broken down their monolithic ERP product towards using microservice architecture and providing their product as a service on the cloud. One of their main goals is to avoid pitfalls and (remove) debts while migrating their product.

Company I provides IT and product engineering services, with approximately twice as many employees as Company H. Company I serves thousands of customers in more than 90 countries. The branch with which we conducted our study focuses on financial services, such as banking solutions, and is located in a Nordic country. They sell and maintain complex banking solutions for many banks and have continuously developed and modernized their products, focusing more recently on microservice architecture and modern software development. MS-ATD is a considerable concern that software teams want to control better.

Company J is one of the largest financial services groups in the Nordic region (mainly banking) with 9000+ full-time employees serving several millions of customers. The software teams we interacted with were at the core of their in-house software department, specialized in architecture and technology. This software department has a good tradition of handling TDs and has a reputation for allowing engineers to take software engineering courses.

7.3.3 Data collection

The data collection occurred as illustrated in Steps 3 and 5 in Figure 7.3. A total of 47 participants, distributed as shown in Table 7.2, participated in the first data collection (Step 3). The participants had different backgrounds and experiences with microservices. For each MS-ATD presented, we asked the participants the following three questions:

(i) Have you encountered this MS-ATD in your current project? (RQ1)

(ii) Do you foresee this MS-ATD in the future of the project? (RQ2)

(iii) Do you know how to avoid or mitigate this MS-ATD? (RQ3)

The first two questions were answered with *yes*, *not sure*, or *no*. The third question was answered with *yes*, *partially*, *no*, or *not applicable (n/a)*. *Not applicable* was used by the practitioners that considered the MS-ATD as irrelevant or out of context in their projects. For example, insufficient metadata in messages is not applicable to contexts in which messages are not used.

Table 7.2: Attendees for the first presentation.

Roles	**Number of attendees**			**Total**
	Company H	**Company I**	**Company J**	
Developer	1	14	7	**22**
Architect	6	1	11	**18**
Manager	2	2	1	**5**
Other	0	2	0	**2**
Total	**9**	**19**	**19**	**47**

We also asked the participants to report whether they understood the explanation of the MS-ATD. Only three participants from Company I said that they did not understand the explanation, which concerned MS-ATD 1, MS-ATD 2, and MS-ATD 5, as described in Table 7.1. In general, our explanation of the MS-ATDs was well accepted and understood by the participants.

The practitioners perceived some MS-ATDs as riskier than others and, as such, more important to them. At the end of the presentation, we asked the participants to rank the three most important MS-ATDs according to their point of view on the project (RQ4).

After data analysis (Section 7.3.4), we invited the most experienced interviewees from the previous session to a new group interview to discuss the results of the previous interviews. These experts constituted approximately 30% of the original participants. We presented the results, asked for clarifications of context, and discussed the prioritization of MS-ATDs in future development (RQ5). More specifically, we asked the following questions:

(i) What are your considerations regarding these results?

(ii) What are the causes of these results?

(iii) Do you agree with these results? Why?

These experienced interviewees had a good overview and understanding of their projects and thus provided additional helpful information for our analysis.

7.3.4 Data analysis

Descriptive statistics were used to compare the results for the different companies. We created rankings for each question, for example, from the most found to the last found MS-ATD in RQ1. To create the rankings, we transformed the categorical answers into numeric values, as follows:

- Every MS-ATD reported as found, foreseen, or difficult to solve was counted as 1 for each answer.
- MS-ATDs reported as not found, unforeseen, not difficult to solve, or not applicable to their context were counted as 0.
- Partial answers, i.e., "not sure" or "partially," were counted as 0.5. This value represents a 50% probability of a debt being found. It is reasonable to assume that some practitioners are more confident than others when answering these questions. Thus, 0.5 is an approximation to balance all answers. Future studies may find better measures of practitioner confidence in answering these questions.

The remainder of this section explains the analysis procedure in detail. The set of MS-ATDs in Table 7.1 is denoted by $D = \{d_1, d_2, ..., d_7\}$.

7.3.4.1 The ranking of the most encountered MS-ATDs

The question about the MS-ATDs encountered so far had three possible answers: *yes*, *not sure*, and *no*. We counted the number of votes for each answer: e_{yes}, e_{no}, and $e_{not\,sure}$. We weighted each answer: 1 for **yes**, 0 for *no* (because they represent MS-ATDs that were not encountered and, thus, are not relevant for our ranking), and 0.5 for *partially*.

Based on the e_{yes}, e_{no}, and $e_{not\,sure}$, and on the respective weights, we computed the weighted sum of votes v_{e_i} for each MS-ATD encountered $d_i \in D$, as defined in Equation 7.2.

$$v_{e_i} = (1 \times e_{yes}) + (0.5 \times e_{not\,sure}) + (0 \times e_{no}) \tag{7.2}$$

The set with all values v_{e_i} was used to compute the ranking $R_{encountered}$ of the MS-ATDs encountered by practitioners. The ranking is defined in

Equation 7.3 and represents the ordered set of debts $d \in D$ according to the respective values previously calculated: d_i is higher than d_j if v_i is greater than v_j, which means that d_i is encountered more than d_j. For cases in which the weighted sum of votes is numerically the same for more than one MS-ATD, we consider as most encountered the debt with less uncertainty, i.e., with less "not sure" answers.

$$R_{encountered} = order(D, \{v_{e_1}, v_{e_2}, ..., v_{e_7}\}) \tag{7.3}$$

7.3.4.2 The ranking of the most foreseen MS-ATDs

The question regarding the most foreseen MS-ATDs in the project's future had the same possible answers (*yes*, *not sure*, and *no*) as for the question about encountered MS-ATDs. We used the same reasoning as before to calculate Equations 7.4 and 7.5.

$$v_{f_i} = (1 \times f_{yes}) + (0.5 \times f_{not\,sure}) + (0 \times f_{no}) \tag{7.4}$$

$$R_{foreseen} = order(D, \{v_{f_1}, v_{f_2}, ..., v_{f_7}\}) \tag{7.5}$$

7.3.4.3 The ranking of the MS-ATDs that the participants do not know how to solve

The third question, which asked whether the participants knew how to solve each MS-ATD, had four possible answers: *no*, *partially*, *yes*, and *not applicable (n/a)*. We aimed to have a final ranking in which the first MS-ATD was the one in which the practitioners did not have the complete solution (since they could also answer *partially*) or did not know how to avoid or mitigate. We counted the number of votes for each answer, k_{no}, $k_{partially}$, k_{yes}, and $k_{n/a}$, and applied weights for them: 1 for *no*, 0 for *yes* and *n/a* (because they represent MS-ATDs for which the companies already have a solution or that do not apply to their cases), and 0.5 for *partially* (because they represent a solution that does not entirely repay the MS-ATD, but that at least reduces its risk).

Based on k_{no}, $k_{partially}$, k_{yes}, and $k_{n/a}$, and on the respective weights, we computed the values v_{k_i} for each $d_i \in D$, as defined in Equation 7.6, which were used to compute the ranking R_{known} of the MS-ATDs, as described in Equation 7.7.

$$v_{k_i} = (1 \times k_{no}) + (0.5 \times k_{partially}) + (0 \times k_{yes}) + (0 \times k_{n/a}) \tag{7.6}$$

$$R_{known} = order(D, \{v_{k_1}, v_{k_2}, ..., v_{k_7}\}) \tag{7.7}$$

7.3.4.4 The ranking of the importance of the MS-ATDs for the participants

The ranking of the importance of the MS-ATDs given by the participants is formed by the ordered set of MS-ATDs $d_i \in D$. d_i is higher than d_j if v_{i_i} is greater than v_{i_j}; v_{i_i} is the number of votes for MS-ATD i. In other words, we have an ordered list from the most important to the least important MS-ATD (see Equation 7.8).

$$R_{importance} = order(D, \{v_{i_1}, v_{i_2}, ..., v_{i_7}\}) \tag{7.8}$$

7.3.4.5 The final ranking of importance for the MS-ATDs

TD can be threatened as a software risk because of the uncertainty of interest payments [Guo+11; MB16b]. Therefore, a risk analysis is appropriate for prioritizing our MS-ATDs; the first MS-ATD to be paid is the one that poses a higher risk to the company and the project. As presented in Equation 7.1, the risk of an MS-ATD can be calculated as the probability of the debt to occur multiplied by the impact of that debt. Therefore, we computed the priority score p_i for each MS-ATD i in Table 7.1 based on this risk definition. Such a priority score is calculated using Equation 7.9: (i) the probability of having the debt is calculated as the product of the number of practitioners who believe the debt will happen in the future and the number of people who do not know how to solve the debt; (ii) the impact is represented by the importance given by the practitioners for the debt (we add 1 to the number of votes on the importance to prevent multiplication by zero if no practitioner has voted for the debt as important).

Our approach, represented by Equation 7.10, uses the priority scores to compute the priority ranking. However, other authors may consider different methods.

$$p_i = (v_{f_i} \times v_{k_i}) \times (1 + v_{i_i}) \tag{7.9}$$

$$R_{priority} = order(D, \{p_1, p_2, ..., p_7\}) \tag{7.10}$$

7.3.4.6 Visualizing the priority ranking

We used the score defined in Equation 7.9 for the prioritization ranking. However, for visualization and readability purposes, we applied the transformation [Cle85] defined in Equation 7.11 to the score.

The maximum value of the priority score depends on the number of participants and votes, and there is no upper limit. Therefore, we normalized the score between 0 and 1 by dividing it by the maximum score. We used a logarithmic transformation to reduce the differences between the values. Using two as the logarithm base is reasonable when the data range is less than two powers of 10 [Cle85]. We add 1 to the normalized value to ensure that we do not have negative numbers after our transformation (if the priority is zero, we have $log_2(1) = 0$ as the minimal value possible). Finally, we transformed the score into a scale between 1 and 10 by multiplying the results by 10.

$$s_i = \log_2 \left(\frac{p_i}{max(\{p_1, p_2, ..., p_7\})} + 1 \right) \times 10 \tag{7.11}$$

7.3.4.7 The qualitative analysis

We used the recorded interviews to identify contextual information that could explain the practitioners' decisions. Owing to the limited interview time, the practitioners focused on the MS-ATDs they considered the most important while discussing each RQ. In a few cases, the practitioners also discussed debts that were less important to them.

For each MS-ATD in our ranking, we looked for a mention of the debt during the interviews. We found explanations and quotations that helped us interpret the results. For example, when asked about the reasons for having shared databases, one interviewee from Company H said, "*We decided to share the database at the beginning of the migration to speed up the process.*" Thus, the debt exists at the top of their rankings because they explicitly decided to have it, and they are paying the respective costs.

7.4 Results and discussion

Table 7.3: The raw answers for the most encountered MS-ATDs, the most foreseen MS-ATDs, the MS-ATDs practitioners do not know how to avoid, and the importance for each MS-ATD according to practitioners.

ATD	Comp.	MS-ATDs encountered			MS-ATDs foreseen			MS-ATDs practitioners do not know how to avoid				Votes for importance
		Found	Not sure	Not found	Foreseen	Not sure	Not foreseen	No	Partially	Yes	N/A	
Insufficient metadata	H	2	2	5	4	2	3	4	1	1	3	1
	I	1	6	9	3	7	5	2	5	0	9	4
	J	6	5	7	9	7	1	1	6	3	5	5
Microservice coupling	H	4	1	4	7	1	1	4	3	1	1	9
	I	8	4	4	12	4	0	6	5	1	5	8
	J	10	1	3	10	1	2	5	6	2	1	13
Inadequate use of APIs	H	6	0	2	5	1	3	2	4	2	1	6
	I	6	3	5	7	4	2	0	6	5	3	6
	J	9	1	3	11	1	1	0	12	1	0	4
Excessive diversity	H	2	0	6	3	0	5	3	3	1	1	0
	I	1	1	12	6	0	7	0	8	2	2	1

Continued on next page

Table 7.3 – continued from previous page

ATD	Comp.	MS-ATDs encountered			MS-ATDs foreseen			MS-ATDs practitioners do not know how to avoid				Votes for importance
		Found	Not sure	Not found	Foreseen	Not sure	Not fore-seen	No	Partially	Yes	N/A	
	J	7	0	6	8	2	3	4	6	3	1	6
Unplanned data sharing/sync	H	6	1	2	5	3	1	1	7	0	1	7
	I	7	4	3	10	3	0	4	6	1	2	7
	J	4	4	6	6	4	3	3	7	2	2	6
Misusing shared libraries	H	6	1	1	6	2	1	4	3	1	1	2
	I	10	0	3	11	1	1	5	6	2	0	10
	J	9	0	3	11	3	0	6	6	1	1	3
Unnecessary settings	H	6	0	1	6	1	1	1	4	3	1	2
	I	9	2	4	8	1	5	5	4	3	1	3
	J	9	2	2	10	1	2	2	9	1	1	2

Table 7.3 shows our raw data with the number of participants who voted for each answer in the first three RQs (i.e., for the MS-ATDs encountered so far, foreseen in the future, and that the practitioners do not know how to avoid), and the number of votes for importance, for each company. The raw data may be used by the reader to replicate our study or to use them in different ways as those proposed in this work.

Figures 7.4, 7.6, and 7.8 show the percentage of practitioners who voted for each answer in each MS-ATD regarding the respective RQs.

Tables 7.4, 7.5, 7.6, and 7.7 present the rankings calculated from the data in Table 7.3 using the formulas described in Section 7.3.4. These tables contain colors to facilitate the identification of the MS-ATD for the distinct companies, i.e., the same MS-ATD has the same color for all the rankings, facilitating the comparison among them. We focused on the top-3 debts of the rankings because they were the debts the companies mainly discussed in our follow-up interviews and were thus most relevant to the companies.

7.4.1 Which MS-ATDs do companies encounter during early migration to microservices? (RQ1)

Figure 7.4 shows the percentage of practitioners who voted for each answer. Table 7.4 shows a ranking calculated as defined by Equation 7.3, ordered from the most found MS-ATD to the less found MS-ATD. Figure 7.5 shows the values used to calculate the rankings and is used to support our discussion.

Figure 7.4: Percentage of practitioners who voted for each answer regarding the debts found on each company. Not all practitioners voted for all questions for each company.

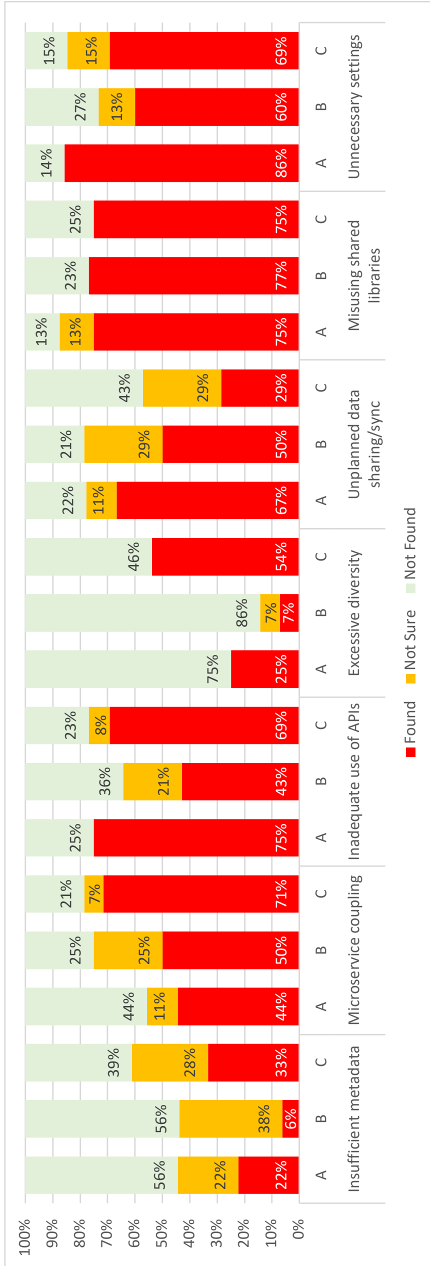


Table 7.4: Ranking of the most encountered MS-ATDs calculated through Equation 7.3. Each MS-ATD is associated with the same color to facilitate identifying it across the rankings from the distinct companies.

	Company H	**Company I**	**Company J**
1	Misusing shared libraries	Unnecessary settings	Microservice coupling
2	Unplanned data sharing/synchronization	Misusing shared libraries	Unnecessary settings
3	Unnecessary settings	Microservice coupling	Inadequate use of APIs
4	Inadequate use of APIs	Unplanned data sharing/synchronization	Misusing shared libraries
5	Microservice coupling	Inadequate use of APIs	Insufficient metadata
6	Insufficient metadata	Insufficient metadata	Excessive diversity
7	Excessive diversity	Excessive diversity	Unplanned data sharing/synchronization

7.4.1.1 Misusing shared libraries

"Misusing shared libraries" is among the most encountered MS-ATDs for Companies H and I, and fourth for company C. This debt has the least uncertainty among the practitioners (see Figure 7.4). Only 13% of the participants from Company H, the company with the fewest participants, answered "Not sure." Among the participants from all three companies, 75–77% answered "Found."

A practitioner from Company H said, "*We have created a lot of smaller projects ourselves that we use in our solutions as packages [as shared libraries] in the monolith.*" Another practitioner complemented with an example to justify why they use shared libraries: "*During the migration, we are still sharing the database with the monolith. We have some encrypted data that must be accessed by the same decryption algorithms available as a library. So, we share the library among the microservices.*" Thus, in this case, the need for such shared libraries is caused by the dependency on the monolith.

Company I has internal restrictions on how many times a service should be called, as explained by one interviewee: "*If you try to run this external validation several times an hour, suddenly you get a call from those running that service saying that you cannot do this.*" They work around these issues using shared libraries instead of relying on external services, which makes them

use more shared libraries and might indicate an infrastructure that is not yet prepared for distributed systems.

For Company J, the weighted sum of votes for this debt is similar to that for top-3 debts (see Figure 7.5c). Figure 7.4 shows that as many as 75% of the practitioners encountered it in their projects. As a financial services company, they have several legacy systems. These systems potentially share code with microservices through libraries, increasing the likelihood of having this debt.

7.4.1.2 Unnecessary settings

"Unnecessary settings" is the only MS-ATD in the top-3 for all companies. The number of practitioners unsure about the presence of this debt is relatively small compared with other debts, and 60–86% of the practitioners reported it as found (see Figure 7.4). Thus, this debt is relatively common across companies and easy to recognize.

One interviewee from Company H said, "*The situation regarding configuration settings today is chaotic*" while trying to explain that there was no approach to control the addition of unnecessary settings to the services, and, thus, the debt was common to be found. Companies I and J reported that this debt is so common that it must be accepted when using microservices. One interviewee from Company I, for example, said, "*This is an expected consequence of having many small services, each with its own settings.*" Only practitioners from Company H reported that they would like to mitigate this debt in the future.

7.4.1.3 Microservice coupling

"Microservice coupling" is among the three most encountered MS-ATDs for Companies I and J. A considerable number of practitioners are unsure of the existence of this debt (see Figure 7.4). Possible reasons are that the practitioners did not perceive the costs of this debt in the current stage of their projects, that they did not have a tool to make those dependencies visible, or that they might not be sure about the design of their services.

Company H explained that the new microservices are one way to reduce coupling from the previous system. However, they do not seem concerned with coupling among the microservices themselves in the current stage because they are in an early stage of migration, with only a small part of a monolithic architecture migrated to microservices. Answering this question requires observing Company H again in the future to understand the evolution of microservice coupling.

Companies I and J seem to have a higher number of microservices and teams involved with the services than Company H. Therefore, it was easier for them to visualize microservice coupling.

Our impression is that the practitioners only start to think about microservice coupling at a later stage when more couplings have been created.

7.4.1.4 Unplanned data sharing/synchronization

Microservices recommend decentralized data management; however, some practitioners do not entirely agree with this recommendation. When centralizing data management, some additional services may have to share data with other services in a way that was not previously planned. On the other hand, practitioners might not properly plan the database synchronization properly when decentralizing data management. These situations lead to "unplanned data sharing/synchronization," which only appears on the top-3 list for Company H. This debt represents the difficulties of splitting a large database that is running for a long time or synchronizing distinct databases. Many practitioners were uncertain about the existence of this debt (see Figure 7.4), indicating that identifying it is more challenging than identifying others.

Several practitioners have stated that they wondered whether sharing databases across microservices is an incorrect decision. One reason for that is the trade-off represented by this debt: splitting the database increases issues with synchronization among services.

Company H decided to split the database at a later stage of its migration to microservices. Thus, they deliberately acquired this debt to accelerate the initial steps of their migration and plan to repay it later.

For Company I, despite this debt being ranked fourth, at least 50% of the practitioners reported it and 29% were unsure (see Figure 7.4). This debt is almost as important as the other debts.

It is not clear from our data why this debt is the last on the list for Company J. It might be that the practitioners from Company J who participated in our interviews were primarily involved in well-designed services that had their own databases and did not have to synchronize with other services. This can only be confirmed by a more in-depth study.

7.4.1.5 Inadequate use of APIs

"Inadequate use of APIs" only appears on the top-3 list for Company J.

This debt was ranked fourth in Company H, but with as many as 75% of developers reporting it, it was close to the top three debts. In Company I, this debt has been reported less.

Company J could not explain why this debt was among the most found when asked during the follow-up interviews. However, we might have identified a disagreement between technical leaders and other practitioners. The company will discuss it internally.

7.4.1.6 Insufficient metadata

"Insufficient metadata" is the debt with most uncertainty among all the debts. In practice, many practitioners could not connect the debt with their examples. It is unclear whether additional metadata can resolve the current issues. To be repaid, this debt may require a global overview of the architecture. However, many practitioners focus on their own services, and only a few have such a global overview of the architecture.

7.4.1.7 Excessive diversity

"Excessive diversity" is the debt in which the majority of the participants had a strong opinion about its existence: only 7% of the participants from Company I reported not being sure about it, while all the other practitioners answered "found" or "not found" (see Figure 7.4). However, in our interviews, we noticed a lack of consensus on the extent to which this is a debt. Some practitioners believe that such technology diversity is acceptable, while others believe it incurs high costs.

Only 25% of the practitioners from Company H reported this debt as found. Company H has a well-defined set of technologies and platforms for the development of its microservices. Moreover, they have only migrated a small part of their monolith to microservices, and used the same .NET Core technology stack. Thus, Company H expected to have fewer complaints about this debt than the other companies.

Company I was satisfied with its current policy on the diversity of technologies while using microservices. On the other hand, Company J is divided on their opinions; about half of the practitioners believe there is a problem with their policy on diversity, but in spite of that, they did not want to limit the technologies used by other teams.

Main findings

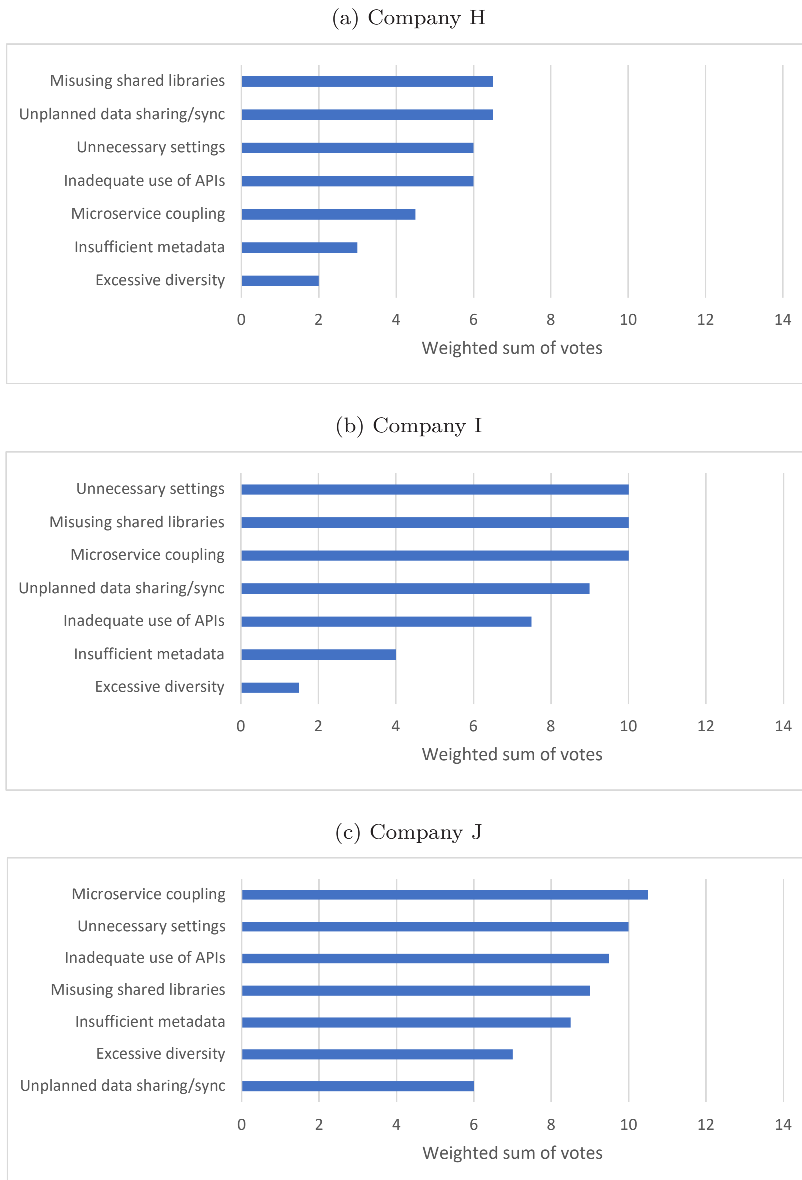
F1. The use of shared libraries starts at the early migration stages and is usually related to the convenience of reusing code from the original architecture. However, companies may misuse the shared libraries.

F2. Unnecessary settings are common during the early stages of migration. Some practitioners try to find ways to mitigate this debt (sooner or later), while others find it more convenient to maintain the debt and its extra cost.

F3. The costs of microservice coupling are not recognizable in the initial stages of migration, and the debt is down-prioritized. Practitioners may not prioritize this debt and may tend to postpone repayment.

F4. Compared to the other debts, insufficient metadata and unplanned data sharing/synchronization are the debts in which more practitioners are unsure about their existence, which might indicate that identifying these debts is more challenging than identifying others.

Figure 7.5: Values for the calculation of the ranking of MS-ATDs found for each company.



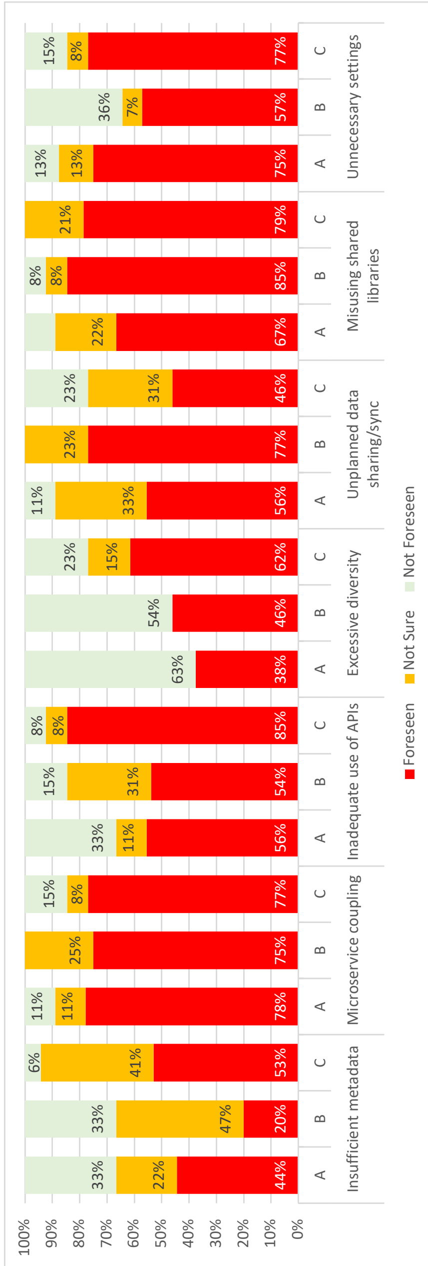


Figure 7.6: Percentage of practitioners who voted for each answer regarding the debts foreseen on each company. Not all practitioners voted for all questions for each company.

7.4.2 Which MS-ATDs do companies foresee in the future of the migration? (RQ2)

Table 7.5: Ranking of MS-ATDs foreseen in each company calculated through Equation 7.5. Each MS-ATD is associated with the same color to facilitate identifying it across the rankings from the distinct companies.

	Company H	Company I	Company J
1	Microservice coupling	Microservice coupling	Misusing shared libraries
2	Misusing shared libraries	Misusing shared libraries	Insufficient metadata
3	Unnecessary settings	Unplanned data sharing/synchronization	Inadequate use of APIs
4	Unplanned data sharing/synchronization	Inadequate use of APIs	Unnecessary settings
5	Inadequate use of APIs	Unnecessary settings	Microservice coupling
6	Insufficient metadata	Insufficient metadata	Excessive diversity
7	Excessive diversity	Excessive diversity	Unplanned data sharing/synchronization

Figure 7.6 presents the percentage of practitioners who voted for each answer regarding MS-ATDs foreseen in the next steps of migration. More practitioners are not sure about the debts in the future than when compared to the debts in the present, as detailed in Section 7.4.1. Such an increase in the number of “not sure” answers is expected because practitioners are reasoning about a possible future.

The remainder of this section discusses the most foreseen MS-ATDs extracted from the results for each company according to our data and ranking calculation defined in Equation 7.5. The rankings are presented in Table 7.5, ordered from the most foreseen to the least foreseen debt. The values used to calculate the rankings are shown in Figure 7.7.

7.4.2.1 Microservice coupling

“Microservice coupling” is the top item in the ranking for Companies H and I, and the fifth debt in the ranking for Company J. Compared to the currently found MS-ATDs in Section 7.4.1, 75–77% of the practitioners from all three companies estimate that this debt will increase.

In the follow-up interview with Company H, senior practitioners did not expect such a result because they planned to reduce microservice coupling in the future. This result highlights that other participants may not share the same point of view. Thus, they may internally discuss the reasons for such concern, as explained by one of the practitioners: "*These numbers inform us that we have an important job in informing everyone about what we want to do*".

One practitioner from Company I explained that this is expected: "*We are going to see more on coupling if nothing is done today to change [the process]*".

For Company J, "Microservice coupling" is among the less foreseen debts (see Table 7.4). However, all debts had substantial votes by practitioners from Company J: 77% of the practitioners reported this debt as foreseen, and only 8% were unsure (see Figure 7.6). However, our ranking provides a starting point for prioritizing the debts.

7.4.2.2 Misusing shared libraries

"Misusing shared libraries" is in the top-3 for all companies in the ranking. We present the reasons for this for each company below.

One interviewee from Company H said, "We have a lot of dependencies on other parts of the system. Instead of implementing something new, we use these dependencies to focus on the main goal. Most of these dependencies will be addressed in the end, but you cannot address them all during the migration process." Therefore, they will still use libraries they believe are necessary and will postpone their removal.

Company I reported no plan to reduce the usage of shared libraries today; they foresee this debt coming again in the future.

Company J started a discussion on whether the shared libraries were an issue. One practitioner said, "*We first need to discuss whether shared libraries are necessarily bad, are they?*" They do not seem to have plans to change how they use these libraries. Thus, they might prevent the costs of misusing shared libraries by closely following library usage. Our discussion on this topic may have increased practitioners' awareness of the debt.

7.4.2.3 Unnecessary settings

"Unnecessary settings" is in the top-3 for Company H only.

Company H visualizes the need to reduce unnecessary settings in the future but believes that the problem will first increase before they have a better

approach to handle it. They wanted a solution to the problem, but that was not a priority.

Companies I and J expect this debt, but they accept it and do not have plans to mitigate it. They are not concerned with the costs of this debt. It is possible that it is better to pay interest in this type of debt than to repay it.

7.4.2.4 Unplanned data sharing/synchronization

"Unplanned data sharing/synchronization" is in the top-3 for Company I only. This debt is still among those with more participants who are unsure about the debt. The reasons may be the same as those explained in Section 7.4.1: practitioners are still questioning whether sharing databases is always a bad practice because they foresee cases in which this seems to be a good solution for the problem.

Company H decided to postpone the migration of the database to the microservices. Thus, they currently have the costs of using a centralized database and do not foresee the complete migration of the database. However, Figure 7.7a shows that there is no difference in the value used to calculate the rankings between this debt and the "Unnecessary settings," one of the top-3 debts in the ranking for Company H.

Company I saw this debt as a challenge that will increase if nothing else is done to reduce it. Thus, some practitioners have already observed that the company needs to work on a solution for the debt.

It is unclear from our data why this debt goes to the bottom of the list for Company J, a result similar to that described in Section 7.4.1.

7.4.2.5 Inadequate use of APIs

"Inadequate use of APIs" is in the top-3 for Company J only.

Company J explained the same as reported in Section 7.4.1: they could not explain why this debt was among the most found when asked during the follow-up interviews, so they went back to investigate this further with developers. Companies H and I did not provide any further comments.

7.4.2.6 Excessive diversity

"Excessive diversity" is the debt in which the majority of the participants had a strong opinion about its existence: only 15% of the participants from Company J reported not being sure about it, while all the other practitioners answered "found" or "not found" (see Figure 7.6). Compared with the described in

Section 7.4.1, the participants from all companies believe that this debt will increase in the future.

Company H has the fewest participants among the companies in this study. Such a result is expected because they seem to have reasonable control of technology diversity in their current stage of development.

Company I reported that they do not have proper control of such diversity, which might lead to an increase in this debt in the future, indicating that they should be aware of the issue and control it in advance.

Company J already saw the costs of this debt, and they believed that the problem would increase because there was no plan to limit such diversity.

7.4.2.7 Insufficient metadata

"Insufficient metadata" keeps being the debt with the higher number of practitioners not sure about it. The possible reasons are explained in Section 7.4.1: practitioners have difficulties seeing this debt in their contexts and are not sure whether additional metadata is the right solution for the cases they observed. The number of practitioners who answered "not sure" increased more than for other debts. Again, this debt seems difficult to identify and estimate in the future and might concern architects more than developers, who are only involved with the development of microservices.

Companies H and J foresee cases in which they need additional metadata and consequently increase the probability of having this debt. On the other hand, Company I is mostly uncertain about this debt.

Main findings

F1. The practitioners seem to foresee an increase in microservice coupling. Microservice coupling might increase unnoticed in the early stages of migration, and suddenly become visible with many microservices.

F2. The practitioners foresee the use of shared libraries because they plan to use libraries to accelerate migration. Therefore, they may have to deal with the misuse of such libraries later.

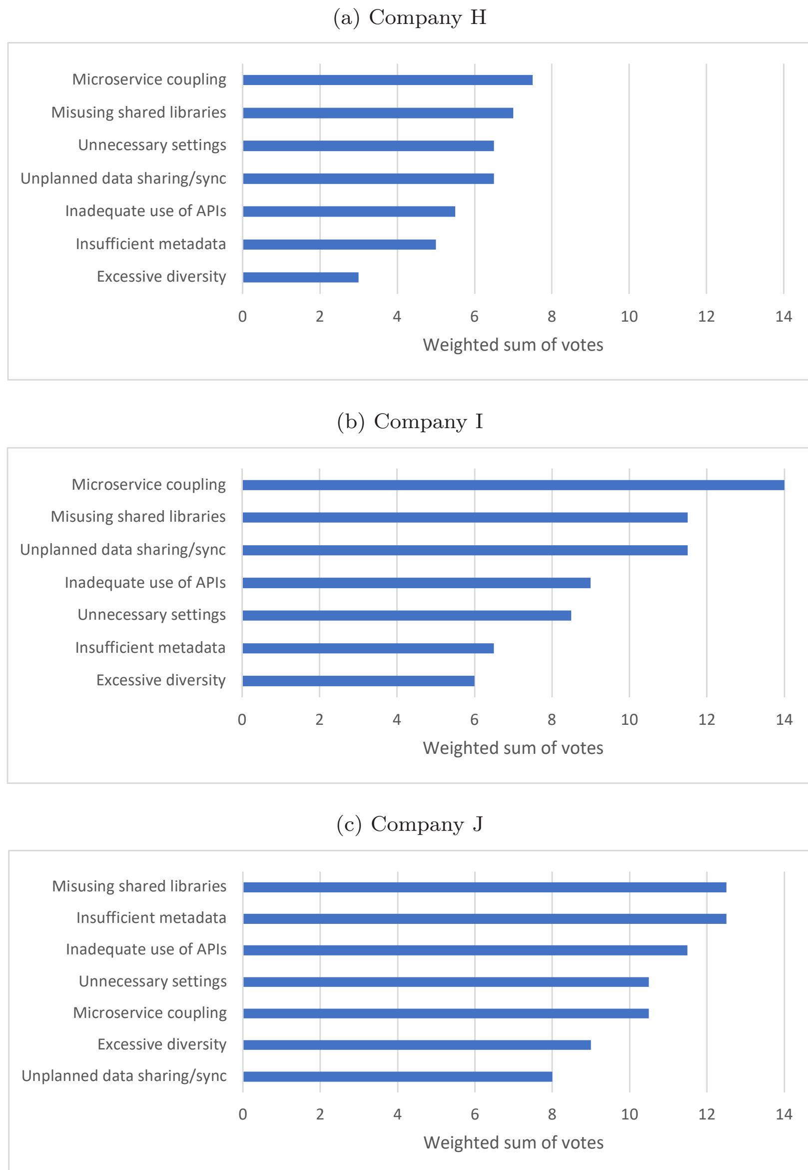
F3. The practitioners accepted the extra costs of "Unnecessary settings." Therefore, they foresee the presence of such debt.

F4. The practitioners are most uncertain about to what extent "unplanned data sharing/synchronization" is a debt. They foresee the debt because they are unsure how to repay it.

Table 7.6: Ranking of MS-ATDs that companies do not know how to avoid calculated through Equation 7.7. Each MS-ATD is associated with the same color to facilitate identifying it across the rankings from the distinct companies.

	Company H	**Company I**	**Company J**
1	Misusing shared libraries	Microservice coupling	Misusing shared libraries
2	Microservice coupling	Misusing shared libraries	Microservice coupling
3	Unplanned data sharing/synchronization	Unnecessary settings	Excessive diversity
4	Excessive diversity	Unplanned data sharing/synchronization	Unnecessary settings
5	Insufficient metadata	Insufficient metadata	Unplanned data sharing/synchronization
6	Inadequate use of APIs	Excessive diversity	Inadequate use of APIs
7	Unnecessary settings	Inadequate use of APIs	Insufficient metadata

Figure 7.7: Values for the calculation of the ranking of MS-ATDs foreseen for each company.



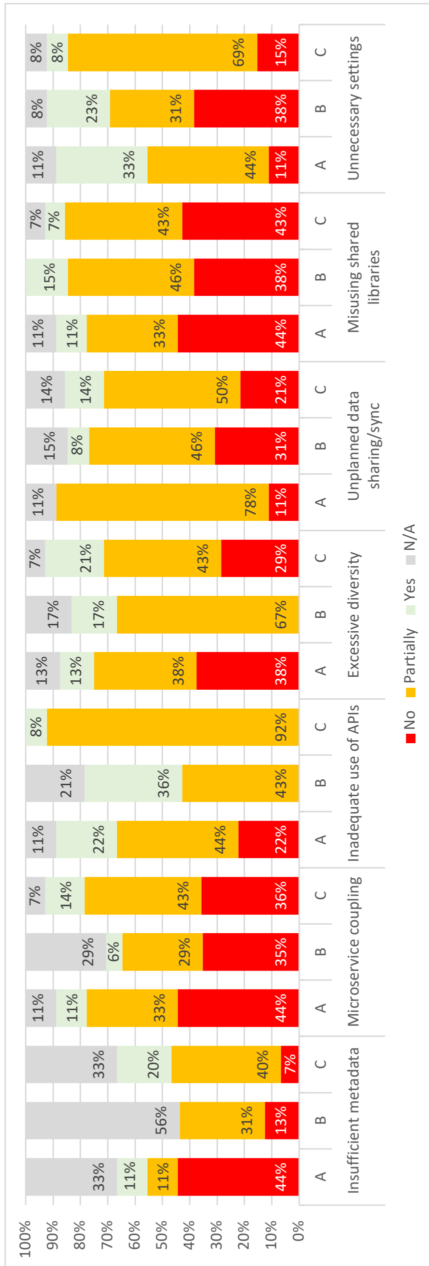


Figure 7.8: Percentage of practitioners who voted for each answer regarding the debts practitioners know how to avoid on each company. Not all practitioners voted for all questions for each company.

7.4.3 Which MS-ATDs do the practitioners find difficult to solve? (RQ3)

Figure 7.8 presents the percentage of practitioners who voted for each answer regarding MS-ATDs they did not know how to avoid. Most practitioners are not confident about solving the problem, which means that they do not know whether what they are using is a solution to prevent such debts in most cases.

The remainder of this section discusses the top-3 most found MS-ATDs extracted from the results for each company, according to our data and ranking calculation defined in Equation 7.7. The rankings are listed in Table 7.6. The MS-ATD on the top of the list is the debt that most practitioners do not know how to prevent. The values used to calculate the rankings are shown in Figure 7.9.

7.4.3.1 Misusing shared libraries

"Misusing shared libraries" is in the top-3 for all companies in the ranking. None of the companies seemed to have considered good alternatives for shared libraries in the early stages of migration.

Company H reported that it is difficult to avoid these libraries because of the dependencies they have on the original architecture. These libraries ensure that all services behave in the same way when dealing with a centralized database and old pieces of software. Thus, they believe that this is difficult to deal with now, and their removal will be postponed.

Company I reported that the debt would increase because there is no plan to reduce the usage of shared libraries today. One participant from this company said, "*There is really no gold solution; everything is a trade-off.*" It is difficult for them to avoid using shared libraries.

Company J was not convinced about reducing the usage of shared libraries. When discussing the implementation of functionality as a service, one practitioner said, "*This will add latency everywhere.*" For him, this is unacceptable, and a shared library solves the problem without additional latency. However, companies with more mature microservice architectures have reported increased maintenance costs owing to the growing use of shared libraries [dMS21]. Company J may have a different context from the companies studied by de Toledo et al. [dMS21], and how they use shared libraries does not lead to problems. However, it is also possible that they are just down-prioritizing the debt, and they might incur high costs later. A more in-depth study would be useful to help companies such as Company J achieve a good trade-off between performance and maintainability while using shared libraries.

7.4.3.2 Microservice coupling

“Microservice coupling” is in the top-3 for all companies in the ranking. This result highlights that practitioners do not know how to properly prevent this debt from occurring. Thus, it is important to invest in training and techniques to prevent or reduce coupling in the early stages and, therefore, to reduce the growth of the debt interest.

Company H reported that coupling is difficult for practitioners to solve, and that more code reviews should help to reduce it.

Companies I and J reported that they did not have a good approach to solving this debt.

7.4.3.3 Unplanned data sharing/synchronization

“Unplanned data sharing/synchronization” is among the top-3 for Company H only.

Company H had a complex database that was difficult to split. Practitioners postponed splitting the database because it is challenging and costly to do so immediately.

Company I has this debt as fourth in its ranking. However, as shown in Figure 7.9b, there is no difference in its value with the third element in the ranking despite more votes for partial solutions (see Figure 7.8). Nevertheless, the differences are minimal (a small difference in uncertainty compared to the third debt in the list), and this debt is almost as important as the third one in the list for Company I.

Several practitioners from Company J also voted for this debt as difficult to solve despite the existence of other debts with more votes (see Figure 7.8).

7.4.3.4 Unnecessary settings

“Unnecessary settings” is in the top-3 for Company I only.

Company H did not discuss how it planned to reduce unnecessary settings in their services.

Company I had difficulties finding solutions for the increasing number of unnecessary settings, and considered the debt difficult to handle. One of the practitioners said, “*Approaches to control the growth of these settings, such as peer review, might introduce time-demanding formal procedures, such as waiting for external teams’ reviews and additional coordination.*” Thus, Company I does not see good approaches to dealing with this issue, placing this debt as one of the top-3 in the list.

Company J has this debt as the fourth in the ranking, but it is as difficult as the third in the ranking (see Figure 7.9c).

7.4.3.5 Excessive diversity

"Excessive diversity" is in the top-3 for Company J only.

For Company H, this debt is fourth in the ranking, but it is as difficult as the third in the same ranking (see Figure 7.9a). They only migrated a small part of the monolithic architecture to microservices and used the same .NET Core technology stack, reducing the diversity of technologies. However, they also reported having outsourced teams, and the new microservice architecture would allow other teams to use other technology stacks. The debt may worsen in the future.

Company I seems to have this diversity controlled, but it is not clear how they performed this control. The diversity of technology stacks for Company I stems from their various projects for different customers, but not from the same project with diverse microservices using different stacks.

Company J, on the other hand, reported that it is difficult to limit the technologies and languages without receiving complaints from teams already using a distinct set of technologies throughout the company. They accepted the debt now, but might need to develop company-level guidelines to keep it under control in the future. This seems to be a social rather than technical problem that is difficult to solve.

7.4.3.6 Insufficient metadata

Answers for "insufficient metadata" carries a lot of uncertainty. Since practitioners have difficulties visualizing the solution in practice, they are not sure whether solving it is difficult.

Company H was the company in which most practitioners who voted for the debt were sure about it being difficult to solve (see Figure 7.8). Since Company H is also the company with the fewest microservices in the migration, it has had a hard time seeing cases where this debt applies.

Companies I and J seem to have more microservices that use messages in which the debt could apply. Thus, they seem to have a better overview of how the metadata could be implemented in their cases than Company H. However, they were uncertain about whether what they thought as a solution would help them solve the problem. Therefore, there is a large amount of uncertainty.

7.4.3.7 Inadequate use of APIs

The “inadequate use of APIs” is a debt with a high degree of uncertainty, but it is also one of the debts with more participants answering they know how to solve. Practitioners from all companies reported that this was a matter of education and training for creating good APIs. During our follow-up interviews, senior practitioners and architects reported knowing the good practices for developing APIs. They informed us that they were already investing in spreading knowledge on the subject throughout the companies. One possible interpretation of these results is that the companies still have practitioners learning about topics such as how to control API versioning and deprecation. Such cases generated some uncertainty in the teams, but they knew how to repay the debt.

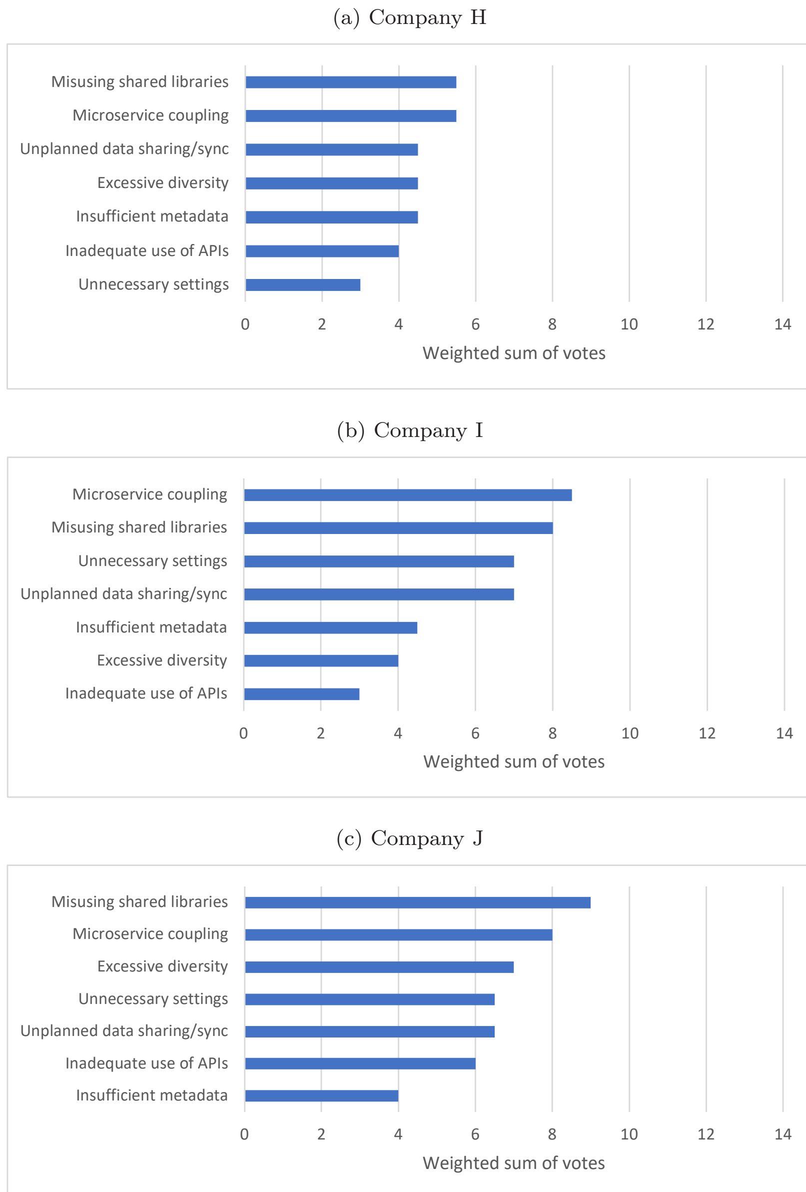
Main findings

F1. Practitioners consider “microservice coupling” difficult to solve. Therefore, it is important to invest in training and techniques to prevent or reduce this debt early on in a project.

F2. Companies share libraries to reuse code from the original architecture. However, they do not consider the costs of misusing them in the early stages of migration, which might incur high costs later.

F3. “Excessive diversity” is a debt that is difficult to mitigate after practitioners start using distinct technologies. Having some agreement regarding the technologies to use in the early stages of migration would facilitate dealing with this debt later.

Figure 7.9: Values for the calculation of the ranking of MS-ATDs the practitioners know how to avoid for each company.



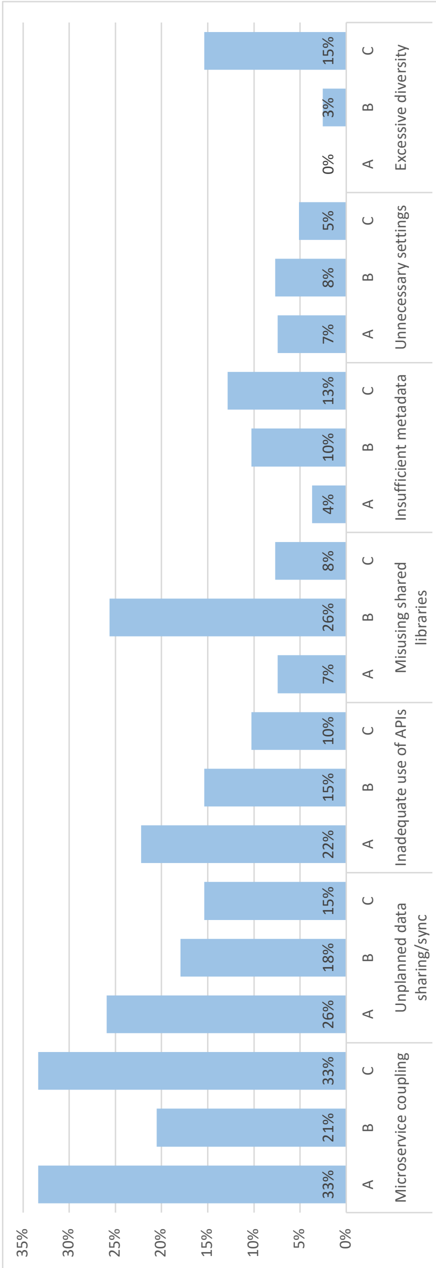


Figure 7.10: Importance of the MS-ATDs as perceived by the practitioners calculated by the Equation 7.8.

7.4.4 How important do practitioners perceive MS-ATDs? (RQ4)

Table 7.7: Ranking of the most important MS-ATDs according to the practitioners calculated through Equation 7.8. Each MS-ATD is associated with the same color to facilitate identifying it across the rankings from the distinct companies.

	Company H	**Company I**	**Company J**
1	Microservice coupling	Misusing shared libraries	Microservice coupling
2	Unplanned data sharing/synchronization	Microservice coupling	Excessive diversity
3	Inadequate use of APIs	Unplanned data sharing/synchronization	Unplanned data sharing/synchronization
4	Unnecessary settings	Inadequate use of APIs	Insufficient metadata
5	Misusing shared libraries	Insufficient metadata	Inadequate use of APIs
6	Insufficient metadata	Unnecessary settings	Misusing shared libraries
7	Excessive diversity	Excessive diversity	Unnecessary settings

Figure 7.10 and Table 7.7 show the importance of the MS-ATDs as perceived by the practitioners and calculated using Equation 7.8 for each company. Figure 7.10 groups the answers by MS-ATD and shows the percentage of votes per company. Table 7.7 presents the MS-ATDs ordered by the company.

There are clear differences among the companies, indicating that the importance of MS-ATDs is context dependent. For example, "excessive diversity," the last in the rankings for Companies H and I, is in the top-3 for Company J. Two debts are consistently among the most important for all three companies: "microservice coupling" and "unplanned data sharing/synchronization." The remainder of this section discusses each MS-ATD and possible reasons for these results.

7.4.4.1 The debts that were important for all three companies

All three companies highly reported two MS-ATDs: "microservice coupling" and "unplanned data sharing/synchronization" (see Table 7.7). These debts are explicitly mentioned in the definition of microservices: low-coupled services with their own databases. One possible interpretation of these results is that practitioners may clearly identify debts against the definition of microservices.

As reported in Section 7.4.1, Company H reported that "microservice coupling" does not exist at present, and they do not see its costs. However, with the increasing complexity of the software, they foresee it coming (see Section 7.4.2) and recognize it as difficult to solve (see Section 7.4.3). On the other hand, companies B and J have reported "microservice coupling" as found in the present, expected in the future, and difficult to solve. Therefore, this debt was expected to be considered important by practitioners.

Regarding "unplanned data sharing/synchronization," we found that all three companies reported it as important, but they have very distinct answers to the previous RQs. Company H postponed working on the database; thus, it is important to address this issue soon. Company I reported this debt for all previous rankings, but they considered other debts more important than this one. One possible interpretation of this result is that some of the practitioners in Company I believe that database sharing or synchronization approaches are adequate for their needs. In contrast, others are more skeptical of those approaches: they seem to discuss data sharing/synchronization approaches, but not all agree on doing so. Our method identified a disagreement among practitioners from the company. With this information, they could now discuss it internally. Company J, on the other hand, has this debt as the last for the first two rankings (found and foreseen) and as one of the last debts in the ranking of difficulty to solve. However, they consider this debt one of the three most important for them. One possible interpretation of the results from Company J is that they believe that this debt is important, but they have it under control. They shared databases and performed synchronizations among different services, but deliberately did so in situations they presumed they were right. Thus, they seemed to have planned these cases carefully.

7.4.4.2 The debts the practitioners mostly disagree

There are three debts for which the companies have distinct points of view: "misusing shared libraries," "inadequate use of APIs," and "excessive diversity." The causes of these differences may be the context of the projects or other unknown factors.

Company H has already reported the use of shared libraries to accelerate the development process (see Section 7.4.2). Company J reported that it was common to use such libraries. Companies H and J reported that it was challenging to remove misused shared libraries later. However, they do not consider this debt very important because they believe most of the shared libraries' use was correct. If this is the case, an in-depth study would help identify good practices while using shared libraries in microservices. If this

is not the case, they may suffer from the costs of this debt in the future, as reported by companies with more mature microservice architectures [dMS21]. On the other hand, Company I identifies "misusing shared libraries" as one of the most important debts to be mitigated.

"Inadequate use of APIs" varies among companies. For Company H, this is one of the three most important debts; for Company I, it has medium importance; and for Company J, this debt is one of the three less important debts. In general, practitioners reported that it is important to have good APIs, but they did not make further comments regarding that specific debt, even though we asked.

Regarding the "Excessive diversity," there are considerable differences among companies. No practitioner from Company H voted for this debt. Company H seems to have good control of diversity, which might explain the number of votes for that debt (see Figure 7.10). Company I reported that they have some diversity but that it is not important. They did not have any costly issues related to this debt. Finally, Company J reported a very distinct result compared to the other companies: this debt was one of the most important to them. Company J appear to be the company with the most diverse set of technologies. Some practitioners have reported concerns regarding their current policy of not limiting the technologies used by their microservices.

7.4.4.3 The less important debts

Other two debts were considered less important by practitioners: "unnecessary settings" and "insufficient metadata." However, there are some differences among the companies. We discuss these differences below.

None of the three companies is concerned with the "unnecessary settings" because they report it as a problem that cannot be avoided in microservices. Company H was the only company to comment that might try to mitigate it in the future, without explaining how, since it is a future concern.

"Insufficient metadata," on the other hand, seems to be more context-dependent and had distinct votes from the different companies. The low number of votes might be related to the uncertainty in the previous responses (see Sections 7.4.1 and 7.4.2).

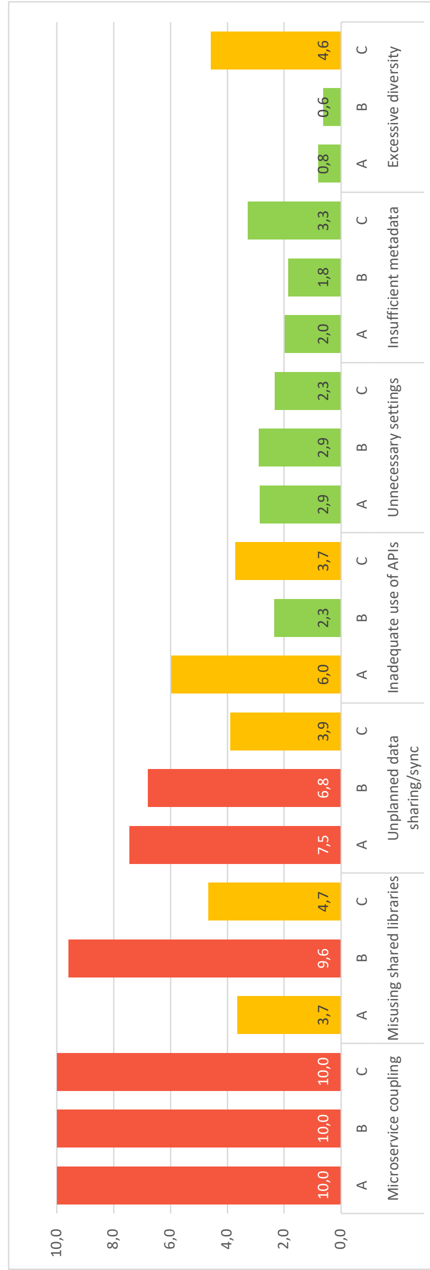


Figure 7.11: Priority ranking calculated through Equations 7.9 and 7.10, normalized between 1 and 10. Red bars indicate debts with high priority. Orange bars indicate debts with medium priority. Green bars indicate debts with low priority.

7.4.5 How can companies prioritize which MS-ATDs to avoid or repay? (RQ5)

Table 7.8: Priority rankings of the proposed MS-ATDs calculated through Equation 7.10.

	Company H	Company I	Company J
1	Microservice coupling	Microservice coupling	Microservice coupling
2	Unplanned data sharing/synchronization	Misusing shared libraries	Misusing shared libraries
3	Inadequate use of APIs	Unplanned data sharing/synchronization	Excessive diversity
4	Misusing shared libraries	Unnecessary settings	Unplanned data sharing/synchronization
5	Unnecessary settings	Inadequate use of APIs	Inadequate use of APIs
6	Insufficient metadata	Insufficient metadata	Insufficient metadata
7	Excessive diversity	Excessive diversity	Unnecessary settings

As mentioned in Equation 7.1, the risk of an MS-ATD is the probability of its occurrence multiplied by its negative impact. The more people foresee the debt, the more likely it is to happen, and the fewer people know how to solve the debt, the more likely it is to persist for a longer time. Therefore, we combine (multiply) the weighted sum of votes for the foreseen debts (defined in Equation 7.4) and the weighted sum of votes for the debts practitioners do not know how to solve (defined in Equation 7.6) as the probability of an MS-ATD to occur. Additionally, we used the importance given by the practitioners as the impact of the debt because it is more likely that practitioners consider important debts that have a more significant impact on their contexts. This interpretation is the basis for the calculation of Equation 7.5.

Figure 7.11 presents both the ranking calculated from Equation 7.10 and the normalized priority scores for each company and debt defined at Equation 7.11. Additionally, Figure 7.11 shows colors for the debts with high, medium, and low priorities. Table 7.8 presents another visualization of the ranking calculated from Equation 7.10 for each company.

Companies can use our ranking to prioritize their debts in their own contexts. Below, we present four examples to explain how our prioritization ranking can be helpful.

Based on the information presented by practitioners, our method found that microservice coupling is a major concern for all three companies. This debt received many votes from all companies as foreseen, difficult to solve, and important to avoid. Combining these votes into our ranking makes this debt stand out with the higher priority score among all the studied MS-ATDs. Other companies with mature microservice architectures have also reported this MS-ATD to be important [dMS21]. Thus, the results of this study underscore that microservice coupling arises very early and should be addressed as soon as possible to prevent the debt from increasing. As the number of microservices increases, it becomes increasingly difficult to remove the coupling in microservices.

The next example shows how our method can capture contextual information regarding MS-ATDs. “Excessive diversity” is the last MS-ATD recommended for Companies H and I, but it is one of the three first debts recommended to be prioritized for Company J. Our qualitative data revealed that Company J had internal concerns regarding diversity, but the leaders did not want to enforce any limitations on the teams. The same concerns did not exist for Companies H and I. On the other hand, for Company J, our method captured internal concerns, allowing them to discuss how to address the issue internally.

In the next example, “Insufficient metadata” was identified as having a low priority. Practitioners were not convinced that this debt could be a problem in their projects. This debt involves understanding the big picture of the microservices, including their communication and the impact of the costs across many microservices and teams. Practitioners in the early stages of migration to microservices may not have enough information to be certain about this debt. They might not have a good overview of the architecture because they are focused on their own services, or they might not have enough microservices for this debt to be visible. Whatever the case, there is much uncertainty, and this debt does not seem to be sufficiently significant at the current stage of development.

Finally, as the last example, “Misusing shared libraries” was recommended with high priority to Company J, despite only a small percentage of practitioners having voted for it as important 7.10. This is an interesting result because our method found reasons to believe this debt could be a problem and properly warned practitioners of such concerns. This debt is the most foreseen (see Table 7.5) and difficult to solve (see Table 7.6) for Company J. There is a high probability that this debt will arise and have high costs if not properly mitigated. The costs reported by mature companies regarding this debt are considerable [dMS21], and the practitioners were properly warned and may discuss it internally.

7.4.6 Overall remarks

Practitioners tend to ignore “unnecessary settings.” They believe that this debt cannot be removed while using microservices and they just accept it. A similar circumstance happens with the “excessive diversity,” another debt practitioners do not give much importance. When the debt arrives, it is difficult to repay because practitioners might resist not using the technologies they have adopted until the present date.

Other debts are deliberately taken during the migration to microservices: “unplanned data sharing/synchronization” and “misusing shared libraries.” Several practitioners have reported that using existing databases instead of creating new ones for the microservices may accelerate the migration process. Therefore, they neither plan to have separate databases nor plan the data sharing adequately. There has been less discussion regarding the synchronization of separated databases because they mostly share it to avoid synchronization issues. Regarding the debt “misusing shared libraries,” practitioners hardly believe that the libraries they use are misused. Still, they use them to accelerate the development process and avoid dealing with latency across services. Practitioners rarely think about the number of libraries being used, nor track down the libraries that have constant updates and might block several teams on the project. Practitioners should consider alternatives and use shared libraries when none of the other options is feasible.

Debts such as “insufficient metadata” are difficult to identify. A possible reason is that this is a context-dependent issue or that a broader view of the project is required, usually shared by its architects, and frequently overlooked by microservice developers, who are more concerned with the single services they are developing. Such a situation is both an advantage and a disadvantage of the microservice architecture: practitioners may focus on developing their own microservices without worrying about the work of other teams; however, they lose the overview of the big picture of the project.

“Inadequate use of APIs” is a debt easy to remove and that is always in the middle of our rankings: it is never the most found or foreseen debt, but it is also never in the bottom of the rankings. Therefore, it is a common debt of medium importance. This debt can be mitigated through good practices in developing APIs.

Finally, “microservice coupling” is one of the most important issues that need to be avoided by teams. Our previous study [dMS21] also encountered the problem of coupling in more mature microservice projects than those studied here. One possible cause of coupling might be insufficient experience by teams on how to delimit microservice boundaries, but other causes may also be worth

investigating.

The companies found our results useful in helping them address MS-ATDs in their contexts. As future work, it would be interesting to draw a threshold in our priority ranking above which the debts should be fixed and below which it is safe to postpone debt repayment.

7.4.7 Modeling uncertainty

As presented in Table 7.3, some practitioners answered “not sure” for the MS-ATDs found (RQ1) and foreseen (RQ2). In our analysis (Section 7.3.4), we considered those answers to have a 50% probability of the MS-ATD occurrence. Such a decision entails that two “not sure” count as one “yes.” However, practitioners may have meant more or less than 50%. Although we do not have more information to model this uncertainty, we could have chosen other more or less conservative weights. Here, we discuss the changes in our prioritization when “not sure” is assigned such other weights.

Other areas of research, such as health research (see, for example, the results for the Behavioral Risk Factor Surveillance System, BRFSS, a health-related survey from U.S. residents [Cen20]), completely ignore the uncertainty in their analysis. On the other hand, in our case, the practitioners wanted to express a chance of the debt to happen, although they were not entirely sure. In other words, they expressed a chance, a probability that leads us to the next subject to discuss: the value of the probability. Nevertheless, we want to see what would change if we use this approach in our prioritization. Therefore, we compile the results where “not sure” is given a value of 0.

In addition to the previous consideration, we want to see what would change if we consider that the participants are almost sure about the existence of the debt. Therefore, we used a probability of 0.8. Similarly, we want to see the changes if we consider that participants are skeptical about the existence of the debt. For the last case, we used a probability of 0.2. Table 7.9 shows the changes in all our rankings, including the final prioritization proposal, for all these values compared to the value of 0.5, which we used in our previous analysis. The numbers indicate the changes in the positions in the ranking. The arrows indicate the direction of the change, up or down, in the ranking.

Table 7.9: The changes in the rankings when using different probabilities (0, 0.2, and 0.8) for the uncertainty ("not sure") answers, in comparison with the results with probability 0.5, for each MS-ATD.

Debt	Comp.	MS-ATDs encountered (see Table 7.4)			MS-ATDs foreseen (see Table 7.5)			Final prioritization ranking (see Table 7.8)		
		0	0.2	0.8	0	0.2	0.8	0	0.2	0.8
Insufficient metadata	H									
	I				1 ↓	1 ↓				
	J				3 ↓	1 ↓	1 ↑			
Microservice coupling	H									
	I			2 ↑				1 ↓	1 ↓	
	J			2 ↑	2 ↑	1 ↑	1 ↑			
Inadequate use of APIs	H	1 ↑	1 ↑	1 ↑						
	I				1 ↓	1 ↓				
	J				1 ↑	1 ↑		1 ↑	1 ↑	
Excessive diversity	H									
	I				1 ↑	1 ↑				
	J									
Unplanned data sharing/synchronization	H	1 ↑	1 ↑	1 ↑			1 ↑			
	I			1 ↑			1 ↑			
	J			1 ↑				1 ↓	1 ↓	
Misusing shared libraries	H	1 ↓	1 ↓	1 ↓						
	I	1 ↑	1 ↑	2 ↓			1 ↓	1 ↑	1 ↑	
	J			2 ↓			1 ↓			
Unnecessary settings	H	1 ↓	1 ↓	1 ↓			1 ↓			
	I	1 ↓	1 ↓	1 ↓	1 ↑	1 ↑				
	J			1 ↓		1 ↓	1 ↓			

As shown in Table 7.9, most of the changes are plus or minus 1, and there are even fewer changes in the final prioritization rankings. Therefore, our approach is not highly affected by changes in the uncertainty probabilities. Since we only collected which participants were unsure, we believe that 0.5 would give us a better chance to balance out skeptical and pessimistic opinions from all possible values. The construction of the final prioritization ranking required us to select one of these values. Otherwise, showing different rankings with different probabilities would confuse participants.

As a future improvement, we suggest using a Likert scale to gather more precise data on practitioners' uncertainty. One could model values or categories from "very improbable" to "very probable." Additionally, a "do not know" option would help to exclude invalid answers to RQ1 and RQ2, if any. However, these additions require caution because they demand more attention and time from practitioners on each debt. Therefore, they might increase the participants' response times and affect their participation ratings in the questionnaire.

7.4.8 Implications for research and practice

We received positive feedback from practitioners, indicating that our approach is helpful in decision-making regarding the prioritization of MS-ATDs. A few examples: "We would like to try it again with a more mature project." (Company H); "We are of course interested in going deep. (...) [We need] a serious internal discussion among the developers, architects, and management." (Company I). Company J is going to adjust its priorities based on the MS-ATDs we discussed and asked for additional information on the debts.

Our prioritization method is lightweight and can be applied periodically by companies to manage the risk of ATD during the development process, for example, within an agile architecture framework such as CAFFEA [MB16a]. The risk of a debt changes according to project needs and should be re-evaluated. The requirements for this method are to have a list of MS-ATDs that can be discussed (for example, as in this study, one can use an existing catalog [dMS21]) and one facilitator (researcher or practitioner) who is familiar with the debts and can present them to the remaining participants, collect the answers, and compute the rankings.

Researchers may apply the prioritization method in this study to investigate ATDs from other studies. We envision that the technique can be applied not only for MS-ATDs, but also for any ATD within an organization. However, more research is needed. Researchers also have the raw data available to facilitate comparisons with their own findings. Additionally, there are several suggestions for future and in-depth work that might be useful for research

purposes, such as supporting the early identification of microservice coupling, shared libraries, and database sharing and synchronization, which are some of the most costly debts in our list, or investigating the benefits of using our prioritization approach continuously in agile development processes.

7.5 Limitations

Given the availability constraints of companies and practitioners, we used a subset of MS-ATDs identified in a previous study. Therefore, the final prioritization results may be partial. Still, we selected the most common MS-ATDs found in seven companies experienced with microservices from de Toledo et al. [dMS21]: we considered them also to be the most likely to be relevant to the companies participating in this new study. In addition, the practitioners in this study acknowledged the importance of the selected MS-ATDs during the interviews. The companies involved can also follow a similar procedure to prioritize additional debts and enrich the existing prioritization list. In this case, there is no need to collect the answers for RQ1 again, RQ2, and RQ3 for the MS-ATDs investigated previously.

The prioritization procedure presented here is based on our interpretation of the research context. The value 0.5, used for the answers "not sure" and "partially" impacts our results, especially for responses that are mainly composed of those options. Our interpretation of this result is that, since it is not possible to obtain a better approximation of the uncertainty or the partiality of a solution, the probability of the value being close to 0.5 is high because some practitioners might be very uncertain. By contrast, others might be very sure of their answers. Therefore, on average, the values tend to be 0.5. Further studies could attempt to obtain a more precise estimation of this probability.

The practitioners might not know the debts or might have a different understanding of a specific debt (e.g., shared databases might be understood as coupling by some practitioners and as a very distinct debt by others), which might affect their answers regarding the existence of the debt, or whether they know how to solve it. To reduce this threat, we presented and described each MS-ATD before asking questions. The practitioners were asked to provide their answers immediately after the explanation. We also collected information on practitioners' satisfaction with our descriptions and interpretations. Only three interviewees reported difficulties in understanding our explanations for one question each.

It was also challenging to interpret the situations in which the participants answered: "not sure" or "partially." We considered those as a 50% probability

of the answer being "yes." However, this interpretation may not be accurate. Practitioners and researchers should interpret these rankings cautiously. For example, the second element in the priority ranking might be the best to start with for a specific company. However, such an interpretation is a reasonable approximation of all the answers because some practitioners have more knowledge than others and some solutions are less partial than others. Therefore, we considered that, on average, the answers converged to 50%. Our rankings are suggestions for discussion by practitioners and researchers. The raw data are provided in Table 7.3 for further interpretation.

Finally, it is possible that other companies not included in our study have successfully implemented a different prioritization or refactoring strategy and applied it to different MS-ATDs than those discussed in this work. However, none of the companies in this or previous studies reported a similar prioritization approach. We performed a lightweight literature review at the beginning of this study, but we did not find any other strategies for prioritizing MS-ATDs. Our work is exploratory and can be considered a starting point for future research. More studies are necessary to investigate additional MS-ATDs and to further validate and potentially improve our approach.

7.6 Related Work

Lenarduzzi et al. [Len+20] monitored the evolution of code TD in a small to medium-sized company that migrated a monolithic system to microservices. They concluded that the migration reduced the overall code TD. The authors did not study architectural TD, which was the focus of this study.

Taibi et al. [TLP20] defined a taxonomy of 20 microservice anti-patterns, based on 27 interviews with experienced practitioners. Several of these anti-patterns are related to migration: "no DevOps tools," "too many technologies" (the same as excessive diversity in our paper), "I was taught to share" (which we defined as misusing shared libraries), "static contract pitfall" (i.e., APIs that are not properly versioned), "mega-service," "shared persistence" (part of our unplanned data sharing or synchronization), "sloth" (too many coupling among the microservices, resulting in a distributed monolith), and "trying to fly before you can walk" (i.e., migrating to microservices while the practitioners lack the necessary skills). In our study, we considered the overlapping anti-patterns to be MS-ATDs. (For discussing the relationship between architectural anti-patterns and ATDs, see de Toledo et al. [dMS21]). Additionally, we created rankings for the identified MS-ATDs and investigated how to prioritize them.

Martini et al. [Mar+18] identified and prioritized ATDs through architec-

tural smells in a multiple case study. They analyzed four software projects in the same company. Their approach consisted of using a tool called Arcan [Fon+17] to identify architectural smells leading to ATD and then asking the practitioners to prioritize the debts found. Their approach was limited to ATDs that can be identified through architectural smells. On the other hand, our approach was tested with microservice-specific ATDs and used a risk-based approach for prioritization, combining information from the present, the foreseen future, and the importance given by practitioners.

Pigazzini et al. [PAM19] presented an approach also supported by Arcan [Fon+17] to identify candidate microservices in monolithic Java projects. The approach begins with the identification of architectural smells that can hinder migration. Despite their relation to migration, the identified anti-patterns were detected in the monolithic architecture before migration. By contrast, we only considered ATDs in microservices during and after migration.

Panichella et al. [PRT21] proposed metrics to compute and visualize the coupling between microservices, which may help evaluate the cost of coupling in a given context. Their approach can help practitioners find and measure microservice coupling, but it is focused on a single MS-ATD and does not propose ways to prioritize debts. Our case study highlights situations in which practitioners might not have promising approaches for measuring microservice coupling. The method proposed by Panichella et al. [PRT21] may be useful in such cases.

7.7 Conclusion

This paper presents our investigation of how ATD issues specific to microservices, MS-ATDs, accumulate in three companies that are migrating to such an architectural style. We carefully conducted our study with the practitioners by giving them presentations on the MS-ATDs and double-checking that they were on the same page with the explanation of the MS-ATDs. We then asked practitioners which MS-ATDs they considered difficult to remove (tackle) and which ones were the riskiest in the present and future of their projects. We discussed the answers given by the practitioners considering their contexts and created rankings for the most found, foreseen, difficult to solve, and important MS-ATDs for each of the participating companies. We also proposed an approach to prioritize MS-ATDs based on risk and discussed it with participants. The participants also reported that the results and the prioritization method were useful and may be used in their contexts. For example, participants from one company said that our approach would help

people become aware of the issues they have not yet seen. Thus, they are be able to take action to mitigate the negative consequences (interest) of the debts.

Our most important findings related to individual MS-ATDs were as follows: (i) “Misusing shared libraries” is a common debt during migration to microservices. Companies start using shared libraries because of the convenience of reusing code from their original architectures. However, these companies may be good candidates for high costs due to misuse of such libraries in the future, as identified in previous studies [dMS21; TLP20]. (ii) It is common for companies to share databases during the early stages of migration to microservices to accelerate the migration process. (iii) Microservice coupling occurred frequently, possibly related to practitioners’ lack of experience in creating microservices. However, they postpone the discussion of the debt to the late stages, possibly because the costs of the debt are small at the beginning of the migration. One possible interpretation of this result is that practitioners might not have proper ways or tools to measure the growth of microservice coupling.

Given that “Misusing shared libraries” and “Microservice coupling” were common MS-ATDs in our study, other organizations may also consider them. The procedure of creating rankings reported in this paper may help other organizations prioritize fixing or avoiding MS-ATDs during migration before their costs increase. Overall, we believe that our results will help understand the relationship between ATDs and microservices.

Our study is fully replicable by researchers. The raw data are also available, allowing researchers to use such data in their approaches and facilitating comparison of the results.

Future work includes: running in-depth studies on the companies participating in this study to understand the consequences of their current architectural decisions; analyzing other companies migrating to microservices; investigating possible metrics; quantifying debts and costs; exploring deeper the aspects that practitioners emphasize when they consider an MS-ATD important.

Authors’ addresses

Saulo S. de Toledo University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, saulos@ifi.uio.no

Antonio Martini University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, antonima@ifi.uio.no

Phu H. Nguyen SINTEF, Forskningsveien 1, 0373 Oslo, Norway phu.nguyen@sintef.no

7. Accumulation and prioritization of Architectural Debt in three companies migrating to microservices

Dag I. K. Sjøberg University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, dagsj@ifi.uio.no

Chapter 8

Reducing Incidents in Microservices by Repaying Architectural Technical Debt

Saulo S. de Toledo, Antonio Martini, Dag I. K. Sjøberg, Agata Przybyszewska, Johannes Skov Frandsen

Published in *Proceedings of the 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021*, 2021. DOI: 10.1109/SEAA53835.2021.00033.

Abstract

Introduction: Architectural technical debt (ATD) may create a substantial extra effort in software development, which is called interest. There is little evidence about whether repaying ATD in microservices reduces such interest.

Objectives: We wanted to conduct a first study on investigating the effect of removing ATD on the occurrence of incidents in a microservices architecture.

Method: We conducted a quantitative and qualitative case study of a project with approximately 1000 microservices in a large, international financing services company. We measured and compared the number of software incidents of different categories before and after repaying ATD.

Results: The total number of incidents was reduced by 84%, and the numbers of critical- and high-priority incidents were both reduced by approximately 90% after the architectural refactoring. The number of incidents in the architecture with the ATD was mainly constant over time, but we observed a slight increase of low priority incidents related to inaccessibility and the environment in the architecture without the ATD.

Conclusion: This study shows evidence that refactoring ATDs, such as lack of communication standards, poor management of dead-letter

queues, and the use of inadequate technologies in microservices, reduces the number of critical- and high-priority incidents and, thus, part of its interest, although some low priority incidents may increase.

Contents

8.1	Introduction	166
8.2	Background	167
8.3	Methodology	175
8.4	Results	182
8.5	Discussion	188
8.6	Related work	190
8.7	Conclusions and future work	190

8.1 Introduction

Software is a competitive factor for many companies. They compete to deliver value faster and timely and frequently prioritize feature delivery over architecture work. Such prioritization, in turn, may lead to unexpected rework costs in the future, which will gradually reduce the value of the product over time, culminating in technical debt (TD).

TD is a metaphor coined to explain the trade-offs between short-term and long-term decisions in software development [Avg+16]. TD can be found in all software lifecycle phases, from requirements and architecture specification to development and testing [KNO12]. A TD is a sub-optimal solution with short-term benefits that incur an extra cost called *interest*. The cost of developing or refactoring a solution to avoid the debt is called the *principal* [Avg+16].

A challenging and costly type of TD is architectural technical debt (ATD), which is caused by architectural decisions that affect quality attributes such as maintainability and evolvability [Avg+16], and other qualities such as reliability [BMB17b]. Core financial software services, for example, must be reliable to ensure the money of their clients is handled correctly. Otherwise, the monetary losses may spread to the clients, the company, and its investors.

Some of the most recently embraced software development techniques to deal with the growing complexity of software in large companies are related to adopting a microservices architecture. Such an architecture has been successful in splitting an entire software solution into smaller and more manageable pieces of software called microservices, which are easier to develop, test, and deploy than larger pieces of software like a monolith. However, like any other

architectural style, it has drawbacks, such as extra operational complexity and an extra effort to manage distributed systems [Fow15]. There is scarce evidence on ATD in Microservices.

In this paper, we conducted a case study in a financing company to investigate whether repaying ATD in microservices increases the reliability of a system measured in terms of the number of incidents. Incidents are unwanted or unexpected interruptions of a system's services or a reduction in its quality. They cause extra maintenance costs, decreasing teams' productivity, software reliability, and availability. Thus, they are part of the interest of the debts that cause them. The company in this study repayed ATD with a refactored architecture. We propose the following research questions (RQs) in the context of microservices architecture:

1. **RQ1:** Does repaying the ATD change the type of incidents that occur?
2. **RQ2:** Does repaying the ATD reduce the number of critical- and high-priority incidents?
3. **RQ3:** Does repaying the ATD change the distribution of the incidents over time for the original and refactored architectures?

RQ1 investigates whether the types of incidents change after the removal of ATD. An overview of the types of incidents in both architectures is important to understand which types of incidents have decreased and which ones have increased.

RQ2 investigates the priority of the incidents. A single incident with high priority might be much more costly than several lower priority incidents. Therefore, answering this question helps us understand whether the refactoring caused a substantial cost change in the project due to critical- and high-priority incidents.

RQ3 investigates the distribution of the various types of incidents over time. We aim to understand which types of incidents happen when, and whether they are recurrent or periodic. Such information may help us understand how ATD removal affects the occurrence of the different types of incidents.

8.2 Background

8.2.1 Microservices architecture

The microservices architectural style is used by many companies, including the one participating in this study. Lewis and Fowler [LF14] define 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." The microservices architectural style is frequently described as an alternative to monolithic applications built and deployed as a single unit. Microservices have some advantages over monoliths: they are easier to scale, they have shorter cycles for test, build and release, and are less frequently affected by downtime [Fow15]. Microservices are also particularly useful for continuous delivery because they are usually independent of each other and may be tested and delivered separately, reducing lead time. There are, however, some drawbacks and challenges with microservices, such as the extra effort for handling distributed systems (because each service is deployed separately) and increased operational complexity [Fow15].

Another concept tightly related to microservices is Service Oriented Architecture (SOA). In this paper, we consider microservices as one way of implementing SOA, although there are different views about this topic [Zim17]. In fact, SOA might describe a set of applications that cannot be considered microservices, such as those using an Enterprise Service Bus (ESB). An ESB is an infrastructure that mediates requests among services, intercepting their communications and providing transformation capabilities [NG05]. On the other hand, microservices invest in a lightweight communication mechanism [LF14]. Both architectural styles share many concepts and techniques, such as circuit breakers, service discovery, and service registry [MW16].

8.2.2 Architectural Technical Debt (ATD)

ATD is a type of TD that focuses on the product's architecture. ATD might slow down the addition of new functionalities and increase maintenance and other costs. ATD is considered the most challenging type of TD to be unveiled and managed [BMB17a; Ern+15; KNO12]. There are three main concepts on TD: debt, interest, and principal [Avg+16]:

- **Debt:** The debt is a sub-optimal solution with short-term benefits but will, however, require the payment of interest in the future. For example, suppose that a development team can (i) spend time planning the software architecture for scalability or (ii) start the development right away without a well-designed architecture. Choosing (ii) has the benefit of having a final version of the software ready earlier, but it might require the software to be rewritten from scratch later when scalability is required in production, for example.

- **Interest:** The interest is the extra cost that must be paid because of the presence of a debt or, from another perspective, the amount that will be saved if there is no such debt. In the example for the debt above, the interest is the extra cost for dealing with the lack of scalability, such as deploying additional servers for supporting more users and costs with their maintenance.
- **Principal:** The principal is the cost of developing a solution that avoids the debt or refactoring a solution to remove the debt. In the previous example, the principal is the cost in time or any other resources required to plan the software architecture for scalability in advance.

The importance of ATD motivated static analysis tools to offer a way of measuring it, mostly through architectural smells [Avg+21]. However, the interest of ATD has proven difficult to measure, and only a few tools and studies attempt to quantify it. For example, Martini et al. [MSM18] use productivity loss, while Xiao et al. [Xia+16] use bugs to quantify the ATD interest. These approaches do not capture the whole interest because they focus on specific artifacts. In fact, the whole interest generated by ATD may consist of several factors [Len+21], ranging from productivity loss to reliability issues to the degradation of developers' morale.

Part of the interest in ATD might be visible in the form of incidents, which decrease the software reliability and availability and cause extra maintenance costs, which also slows down teams. Other interests, such as delays in lead time or feature delivery, impossibility to create new features, and others, are not quantified in this paper. We investigate the effect of repaying ATD in microservices on the types and number of incidents after refactoring a microservices architecture (see Figure 8.1).

8.2.3 Company context

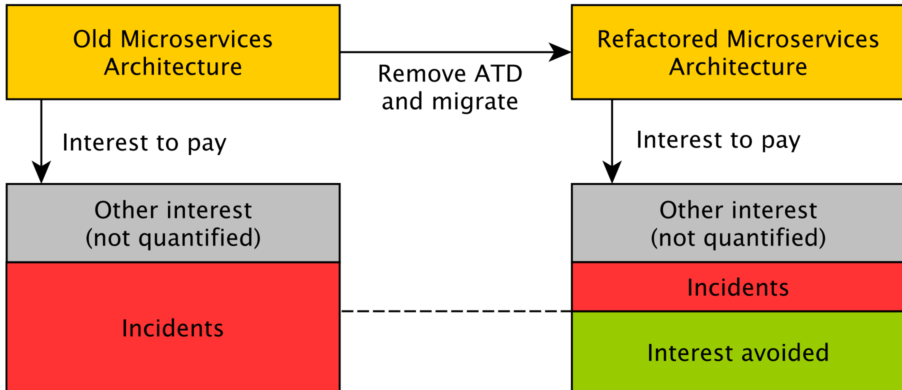
This study was conducted in a large, multinational financial services company with a complex and heterogeneous IT system landscape. The core business is about performing financial services within the boundaries of risk and compliance.

8.2.3.1 Company business

The financial sector is subject to a series of tight regulations. Different governance bodies have different legislations (Basel, MiFID[1] I, MiFID II,

[1] Markets in Financial Instruments Directive

Figure 8.1: The old microservices architecture had many incidents caused by the presence of ATD. Our goal is to quantify how many of these incidents were removed by repaying the debts.



FRTB[2]) that are prerequisites to having a banking license. Such regulations define how to report on risk timely; how the company should be exposed in different markets, currencies, and clients; the limits on risk; data lineage; and how to report trade activities to authorities. The company operates in several countries with different national legislations. On top of those, there are super-national regulations such as those given by the European Union (EU) that raise another dimension of IT complexity.

The risk computations give a significant business advantage if properly managed. That is, they should be performed as close to real time as possible, and the different computation-heavy simulations should be performed in as much detail as feasible. The business demand on computing needs to be consistent across a data set of around 20 million daily transactions.

8.2.3.2 Company organization

The company has introduced a separation of duties, where employees either belong to *business*, *IT*, or *operations*. Every application has (i) an *application owner* in the business that is the responsible sponsor and drives the business requirements, and (ii) an application provider in the IT organization that is responsible for the application development and delivery. Finally, the

[2]Fundamental Review of the Trading Book

operations team takes accountability for operating the application/service, being responsible for the Service Level Agreement (SLA) [PMI13]. The majority of the teams are virtual and operate across national borders.

Over the years, the company faced many organizational changes and merges, resulting in heterogeneous IT systems, a lack of ownership for older artifacts, and orphaned business logic in the IT systems.

8.2.3.3 Software architecture

The software architecture of the company's division that was part of our study consists of about 1,000 microservices, of which about 150 are business-critical and form the core of the solution. The services communicate in different ways: a legacy communication layer solution, a new communication layer solution, point-to-point service calls, database pumps, direct database connections, file transfers and mails, external banking networks, and mainframe gateways. The connectivity layers often contain logic that performs ETL[3] and sometimes business logic.

8.2.3.4 Company process

The need for new compliance and legislation requirements and the inability to upgrade systems when needed because of TD's accumulation led to the start of a large program that built the new communication layer solution introduced before. The program aimed to simplify some of the complexity, reduce TD and restore business agility and business reliability.

The focus of the new program was the data foundation and fixing the lack of accountability, as appointed by the *data asset owners*, who were responsible for governing and mandating the processing of various data assets throughout the organization. A DevOps culture was introduced to deal with the challenging changes in the process required by the new complications in the logic and difficulties to test changes.

Originally, the company's governance model was waterfall-based, inspired by the PRINCE2 methodology [AXE17], with a series of approval gates: early architecture approval, solution design approval, and a run gate. Changes were not possible after a project was in its run state.

The new governance model moved away from the traditional waterfall methodology to agile with Scrum. Later, the new program embraced the Scaled

[3] Extraction, Transformation, and Loading; three steps to combine data from multiple sources.

Agile Framework (SAFe) with its Agile Release Trains (ART) construct because of the need to coordinate many projects.

8.2.4 ATD removal (repayment)

The company had an overwhelmingly complex old architecture with thousands of point-to-point interfaces and a complex and orphaned logic in the integration among them. Such characteristics led to the occurrence of several incidents, which created issues related to reliability, availability, and evolvability, increasing maintenance costs, and reducing time for developing new features. In a previous study [dMS21], we qualitatively identified a set of debs, their interest, and principal in several companies, including the one in this paper. Particularly for the company in this study, the interviewees reported that the following ATDs were the main causes of several incidents:

- **ATD 1:** Poor dead-letter queue growth management, in which many messages were lost, creating incidents.
- **ATD 2:** Microservice coupling, one of the main causes of cascading failures, propagating and multiplying incidents.
- **ATD 3:** Lack of communication standards among microservices, in which the conversions of formats would cause incidents.
- **ATD 4:** Use of business logic in communication among services, which would cause incidents if not properly updated when the involved services were changed.
- **ATD 5:** Unnecessary diversity in the technologies chosen to handle communication among the services, in which they had many queuing mechanisms from different vendors spread across the services, leading to complexity, difficulties in communication among services, and potential incidents.
- **ATD 6:** Many services using different versions of the same internal shared libraries, potentially leading to incidents due to incompatibilities or deprecation.

Additional details regarding those ATDs, such as how we identified them, their interest, and their principal, might be found in [dMS21].

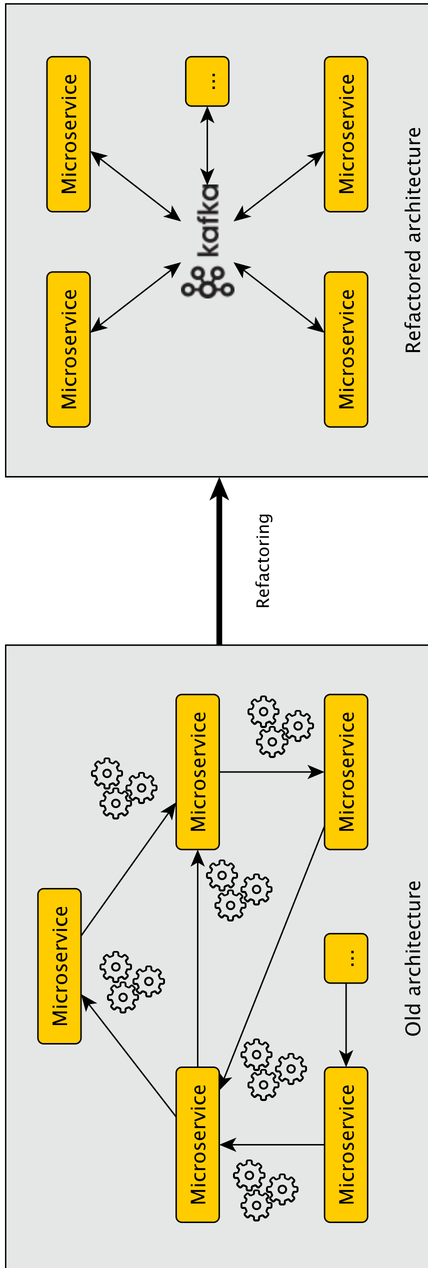


Figure 8.2: Solution to solve ATDs 2, 4, and 5.

The company decided to remove such ATDs by simplifying the architecture and transforming data access to a publish/subscribe model (reducing ATD 2), where data is published once in a standard format (removing ATDs 3 and 4, the last because there was no need for business logic for transforming data) and made available via a distributed streaming platform (in this case, Kafka[4], solving ATD 5). Figure 8.2 presents an overview of the refactoring for some of the ATDs.

The publish-subscribe pattern was promoted by changing the architecture into a message-driven style where data was made accessible as messages in a uniform message format. These messages were to be consumed by microservices either in real-time or from data stores. The data quality and the canonical message formats were the accountability of Data Owners, a new role introduced into the organization.

The queues were decentralized other than the dead-letter queue, which was removed. The responsibility for the message deliveries was moved to the microservices themselves, solving ATD 1. Additionally, the company reduced the use of internal shared libraries, reducing ATD 6.

The refactored architecture was implemented with the following design goals:

- **Always up:** Create a message broker that is always online, avoiding the complex logic of ensuring that a recipient of a message is alive for every single point-to-point integration. A continuously online message broker addresses the operational stability issues caused by the microservices' time coupling.

- **Never lose a message:** Guarantee message delivery, addressing the cascading failures issues of which lost messages, time coupling, and communication failures were the cause.

- **Self-service:** Bake in responsibilities and governance to a self-service to help ensure that data ownership, access, and data quality issues are followed through the technical implementation. Such a self-service addresses the lack of organizational accountability issues.

- **No gossip:** Publish all messages in a canonical message format curated by a responsible data owner. A system is allowed to share only facts about the information of which the system itself is the golden source.

[4] https://kafka.apache.org/

Other information is passed by reference. These guidelines address the poor-data-quality issues.

Currently, the refactored architecture is serving 100 million daily messages.

8.3 Methodology

We conducted an exploratory single-case study of a large international financial services company. The research consisted of quantitative and qualitative analyses of the number of incidents before and after refactoring an old microservices architecture. During our research, parts of the system remained in the old architecture, while other parts were migrated to the refactored one.

8.3.1 Case study design

We compared an old architecture with a refactored architecture in the same product in the studied company. Both architectures were still in use over the period of this study. The refactored architecture was developed to repay ATDs identified as the source of several issues, including incidents. According to the company, incidents are unexpected events causing malfunction in the system, such as network, credential, and database issues.

We designed our case study as follows: (i) a set of interviews to identify the ATDs; (ii) additional interviews with architects to discuss incidents; (iii) quantitative data collection on incidents; (iv) interpretation and validation of the quantitative data with interviewees from (ii); and (v) results and interpretation of the data.

This case study is particularly useful because: (i) despite some migration happening from one architecture to another (which is expected in such cases), both architectures were being used by the same group of users at the same time; (ii) the system used to report the incidents for both architectures is the same, making it easier to compare the data; and (iii) both architectures share the same vocabulary, such as incidents, priorities, and others.

8.3.2 Data collection

Figure 8.3 presents an overview of our data collection. Each box is explained in the subsections below.

8.3.2.1 Preliminary interviews

We started by interviewing 10 subjects from the company to identify the ATDs, their interest, and their principal. The detailed results from these interviews are reported in [dMS21]. A summary of the ATDs reported in those previous interviews related to the incidents is presented in Section 8.2.4. Later, we interviewed again one architect among those previous subjects and another architect that we have not met before to understand what kind of information we could use to measure the interest of the ATD.

All the interviewees were selected by convenience sampling according to their involvement with the project and availability. The interviews were recorded and lasted about one hour each.

8.3.2.2 Quantitative data collection

The company provided an architect involved in the project to collect the data requested by the authors. We requested data specific to incidents in both architectures as reported in our preliminary interviews. We obtained two datasets, one for each architecture.

We collected 8330 incidents registered in the company's internal project management tool from October 2016 to June 2019, a period in which both architectures were in operation. Among the incidents, 7571 were reported in the version with the old architecture, and 759 were reported in the refactored architecture. The incidents were recorded in two ways: automatically by an end-to-end monitoring tool or manually for those that could not be identified by the monitoring tool. The incidents for both architectures were reported with the same management tool, in the same time frame, and by the same teams, using similar processes and assessment methods, making the two datasets easily comparable.

The incidents collected contained the date of the report, a descriptive summary, the report submitter (a tool used by the company or a person), the priority (critical, high, medium, or low), and the related architecture (the old or the refactored one).

8.3.2.3 Data filtering and cleaning

The data we collected was filtered and cleaned based on the interviewees' information and our understanding of the data we received, as detailed in the remainder of this section. We finished this phase with a total of 3,383 incidents to be used in our investigation.

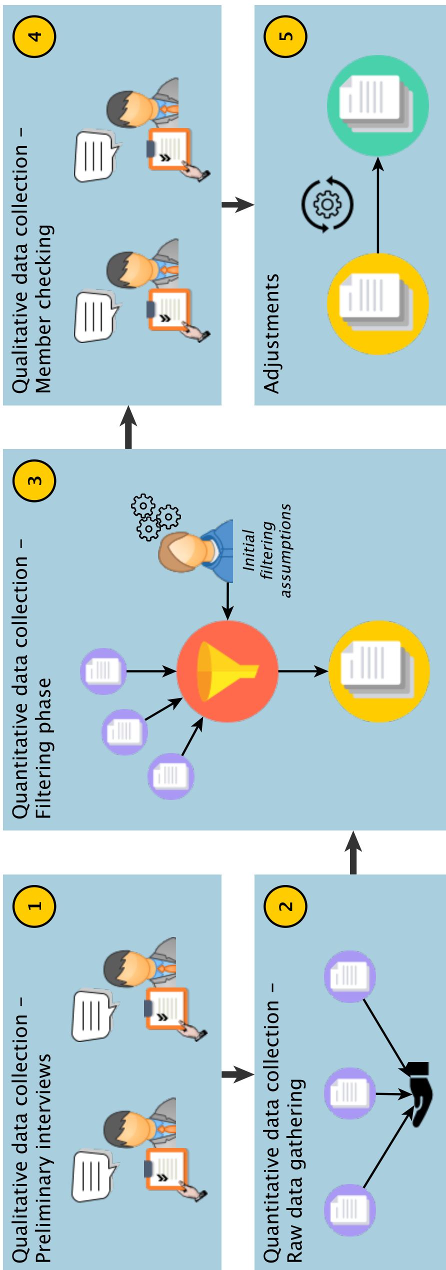


Figure 8.3: Data collection overview.

We started by *removing repeated incidents.* The automatic tools were running continuously and reported many incidents repeatedly until new versions were deployed into production by the developers.

According to our preliminary interviews, cascading failures should be considered a single incident. In fact, the automatic tools reported the same incident at the same date and time for multiple services. We proceeded by *reducing those incidents to a single one.* Our interviewees reported that the probability of having two or more incidents that were not related was very low, so it was safe to proceed with this step.

After the previous step, smaller sets of incidents had distinct messages but were reported at the same minute. Thus, *we reduced those sets to single incidents* because the interviewees identified them as cascading incidents with the same root cause. In the cases where we had distinct priorities, we selected the incident with the most critical priority.

Furthermore, we *grouped the same incidents with different thresholds* (e.g., more than 80% of disk usage or queue fill percentage). In fact, for example, the same incident was reported with 80%, 90%, and 100% of disk usage or queue fill percentage.

Finally, we *considered all incidents that had the exact same description within the same day as single incidents.* This was supported by our interviewees. They said that in those cases, considering the same day, the probability of having different incidents was irrelevant given the tools they were using for such reports. This consideration was reasoning that the same incident could be repeatedly reported until it was fixed. If the incident was reported the next day, there was a possibility that it was a new incident because they should have fixed the issue within the same day. Despite the existence of exceptions, it was not safe to remove those incidents on different days.

8.3.2.4 Member checking

The incidents were categorized according to the process described in Section 8.3.3. The results and the categories were discussed and validated in follow-up interviews with the last two subjects from the preliminary interviews, a technique known as *member checking* [Run+12]. In the new interviews, which were recorded and lasted one hour each, the interviewees visualized a sample of the data. They were asked if they agreed with the categories created to group the incidents and whether the incidents we considered as duplicates were indeed the same.

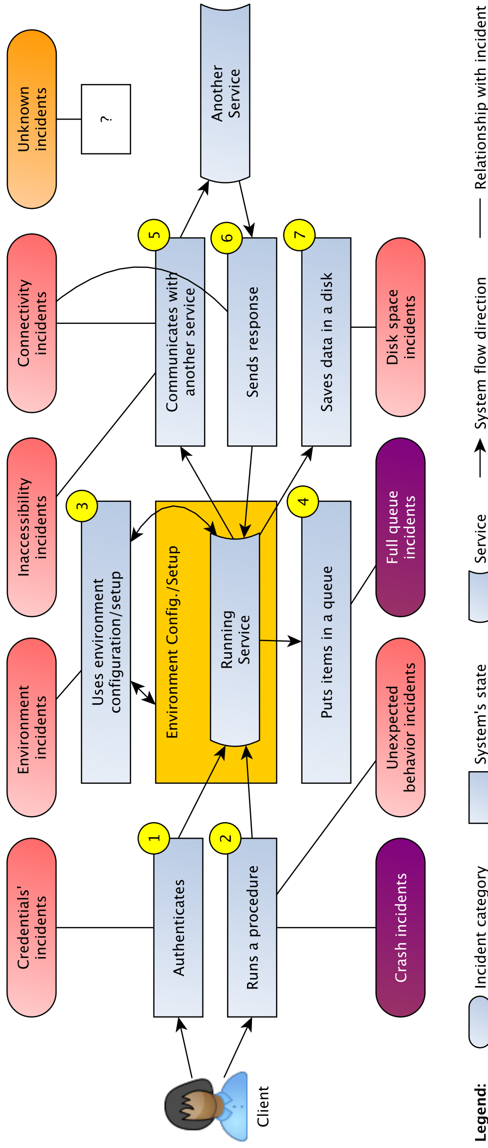


Figure 8.4: Categories of incidents.

8.3.2.5 Adjustments

As a result of previous steps, we updated some categories of incidents and slightly adjusted how the data was cleaned up.

8.3.3 Data analysis

RQ1: Does repaying ATD change the type of incidents that occur?

We organized the incidents into categories according to our understanding of the data and the information acquired in the first interviews using thematic analysis [Run+12]. We used the incident descriptions to classify the incidents into nine categories represented by rounded squares in Figure 8.4. Such categories were created iteratively by reviewing the incidents, as in the following example. Suppose we have the list of incidents below:

1. Unable to login at service-01
2. Service connectivity lost
3. Access denied

We will start with the first incident by identifying it as a "credential incident," our first category. We consider the second incident as a "connectivity incident" for the second category. When analyzing the third incident, we notice that it can be categorized as a "credential incident," so there is no need to create a new category. After repeating such steps for all the incidents in our database, we created nine different categories. The categories were reviewed during the process and later validated with one interviewee who worked with those incidents. A few incidents were reclassified after the interviews.

We explain the types of incidents below. The numbers in parentheses refer to the yellow indications in Figure 8.4:

- **Credential incidents:** those that happen when users or services try to authenticate themselves into a service (1), such as login failures or certificates expiration.
- **Unexpected behavior incidents:** unexpected results, such as incorrect data processing, invalid results, and queue errors, whenever a user runs a procedure (2).

- **Environment incidents:** those related to the environment on which the service was running. Examples are incidents related to containers, HTTP servers, virtual disks and machines, and environment variables. Such incidents might happen, for example, when a service tries to load or save the information in an environment variable (3).
- **Full queue incidents:** those that may cause data loss or other problems due to full queues. Queues are used widely by both systems for communication among the services and other tasks (4).
- **Inaccessibility incidents:** those reported when a service is unreachable, such as when a service tries to communicate with another service (5).
- **Connectivity incidents:** those reported when the target service is available, but there are issues in the connectivity, such as package loss and long delays. It might happen when a service starts to communicate with another service (5) or when it waits for a response from that service (6).
- **Disk space incidents:** those regarding lack of space in the disk for services that use the disk (7) to save data.
- **Crash incidents:** those that happen when a service aborts unexpectedly, usually when a user or a service tries to run a procedure (2) but fails.
- **Unknown incidents:** any kind of incident that we could not classify appropriately because insufficient information was available.

We compared the architectures by counting the number of different types of incidents for both architectures.

RQ2: Does repaying the ATD reduce the number of critical- and high-priority incidents?

The company's internal incident management team defines four different priority levels for the incidents that are used to prioritize the efforts and allocate personal to solve the issues caused by the incidents. The levels are presented below.

- **Critical:** a widespread incident on a business-critical application, making it inaccessible to users or other services.
- **High:** an incident making a system partially inaccessible in a single location.

- **Medium:** a non-critical incident causing inconveniences in a particular system.
- **Low:** a non-critical incident affecting a single user.

Critical- and high-priority incidents are particularly damaging to the company because of potential monetary losses and have a high cost for fixing, such as extra personnel, infrastructure, and time costs. Medium and low priority incidents cause non-critical issues that will not prevent end-users from using the system. The issues may not even be visible to the end-users.

The priority for each incident is already in our dataset. We analyzed the number of incidents for each priority level for both the old and the refactored architectures.

RQ3: Does repaying ATD change the distribution of the incidents over time for the original and repaid architectures?

After the refactored architecture was ready, the services were migrated progressively over time. Consequently, we should expect to see different events at different points in time. Therefore, we aim to find key events related to the refactoring and repayment of ATD. External events may increase some kinds of incidents, such as the need for adequacy to a new regulation. Before we start our analysis, we presuppose the following patterns:

(a) If the incidents are equally distributed over time, they have a recurrent cause that may be analyzed.

(b) If the incidents are concentrated at the same point in time, a common cause might be investigated. If they were found particularly in the last months in our data, it is possible that new causes just emerged, and we might expect an increase of this type of incidents in the future.

We then grouped the incidents by quarters of the year.

8.4 Results

Table 8.1 summarizes the data we will discuss in Sections 8.4.1 and 8.4.2 regarding RQ1 and RQ2, respectively, and introduces the basis for answering RQ3 in Section 8.4.3.

Table 8.1: Incidents by group and priority in both architectures and their differences. Green indicates a reduction in the refactored architecture; red an indicates increase in the refactored architecture.

Priority	*Critical*			*High*			*Medium*			*Low*			**Total**		
Category	*Old*	*Ref.*	*Diff.*	*Old*	*Ref.*	*Diff.*	*Old*	*Ref.*	*Diff.*	*Old*	*Ref.*	*Diff.*	*Old*	*Ref.*	*Diff.*
Credentials	0	0	0	0	0	0	5	4	-1	1	0	-1	6	4	-2
Environment	0	0	0	0	0	0	8	53	45	11	1	-10	19	54	35
Inacessibility	5	1	-4	8	1	-7	122	82	-40	94	125	31	229	209	-20
Conectivity	0	0	0	5	1	-4	9	3	-6	1	0	-1	15	4	-11
Unexpected behavior	2	0	-2	5	0	-5	30	7	-23	35	0	-35	72	7	-65
Disk space	0	0	0	0	0	0	25	64	39	32	117	85	57	181	124
Queue full	0	0	0	0	0	0	92	0	-92	96	0	-96	188	0	-188
Crashes	1	0	-1	13	0	-13	402	0	-402	1891	0	-1891	2307	0	-2307
Unknown	0	0	0	2	1	-1	7	7	0	10	4	-6	19	12	-7
Total	8	1	-7	33	3	-30	700	220	-480	2171	247	-1924	2912	471	-2441

8.4.1 RQ1: Does repaying the ATD change the type of incidents that occur?

From our analysis, we can notice three main highlights regarding RQ1:

1. **The total of incidents decreased:** A total of 2912 was reported in the old architecture, against 471 in the refactored one. This represents a reduction of 84% in the total number of incidents.
2. **Despite the overall reduction, two categories of incidents actually increased:** The refactored architecture has more environment and disk space incidents than the old one. This is due to the new way the disk and the environment are managed in the new architecture.
3. **Two categories of incidents were completely removed:** From the 2912 incidents in the old architecture, 2307 were *crashes*, and 188 were *full queues*. Those categories of incidents do not happen anymore in the refactored architecture. These incidents' disappearance indicates that the refactored architecture has better recovery from failures and that the queues' decentralization eliminated full queues incidents.

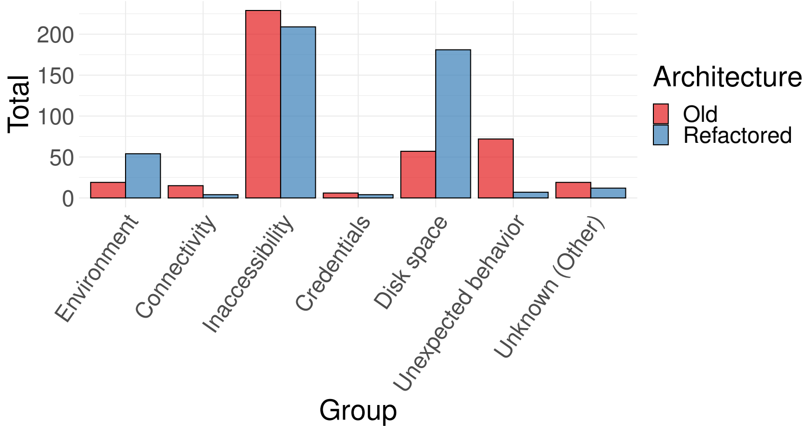
Figure 8.5 presents the remaining types of incidents distributed over time for both architectures, organized by categories. In summary, we observed a shift from some types of incidents to others after the ATD repayment.

8.4.2 RQ2: Does repaying the ATD reduce the number of critical- and high-priority incidents?

The total number of critical- and high-priority incidents for both architectures is 45, less than 2% of the total number of incidents. Still, according to our interviewees, the impact of a single critical- or high-priority incident is much more than the impact of several medium- and low-priority incidents together because they cause downtime. Moreover, medium- and low-priority incidents are much easier to fix than critical- and high-priority ones. Thus, we can highlight the following results regarding RQ2. There was a **reduction** of the number of:

1. **critical incidents**: Only one critical incident was reported in the refactored architecture against eight in the old one.

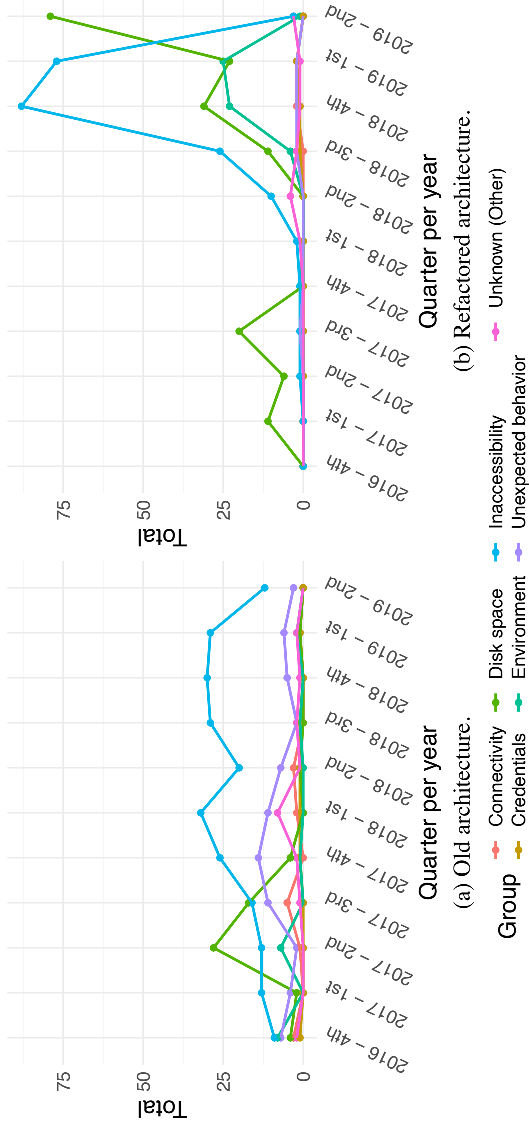
Figure 8.5: Incidents by category found in both architectures.



2. **high-priority incidents**: Only three high-priority incidents were reported in the refactored architecture against 33 in the old one.

8.4.3 RQ3: Does repaying ATD change the distribution of the incidents over time for the original and repaid architectures?

Figure 8.6a shows that the incidents are, to some extent, uniformly distributed over time for the old architecture, whereas Figure 8.6b shows occasional peaks in the refactored architecture. The data we have for June 2019 (part of the second quarter of that year) is incomplete. Despite this, it is unlikely that there is a huge increase of incidents in the missing two weeks of data. Considering that the missing days would have the same number of incidents as the previous period, we expect to have missed about 17% more incidents in that period, keeping the same trends as before. Such an interpretation is also supported by one of the interviewees who helped us review the results. Some categories of incidents deserve special attention in the **old architecture**.



(a) Old architecture.

(b) Refactored architecture.

Figure 8.6: Incidents per type distributed over time for both architectures.

- The *unexpected behaviors* had a slight increase in the second quarter of 2017. According to one of our interviewees, a new regulation was established, and many services had to be updated. Thus, the changes required by the regulation increased the probability of unexpected incidents. Thus, there is a chance that such an increase was an occasional event.
- Unknown causes triggered an increase in *disk space incidents* during the second quarter of 2017. Those incidents, however, seem to have been resolved later.
- *Inaccessibility incidents* seem to have been common during the whole lifetime of the old architecture.

On the other hand, Figure 8.6b shows that only three categories of incidents require immediate attention in the **refactored architecture**.

- *Disk space incidents*, which, for reasons unknown to our interviewees, had a decrease over six months between 2017 and 2018 but have increased substantially since the second quarter of 2018.
- *Inaccessibility incidents* increased in number from the first quarter of 2018 onward but lowered again toward the end of the second quarter of 2019.
- *Environment incidents* had an increase from the second quarter of 2018 but had a sharp decrease in the last months of our data collection. Also, these incidents seem to be more common in the refactored architecture than in the old one.

In summary:

1. **The refactored architecture has a lower total number of incidents**: Only 471 compared to 2912 from the old architecture.
2. **The incidents that increased are not critical- or high-priority**: Although the refactoring seems to have increased some categories of incidents, those incidents were not high-priority. The priority of the incidents is assessed internally by the company according to the number of services impacted and their criticality for the company and clients.

3. **There are many fewer critical- and high-priority incidents after the refactoring**: There was a shift from a considerable number of critical- and high-priority incidents to more medium- and low-priority ones.

4. **Other considerations**: Only the disk space incidents keep increasing over time in our dataset. Inaccessibility and environment incidents seem to have been reduced in the second quarter of 2019.

8.5 Discussion

8.5.1 Did removing ATD decrease the number of incidents?

We observed a huge reduction in the number of incidents for most categories, especially on critical- and high-priority incidents, reducing downtime and maintenance costs, and increasing the system's overall reliability.

The number of environment and disk space incidents increased (see Table 8.1), and the inaccessibility incidents, although their total is lower in number in the refactored architecture, are concentrated in the same period of time (see the first quarter of 2018 in Figure 8.6b). Still, those incidents have a lower priority level, which led our interviewees to affirm that such a trade-off was acceptable for migrating to the refactored architecture. One of our interviewees also mentioned that most of those incident increases resulted from the migration from the old architecture. Such a fact might explain why the environment and inaccessibility incidents reduced in the second quarter of 2019 (see Figure 8.6b).

On the other hand, disk space incidents deserve more attention due to a constant increase in our data and might be a source of problems if not properly controlled. We remind, however, that disk space incidents are triggered by thresholds and do not necessarily point to a real problem but a possible one. This also explains why such incidents have lower priorities in our dataset.

8.5.2 The actual cost of the interest

The ultimate goal of measuring ATD is to calculate a monetary gain. This paper does not provide such a calculation for confidentiality reasons. However, a cost can be associated with the various types of incidents and the effort spent on refactoring.

As an example regarding critical incidents, a one-day disruption of forex trading services, with a revenue stream of 80 million Euro/year, can cost about

200,000 Euro/day. Similar costs per day would be incurred by disruption to other business-critical services. Medium- and low-priority incidents also have an internal cost for fixing them, despite being much lower than when a disruption is the consequence. Thus, we can infer that decreasing the number of incidents reduced a high interest cost generated by the ATD.

8.5.3 Implications for research

Refactoring a huge microservices architecture is costly. Without proper cost-benefit evidence, companies tend not to risk proceeding with the refactoring, making cases like the one we studied hard to be accessible by researchers. Our case study reports empirical evidence for researchers regarding one such valuable case and provides arguments researchers may use to justify further research on repayment of ATD in microservices, which might be successful and reduce development costs, and its progress might be followed by a decrease in the number of incidents. Measuring part of the interest in ATD by using incidents may be a promising approach.

Our results also present a concrete and reproducible approach researchers may use to quantify part of the interest in ATD in microservices.

8.5.4 Implications for industry

Many practitioners seek ways to quantify the benefits of removing ATD and reducing debt costs. However, measuring the impact of architectural changes is challenging, and there is a lack of approaches in the literature about how to do it. The current case study shows empirical evidence from an industry case that it is possible to pay less interest by refactoring microservices. Companies with a similar context may consider these results relevant for understanding their cases and promoting refactoring, for example. The study also presents a concrete and reproducible approach for companies to quantify part of the interest in ATD in microservices by using incidents to visualize part of the ATD interest.

8.5.5 Limitations and threats to validity

Our approach is based on several incidents and a limited set of interviews. We are not considering other variables in which we could measure the effect of refactoring ATD, such as the number of changes in the system and the experience of individuals and teams with the technologies involved.

Another limitation is the lack of data to analyze the causes for the peaks seen in Figure 8.6b, especially the increase of disk space incidents. We asked our interviewees about good reasoning for those cases, but they could not point to any specific information we could verify in our dataset. These are good starting points for practitioners to investigate some possible future issues.

The removal of the ATD might also have other effects that we have not considered in this work, such as the number of changes required, the maintenance, and the management effort. Investigation of such effects in future works would require different case study setups.

8.6 Related work

de Toledo et al. [dMS21] is a qualitative study reporting the ATDs, their principal, and their interest found in seven large companies, including the one in this study. A subset of the ATDs reported by the company in that previous study was identified here as the studied ATDs that were generating incidents.

Xiao et al. [Xia+16] propose an approach to quantify the ATD interest by mining error-prone files from a project's revision history. They propose a mathematical approach to calculate the interest based on what they define as the Design Rule Space, "*a new form of architectural representation that uniformly captures both architecture and evolution structures to bridge the gap between architecture and defect prediction*." Our approach quantifies the interest in terms of the number of incidents. We also focused on microservices and looked into the different categories of incidents.

Nord et al. [Nor+12] use qualitative expert judgment to calculate rework and total ownership costs based on the software architecture to assist ATD management. Lenarduzzi et al. [Len+20] quantify the TD in microservices using SonarQube, a tool for inspection of code quality. Their goal is to quantify the TD before and after a migration from a monolithic to a microservices architecture to report whether the migration reduced the costs with TD. Since they use source code, they do not focus on the software architecture. Fontana et al. [FFZ15] use architectural smells to evaluate ATD. None of these works quantify the actual interest. Our work, instead, uses quantitative data from incidents to quantify the interest of the ATD.

8.7 Conclusions and future work

We reported a case study providing evidence that refactoring ATD in microservices might reduce incidents and downtimes. A reduced amount

of incidents leads to an increase in overall system reliability and availability, consequently reducing maintenance effort and development costs.

We also proposed a way to quantify parts of the interest of ATD in microservices through the occurrence of incidents. We hope such an approach is useful for practitioners and researchers while investigating measurements on ATD.

A few directions remain to be investigated in future works. The first one is related to the sudden increase of incidents in the refactored architecture between 2018 and 2019; we need more information to be able to investigate the causes for it. Another direction is to investigate additional interest costs using other data sources besides the incidents, such as the costs of implementing new functionalities and features, which may increase or decrease with a refactored architecture. Finally, a last direction is to investigate the principal of the ATD, which comprises the costs for architecture refactoring.

Authors' addresses

Saulo S. de Toledo University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, saulos@ifi.uio.no

Antonio Martini University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, antonima@ifi.uio.no

Dag I. K. Sjøberg University of Oslo, Postboks 1080 Blindern, 0316 Oslo, Norway, dagsj@ifi.uio.no

Agata Przybyszewska IT University of Copenhagen, Copenhagen, Denmark agpr@itu.dk

Johannes Skov Frandsen Copenhagen, Denmark johannesfrandsen@me.com

Chapter 9

Discussion and Conclusions

In this chapter, we discuss the contributions of this thesis concerning the RQs introduced in Chapter 3. Later we highlight possible future work and present our conclusions.

9.1 Research questions and contributions of the thesis

9.1.1 RQ1: What are MS-ATDs?

We investigated this first research question (RQ1) into the following sub-questions:

- **RQ1.1: What are the most critical MS-ATDs?**
- **RQ1.2: What is the negative impact of such MS-ATDs?**

We dedicated two studies to answer RQ1: Studies 1 and 2. In Study 1, we created a catalog of the most critical MS-ATDs in the context of companies using microservices (RQ1.1) through a cross-company study. Our catalog included 16 MS-ATDs. Some examples are microservice coupling, lack of communication standards among services, poor API design, and misuse of shared libraries. Previous authors created related catalogs but with different purposes. Taibi et al. [TL18] proposed a catalog of bad smells on microservices. Taibi et al. [TLP20] defined a taxonomy of microservice anti-patterns (including solutions) based on interviews with experienced practitioners. Bogner et al. [Bog+19a] reported several issues related to MS-ATDs in a qualitative study with companies to explore evolvability on microservices. However, none of these works discuss the extent to which the bad smells and anti-patterns reported are related to MS-ATDs. Bad smells and anti-patterns are debts if they have an interest. In some cases, they do not need fixing. In other cases, their cost is high and requires immediate attention. Additionally, the costs and solutions vary according to the context of the project. This thesis differs from those works by systematically investigating debt, interest, and principal in microservices architectures. Additionally, we identified several MS-ATDs that were not related to any of the previously identified bad smells or anti-patterns.

We performed exploratory studies, and it was not possible to calculate precise values for the interest and the principal of the debts. We asked for quantitative data to support such a calculation, but practitioners could not provide it. However, we represented each debt's interest by negative impact and each debt's principal by possible solutions (RQ1.2). The actual costs for principal and interest may vary according to the context of the project. However, our description of the impact and solutions may help practitioners mitigate costs with MS-ATDs in their projects even without a precise calculation that would not be valid in other contexts anyways.

Study 2 presented a deeper investigation of one of the MS-ATDs from Study 1. We discussed the impact of the debt under investigation in the development agility. The topic is important because the microservice architecture should improve development agility.

Among the negative impact of some MS-ATDs in our catalog, we found potential breaks on microservices, dependencies among teams (which may impact their efficiency), management overhead, and difficulties in providing new features. We presented solutions reported by the informants for many of the debts, such as investing in a good API design, identifying and replacing shared libraries with microservices, or defining standards for communication.

9.1.2 RQ2: How do MS-ATDs occur?

We investigated this second research question by answering (RQ2) the following sub-questions:

- **RQ2.1: How do MS-ATDs occur in early-stage microservices?**
- **RQ2.2: How do MS-ATDs occur in mature microservice systems?**

We answered RQ2.1 in Study 3 and RQ2.2 in Studies 1 and 2.

Previous work described bad smells [TL18], anti-patterns [TLP20], and other issues [Bog+19a] in microservices. Still, they did not investigate whether those issues are related to MS-ATDs, their relation with the company contexts, and what are the causes of the accumulation of debts. MS-ATDs have interest and principal that may vary depending on the project context. For example, companies may decide to share databases among services due to licensing costs. The licensing fees might be greater than the costs of sharing the database. However, failures in the database may affect all services at the same time. Therefore, the companies must decide what to do based on the project needs.

This thesis investigates the MS-ATDs and their reasons in early-stage and mature microservice architectures.

Some MS-ATDs in mature microservice systems were more common than others, and some debts seem to be more context-dependent. Several companies reported the existence of unnecessary settings and microservice coupling, but only a few reported the inadequate use of APIs, for example. Microservice coupling seems to be a debt less context-dependent than the inadequate use of APIs. Although we cannot claim causality, it seems reasonable to think that companies making extensive use of particular APIs are more susceptible to using them in a non-optimal way than companies that do not use them. Therefore, practitioners may find it helpful to start reading our report guided by the MS-ATDs from companies with a context similar to their companies. Similarly, researchers may find it beneficial to understand the occurrence of the debts according to different contexts. They may use the contexts to drive new research in new companies.

Many research studies on microservices investigate migration techniques from previous architectures to microservices [DLM19], but we did not find studies investigating the accumulation of MS-ATDs in early stages of migration. Study 3 investigated the accumulation of MS-ATDs in three companies in the early stages of migration to microservices that aimed to modernize their software architecture (from monolithic, SOA, etc.). We examined a subset of MS-ATDs from Study 1 in the context of these other companies. There were not many microservices for the companies in Study 3 since they were in the early stages of migration. However, we found evidence that some MS-ATDs may already occur in those early stages. Therefore, some MS-ATDs may be avoided or mitigated in the early stages of migration before they get too costly. On the other hand, some debts speed up the migration process, but practitioners must be aware of their costs to address them timely.

Study 3 may also help researchers address the occurrence of MS-ATDs in the early stages of migration to microservices. They can extend the study in other companies, add more debts, or choose one or more of those reported debts for further investigation.

Studies 1, 2, and 3 combined help us know which debts occur earlier and their costs. These studies give us an overview of MS-ATDs in companies experienced with microservices and companies starting to use this architectural style. This knowledge may help practitioners manage the MS-ATDs repayment since they know more about which debts are likely in the short and long terms and may be ready for those MS-ATDs beforehand.

9.1.3 RQ3: How to prioritize MS-ATDs?

Prioritization is one of the main activities for managing ATD [LAL15]. Previous studies addressed the prioritization of ATD from distinct points of view. Martini and Bosh [MB16b] proposed a method and a tool (AnaConDebt) to support prioritization and decision-making for ATDs. The proposed method applies to ATD in general and has a learning curve. The tool is commercial and currently not accessible to the general public. We looked for a simpler method to prioritize MS-ATDs in this work. Martini et al. [Mar+18] identified and prioritized ATDs through architectural smells identifiable by another tool called Arcan [Fon+17]. Arcan can detect a limited set of architectural smells and did not detect smells specific to microservices at the time of that research.

For Martini and Bosch [MB16b], ATD management is mainly a risk assessment practice. Therefore, we investigated RQ3 from a risk (of the interest) perspective and assessed methods to prioritize MS-ATDs. We can divide RQ3 into the following sub-questions, both addressed in Study 3:

- **RQ3.1: Which MS-ATDs do practitioners consider risky?**
- **RQ3.2: Which methods can companies use to prioritize the avoidance or repayment of MS-ATDs?**

Some MS-ATDs are riskier than others. The risk of the debt also depends on the context of the project. For example, the impact caused by a confidential data leak due to using an insecure authorization library in the code is more significant in an application publicly exposed than in an application used in a private, secure network.

Study 3 answers RQ3.1 by calculating a risk score for MS-ATDs based on the probability of the MS-ATDs to occur, the difficulty of repaying it, and the importance of the debt for the practitioners. We found which debts are riskier than others according to the project context. Our method also highlights that some MS-ATDs are less context-dependent than others. For example, the microservice coupling as MS-ATD has the higher risk scores for all companies in Study 3. In contrast, the debt “excessive diversity” has different risk levels for distinct companies. Finally, Study 3 provides a method for prioritization of the MS-ATDs based on the risk scores. The approach proposed in Study 3 is new and completely different from previous approaches for debt prioritization. It is also lightweight and does not require tools to be used.

Practitioners involved in the study well accepted the approach, which can be easily adapted to other projects in their contexts. Researchers may also

benefit from the study by having a new approach for prioritizing MS-ATDs and several discussion points that can support further research. Researchers can also investigate the usefulness of our approach with different instances of MS-ATDs, such as the others presented in the catalog from Study 1 and not studied during Study 3.

9.1.4 RQ4: What are the solutions and effects of repaying or avoiding MS-ATDs?

After deciding what MS-ATDs to repay, practitioners must know how to do it (or avoid it in the future). Additionally, knowing the effects of the MS-ATD repayment may help practitioners understand better how to make this repayment. Studies 1, 2, and 4 investigate RQ4 from the perspective of the following sub-questions:

- **RQ4.1: What are possible solutions to repay or avoid MS-ATDs?**
- **RQ4.2: What are the effects of repaying MS-ATDs on their interest?**

Studies 1 and 2 present several solutions for the MS-ATDs identified in RQ1 and thus answer RQ4.1. Taibi et al. [TLP20] also proposed solutions for microservices anti-patterns that are related to MS-ATDs. However, our catalog is distinct from theirs, and we focused on MS-ATDs, while they focused on anti-patterns. The list of solutions we described is not exhaustive, i.e., there might exist other solutions beyond those presented. However, it comes from mature and large microservice projects, so it might be helpful for other practitioners. Examples of solutions are:

- Redesigning microservices to use messaging approaches to communicate when there are many complex API calls.
- Limiting the set of technologies when the technology diversity becomes hard to manage from a project perspective.
- Replacing specific shared libraries with microservices to avoid cascading upgrades. Or replacing other shared libraries with code within the services for smaller changes, i.e., when the effort is less costly than the cascading upgrades.

- Using an API first approach to design the microservices to reduce the coupling generated by poorly designed interfaces among the services.

Companies running microservices may avoid redesigning working software if they find that the cost of MS-ATDs interest is lower than the cost of their principal. Therefore, Study 4 presented the reduction of incidents after repayment of the debts in a large company. There were no similar studies regarding MS-ATDs. The incidents were a proxy of part of the debts' interest. Our study shows that the solutions applied by the studied company drastically reduced the number and the criticality of incidents.

Study 4 is mainly informative to practitioners as an example that MS-ATDs repayment may be beneficial in the long run. On the other hand, researchers may use it as an example to indirectly measure interest, a proposal that is also new in TD research. No previous work on TD used indirect measures to measure debt interest. Measuring the interest of MS-ATDs is difficult in practice, and Study 4 uses the occurrence of incidents as a proxy of the interest.

9.2 Future work

This thesis contributed to the body of knowledge of ATD and microservice areas. However, that also opened room for several other research questions that remain open and require future work. Future research can expand the catalog introduced in Study 1 with new debts, costs, and solutions in contexts that we have not studied. Additionally, a survey to gather more data about the MS-ATDs already reported would help increase the data's generalizability. It is also possible for future works to dive deep into each of the MS-ATDs to investigate measures for costs, methods for identification and mitigation, impact from different perspectives (such as on Agility as done in Study 2), or the characteristics of the debt in different contexts. All those suggestions may also target supporting decision-making on MS-ATDs repayment.

Future research may also improve the prioritization approach proposed in Study 3. Our first suggestion is to replace the answers with a Likert scale. So the method can capture a more precise measure of the respondents' uncertainty when they answer "partially" or similar responses. Study 3 also does not identify which debts must be paid from those that do not need repayment. In a list of seven debts, the practitioners have a priority ranking but not an indication of how many of the top items need immediate repayment. Further research may try to improve such recommendations to practitioners. Additionally, long-term studies on companies using our prioritization approach would help understand

the implications of adopting it. Future work may extend our method by considering additional prioritization aspects.

Measuring the costs of any ATD is hard, and researchers have the challenge of finding measures that do not require too much overhead, learning or process changes in organizations' software development. Study 4 used incidents as a proxy to measure part of MS-ATDs' interest indirectly. Future research may investigate the approach in other studies or find other proxies measures. Further work may also investigate the principal. Finally, repaying MS-ATDs may impact other areas not explored in the study, such as software maintenance or management effort. Future works may take those aspects under consideration.

9.3 Conclusion

Microservices are being widely adopted by software development organizations. The market competitiveness and pressure to meet business requirements demand applications to evolve continuously, adding new features and being always available. The maintenance windows that were typical in many software applications in the past are no longer acceptable today. Microservices allow companies to meet these and many other requirements.

However, software development organizations are still learning how to use microservices. Their inexperience with this architectural style leads to debts with high interest. We created a catalog containing 16 different MS-ATDs, their costs, and solutions to help practitioners and researchers understand MS-ATDs and reduce the current knowledge gap that leads to high costs. We also proposed an approach for prioritizing MS-ATDs that practitioners and researchers can use.

Finally, we reported a case study providing evidence that repaying MS-ATDs might reduce incidents and software downtime. Fewer incidents increase system reliability and availability, thus reducing maintenance effort and development costs. We also used incidents to quantify part of the MS-ATDs' interest. Practitioners and researchers can use such a novel approach to measure ATD.

In this thesis we presented new knowledge about occurrences of MS-ATDs, their negative costs and solutions, and proposed a prioritization approach for MS-ATDs and a proxy for quantifying the MS-ATD interest based on incidents.

Bibliography

[AB13] Alzaghoul, E. and Bahsoon, R. "CloudMTD: Using real options to manage technical debt in cloud-based service selection". In: *2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings*. IEEE Computer Society, 2013, pp. 55–62.

[Alv+14] Alves, N. S. R. et al. "Towards an ontology of terms on technical debt". In: *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*. Institute of Electrical and Electronics Engineers Inc., Dec. 2014, pp. 1–7.

[Alv+16] Alves, N. S. R. et al. "Identification and management of technical debt: A systematic mapping study". In: *Information and Software Technology* vol. 70 (Feb. 2016), pp. 100–121.

[Avg+16] Avgeriou, P. et al. "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)". In: *Dagstuhl Reports* vol. 6, no. 4 (2016). Ed. by Avgeriou, P. et al., pp. 110–138.

[Avg+21] Avgeriou, P. C. et al. "An Overview and Comparison of Technical Debt Measurement Tools". In: *IEEE Software* vol. 38, no. 3 (2021), pp. 61–71.

[AXE17] AXELOS. *Managing Successful Projects with PRINCE2*. Ed. by The Stationery Office. 6th ed. The Stationery Office, 2017, p. 400.

[Bes+18] Besker, T. et al. "Embracing Technical Debt, from a Startup Company Perspective". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2018, pp. 415–425.

[BHJ16] Balalaie, A., Heydarnoori, A., and Jamshidi, P. *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture*. May 2016.

[BMB16] Besker, T., Martini, A., and Bosch, J. "A Systematic Literature Review and a Unified Model of ATD". In: *Proceedings - 42nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2016*. IEEE, Aug. 2016, pp. 189–197.

[BMB17a] Besker, T., Martini, A., and Bosch, J. "The pricey bill of Technical Debt: When and by whom will it be paid?" In: *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*. IEEE, Sept. 2017, pp. 13–23.

[BMB17b] Besker, T., Martini, A., and Bosch, J. "Time to pay up: Technical debt from a software quality perspective". In: *CIbSE 2017 - XX Ibero-American Conference on Software Engineering*. 2017, pp. 235–248.

[BMB18] Besker, T., Martini, A., and Bosch, J. "Managing architectural technical debt: A unified model and systematic literature review". In: *Journal of Systems and Software* vol. 135 (Jan. 2018), pp. 1–16.

[Bog+19a] Bogner, J. et al. "Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Cleveland, Ohio, USA, 2019, pp. 546–556.

[Bog+19b] Bogner, J. et al. "Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells". In: *Proceedings - 2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019*. Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 95–101.

[Bus+21] Bushong, V. et al. "On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study". In: *Applied Sciences* vol. 11, no. 17 (Aug. 2021).

[Cen20] Centers for Disease Control and Prevention. *Calculated Variables in the 2019 Behavioral Risk Factor Surveillance System (BRFSS) Data File*. Department of Health and Human Services, July 2020.

[Cle85] Cleveland, W. S. *The Elements of Graphing Data*. Ed. by Monterey. California, United States: Wadsworth Advanced Books and Software, 1985, p. 323.

[CLM21] Ciolkowski, M., Lenarduzzi, V., and Martini, A. *10 Years of Technical Debt Research and Practice: Past, Present, and Future*. Nov. 2021.

[CS15] Corbin, J. M. and Strauss, A. L. *Basics of qualitative research: techniques and procedures for developing grounded theory*. Fourth edition. SAGE, 2015.

[Cun92] Cunningham, W. "The WyCash portfolio management system". In: *SIGPLAN OOPS Messenger* vol. 4, no. 2 (1992), pp. 29–30.

[DB12] De Silva, L. and Balasubramaniam, D. “Controlling software architecture erosion: A survey”. In: *Journal of Systems and Software* vol. 85, no. 1 (Jan. 2012), pp. 132–151.

[de +19] de Toledo, S. S. et al. “Architectural Technical Debt in Microservices: A Case Study in a Large Company”. In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. Montreal, Quebec - CA: IEEE, May 2019, pp. 78–87.

[DLM18] Di Francesco, P., Lago, P., and Malavolta, I. “Migrating Towards Microservice Architectures: An Industrial Survey”. In: *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*. IEEE, Apr. 2018, pp. 29–38.

[DLM19] Di Francesco, P., Lago, P., and Malavolta, I. “Architecting with microservices: A systematic mapping study”. In: *Journal of Systems and Software* vol. 150 (Apr. 2019), pp. 77–97.

[DML17] Di Francesco, P., Malavolta, I., and Lago, P. “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. Apr. 2017, pp. 21–30.

[dMS21] de Toledo, S. S., Martini, A., and Sjøberg, D. I. K. “Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study”. In: *Journal of Systems and Software* vol. 177 (July 2021).

[Dra+17] Dragoni, N. et al. “Microservices: Yesterday, today, and tomorrow”. In: *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, 2017. Chap. 12, pp. 195–216.

[Ern+15] Ernst, N. A. et al. “Measure it? Manage it? Ignore it? Software practitioners and Technical Debt”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 50–60.

[Eva04] Evans, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004, p. 529.

[FFZ15] Fontana, F. A., Ferme, V., and Zanoni, M. “Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations”. In: *Proceedings - 2nd International Workshop on Software Architecture and Metrics, SAM 2015*. IEEE, July 2015, pp. 1–7.

[Fon+17] Fontana, F. A. et al. “Arcan: A Tool for Architectural Smells Detection”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017, pp. 282–285.

[Fow15] Fowler, M. *Microservice Trade-Offs*. July 2015.

[Fur+18] Furda, A. et al. “Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency”. In: *IEEE Software* vol. 35, no. 3 (May 2018), pp. 63–72.

[FY97] Foote, B. and Yoder, J. W. “Big Ball of Mud”. In: *4th Pattern Languages of Programming Conference (PLoP 1997)*. Monticello, Illinois, USA, 1997.

[GS11] Guo, Y. and Seaman, C. “A portfolio approach to technical debt management”. In: ACM Press, 2011, pp. 31–34.

[GS87] Garcia-Molina, H. and Salem, K. “Sagas”. In: *ACM SIGMOD Record* vol. 16, no. 3 (1987), pp. 249–259.

[Guo+11] Guo, Y. et al. “Tracking technical debt - An exploratory case study”. In: *IEEE International Conference on Software Maintenance, ICSM*. 2011, pp. 528–531.

[HS17] Hasselbring, W. and Steinacker, G. “Microservice architectures for scalability, agility and reliability in e-commerce”. In: *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*. Institute of Electrical and Electronics Engineers Inc., June 2017, pp. 243–246.

[HW12] Hohpe, G. and Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 1st. Addison-Wesley, 2012.

[KH19] Knoche, H. and Hasselbring, W. “Drivers and Barriers for Microservice Adoption - A Survey among Professionals in Germany”. In: *Enterprise Modelling and Information Systems Architectures (EMISAJ)* vol. 14 (Jan. 2019), pp. 1–35.

[KNO12] Kruchten, P., Nord, R. L., and Ozkaya, I. “Technical debt: From metaphor to theory and practice”. In: *IEEE Software* vol. 29, no. 6 (2012), pp. 18–21.

[LAL15] Li, Z., Avgeriou, P., and Liang, P. “A systematic mapping study on technical debt and its management”. In: *Journal of Systems and Software* vol. 101 (Mar. 2015), pp. 193–220.

[Len+20] Lenarduzzi, V. et al. “Does migrating a monolithic system to microservices decrease the technical debt?” In: *Journal of Systems and Software* vol. 169 (Nov. 2020).

[Len+21] Lenarduzzi, V. et al. “A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools”. In: *Journal of Systems and Software* vol. 171 (Jan. 2021).

[LF14] Lewis, J. and Fowler, M. *Microservices: a definition of this new architectural term*. Mar. 2014.

[Li+19] Li, W. et al. “Service Mesh: Challenges, state of the art, and future research opportunities”. In: *Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019*. Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 122–127.

[LLA14] Li, Z., Liang, P., and Avgeriou, P. “Architectural Debt Management in Value-Oriented Architecting”. In: *Economics-Driven Software Architecture*. 2014, pp. 183–204.

[LLA15] Li, Z., Liang, P., and Avgeriou, P. “Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios”. In: IEEE, May 2015, pp. 65–74.

[LT18] Lenarduzzi, V. and Taibi, D. “Microservices, Continuous Architecture, and Technical Debt Interest: An Empirical Study”. In: *Euromicro SEAA. Work in Progress* (Oct. 2018). arXiv: 1810.10855.

[MA18] Marquez, G. and Astudillo, H. “Actual Use of Architectural Patterns in Microservices-Based Open Source Projects”. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. Vol. 2018-Decem. IEEE Computer Society, July 2018, pp. 31–40.

[Mar+18] Martini, A. et al. “Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11048 LNCS. Madrid, Spain, 2018, pp. 320–335.

[MB15] Martini, A. and Bosch, J. "Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners". In: IEEE, Aug. 2015, pp. 422–429.

[MB16a] Martini, A. and Bosch, J. "A multiple case study of continuous architecting in large agile companies: current gaps and the CAFFEA framework". In: *Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016*. Institute of Electrical and Electronics Engineers Inc., July 2016, pp. 1–10.

[MB16b] Martini, A. and Bosch, J. "An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt". In: *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 31–40.

[MB17] Martini, A. and Bosch, J. "On the interest of architectural technical debt: Uncovering the contagious debt phenomenon". In: *Journal of Software: Evolution and Process* vol. 29, no. 10 (2017), pp. 1–18.

[MBC15] Martini, A., Bosch, J., and Chaudron, M. "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study". In: *Information and Software Technology*. Vol. 67. Elsevier, Nov. 2015, pp. 237–253.

[McB07] McBride, M. R. "The software architect". In: *Communications of the ACM* vol. 50 (5 May 2007), pp. 75–81.

[Mo+19] Mo, R. et al. "Architecture Anti-patterns: Automatically Detectable Violations of Design Principles". In: *IEEE Transactions on Software Engineering* (Apr. 2019), pp. 1–1.

[Mos+18] Mosqueira-Rey, E. et al. "A systematic approach to API usability: Taxonomy-derived criteria and a case study". In: *Information and Software Technology* vol. 97 (May 2018), pp. 46–63.

[MSM18] Martini, A., Sikander, E., and Madlani, N. "A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component". In: *Information and Software Technology* vol. 93 (Jan. 2018), pp. 264–279.

[MVA18] Márquez, G., Villegas, M. M., and Astudillo, H. "A pattern language for scalable microservices-based systems". In: *ACM International Conference Proceeding Series*. 2018, pp. 1–7.

[MW16] Montesi, F. and Weber, J. "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *CoRR* vol. abs/1609.0 (2016). arXiv: 1609.05830v2.

[New17] Newman, S. *Building Microservices: Designing Fine-Grained Systems*. 1st. O'Reilly Media, Inc., 2017.

[New19] Newman, Sam. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. 1st. O'Reilly Media, Inc., 2019, p. 221.

[NG05] Niblett, P. and Graham, S. "Events and service-oriented architecture: The OASIS web services notification specifications". In: *IBM Systems Journal* vol. 44, no. 4 (2005), pp. 869–886.

[Nor+12] Nord, R. L. et al. "In search of a metric for managing architectural technical debt". In: *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012*. IEEE, Aug. 2012, pp. 91–100.

[PAM19] Pigazzini, I., Arcelli Fontana, F., and Maggioni, A. "Tool support for the migration to microservice architecture: An industrial case study". In: *LNCS*. Vol. 11681 LNCS. Springer Verlag, Sept. 2019, pp. 247–263.

[PMI13] PMI, ed. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. 5th ed. Newtown Square, PA: Project Management Institute, 2013.

[PRT21] Panichella, S., Rahman, M., and Taibi, D. "Structural Coupling for Microservices". In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, May 2021, pp. 280–287. eprint: 2103.04674.

[RH08] Runeson, P. and Höst, M. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* vol. 14, no. 2 (Dec. 2008), p. 131.

[Ric16] Richards, M. *Microservices vs. Service-Oriented Architecture*. Ed. by Barber, N. and Roumeliotis, R. First. O'Reilly Media, Inc., 2016, p. 45.

[RK98] Reddy, P. K. and Kitsuregawa, M. “Reducing the blocking in two-phase commit protocol employing backup sites”. In: *Proceedings - 3rd IFCIS International Conference on Cooperative Information Systems, CoopIS 1998*. Vol. 1998-Augus. Institute of Electrical and Electronics Engineers Inc., 1998, pp. 406–415.

[Rob02] Robson, C. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers*. 2nd ed. Blackwell Publishing, 2002.

[RSZ17] Rademacher, F., Sachweh, S., and Zundorf, A. “Differences between model-driven development of service-oriented and microservice architecture”. In: *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*. Institute of Electrical and Electronics Engineers Inc., June 2017, pp. 38–45.

[Run+12] Runeson, P. et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. Wiley Publishing, 2012.

[SB21] Sjøberg, D. I. K. and Bergersen, G. R. “Construct Validity in Software Engineering”. In: (Mar. 2021).

[Sch13] Schmid, K. “A formal approach to technical debt decision making”. In: *QoSA 2013 - Proceedings of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures*. New York, New York, USA: ACM Press, 2013, pp. 153–162.

[SDJ07] Sjøberg, D. I. K., Dybå, T., and Jørgensen, M. “The future of empirical methods in software engineering research”. In: *FoSE 2007: Future of Software Engineering*. 2007, pp. 358–378.

[SS20] Schwaber, K. and Sutherland, J. *The Scrum Guide: The Definitive Guide to Scrum*. 2020.

[STV18] Soldani, J., Tamburri, D. A., and Van Den Heuvel, W. J. “The pains and gains of microservices: A Systematic grey literature review”. In: *Journal of Systems and Software* vol. 146 (Dec. 2018), pp. 215–232.

[Thö15] Thönes, J. “Microservices”. In: *IEEE Software* vol. 32, no. 1 (Jan. 2015), p. 116.

[TL18] Taibi, D. and Lenarduzzi, V. “On the Definition of Microservice Bad Smells”. In: *IEEE Software* vol. 35, no. 3 (May 2018), pp. 56–62.

[TLP20] Taibi, D., Lenarduzzi, V., and Pahl, C. “Microservices anti-patterns: A taxonomy”. In: *Microservices: Science and Engineering*. Ed. by Bucchiarone, A. et al. Cham: Springer International Publishing, 2020, pp. 111–128.

[UKS19] Unger-Windeler, C., Klunder, J., and Schneider, K. “A Mapping Study on Product Owners in Industry: Identifying Future Research Directions”. In: IEEE, May 2019, pp. 135–144.

[VB02] Van Gurp, J. and Bosch, J. “Design erosion: Problems and causes”. In: *Journal of Systems and Software* vol. 61, no. 2 (Mar. 2002), pp. 105–119.

[Ver+21] Verdecchia, R. et al. “Building and evaluating a theory of architectural technical debt in software-intensive systems”. In: *Journal of Systems and Software* vol. 176 (June 2021), p. 110925.

[VML18] Verdecchia, R., Malavolta, I., and Lago, P. “Architectural technical debt identification: The research landscape”. In: *Proceedings - International Conference on Software Engineering*. 2018, pp. 11–20.

[Vog08] Vogels, W. “Eventually consistent”. In: *Queue* vol. 6, no. 6 (Oct. 2008), pp. 14–19.

[Xia+16] Xiao, L. et al. “Identifying and quantifying architectural debt”. In: *Proceedings - 38th IEEE International Conference on Software Engineering*. Austin, Texas, USA: IEEE Computer Society, May 2016, pp. 488–498.

[Yin18] Yin, R. K. *Case Study Research and Applications: Design and Methods*. 6th ed. Sage Publications, Inc, 2018, p. 352.

[YM20] Yoder, J. W. and Merson, P. “Strangler Patterns”. In: *Proceedings of the 27th Conference on Pattern Languages of Programs*. The Hillside Group, 2020, p. 25.

[Zim17] Zimmermann, O. “Microservices tenets: Agile approach to service development and deployment”. In: *Computer Science - Research and Development* vol. 32, no. 3-4 (July 2017), pp. 301–310.

Appendices

Appendix A

Study 1 Interview guide

We present the interview guide for Study 1 in Table A.1.

Table A.1: Interview guide for Study 1.

ID	Question
1	Tell us about the organization and its divisions.
2	Describe the project, its duration, its size, its technologies, its goals and your role on it.
3	Talk more about the parts of the project that use microservices or another service-oriented architecture.
4	What challenges regarding the architecture have you faced recently? What were their causes and impacts? Did you manage to avoid any of them? How?
5	Are you migrating from an old solution? What challenges did you face during the migration? What were the costs of the migration? What were the costs of not migrating?
6	Did you have challenges regarding the communication among services? Did you have business logic outside the services in their communication? How did you manage to handle it?
7	Did you use any standard for the APIs/message format? Did you have any issues due to your choice of using/not using such standards? How did you manage to solve it?
8	How did you manage your source code and documentation?
9	Did you have any issues regarding third-party licenses? What were the costs and how did you manage to solve it?
10	Did you have any issues regarding shared libraries? What were the costs and how did you manage to solve it?
11	Did you have any issues regarding data storing? What were the costs and how did you manage to solve it?
12	Could you mention other situations with issues that (including why and how you managed to solve it): • reduce development speed? • cause more bugs? • have a negative impact on other system qualities? • impact many developers? • will become worse in the future?
13	Do you have any additional issues we did not covered before?