

Universität Stuttgart

# Architectural Refactoring to Microservices: A Quality-Driven Methodology for Modernizing Monolithic Applications

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung


Vorgelegt von

Jonas Fritzsch

geboren in Erlabrunn, Deutschland


| | |
|---|---|
| **Hauptberichter:** | Prof. Dr. Stefan Wagner |
| **Mitberichter:** | Prof. Dr. Alfred Zimmermann |
| | Prof. Dr. Wilhelm Hasselbring |
| **Tag der mündlichen Prüfung:** | Dienstag, 24.09.2024 |


Institut für Software Engineering



2024

# Zusammenfassung

**Kontext und Problemstellung:** Der Microservices-Architekturstil als De-facto-Standard für Cloud-basierte Anwendungen hat die Art und Weise revolutioniert, in der moderne Softwaresysteme entwickelt und betrieben werden. Das Konzept kleiner, gekapselter und autonomer Services ermöglicht einen agilen Entwicklungsprozess, der durch einen hohen Automatisierungsgrad und unabhängige Teams gekennzeichnet ist. Die gewonnene Flexibilität zeigt sich in kurzen Release-Zyklen und wird oftmals mit einer verbesserten Produktqualität assoziiert. Bestandssysteme sind jedoch oft als Monolithen konzipiert. Neben langen Release-Zyklen und problematischer Wartbarkeit ist der Architekturstil nicht geeignet, die Vorteile von Cloud-Umgebungen auszuschöpfen. Der Einsatz von Microservices erfordert jedoch ein Architektur-Refactoring, was mitunter die Neuentwicklung von Teilen oder sogar der gesamten Anwendung zur Folge hat.

Microservices sind nicht von vornherein für jedes System geeignet. Der teils enorme Aufwand einer Modernisierung erfordert daher eine systematische Entscheidungsfindung, detaillierte Planung und bewährte Verfahren. Aus technischer Sicht erfordert die Architekturtransformation hin zu Microservices die Zerlegung des Systems in einzelne, in sich abgeschlossene Services. Eine zielgerichtete, qualitätsorientierte und teilautomatisierte Dekomposition ist jedoch oft problematisch und mit hohem Aufwand verbunden.

Architektinnen und Architekten von Anwendungssystemen suchen daher nach ressourceneffizienten Wegen, um Orientierung und prognostizierbare Ergebnisse in diesem höchst individuellen Prozess zu erlangen. Die akademische Forschung hat in letzten Jahren ein umfangreiches Wissen über die Erstellung performanter, zuverlässiger, wartbarer und sicherer Microservices-basierter Systeme hervorgebracht. Dieses Wissen wird durch zahlreiche Migrations- und Refactoring-Verfahren für bestehende Systeme ergänzt. Es zeigt sich jedoch, dass diese vielfältigen Ressourcen in der Praxis wenig genutzt werden oder teilweise unbekannt sind.

**Zielsetzung:** Wir streben eine ganzheitliche Methodik zur systematischen und qualitätsgesteuerten Modernisierung bestehender Softwaresysteme hin zu Microservices an. Dabei suchen wir nach Lösungen für die gezielte und automatisierte Dekomposition in Services, eine besondere Herausforderung im Architektur-Refactoring. Um sicherzustellen, dass die resultierende Architektur vordefinierte Qualitätsziele erfüllt, soll deren qualitätsorientiertes Design durch etablierte Entwurfsmuster und bewährte Praktiken unterstützt werden. Darüber hinaus zielt unsere Arbeit darauf ab, Industrie-relevante Methoden bereitzustellen, um den Wissenstransfer zwischen Wissenschaft und Praxis zu fördern.

**Methoden:** In einem übergreifenden Design-Science-Forschungsprozess entwickeln wir eine Migrationsmethodik, ergänzt durch eine Implementierung als webbasierte Anwendung. Für die Analyse und Evaluation setzen wir etablierte Methoden des empirischen Software Engineerings ein, wie z.B. Fallstudienforschung, Umfragen und Experteninterviews. Unsere Sekundärforschung zur Zusammenfassung des aktuellen Stands der Wissenschaft stützt sich auf konsekutive Literaturrecherchen und Rapid Reviews – einer leichtgewichtigen Methode, die sich von systematischen Literaturrecherchen ableitet.

**Forschungsbeiträge:** Auf Basis zweier Interview-Studien konnten wir Erkenntnisse über die Intentionen, Strategien und Herausforderungen im Migrationsprozess gewinnen. In unserem ersten Beitrag sprachen wir mit 16 Software-Experten aus 10 deutschen Unternehmen über insgesamt 14 Migra-

tionsszenarien. Die anschließende Fallstudie im Bereich Cyber-Physical Systems (CPS) erbrachte in Interviews mit 9 Branchenexperten und -expertinnen zusätzliche Anforderungen hinsichtlich Tool-Unterstützung und Automatisierung. In vier Iterationen haben wir 110 wissenschaftliche Publikationen zum Architektur-Refactoring und Migrationen im Microservices-Kontext aufgearbeitet. Aus dem entwickelten Klassifizierungsschema wurde eine Taxonomie abgeleitet, die eine schnelle Einordnung neuer Lösungsansätze ermöglicht. Zur Anleitung und Unterstützung des Migrationsprozesses haben wir ein Framework mit dediziertem Qualitätsmodell konzipiert, das einen qualitätsgetriebenen Migrationsprozess abbildet. Auf der Basis aktueller Technologien und eines modernen Benutzeroberflächen-Designs wurde unser Konzept als webbasierte Anwendung in einem agilen Entwicklungsprozess realisiert. In einer mehrstufigen Evaluation haben wir untersucht, inwieweit unsere Methodik Orientierung und Unterstützung für die Praxis bieten kann. Dazu führten wir drei Befragungsstudien und eine Interviewstudie mit insgesamt 26 Teilnehmern durch, ergänzt durch zwei Längsschnittstudien im industriellen Kontext.

**Fazit:** Wir präsentieren eine ganzheitliche Methodik für die Modernisierung von monolithischen Systemen hin zu Microservices. Diese umfasst ein Framework mit dediziertem Qualitätsmodell und implementiert einen qualitätsgetriebenen Migrationsprozess. Unsere Beiträge unterstützen zunächst die systematische Entscheidungsfindung bei der Einführung von Microservices. Im Anschluss führt unsere Methodik durch den Transformationsprozess mit Fokus auf das Architektur-Refactoring. Die durchgeführten Evaluationen ergaben ein insgesamt positives Ergebnis in Bezug auf Wirksamkeit, Nützlichkeit und Benutzerfreundlichkeit, während beide Fallstudien eine erfolgreiche Anwendung im industriellen Umfeld zeigten. Indem wir wichtige Studienartefakte zur Verfügung stellen, unterstützen wir Forscherinnen und Forscher bei der Entwicklung ähnlicher Industrie-fokussierter Methoden. Darüber hinaus betrachten wir unseren Design-Science-Ansatz zur Nutzung akademischer Forschung in der Praxis als übertragbar auf andere Wissenschaftsdisziplinen.

# Abstract

**Context and Problem:** The microservices architectural style has revolutionized the way modern software systems are developed and operated, and has become the *de facto* standard for cloud-based applications. The concept of small, encapsulated, and self-contained services allows for an agile development process with a high degree of automation, independent teams, and freedom of technological choice. Such systems offer high flexibility through reduced time-to-market and are often associated with better overall product quality. However, existing systems are often designed as monoliths, which are associated with inflexible processes, long release cycles, and an architecture incapable of leveraging the advantages of cloud environments. The adoption of microservices would require an architectural refactoring, entailing redevelopments of parts or even the entire application. Yet, microservices are not inherently a good fit for every system, and the enormous effort involved requires systematic decision-making, detailed planning, and proven approaches to such a modernization. Technically, transforming the architecture of a monolithic system into microservices requires decomposing it into a set of self-contained services. Often associated with extensive manual effort, the targeted, quality-oriented, and semi-automated decomposition remains problematic. Software architects look for resource-efficient ways to provide predictable results and guidance in this highly individual process.

Academia has accumulated a significant body of knowledge on how to build high-performance, reliable, maintainable, and secure microservices-based systems, complemented by numerous migration and refactoring approaches to microservices. However, we found that practitioners hardly leverage this diverse spectrum of resources, or are even unaware of it.

**Objective:** To systematically guide software architects and developers in modernizing their software systems, we seek to provide a holistic methodology for systematic and quality-driven migrations towards microservices. As part of it, we search solutions for the targeted and automated decomposition into services, a particular difficulty during an architectural refactoring. To ensure that the resulting architecture meets predefined quality goals, we seek for ways to support a quality-oriented design based on established patterns and best practices. Moreover, our work aims to provide industry-relevant methods that address the gap between academia and practice by facilitating the transfer of knowledge.

**Methods:** In an overarching design science research process, we create a migration methodology that we implement as a web-based application. For analysis and evaluation, we apply established methods in empirical software engineering, such as case study research, surveys, and semi-structured interviews with experts. Our secondary research to summarize the current state of scientific advances relies on consecutive literature searches and rapid reviews, a lightweight method derived from systematic reviews.

**Contributions:** Based on two primary empirical interview studies, we collected evidence on the intentions, strategies, and challenges of migrating monolithic applications to microservices. In our first analysis contribution, we interviewed 16 software professionals from 10 German-based companies, and talked about 14 migration scenarios. The following mixed-method empirical study in the domain of cyber-physical systems yielded additional requirements for tool support and automation from interviews with 9 industry experts. Over four iterations, we reviewed 110 scientific publications on approaches for architectural refactoring and migration to microservices. From the developed classification scheme, we derived a taxonomy that sup-

ports the swift classification of new solution proposals. To guide architects and developers in a migration process, we conceptualized a framework, along with a dedicated quality model that reflects a quality-driven migration process. Based on latest technologies and a modern user interface design, we realized our concept as a web-based application in an agile development process with early involvement of potential users. In a multi-faceted evaluation, we examined its ability to provide actionable guidance for practitioners. To this end, we conducted three surveys and one interview study with a total of 26 participants, complemented by two longitudinal case studies in an industrial context.

**Conclusion:** We propose a holistic methodology for modernizing monolithic applications to microservices, that comprises a framework and dedicated quality model, thereby implementing a quality-driven migration process. Our contributions support architects in making informed decisions about microservices adoption, and furthermore guide them through a systematic transformation process. The evaluations showed an overall positive result in terms of effectiveness, usefulness, and usability, while both case studies demonstrated a successful application in an industrial environment. By sharing important study artifacts, we support researchers developing industry-focused methods, who can profit from our insights and experiences. Moreover, we regard our design science approach to leveraging academic research by practice as transferable to other scientific disciplines.

# Acknowledgements

The research conducted within this thesis would not have been possible without the support and guidance of several people.

First of all, I want to express my gratitude to my PhD advisors Prof. Dr. Stefan Wagner and Prof. Dr. Alfred Zimmermann. I met Alfred during my master's studies, while he encouraged me to join the doctoral program "Services Computing". His breadth of knowledge, good-tempered way, and thoughtful advice provided a strong anchor and guidance throughout my PhD journey. In his foresight, Alfred soon established the contact to Stefan, who became my first supervisor. I want to thank Stefan for his early support and motivation to join his group. His guidance, advice, and goal orientation are the foundation that made this dissertation possible. I appreciate our numerous constructive discussions in an always serene and pleasant atmosphere. Stefan gave me the freedom to also satisfy my curiosity for other topics and explore adjacent research topics, which broadened the repertoire of my scientific skills. I'm especially tankful for his support in joining the "Software Campus" program, an opportunity that allowed me to support my research activities, acquire interdisciplinary skills, and expand my professional network. I would also like to thank Prof. Dr. Wilhelm Hasselbring for his prompt confirmation to examine this dissertation.

Likewise, I want to thank my colleagues in the Empirical Software Engineering Group, who have accompanied my journey over the past years. I appreciate the fruitful discussions, some of which have led to new ideas and evolved into successful collaborations. In particular, I want to thank Dr. Justus Bogner for our joint research studies that were outstandingly well received by the scientific community. As division lead of our research group, his profound expertise and methodological knowledge have always been a great source of inspiration. Eventually, I want to thank my supervised bachelor's and master's students whose excellent works likewise mark substantial contributions. In this context, I would like to give a special mention to Tobias Haller and Daniel Koch, who laid the foundation for our web-based application.

My thanks also go to Dr. Filipe Correia from the University of Porto. Our meeting during the Microservices Conference 2019 was the starting point of a successful collaboration, with several publications over the following years. Special thanks go the team at Siemens AG, Carolin Rubner, Matthias Saft, and Horst Sauer, for their support in the course of our conducted case study. Likewise, I very much appreciate the support from Stefan Melles at DATEV eG throughout our collaboration in the "Software Campus" project. I would also like to thank Dr. Alexander Eisold and his former employer L-mobile, and Frank Intorp at levigo solutions, for their active involvement throughout our evaluation case studies.

Last but not least, my gratitude goes to my family and friends whose support I could always count on. I particularly thank Ganga for her refreshing thoughts while patiently accompanying my writing process with all its ups and downs.

Jonas Fritzsch, May 2024

# Contents


**I. MOTIVATION AND FOUNDATIONS** **19**

**1. Introduction** **21**
- 1.1. Software Modernization . . . . . 21
- 1.2. The Challenge of Moving to Microservices . . . . . 23
- 1.3. The Academia-Industry Gap . . . . . 25
- 1.4. Research Objective . . . . . 26
- 1.5. Research Methodology . . . . . 27
- 1.6. Contributions . . . . . 30
- 1.7. List of Publications . . . . . 33

**2. Background and Related Work** **35**
- 2.1. Software Architecture . . . . . 37
  - 2.1.1. Architectural Styles . . . . . 38
  - 2.1.2. The Monolith . . . . . 40
  - 2.1.3. Microservices . . . . . 41
  - 2.1.4. Migration to Microservices . . . . . 45
- 2.2. Architectural Refactoring . . . . . 48
  - 2.2.1. Refactoring Techniques . . . . . 49

2.2.2. A Classification Scheme for Approaches . . . . . 54
2.3. Software Quality in Microservices Architectures . . . . . 56
2.3.1. Quality Attributes . . . . . 58
2.3.2. Design Properties . . . . . 60
2.3.3. Quality Trade-Offs . . . . . 64
2.4. Quality-Driven Migration to Microservices . . . . . 65
2.4.1. Architecture Assessment and Migration Decision . . 66
2.4.2. Quality-Oriented Decomposition . . . . . 70
2.4.3. Quality-Oriented Design of Microservices . . . . . 70
2.5. Patterns and Best Practices for Microservices . . . . . 72
2.5.1. Mapping Quality Attributes to Patterns . . . . . 74
2.5.2. Quality Trade-Offs for Patterns . . . . . 76
2.5.3. Best Practices . . . . . 77

**II. STATE OF THE ART ANALYSIS 79**

**3. State of Practice: Microservices Migration in Industry 81**
3.1. Context: Intentions, Strategies, and Challenges . . . . . 82
3.2. Method: Semi-structured Interviews . . . . . 83
3.3. Interview Case Descriptions . . . . . 85
3.4. Results and Discussion . . . . . 97
3.4.1. Migration Intentions (RQ1) . . . . . 97
3.4.2. Migration Strategies and Decomposition (RQ2) . . 99
3.4.3. Technical Challenges (RQ3) . . . . . 101
3.4.4. Organizational Challenges (RQ3) . . . . . 102
3.5. Threats to Validity . . . . . 103
3.6. Conclusion . . . . . 104
3.7. Subsequent Work . . . . . 105

**4. State of Practice: A Case Study in the CPS Domain 107**
4.1. Context and Goals: Microservices in the CPS Domain . . . . 108
4.2. Method: Rapid Review and Interview-Based Case Study . . 111
4.2.1. Rapid Review . . . . . 111

4.2.2. Interview-based Case Study . . . 115
4.3. Rapid Review Results . . . 117
4.3.1. Challenges (RQ1) . . . 117
4.3.2. Practices (RQ2) . . . 119
4.4. Interview Results . . . 122
4.4.1. Challenges (RQ1) . . . 124
4.4.2. Practices (RQ2) . . . 127
4.4.3. Rationales (RQ3) . . . 129
4.4.4. Requirements for Tool Support (RQ4) . . . 130
4.5. Discussion . . . 134
4.5.1. Comparing CPS with Enterprise Applications . . . 134
4.5.2. Comparing Rapid Review and Interview Results . . . 138
4.5.3. Migration Guidance and Automation . . . 140
4.5.4. Main Takeaways . . . 141
4.6. Threats to Validity . . . 144
4.7. Conclusion . . . 146

**5. State of Research: A Classification of Refactoring Approaches 147**
5.1. Context: Refactoring Approaches . . . 148
5.2. Method: Literature Review . . . 150
5.2.1. Search Databases and Keywords . . . 150
5.2.2. Inclusion/Exclusion and Data Extraction . . . 151
5.2.3. Review Iterations . . . 152
5.3. Results . . . 153
5.3.1. Required Input . . . 155
5.3.2. Generated Output . . . 160
5.3.3. Process . . . 161
5.3.4. Usability . . . 167
5.3.5. Quality Goals . . . 171
5.4. A Taxonomy for Refactoring Approaches . . . 175
5.5. Limitations . . . 177
5.6. Conclusion . . . 178

**6. State of Research: Automated Refactoring to Microservices 179**
6.1. Context and Goal: Automating the Decomposition . . . . . 180
6.2. Method: Rapid Review . . . . . 181
6.3. Results . . . . . 183
6.3.1. Mono2Micro (socialsoftware) . . . . . 183
6.3.2. Service Cutter . . . . . 184
6.3.3. Mono2Micro (IBM) . . . . . 185
6.3.4. Microservices Identification . . . . . 187
6.3.5. MonoBreaker . . . . . 188
6.4. Limitations . . . . . 188
6.5. Discussion and Conclusion . . . . . 189

**III. MIGRATION METHODOLOGY 191**

**7. Conceptualizing a Microservices Migration Framework 193**
7.1. Context and Goals . . . . . 194
7.2. Method: Design Science Research . . . . . 196
7.3. Related Frameworks and Quality Models . . . . . 197
7.4. Microservices Migration Framework . . . . . 199
7.5. A Quality Model for Migrations . . . . . 203
7.5.1. From Qualities to Refactoring Approaches . . . . . 204
7.5.2. From Qualities to Patterns and Best Practices . . . . 210
7.5.3. Algorithm for Suitability Calculation . . . . . 211
7.6. Discussion and Limitations . . . . . 212

**8. Implementation as Web-based Application 215**
8.1. Context and Goals . . . . . 216
8.2. Method: Agile Development Process . . . . . 218
8.2.1. Requirements Engineering and Prototyping . . . . . 218
8.2.2. Test Strategy . . . . . 219
8.3. Application Model and Prototyping . . . . . 221
8.3.1. Client-Server Application . . . . . 221
8.3.2. Vertical Prototype . . . . . 221

8.4. Extensible Repository for Refactoring Approaches . . . . . 223
8.4.1. Data Model for Refactoring Approaches . . . . . 224
8.4.2. Technology Stack . . . . . 226
8.5. User Interface and Application Front-End . . . . . 229
8.5.1. Technology Stack . . . . . 231
8.5.2. Design and Navigation . . . . . 232
8.5.3. Integration with Back-End . . . . . 240
8.6. Quality-Model Implementation . . . . . 240
8.6.1. Quality Assessment (Phase 1) . . . . . 241
8.6.2. Quality-Based Recommendations (Phases 2 and 3) 243
8.6.3. Configuration Overview . . . . . 244
8.7. Build, Test, and Deployment . . . . . 245
8.7.1. Deployment Architecture . . . . . 246
8.7.2. Containerization with Docker . . . . . 247
8.7.3. Integration Testing . . . . . 249

**IV. EVALUATION AND CONCLUSION 253**

**9. Application Concept and Usability Evaluation 255**
9.1. Survey on Prototype (9 Participants) . . . . . 256
9.1.1. Results . . . . . 257
9.1.2. Discussion . . . . . 261
9.2. Survey on Web-based Application (11 Participants) . . . . . 263
9.2.1. Questionnaire Design . . . . . 264
9.2.2. Results . . . . . 266
9.2.3. Discussion . . . . . 272
9.3. Evidence on Quality-Driven Migration (6 Participants) . . . 275
9.3.1. Survey . . . . . 275
9.3.2. Expert Interviews . . . . . 279
9.3.3. Discussion . . . . . 284
9.4. Limitations . . . . . 287
9.5. Conclusion . . . . . 288

**10. Case Studies in Industry** **291**
10.1. Case Study Research . . . 292
10.2. A Microservices Migration in the Services Domain . . . 293
10.2.1. Methods . . . 294
10.2.2. Results . . . 297
10.2.3. Discussion . . . 312
10.3. Refactoring of a Document Management Solution . . . 313
10.3.1. Methods . . . 314
10.3.2. Results . . . 316
10.3.3. Discussion . . . 330
10.4. Limitations . . . 332
10.5. Conclusion . . . 333
10.6. Outlook . . . 335

**11. Discussion and Conclusion** **337**
11.1. Summary of Contributions . . . 337
11.2. Limitations and Future Research . . . 344
11.3. Final Remarks . . . 346

**Bibliography** **349**

**List of Figures** **389**

**List of Tables** **393**

**A. Patterns and Best Practices** **397**

**B. Refactoring Approaches** **403**

**C. Web Application Requirements** **409**

# Part I.

# MOTIVATION AND FOUNDATIONS

Chapter 1

# Introduction

In the first chapter, we motivate our research by placing the transformation to microservices in the context of general software maintenance and modernization. We highlight the particular challenge of architectural refactoring monolithic systems towards microservices and identify a related gap between academia and industry. Based on the outlined deficiencies, we derive our research objective, which we subsequently refine into four research questions. The research methods applied to answer these questions are presented afterwards, along with an overview of the main contributions in this thesis. A list of publications contributed to the scientific community during this research concludes the chapter.

## 1.1. Software Modernization

In contrast to consumer products that we are used to in our day-to-day life, a software product, once created, is never *done*, but subject to ongoing development. Its lifecycle entails permanent changes that make it more useful and mature over time, until its decommissioning [Sne12]. The notion of software evolution has been established to holistically describe this process,

which starts with a software's initial operation [SS13]. Software evolution, according to Sneed and Seidl, entails activities to correct, change, expand, optimize, renovate, document, and integrate software. While evolution implies that the product stays the same at its essence, this may not be true for more profound changes like transformations to new environments, or introducing new programming languages [SS13]. Changing a software system's architecture likewise involves more comprehensive changes, while we distinguish it from typical maintenance efforts that the notion of software evolution encompasses. However, boundaries between denotations are not always distinct, as such transformation processes can also be performed in small increments, thereby reflecting a continuous evolution process.

Estimates suggest that around 70% of software developers are concerned with maintenance of existing systems [SS13]. The authors state that existing systems are less often decommissioned than new systems are introduced, leading to an ever-growing landscape of systems. Their maintenance, transformation to new platforms, or migration to a different architectural style require specific technical expertise, complemented by organizational and economical considerations. While common computer science curricula mainly focus on developing new systems [SS13], the dominance of maintenance-related demands by industry is proportionally underrepresented in education of software professionals. Academia has made significant advances to address this issue and proposed measures for the ability of a software system to evolve. The associated quality attribute of *evolvability* [Bog20] is regarded as particularly promoted by the service-oriented computing paradigm [BVS13] and its realizing service-oriented architecture. As the predecessor and overarching principle of microservices, it inherently supports the ongoing change of a software system [Sne12]. Complemented by cloud technologies, contemporary software systems thereby represent ecosystems that can constantly evolve and quickly react to changing requirements.

However, most existing systems are designed and implemented in a conventional monolithic style. While such systems can be partly transitioned to cloud platforms, their underlying architecture is often incapable of leveraging the advantages of such environments. A recent magazine survey among 339

German-based IT decision makers revealed that many companies struggle with monolithic legacy systems, while they attest themselves a modernization backlog [CIO22; Mau22]. The results show that 59% of the companies, particularly smaller ones, spend additional efforts on the maintenance and operation of legacy systems, which are usually based on a monolithic architecture [Mau22]. Monolithic systems that have grown over years or decades can become complex and even *fossilize* in later stages, making them practically unmaintainable [SS13]. The problem would be intensified by missing personnel that is familiar with the used legacy technologies. While such systems hardly meet today's requirements, they increasingly become security risks, too. On average, the surveyed companies grade their own IT infrastructure only "satisfactory", with flexibility and adaptability revealing the most deficiencies. The article summarizes that most companies are aware of the situation, and thus regard the modernization of their systems as important.

To fully leverage the advantages of microservices and cloud environments, monolithic applications require an architectural refactoring, that can entail redevelopments of parts or even the entire system. The sometimes enormous efforts of such a transformation involve significant amounts of time and human resources. It stands to reason that systematic decision-making, detailed planning, and proven approaches are in high demand, to avoid costly misinvestments.

## 1.2. The Challenge of Moving to Microservices

The microservices architectural style has been established to date as *de facto* standard for modern cloud-based software systems. The overarching advances towards cloud-native computing massively promote this architectural style, facilitated by technological enablers like containerization. Linthicum and David describe the underlying principle in a bigger context: *"This is service-oriented computing, at its essence, decomposing the application down to the functional primitive, and building it as sets of services that can be leveraged by other applications, or the application itself"* [Lin16]. The concept of

autonomous services allows for their independent development and deployment with a high degree of automation [JPM+18], embedded in a DevOps culture that combines development and operations. It thereby facilitates an agile development process that allows rapidly reducing time to market for new features, while several other product qualities like maintainability or performance likewise profit.

While the advantages of this architectural style seem beneficial in every respect, literature often emphasizes that "*microservices are no free lunch or silver bullet*" [New15]. Adopting microservices entails various challenges related to their nature of a distributed system, like transactions and data consistency [New21], complemented by organizational and technology-related challenges [BNK18; FBWZ19]. As a prominent example, the *Istio* project recently reported on their migration back to a monolith, acknowledging that microservices are not always the best fit [MBMR21]. It underpins the need for a thorough analysis to reason a migration decision first, and identify appropriate strategies and refactoring techniques in the following. The question of an appropriate strategy distinguishes between the development of a new application, and the redevelopment of the existing system. While building a new system from scratch is often regarded too expensive, solutions for resource-efficient refactoring of existing systems are in high demand. Strategies range from a slow-paced gradual adaption in a continuous evolution process, to a more rapid redevelopment using current technologies, while arbitrary mixtures of strategies are thinkable.

Technically, transforming a monolithic system's architecture would require decomposing the monolithic structure into a set of self-contained services. Such an architectural refactoring is considered a major reconstruction that can span several months to years for enterprise-scale systems, and typically goes along with organizational and process-related changes [KH19; KMM18]. Such a challenging endeavor needs to be well-considered [BNK18], while software architects seek for resource-efficient and partly automated modernization approaches that yield predictable outcomes. A variety of design guidelines, pattern languages, and best practices are available for microservices *greenfield* developments [New21; VCG+22], but the process

of migrating an existing system is not easily generalizable. The associated architectural refactoring is a highly individual process and heavily dependent on the system in place. While code-level refactoring can be automated to a great extent [AMNO21; MPB09], adapting a system's architecture is a much more individual process, and the approaches proposed so far do not fit for arbitrary systems [FBZW19].

Early adopters of microservices had to deal with manifold challenges, such as a lack of guidance and best practices, immature tooling, or building up technology-related expertise [FBWZ19]. While pioneers like Amazon, Netflix, Spotify, and even the German retailer Otto published on their journeys of microservices adoption, such exemplary cases do not necessarily qualify as a blueprint. The average enterprise system has often more individual and sophisticated requirements to meet than, e.g., the well-studied retail domain. Moreover, IT budgets are often tight, and strict compliance and quality requirements do not leave much room for experimentation. Especially small and medium enterprises suffer from resource limitations that render the adoption of microservices a particular challenge [ADY19]. While practitioners often find reasonable solutions with extensive manual efforts, a targeted, predictable, and quality-controlled migration, supported by semi-automated decomposition approaches, continues to be problematic.

## 1.3. The Academia-Industry Gap

Over the last few years, scientific research has yielded various methods and approaches for migrations and architectural refactoring to microservices [ASM+21; BBGG21; FBZW19; KBAR21; PMA19; SKM20]. Existing studies also cover a variety of related topics, starting from decision-making [HWB17] over process strategies [BBGG21], to quality assurance [COU19], and organizational aspects [BNK18]. It reflects the great demand for solutions to structure, systematize, and partly automate such transformations. However, their applicability is heavily dependent on several aspects of the system

at hand, e.g., the availability and preparation of input artifacts, targeted quality attributes, or the maturity of accompanying tools.

While academia offers a diverse spectrum of guidance, we found that practitioners hardly leverage these resources [FBH+22; FBWZ19], even specialized consultancy companies rarely consider such knowledge. There may be several aspects to this barrier, e.g., reservations about scientific publications, access limitations to related databases, or concerns regarding the practical applicability and maturity of solutions proposed by academia [FBH+22]. Jamshidi et al. confirm that publications have had "*little if any impact on microservice practice*" [JPM+18], which the authors attribute to potentially limited involvement of industry-scale applications. While a strong interest in migrating legacy systems is widely confirmed [DLM19], academia still struggles in providing answers relevant for industry, such as benefits and drawbacks of microservices in a practical context [ZLC+23]. Hence, a key motivation of our work is bridging the gap between academia and industry by developing methods that focus on practical applicability and facilitate the knowledge transfer into practice.

## 1.4. Research Objective

Based on the outlined problem statement, we frame our research project by formulating a research objective according to the guidelines for design science research by Wieringa [Wie14]:

> *Design, implement, and evaluate*
> ***A practically applicable methodology***
> ***For transforming monolithic applications into microservices***
> *From the viewpoint of software architects and developers*
> *In the context of enterprise software development*

This overarching objective is further refined into four specific research questions that express knowledge goals (RQ1-2), and design goals (RQ3-4).

**RQ1:** What are intentions, strategies, and challenges in the context of migrating monolithic systems to microservices?

**RQ2:** What architectural refactoring techniques are applicable for the decomposition into a set of microservices?

**RQ3:** How can predefined quality goals steer the migration process?

**RQ4:** How can a practically applicable migration methodology guide architects?

While these research questions set the scope for this thesis in its entirety, we formulate more find-grained questions for solving partial problems in the following chapters. Before, we introduce the applied research methodology for this thesis and its individual contributions.

## 1.5. Research Methodology

The methodology followed throughout this thesis relies on established research methods in empirical software engineering, complemented by non-empirical methods that focus on the design of new artifacts. Empirical software engineering applies proven methods from social sciences to study and evaluate software engineering phenomena. Several concepts originate from evidence-based medicine, leading to the synonymously used notion of evidence-based software engineering [KDJ04]. But also various other fields like anthropology, psychology, or industrial engineering shaped this engineering discipline, that experienced a rapid growth during the last two decades [WA11]. The adoption of industry-relevant methods aims for designing and building high-quality software solutions [DKJ05], thereby bridging the gap between academia and practice [SDJ07]. Besides, empirical software engineering acknowledges the strong influence of human factors when constructing complex software systems [ESSD08; WA15].

Easterbrook et al. reviewed commonly used empirical methods, along with theoretical stances behind, and practical considerations for their application [ESSD08]. According to the authors, the key element of an empirical

research design is a clear research question. A philosophical stance will help to reflect the goal and further guide the selection of research methods. The chosen methods then determine the study design and data collection methods. Eventually, formulating a theory can help in data interpretation and subsequent answering the initially stated research question. Wohlin and Aurum similarly emphasize the importance of understanding the research design, process, and possible implications [WA15]. They confirm a broad spectrum of empirical methods for knowledge acquisition, and the inherent challenges of their appropriate selection. To guide researchers, they provide a decision-making structure for research designs in empirical software engineering. For the creation of our research design, we consulted both mentioned guidelines.

While we employ several complementing research methods throughout this thesis, its structure reflects the design science research process [HMPR04; PTRC07; Wie09]. According to Hevner et al., design science "*seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts*" [HMPR04] to understand the problem domain and its solution. Peffers et al. similarly emphasize to build upon prior knowledge and extending it through artifact design, development, and evaluation [PTRC07]. Such artifacts can be constructs, models, frameworks, methods, or instantiations, emphasizing its explorative nature [HMPR04; WA15]. Wohlin and Aurum add that such an artifact aims to address a specific problem, in which it needs to be effective and rigorously evaluated [WA15].

The nature of human knowledge is a contrarily discussed philosophical question [ESSD08], in particular for design science research [NB06]. Related epistemological questions center around ways in which scientific knowledge can be acquired, how suitable research methods can be selected, and how their soundness and rigor can be judged [Gre06]. Hevner et al. describe the design-science paradigm as the question of "*what is effective*" [HMPR04], and add that "*both design-science and behavioral-science paradigms are needed to ensure the relevance and effectiveness*" [HMPR04]. Schön argues that design science research does not particularly locate itself in a positivist domain, but is open to interpretivism as well [Sch91], while Hevner even constitutes

a pragmatic nature of design science [Hev07] that "*adopts an engineering approach to research*" [ESSD08].

Case study research represents one of our essential methods, commonly used in software engineering research to investigate "*contemporary phenomena in its natural context*" [RH08]. By employing multiple data collection methods [WA15], case studies "*offer in-depth understanding of how and why certain phenomena occur*" [ESSD08]. Case study research supports our initial investigations in an explorative manner, and our evaluation activities in a confirmatory manner. As such, Sjøberg et al. emphasize their particular importance for industrial evaluations of methods and tools [SDJ07].

Survey research constitutes another pillar of our research methodology, both as questionnaires and expert interviews [ESSD08]. In the form of technology evaluation surveys, we use self-administered questionnaires [KP08] to support our evaluation activities. Interviews, however, are a means to elicit a more "*vivid picture of a participant's perspective on the research topic*" [WA15]. While fully structured interviews can be compared to questionnaire-based surveys [RH08], semi-structured interviews allow for capturing also unexpected types of information [HA05; Sea08]. Such additional information in the form of rationals or intentions greatly facilitates the analysis of industry practice, but also allows for a targeted artifact evaluation.

Next to the mentioned primary research methods, secondary research is essential to summarize the current state of research and draw conclusions [SF08], thereby setting the base for meaningful contributions to the scientific community. Systematic literature reviews, adopted from the medical domain, represent a cornerstone of evidence-based software engineering [BKB+07; KDJ04]. They are "*a means of evaluating and interpreting all available research relevant to a particular research question, topic area, or phenomenon of interest*" [Kit04]. Its "*highly procedural approach*" [SF08] aims to reduce selection bias and promote replicability. However, due to the partly immense effort involved, more lightweight approaches like rapid reviews increasingly gained popularity [CPS20]. In our thesis, literature reviews build the foundation for analyzing and synthesizing the current state of research, and help in identifying open questions and research gaps [SDJ07]. To op-

timally reconcile efforts and results, we rely on rapid reviews, or conduct several, partly overlapping, review iterations over a certain timeframe.

## 1.6. Contributions

Within our thesis, the before outlined and complementing research methods integrate in an overarching design science research process [HMPR04; PTRC07; Wie09]. Peffers et al. suggest a process model that comprises its six stages: problem identification, definition of objectives, design and development, demonstration, evaluation, and communication. Fig. 1.1 illustrates our research design, that manifests in the structure of this thesis. It symbolizes the outlined process by employing the metaphor of a pyramid. Each building block represents a contribution that utilizes an individual research design. In the following, we briefly introduce the contributions, which are tagged with their chapter number in this thesis.

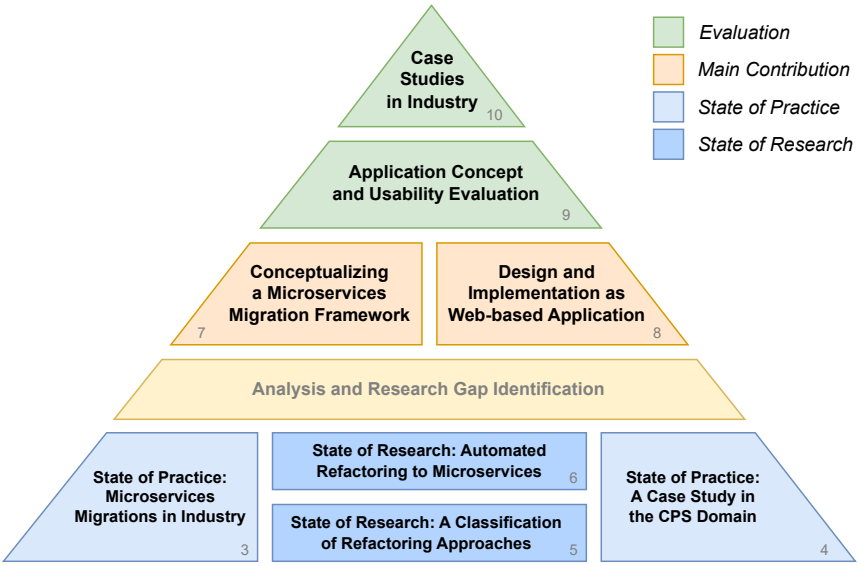



**Figure 1.1.:** Thesis Structure — Contributions and Methods

**State of Practice: Microservices Migration in Industry (3)** In this first contribution to the state of practice, we conducted a qualitative interview study on *intentions*, *strategies*, and *challenges* in the context of migrations to microservices. We investigated the transformation process of 14 systems across different domains and sizes by conducting 16 in-depth interviews with software professionals from 10 German-based companies. While presenting a separate discussion of each case, we take an inventory of the situation, industry is confronted with.

**State of Practice: A Case Study in the CPS Domain (4)** In this second contribution to the state of practice, we performed a mixed-method empirical study that aims at more thoroughly evaluating the state of microservices adoptions. To this end, we conducted a case study in the domain of cyber-physical systems (CPS), during an industry-academia collaboration. A rapid review based on 146 scientific papers on microservices and DevOps in the CPS domain is subsequently validated in an interview-based case study with 9 CPS professionals in different business units at Siemens AG. While the interviews reveal current challenges and applied practices, they also yield requirements for systematizing and automating the migrations process.

**State of Research: A Classification of Refactoring Approaches (5)** In our third analysis contribution, we examine the state of research on architectural refactoring to microservices. Motivated by the reflections from industry, we sought answers and solutions offered by academia. Throughout four iterations, we reviewed 110 academia-proposed refactoring and migration approaches while classifying them based on several criteria, including quality aspects. Besides, we propose a taxonomy for such approaches that acts as a basis for communication among researchers and practitioners, and allows for swiftly classifying new solution proposals.

**State of Research: Automated Refactoring to Microservices (6)** In our fourth contribution, we investigate the advances made by academia in automating a refactoring to microservices. We inspect the previously identified approaches regarding their techniques and tool support, while focusing on practical applicability. From 91 publications, we extracted 22 tools, seven of which address the decomposition into services. For five of them, we generated service cuts using reference applications. To assess the tools from an end-user perspective, we report our experiences on installation, documentation, usability, and support.

**Conceptualizing a Microservices Migration Framework (7)** Our state-of-the-art analysis revealed prevalent challenges in the context of migrating monolithic systems towards microservices. Particular difficulties in the associated architectural refactoring center around a targeted and automated decomposition. Moreover, we discerned a distinct gap between industry and academia when it comes to leveraging research findings in industry practice. To overcome this barrier and provide more systematic guidance, we conceptualize a framework for migrations for microservices. Complemented by a dedicated quality model, we provide a holistic methodology for quality-driven migrations that focuses on practical applicability.

**Implementation as Web-based Application (8)** To guide software architects and developers through a migration scenario, we developed a web-based application that implements our methodology. We adhered to established software engineering standards, including requirements engineering, model design, implementation, and testing. Starting with a vertical prototype, we involved potential users early on in an agile development process through continuous evaluations. The implemented framework and quality model provide the basis for a customizable workflow throughout a transformation, following the concept of a quality-driven migration.

**Application Concept and Usability Evaluation (9)** We report on the evaluation activities throughout the iterative development process of our web-based application. As part of the design science process, evaluations seek to collect empirical evidence on the practicability, soundness, and usability of our methodology and its implementation. We conducted a survey with 9 participants to evaluate the concept and vertical prototype, followed a by a survey with 11 participants targeting the concept and usability aspects. The quality-driven migration process was evaluated in a second and third iteration by a mix of surveys and interviews among six practitioners.

**Case Studies in Industry (10)** We report on two longitudinal case studies that evaluated our framework and web-based application in an industrial context. The first case study on the migration of monolithic application in the *Services* domain was conducted between February and August 2023. Our methodology was systematically applied for developing a migration concept that was subsequently implemented in a proof-of-concept. The second case study aims to evaluate the suitability of our methodology for improving an existing microservices-based architecture of a

document management solution, and was conducted between September 2023 and March 2024. Throughout both studies, we created structured field notes to collect in-depth qualitative data on its applicability and usability.

## 1.7. List of Publications

In the pursuit of conducting the research for this thesis, the following five essential publications have been contributed to the scientific community:

1. J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner. “From Monolith to Microservices: A Classification of Refactoring Approaches”. In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. DEVOPS 2018*. Ed. by J.-M. Bruel, M. Mazzara, and B. Meyer. Cham: Springer International Publishing, 2019, pp. 128–141 [FBZW19]
2. J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. “Microservices Migration in Industry: Intentions, Strategies, and Challenges”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2019, pp. 481–490 [FBWZ19]
3. J. Fritzsch, J. Bogner, M. Haug, A. C. Franco da Silva, C. Rubner, M. Saft, H. Sauer, and S. Wagner. “Adopting microservices and DevOps in the cyber-physical systems domain: A rapid review and case study”. In: *Software: Practice and Experience* (Nov. 2022), pp. 1–20 [FBH+22]
4. J. Fritzsch, F. Correia, J. Bogner, and S. Wagner. “Tools for Refactoring to Microservices: A Preliminary Usability Report”. In: *5th International Conference on Microservices (Microservices 2023), Pisa, Italy*. 2023 [FCBW23]
5. J. Fritzsch, J. Bogner, M. Haug, S. Wagner, and A. Zimmermann. “Towards an Architecture-Centric Methodology for Migrating to Microservices”. In: *Agile Processes in Software Engineering and Extreme Programming – Workshops*. Ed. by P. Kruchten and P. Gregory. Cham: Springer Nature Switzerland, 2024, pp. 39–47 [FBH+24]

Supplementary to this thesis, major contributions to the following publications in the research field have been made, recognized by my co-authorship:

6. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. “Limiting technical debt with maintainability assurance”. In: *Proceedings of the 2018 International Conference on Technical Debt - TechDebt ’18*. New York, New York, USA: ACM Press, 2018, pp. 125–133 [BFWZ18]

7. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. “Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2019, pp. 187–195 [BFWZ19b]

8. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. “Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 546–556 [BFWZ19a]

9. T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo. “Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis”. In: *Lecture Notes in Computer Science*. Vol. 12292 LNCS. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 315–332 [MCF+20]

10. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. “Industry practices and challenges for the evolvability assurance of microservices”. In: *Empirical Software Engineering* 26.5 (Sept. 2021), p. 104 [BFWZ21]

11. G. Vale, F. F. Correia, E. M. Guerra, T. de Oliveira Rosa, J. Fritzsch, and J. Bogner. “Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs”. In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, Mar. 2022, pp. 69–79 [VCG+22]

12. R. Lichtenthäler, J. Fritzsch, and G. Wirtz. “Cloud-Native Architectural Characteristics and their Impacts on Software Quality: A Validation Survey”. In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2023, pp. 9–18 [LFW23]

Chapter 2

# Background and Related Work

In this chapter, we provide the technical background to the contributions of this thesis. As well, we discuss related studies that represent foundations for our work or have similar research goals. Both explications convey fundamental concepts on the current state of research and facilitate the comprehension of our following contributions.

As the central theme of this thesis is bound to software architecture, we start with a brief introduction on software architectural styles. We then delve into the two opposing architectural styles, monolith and microservices, which will be examined from different perspectives throughout this thesis. By discussing their basic concepts, we point out key characteristics and their influence on software product quality. Throughout this thesis, we investigate both architectural styles in the context of software maintenance and evolution, more precisely, the transformation of monolithic architectures towards microservices. We discuss common strategies for such a migration process whilst considering primarily technical but also business-related boundary

conditions. The inherent architecture-related activities in such a transformation are referred to as architectural refactoring. We introduce this concept by discussing different techniques and tactics for its realization. Such aspects represent core features of concrete refactoring approaches. Academia has proposed a variety of such refactoring approaches for migrating monolithic systems towards microservices, whose classification we will briefly touch on.

Upon covering the technical aspects of an architectural refactoring, we widen our perspective and examine the migration process to microservices more holistically. To conduct an architectural refactoring in a targeted manner, a precedent identification of migration intentions and goals is essential. This need arises from the typically long-lasting and resource-intensive migration process that involves challenges also beyond technical aspects. The primary goal of a refactoring activity is the improvement of a software system's non-functional requirements, or in other words, software quality. Both architectural styles, monolith and microservices, promote software quality aspects in different ways. Hence, an improvement of specific quality aspects is often the main-driver for a migration. As a preparatory work, we investigate such qualities for microservices architectures based on the ISO/IEC 25010:2011 Standard [ISO11a] and other related publications. The thereby established foundations allow for conceptualizing a quality model for microservices migrations at a later stage.

As a second groundwork for our main contribution, we introduce the concept of a quality-driven migration to microservices. To this end, we identify the three major phases of a migration. The initial phase aims at eliciting quality goals during an architecture evaluation activity. These goals set the direction for the second phase, in which a strategy for organizing the development and conducting the architectural refactoring is defined. Eventually, established patterns and best practices ensure a quality-oriented design of the emerging microservices architecture in the third phase.

Throughout this chapter, we discuss related studies that pursue similar research goals, or provide foundations that we build on.

## 2.1. Software Architecture

This thesis is anchored in software engineering, a branch of computer science that applies "*a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software*" [IEE90]. The discipline of software architecture as a sub-area of software engineering emerged in the last two decades of the twentieth century. Perry and Wolf's publication on its foundations marked a first important reference for academia in 1992 [PW92]. At that time, software development faced the challenge of growing complexity through the increasing size of software systems. As a consequence, the underlying structure of a system became more important for satisfying various functional and non-functional requirements. Moaven et al. emphasize the importance of software architecture hereto, by stating that "*due to the enlargement and complexity of software systems and the need for maintenance and update, success of systems depends strongly on their architecture*" [MHAK08].

The term software architecture reflects an analogy to the architecture of buildings. While the comparison of brick work and source code seems superficial, the analogy still conveys key concepts as detailed in [Tay18]. Taylor et al. define a software system's architecture as "*the set of principal design decisions made about the system*" [TMD09], whereby *design decisions* would encompass "*every aspect of the system under development, including structure, functional behavior, nonfunctional properties, user interaction, and decisions related to the system's implementation and deployment*" [Tay18].

Bass et al. emphasize the existence of structures and define software architecture as "*set of structures needed to reason about a system. These structures comprise software elements, relations among them, and properties of both*" [BCK21]. The authors put their perspective on software architecture in contrast to other common definitions that emphasize the early decision-making in a development process: In the context of agile development practices, these structures would not predominantly reflect architectural design decisions that may manifest only in later development stages. This abstract notion of *structures* needs a closer inspection — Bass et al. distin-

guish three categories: *Module structures* represent the code and data units, while *Component-and-connector structures* reflect a system's runtime behavior and the interaction between components. Lastly, *allocation structures* map the elements of the first two structures to non-software elements, such as hardware or humans. This definition implies that every software system represents an architecture, independent of its explicit description or specification. It is apparent, however, that the targeted addressing of specific requirements benefits from a well-documented architecture design.

Before embarking into a discussion of architectural styles, we distinguish software architecture from the two adjacent disciplines, but sometimes synonymously used terms *system architecture* and *enterprise architecture*. According to Bass et al., a system architecture is oriented towards "*the totality of hardware, software, and humans*" [BCK21]. While, for instance, a software architecture would allow to reason about qualities like performance and reliability of a system, the system architecture's perspective would involve additional aspects like power consumption, physical dimensions, or human interaction. Enterprise architecture, as the second adjacent discipline, describes "*the structure and behavior of an organization's processes, information flow, personnel, and organizational subunits*" [BCK21]. Hence, it focuses on the synergy between software systems and their role in achieving an enterprise's business goals, e.g., in the process of a digital transformation [ZSS+18]. While the two disciplines have many commonalities with software architecture, both a *system* and an *enterprise* represent environments for accommodating software, with all its interdependencies [BCK21].

### 2.1.1. Architectural Styles

An architectural style in software development comprises a set of principles and patterns for a family of systems, thereby describing their macrostructure. The terms architectural pattern and architectural style are sometimes used synonymously, while design patterns usually describe structures on a lower level. Garlan and Shaw state that an architectural style "*determines the vocabulary of components and connectors that can be used in instances of that style,*

*together with a set of constraints on how they can be combined*" [GS94]. Taylor et al. emphasize the decision-making aspect by defining an architectural style as "*a named collection of architectural design decisions*" [TMD09], with Zimmermann adding that architectural styles are "*typically defined via design intent, principles and patterns [...] or via (technical) constraints*" [Zim16a]. The question arises, which aspects contribute to such a decision-making. In the introduction of their "Architecture Tradeoff Analysis Method" (ATAM), Kazman et al. note that it "*is organized around the idea that architectural styles are the main determiners of architectural quality attributes*" [KKC00]. Taylor et al. similarly state that architectural styles "*elicit beneficial qualities in each resulting system*" [TMD09]. In this regard, the choice of an architectural style can lead to a "*multi-criteria decision-making problem*" [MHAK08] which involves different and sometimes mutually exclusive goals and objectives.

Examples of established architectural styles are client-server, layered, pipes and filters, data-centered, object-oriented, event-based, or REST (REpresentational State Transfer) [GS94; TMD09]. In the early 2000s, service-oriented architecture (SOA) [Erl05] emerged as a new architectural style and enabler of the service-oriented computing (SOC) paradigm [Pap03] that experienced its rise at that time. SOA promotes services as fundamental building blocks for developing applications, and is often regarded as a predecessor or generalization of the microservices architectural style. A detailed discussion of SOA principles and foundations can be found in the work of Zimmermann [Zim09], while Bogner [Bog20] elaborates on the concept of service orientation and its evolution towards microservices.

The microservices architectural style became popular around 2015 and brought significant changes to the landscape of software architectural styles. A survey among 50 architects, project managers, and scrum masters in 2020 ascertained a 50% growth of microservices architectures in web-based applications during the preceding 2 to 3 years [MSV21]. To establish a basic understanding of this architectural style and its characteristics, we provide a brief introduction by starting with a reflection on the opposing monolithic concept, before leading over to microservices.

### 2.1.2. The Monolith

The notion of a *monolith* in software architecture does not precisely represent an architectural style, but merely a traditional unified model for software design, referring to its shape or representation. The term symbolizes the combination of all functionality and application logic within a single unit of software [KM19; LF14; SS17; VF23]. A monolith typically runs in a self-contained manner, and is often accommodated by a single process on a dedicated machine. Thus, the monolithic architecture is well suited for applications of limited size, that consist of tightly coupled components which are present at compile time [KM19]. Commonly, such applications are based on few, if not a single programming language and framework [SS17]. This characteristic allows for quick implementations and facilitates other maintainability aspects, such as testability [KMM18; SS17]. Function calls are typically implemented as in-process calls, which requires little operational overhead [KM19]. Deployment and operation of a monolith are ideally independent of external resources. That way, a small-sized monolith can also handle varying loads if complemented by a load balancer for horizontal scaling of the system [BOP22; KMM18; RWW+18].

While this concept has many advantages, it implies technical limitations like the typically low degree of technology heterogeneity. Today's prevalence of software solutions in a multitude of application fields goes with an ever faster emergence of new technologies [FWBW21]. Hence, contemporary software architectures need to account for and support the quick adoption of new languages and frameworks. In this regard, monolithic structures often require major efforts and comprehensive refactoring, leading to the risk of a technology lock-in [SS17; VF23]. Moreover, monolithic applications that have grown over time, often suffer from related maintainability problems, in particular accumulated technical debt [LLST20]. Besides, build, test, and deployment of large units become more resource-intensive and time-consuming, hindering automation and the adoption of DevOps practices [KM19]. A small change to the code base requires to run the entire build pipeline, thereby hampering an agile development process with short release cycles [KMM18;

LF14]. As well, the operation and handling of varying workloads become more challenging, as horizontally scaling large applications can result in resource conflicts and reduced resource efficiency [KM19; KMM18; MDN20; VF23]. Reliability would be significantly impacted through degraded operational stability, as a single application error can potentially disrupt the entire application's functionality [KM19].

The in 2001 published Agile Manifesto[1] brought far-reaching changes to the hitherto static and monolithic process of software development by introducing a dynamic, flexible, and customer-oriented approach that is nowadays considered the norm. While the monolithic architectural style represents the traditional approach, it still has its field of application in the era of cloud computing. Especially for small and lightweight applications, a system built as a monolithic architecture can profit from its simplicity, maintainability [SS17; VF23], and performance [KM19]. However, with increasing size and complexity, the drawbacks start to appear and eventually tend to outweigh the benefits. Hence, the challenge of building contemporary large-scale software solutions can be broken down into minimizing the proportional increase in complexity with a system's size. Related to the architectural style, many of the outlined drawbacks can be mitigated by a more modular design. The microservices architectural style employs this principle based on current technological advances like containerization technologies [KM19; KMM18; SS17].

### 2.1.3. Microservices

According to a web article in 2016, the term *Micro-Web-Services* was first primed by Dr. Peter Rodgers during a cloud computing conference in 2005, and later in 2011 as *Microservices* itself at a conference of software architects [Hab16]. Lewis and Fowler state that this architectural style has been employed in many projects later on, with promising results [LF14]. Hence, the concept gained growing interest in software industry due to its ben-

[1] https://agilemanifesto.org/

efits over monolithic architectures [DBFP19]. Microservices address the various drawbacks of a monolith by dividing the application into smaller units, so-called *services* [KM19]. Small in this regard means that each service has a single responsibility, also denoted as *single responsibility principle* (SRP) according to the definition of Martin [Mar03; New15]. Lewis and Fowler [LF14], complemented by Newman [New15], have substantially shaped a uniform understanding of microservices by postulating commonly accepted characteristics and principles:

- Componentization via Services [LF14]
- Organized around Business Capabilities [LF14; New15]
- Products not Projects [LF14]
- Smart Endpoints and dumb Pipes [LF14]
- Decentralized Governance & Data Management [LF14; New15]
- Infrastructure Automation & DevOps [LF14; New15]
- Design for Failure [LF14; New15]
- Evolutionary Design [LF14]

Many proponents agree with Sayara et al. [STH17], who consider microservices an evolutionary step from the service-oriented architecture (SOA). Zimmermann confirms that "*microservices indeed can be seen as a development- and deployment-level variant of SOA*" [Zim16a]. In his work, he identifies the commonalities and differences between definitions of microservices and SOA in existing literature, which he then distills into "*seven common themes (tenets)*" [Zim16a] for microservices. In the following, we present them with a brief annotation.

**1. Fine-grained interfaces to single-responsibility units** Small and self-contained units encapsulate data and business logic, while exposing their entire functionality through a fine-grained API. It represents the primary way of inter-process communication between services and employs lightweight mechanisms [LF14]. The predominantly used synchronous REST-based communication [SS17; YPS17] is often complemented by asynchronous

event-driven messaging for performant backend communication. The design as single-responsibility units enables vertical scaling [KMM18; RWW+18], i.e., the individual scaling of services, thereby allowing for resource-efficiently handling varying workloads.

**2. Business-driven development practices** Services are built around business capabilities, which ensures a high cohesion of the implemented functionality, and promotes clear and well-defined boundaries at the same time. Techniques to model business contexts, such as domain-driven design (DDD) [Eva04], are recommended for the identification of such bounded contexts.

**3. Cloud-native application design principles** Zimmermann refers to the design principles by Fehling et al. who summarize them as "IDEAL" (*Isolated state*, *Distribution*, *Elasticity*, *Automated management* and *Loose coupling*) [FLR+14; Zim16a]. The loose coupling of services is a main characteristic [KMM18; SS17] and not only facilitates their independent development and deployment. Moreover, it has implications on the organizational structure of development teams, which, according to Conway's Law, needs to consider the structure of a system to build [Con18].

**4. Polyglot programming and persistence** Loosely coupled services provide developers with the flexibility to use the most suitable technology stack for each microservice [KM19; New15; SS17], thereby facilitating a high degree of technology heterogeneity. In this context, polyglot persistence refers to using multiple databases and storage technologies in an application. It implies eventual consistency [Vog09] that Fowler points out as a microservice trade-off [Fow15].

**5. Lightweight containers** Container technologies like Docker[1] and Kubernetes[2] offer a more resource-efficient virtualization mechanism compared

[1] https://www.docker.com/

[2] https://kubernetes.io/

to, e.g., virtual machines. Accompanying tools for container orchestration enable an automated provisioning, deployment, scaling, load balancing, and monitoring. These technological foundations allow for building microservices-based systems that integrate hundreds or even thousands of small and decoupled services. The integration of containers with CI/CD pipelines ensures a high degree of automation throughout development and operation, thereby facilitating the implementation of a DevOps culture [NOLP21].

**6. Decentralized continuous delivery** As services run in their own process and operate their own database, they allow for independent development, delivery, and deployment [LF14]. While microservices promote building vertical teams around services, decentralization would empower the teams to establish their own build and release cycles, instead of adhering to central governance. An advancement towards continuous deployment even obviates the need for comprehensive product releases in favor of small and frequent increments.

**7. DevOps** The concept of DevOps, which evolved around the rise of microservices, entails a set of practices that promote automation throughout development and operation in a streamlined process [Che18]. The independent development of services eases build, test, delivery, and deployment, allowing for more frequent releases and thus reduced time to market for new features [SS17]. Occurring failures likely affect only a part of the application's functionality, promoting fault tolerance and reliability through isolation and easier location of failures [ALFT21; KMM18; THLL18]. Establishing a DevOps culture is considered more than a set of practices, implying that the traditional roles of developers and operators merge with each other.

The composition of small, autonomous services yields several advantages over a monolithic architecture. Microservices are thus particularly suitable for large and complex applications that require frequent updates and maintenance [RWW+18]. However, this architectural style also has drawbacks which are mainly inherent to its distributed system's nature [BBG+17; BOP22; VGC+15]. While microservices aim at reducing the complexity of

a system though separation into smaller parts, these parts need to communicate in a coordinated manner. This communication complexity increases with the number of microservices that an application comprises. Hence, the system's overall complexity as such does not disappear, but shift from inside the process boundaries to the outside. The communication overhead caused by inter-process communication, particularly across the network boundaries, can directly affect a system's performance [JR19] and constitutes a major challenge in designing microservices-based systems [KMM18]. The services' responsibility for their data management implies aligning different data formats and database systems across the application, which can complicate coordination and data exchange [BOP22]. The implementation of transactions spanning multiple services is severely impeded, and calls to trade consistency for eventual consistency [MM20]. Besides, a potentially high technology heterogeneity can lead to ecosystems that accommodate various languages and frameworks, thereby increasing maintenance efforts.

We conclude that the microservices architectural style offers many advantages, but also poses new challenges of technical and organizational nature. Hence, microservices should not per se be considered the most appropriate architectural style for any "modern" application [KM19]. Pros and cons should be carefully weighed and aligned with software product quality goals, strategic goals, and an organization's resources and capabilities.

### 2.1.4. Migration to Microservices

Many companies migrate their monolithic applications to a microservices-based architecture in the context of moving to cloud-based platforms [FST23b]. Microservices are regarded the most suitable architectural style for cloud infrastructures and implementing DevOps practices [FST23a; HSA19; PD20]. To fully leverage the benefits of cloud environments, monolithic systems need to be transformed towards a microservices-based architecture [FST23a; HOAM23]. Hasan et al. confirm the observation by Mallidi et al. [KSS21], who regard "*legacy/monolithic to microservices*" as the most popular type of application modernization in 2021, while the latter see a similar trend

towards open-source and open platforms. Popular cloud providers such as Amazon Web Services[1], Microsoft Azure[2], or Google Cloud Platform[3] promote this architectural style by offering comprehensive ecosystems for their development and operation. They even offer ways to detach application development from any infrastructural concerns, after the *Serverless* computing paradigm [HR19; TEPN20]. In the following, we reflect on migration intentions and the central theme of service partitioning, before discussing possible migration strategies.

**Intentions** The decision to migrate an existing system towards microservices can result from various intentions and reasons. Companies commonly aim for better maintainability, shorter release cycles, scalability, high availability, cloud readiness, or a more agile development process. Henry and Ridene identified scalability and faster deployments as prevalent drivers by analyzing a developer community survey in 2018, counting 732 participants [HR19]. Taibi et al. identified maintainability, scalability, and independent development teams as the three main drivers for a migration among their 21 interviewed practitioners in 2017 [TLP17]. Several of them stated that migrating to microservices would be the only feasible solution to counteract the growing complexity of their systems. Monolithic systems that have grown over years or even decades can become large and complex, and even fossilize in later stages [SS13], making them practically unmaintainable. In software engineering, the phenomenon is denoted as accumulated *technical debt* [LLST20], which can be reduced eventually through a migration to microservices.

**Service Partitioning** When migrating an existing monolithic application to microservices, the most prominent question centers around an appropriate partitioning and the granularity of services. It essentially marks the difference between both architectural styles, as a monolith can be regarded a

[1] https://aws.amazon.com

[2] https://azure.microsoft.com

[3] https://cloud.google.com

microservices application comprising one service. The number and size of services influence the architecture's efficiency to a great extent [THLL18]. Inappropriate service cuts can, e.g., lead to high coupling and thus degrade performance due to excessive inter-service communication [KM19]. The before outlined characteristics (Section 2.1.3) guide the design of ideal microservices to avoid such inadequacies. For instance, the single responsibility principle suggests that a service should be responsible for one business capability [SFP+19], promoting high cohesion within the service and loose coupling with other services. However, the granularity of such business functionalities is not precisely defined, and thus subject to a specific domain, influenced by the architect's interpretation [STH17; THLL18].

**Migration Strategy** A suitable migration strategy depends on various factors like technical condition of the system, targeted quality goals, available resources, or strategic goals of the enterprise. It frames the migration process by structuring the individual activities in an organizational and chronological manner. Mallidi et al. distinguish two phases in a legacy modernization: assessment and implementation [KSS21]. The initial assessment of the legacy application, combined with an organization's goals, would lead to a concrete transformation strategy. In this regard, they distinguish the "7-R's", namely *Retain*, *Retire/rationalize*, *Rehosting*, *Replace*, *Refactor*, *Re-architecture*, and *Rebuild/rewrite/reengineering*. Frank et al. [FST23b] use an equivalent categorization in the context of could migrations, while stating that only *Refactor*, *Revise (Re-architecture)*, and *Rebuild* would alter the existing application. As *Refactor* and *Revise/Re-architecture* can be considered equivalent in the context of an architectural refactoring, we adhere to the basic strategy types that Bajaj et al. [BBGG21] suggest in the context of migrating monolithic systems to microservices:

1. *Re-Factor* — The restructuring is conducted (often incrementally) by altering the existing application and leveraging its code base. It is often combined with the extension by new functionality.

2. *Re-Build* — The application is redesigned using current technologies that suit best for fulfilling its requirements. It can entail a rewrite of parts or the entire application.
3. *New Application* — A new system is developed in a greenfield approach. An existing system serves merely for the derivation of requirements.

The *Re-Factor* and *Re-Build* strategies are commonly also referred to as *brownfield* developments, while a "New Application" or *greenfield* development would not require an existing system implementation. Knoche and Hasselbring [KH18] similarly distinguish between rewrite, restructuring/refactoring, and incremental approaches. The latter represent a popular strategy for gradually decomposing an existing system. Yoder and Merson provide a detailed discussion of the related *Strangler Application* [Fow04b] pattern and its use in evolving a monolith to the microservices architectural style [YM22]. Bajaj et al. [BBGG21] further add, that brownfield approaches can be grouped into two technique types, namely static and dynamic approaches. This categorization refers to the underlying refactoring techniques, which we will discuss in the following section.

## 2.2. Architectural Refactoring

Software evolves throughout its life-cycle, and the ability to do so efficiently receives much attention in today's fast-moving markets [Bog20]. Software restructuring or refactoring is a means to improve specific quality attributes of a system without changing its behavior [FBBO99]. According to Capuano [Cap21], the migration of an existing monolithic system towards microservices can be regarded as a refactoring of its architecture. Most of the existing research on software refactoring, however, concerns code level refactoring [MT04], that aims to improve the source code's understandability and maintainability [Fow04a].

Architectural refactoring, on the contrary, addresses the higher level structure of a system and deals with documentation involving design decisions, components and connectors, and planning artifacts [Zim17]. Zimmermann

defines architectural refactoring as "*a coordinated set of deliberate architectural activities that [...] improve at least one quality attribute without changing the system's scope and functionality.*" [Zim15], while quality-related conflicts and trade-offs are possible. Such activities would revisit and alter architectural decisions that manifest in subsequent engineering tasks, including structural design changes, implementation, configuration, documentation, and communication [Zim17]. These tasks would be significantly more difficult to automate than code refactoring, as their scale is larger and the level of abstraction is higher [ES22; Zim17].

Several recent studies on architectural refactoring refer to the works of Zimmermann [Zim15; Zim17], who constitutes its under-representation in contemporary software engineering practice, confirmed by Samarthyam [SSS16]. Zimmermann argues that the intangibleness of many involved artifacts would make it difficult to establish universally valid rules and practices. This makes architectural refactoring an effort-intensive process, whose resource demands are difficult to assess in advance [ES22; RR22]. The difficulties are amplified by infrastructural and organizational implications in the case of adopting microservices [TLP17]. As a mitigation, Balalaie et al. [BHJ+18] built a collection of migration patterns from industrial-scale projects that can support the planning of such an architectural refactoring to microservices.

In the following, we review the underlying architectural refactoring techniques that are applicable in the context of a migration to microservices. By combining them with possible migration strategies (Section 2.1.4), we further discuss a classification scheme for such refactoring approaches in a migration context.

### 2.2.1. Refactoring Techniques

Microservices aim at reducing a system's complexity through functional decomposition into self-contained units. In Section 2.1.3, we quoted the *single responsibility* as a measure for the appropriate size of such a unit. However, software architects require more reliable criteria and methods to determine its size and boundaries. Domain-Driven Design (DDD) [Eva04], a software

design approach introduced by Evans in 2004, regained popularity with the rise of microservices [ÖBB23b]. DDD offers a common language that facilitates the communication between stakeholders and is particularly suitable for the design of new systems. As such, it is a commonly used technique for service identification, while initially neglecting the technical details of a subsequent implementation [ÖBB23b]. The authors report in another study on using DDD in the context of refactoring existing systems [ÖBB23a], and emphasize the positive effects of standardizing the refactoring efforts. However, the authors also point out that the complexity of this generally manual technique can be a barrier. It requires profound domain knowledge, and, moreover, involves interpretations and personal judgment of the architects.

Schmidt and Thiry [ST20] differentiate refactoring approaches for microservices based on their use of Domain-Driven Design (DDD) or Model-Driven Engineering (MDE) [Sch06] techniques. For 27 studied approaches, they found 14 of them applying MDE techniques, while only five approaches use DDD. The three of them, which solely rely on DDD, have been identified as exclusively manual approaches with no automation capabilities. While DDD has proven to adequately identify service boundaries and service granularity [VK21], major challenges affecting its use have been pointed out. Rademacher et al. see one reason in the "*domain model informality*" introduced by DDD, and further state that "*domain models are typically underspecified, which poses several challenges when applying DDD to [microservices architectures]*" [RSS18].

When inspecting both methods in the overall system engineering context, the difficulties involved with DDD become more obvious. Brambilla et al. [BCW17] show in Figure 2.1 the interplay between both design techniques. DDD focuses on the accurate modeling of a problem domain in its natural environment, while MDE is a means to transform the resulting model into a concrete system implementation. Here, the employed modeling languages often support the automated transformation into executable artifacts, which would improve precision, quality, and productivity [RSZ17; Sil15].

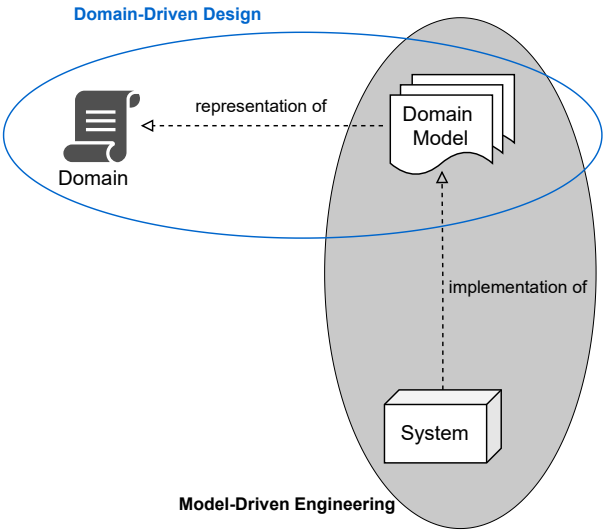



**Figure 2.1.:** Domain-Driven Design (DDD) and Model-Driven Engineering (MDE) in System Development (Adapted From [BCW17])

The substantial effort of a manual technique like DDD may therefore not be practical for industry-scale applications, as its automation is difficult to achieve. When refactoring an existing system, model artifacts can be derived in an automated way by using reverse engineering techniques [DKTT22] and complement existing abstract system representations. We conclude that an architectural refactoring to microservices that leverages existing system artifacts allows for a higher degree of automation, due to the employment of model-driven engineering techniques. Domain-driven design, on the other hand, can be a complementary manual technique, but is more suited to new developments in a greenfield approach.

The demand for reliable splitting criteria in a cost- and resource-efficient architectural refactoring process encouraged researchers to propose a variety of approaches [ASS22; SS21; ST20]. Schröer et al. [SKM20] contributed a comprehensive review of 31 microservices identification approaches. They found that a decomposition into microservices mostly originates from mono-

lithic (code) artifacts, with 17 of them relying on static analysis, sometimes complemented by dynamic analysis. The remainder of approaches relied on requirements or model artifacts as input. Bajaj et al. [BBGG21] similarly reviewed 21 approaches, covering also tool support for automation. They found static analysis techniques in all approaches, dynamic techniques in seven, and hybrid techniques in three approaches.

Kirby et al. [KBAR21] reviewed 13 approaches and collected expectations from 10 practitioners regarding tools that automate the decomposition. The results indicate that such tools should ideally allow for suggesting and comparing different candidate service cuts. As well, they recommend combining multiple relationship types for generating a decomposition, e.g., structural, semantic, or evolutionary. Table 2.1 summarizes the mentioned reviews, supplemented by the study from Abdellatif et al. [ASM+21]. The summary reveals a dominance of static analysis approaches that primarily leverage executable artifacts, but also operate on models or requirements. Dynamic analysis is less common and mostly complements the former techniques for higher precision.

**Table 2.1.:** Analysis Technique Types in Existing Reviews

| Review | Sources | Types of Analysis Techniques |
|---|---|---|
| Schröer et al. [SKM20] (2020) | 31 | Monolith **static** (17), Greenfield (9), Business Process (5), Monolith **dynamic** (4), Monolith first (2) |
| Bajaj et al. [BBGG21] (2021) | 21 | **Static:** Requirement Docs/Models (8), Design Docs (4), Source Code (7), VCS History (2), **Dynamic** (7), Hybrid (3) |
| Kirby et al. [KBAR21] (2021) | 13 | Structural-**static** (11), Structural-**dynamic** (6), Semantic (2), Evolutionary (1) |
| Lapuz et al. [LCA21] (2021) | 8 | **Dynamic** analysis (8) |
| Abdellatif et al. [ASM+21] (2021) | 13 of 41 | **Static** analysis (10), **Dynamic** analysis (5), Lexical analysis (2) |

From the reviewed secondary studies, we can identify three types of techniques used by the covered approaches. Figure 2.2 shows them in a simplified depiction of a refactoring process that combines all relationship types [KBAR21]. While each type will likely yield a different service cut, their aggregation can lead to more reliable results.

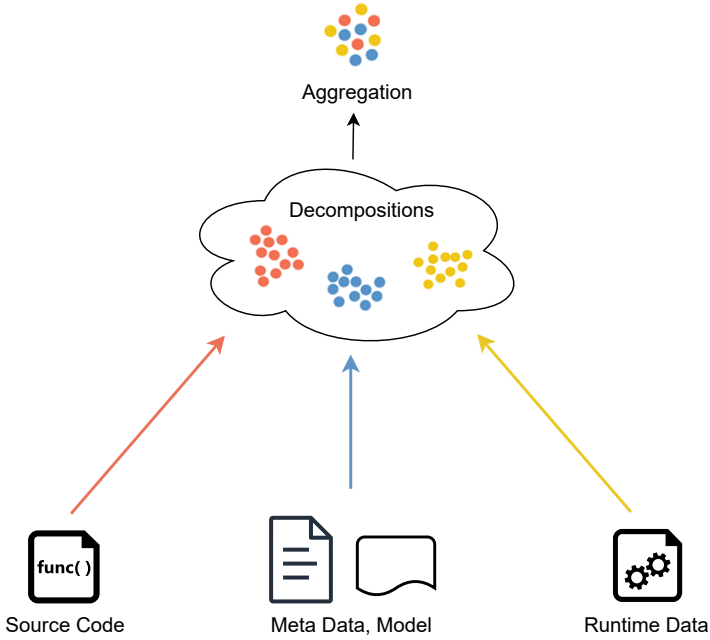



**Figure 2.2.:** Refactoring Techniques Found in Existing Reviews

The most comprehensive review by Abdellatif et al. [ASM+21] (Table 2.1) covers 41 service identification approaches suggested between 2004 and 2019, from which 13 specifically target microservices. Compared to the other studies, the authors classify approaches not only by their employed analysis technique type, but a variety of other attributes relevant in the migration process. As a result, they suggest a detailed taxonomy of such

service identification approaches for legacy software systems modernization, which we discuss in the following.

### 2.2.2. A Classification Scheme for Approaches

To find suitable refactoring approaches for migrating monolithic systems towards microservices, their essential characteristics need to be identified. We therefore elaborate a suitable classification scheme for such approaches, which represents another preparatory work for the following contributions in this thesis. The before discussed review by Abdellatif et al. [ASM+21] yielded a taxonomy for service identification approaches based on the analysis of 41 publications. As it targets service identification in the context of service-oriented architectures (SOA), we consider it a precedent-setting groundwork for an enhancement by microservices-related specifics. Figure 2.3 shows the attributes of such approaches, grouped into the four top-level categories *Inputs*, *Outputs*, *Process*, and *Usability*. It covers the specifics of such approaches in an elaborate and practice-oriented manner. However, microservices alter and refine the concept of SOA in several aspects, as discussed in Section 2.1.3. To account for these differences, we suggested several adaptions to this taxonomy.
Specifically, we consider the following changes:

- **Input**: For easier classification and a flatter hierarchy, we omit the subcategory *Non Executable*, while its branches *Runtime Artifacts* and *Model Artifacts* are promoted to separate subcategories. We also allow additional input artifacts to be reflected, such as repository data from a version control system.
- **Process**: This category will be extended by the subcategories *Process Strategy*, *Technique*, and *Atomic Unit*, to reflect these additional process-related aspects. The subcategory *Process Strategy* would distinguish approaches according to their suitability for *Re-Factor*, *Re-Build*, and *New Application* strategies. The additional *Technique* subcategory will reflect the underlying algorithms and techniques, such as clustering,

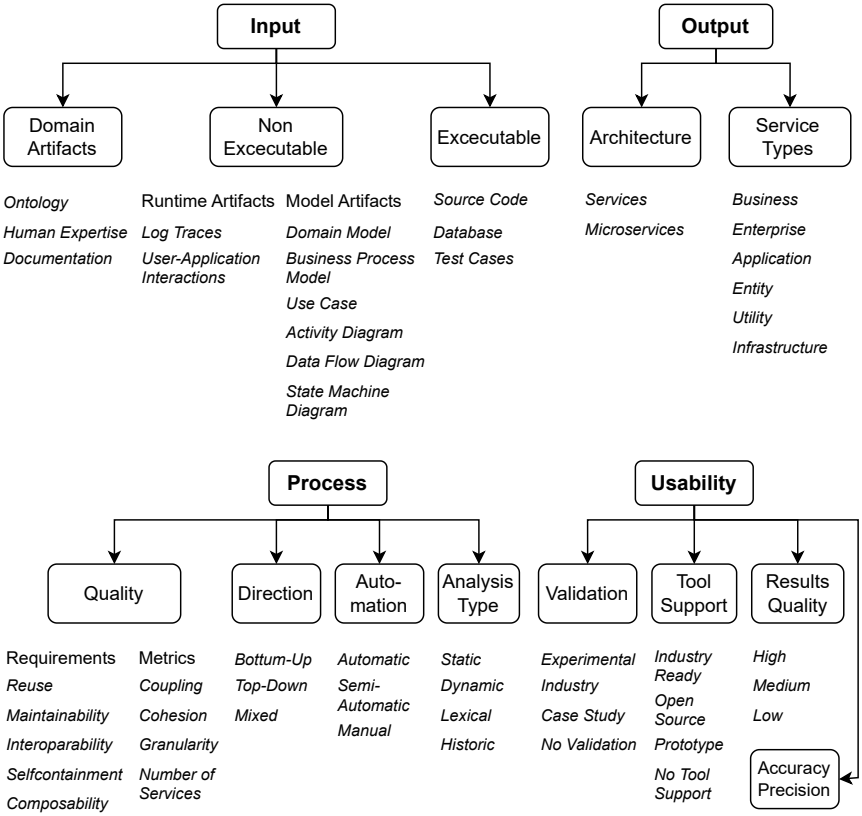



**Figure 2.3.:** Taxonomy of Service Identification Approaches by Abdellatif et al. (Adapted From [ASM+21])

graph-based, data-flow driven, execution-trace modeling, or genetic algorithms. We consider the *Atomic Unit* an essential criterion for such approaches, as it determines the finest possible granularity of resulting microservices. Typical atomic units are, for instance, entity, class, function, interface, use case, or business capability.

- **Output**: We omit the subcategory "Service Types" due to their relevance only in the SOA context. Instead, an extension by a subcategory

*Representation* would characterize the generated output. Approaches could then be classified by generating, e.g., splitting recommendations, a list of services, or even source code for executable microservices.

- **Usability**: We omit the *Results Quality* subcategory, as a rating of approaches in this regard is difficult and prone to subjective judgment. Moreover, we detect an overlap with the *Accuracy Precision* subcategory. We enhance the *Tool Support* subcategory by a basic characterization of accompanied tools, e.g. analysis type, programming language coverage, visualization capabilities, or availability of the tool.
- **Quality**: We promote the *Process* subcategory *Quality* to a top-level category, thereby reflecting its importance in the migration process. Moreover, this category is subject to extension and replacement by a dedicated quality model for microservices migrations. In this regard, we consider replacing the subcategory *Quality metrics* by *Design Properties* which allows for a more precise denotation of these architecture-related characteristics that interrelate with product quality goals. Next to coupling, cohesion, and granularity, we consider complexity, isolation, or technology heterogeneity as additionally relevant properties.

In the following, we provide a more in-depth discussion on the role of quality attributes in the migration process to microservices. To do so, we start with a literature survey on relevant quality attributes for microservices architectures in general.

## 2.3. Software Quality in Microservices Architectures

Delivering high-quality software products is essential for software companies to withstand highly competitive and rapidly changing markets. High quality standards facilitate customer satisfaction – key to retaining existing and gaining new customers. The notion of quality is omnipresent in our daily life, and its concept is perceived self-explanatory to most of us. However, the views on its meaning in a specific context can diverge significantly [Bev95]. Garvin

provides five differing definitions of quality, and hence denotes quality as "*a complex and multifaceted concept*" [Gar84]. According to Bevan [Bev95], quality has an inherently subjective component, as its assessment is dependent on a person's perception. In the context of measuring software quality, however, most approaches would aim to reduce this subjective and "*intrinsically inaccurate judgement*" [Bev95]. The author defines software quality as "*an inherent characteristic of the product determined by the presence or absence of measurable product attributes*" [Bev95], also referred to as *product quality* [Gar84]. Quality models aim to provide a structured view on software quality by defining different aspects or dimensions. According to Wagner, a quality model has "*the objective to describe, assess and/or predict quality*" [Wag13].

Such models allow for determining the degree to which a system or software product satisfies the stated or implied needs of stakeholders, thereby ensuring to provide value [ISO11a]. The ISO/IEC 25010 standard, also referred to as SQuaRE (Software product Quality Requirements and Evaluation), defines a widely accepted quality model that is used in various phases of a software development process [Wag13]. It is composed of five characteristics that cover the "*complete human-computer system*" [ISO11a] including computer systems and software products. To make quality models easier to understand and manage, they typically comprise the overall product quality out of several quality categories. In this regard, ISO/IEC 25010 uses the notion of a quality characteristic and subcharacteristic, respectively. For software product quality, two other notions are often synonymously used in software engineering literature: quality factor [NW14; Wag13] and quality attribute [CM22; COU19; LZJ+21]. In the following, we adhere to the latter terminology by referring to, e.g., *Maintainability* as a quality attribute and *Testability* as one of its sub-attributes.

Of particular interest in the context of software architecture are *low-level* or *design time* quality attributes "*that can be directly calculated from metrics*" [AAC+17]. They are also called design quality attributes [AWS18] or design properties [BNZ+15; NW14; SCKP08]. Shim et al. propose a respective design quality model for service-oriented architectures [SCKP08],

which extends a quality model for object-oriented design. They describe how the components' design influences metrics, which are used to measure design properties. Thereby, design properties comprise "*the features of the system and the relationships between the different components of the system*" [BNZ+15]. Eventually, design properties influence high-level quality attributes, such as maintainability or performance. While reviewing 154 papers on software quality, Arvanitou et al. find that design properties are even more studied than high-level quality attributes, with the most mentioned ones being complexity, cohesion, and coupling [AAC+17].

In the following subsections, we provide a more detailed investigation of quality attributes, design properties, and typical quality trade-offs in the context of microservices.

### 2.3.1. Quality Attributes

When discussing the definition of software architecture, we pointed out that structures are considered "*the primary engineering leverage points of an architecture*" [BCK21]. The authors further state that each structure "*brings along the power to manipulate one or more quality attributes*" [BCK21]. In the context of software restructuring and architectural refactoring, we identified quality attributes as an essential driver and goal [FBBO99; Zim15]. Capuano and Muccini [CM22] emphasize the particular relevance of quality attributes in migrations to microservices, where they play a major role in selecting suitable approaches and techniques [LZJ+21]. Given the fundamentally differing structures of monolith and microservices, we review quality attributes commonly associated with microservices architectures. Their knowing allows us a profound assessment of intentions for adopting microservices, and the role of quality aspects throughout a migration process.

Quality attributes associated with microservices architectures have become increasingly important and receive attention by both industry and academia [LZJ+21]. From surveying 16 software professionals, we found that the impact of microservices-based architectures on software quality is mostly perceived positive [BFWZ19b]. However, not all qualities differ to the

same extent between both architectural styles. To identify the most profiting qualities from using microservices, we performed a literature review that is detailed in [Koc22]. We consulted a total of 13 scientific publications between 2016 and 2022. To ensure a consistent classification, we adhered to the ISO/IEC 25010 standard [ISO11a]. All quality attributes that were found at least twice within all publications, were included in the results. Besides, we documented indications of potential trade-offs between attributes.

From the 13 publications, 10 quality attributes could be identified in the context of microservices architectures, along with 28 sub-attributes [Koc22]. Of the 10 quality attributes, the following six attributes correspond to the ISO/IEC 25010 [ISO11a] classification: *Reliability*, *Performance (Efficiency)*, *Security*, *Compatibility*, *Maintainability*, and *Portability*. Moreover, the following attributes could be identified: *Scalability*, *Independence*, *Execution Cost*, and *Organizational Alignment*. The frequency of our identified quality attributes and their associated sub-attributes among all papers is illustrated in 2.4.

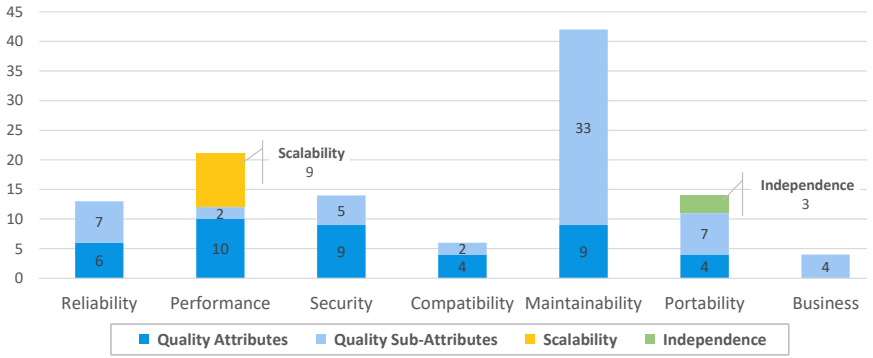



**Figure 2.4.:** Microservices Qualities Extracted From 13 Publications, by Number of Occurrences (From Koch [Koc22])

We found that *Maintainability* is the most prominent attribute in our reviewed publications, followed by *Performance* along with *Scalability*. While

*Reliability*, *Security* and *Portability* received decent attention, *Compatibility* and the additionally identified *Business* attribute were mentioned least.

The attribute *Scalability* is usually subordinated to *Performance* [DML17], but often appears as a separate attribute due to its high relevance in the microservice context [LZJ+21]. While *Independence* is subordinated to *Portability* [ISO11a], it can also be considered a separate attribute due to its relevance in other contexts: For instance, both independent development by individual teams [LS18; Ric15], and independent testing [KCAT21] would reflect organizational aspects. Lastly, the *Business* attribute comprises the aforementioned sub-attributes *Execution Cost* and *Organizational Alignment*.

In Figure 2.5, we summarize our identified quality attributes for microservices architectures. The structure adheres to the ISO/IEC 25010 quality model [ISO11a], with the blue sub-attributes being congruent to the standard. Additional identified sub-attributes are colored magenta. We mapped them to the most appropriate quality attribute, based on the context and evidence found. For the qualities related to *Business* aspects, we introduced a new attribute and assigned its two sub-attributes. According to the above stated peculiarities of *Scalability* and *Independence*, we indicated their treatment as a potentially separate quality attribute.

### 2.3.2. Design Properties

Our review of quality attributes for microservices architectures yielded numerous mentions of low-level attributes, such as *Cohesion*, *Coupling*, and *Granularity*. According to our reflection in Section 2.3, we refer to them as *design properties*. As Shim et al. [SCKP08] describe, they act as mediators for addressing certain product quality attributes in an architectural design [ISO11a]. Our review [Koc22] revealed four design properties, which are commonly associated also with service-oriented architectures [SCKP08]: *Coupling*, *Cohesion*, *Granularity*, and *Complexity*. Additionally, we identified *Isolation*, *Autonomy*, and *Technology Heterogeneity* as often mentioned design aspects for microservices, which we also regard as design properties within our work. In the following, we briefly characterize these design properties

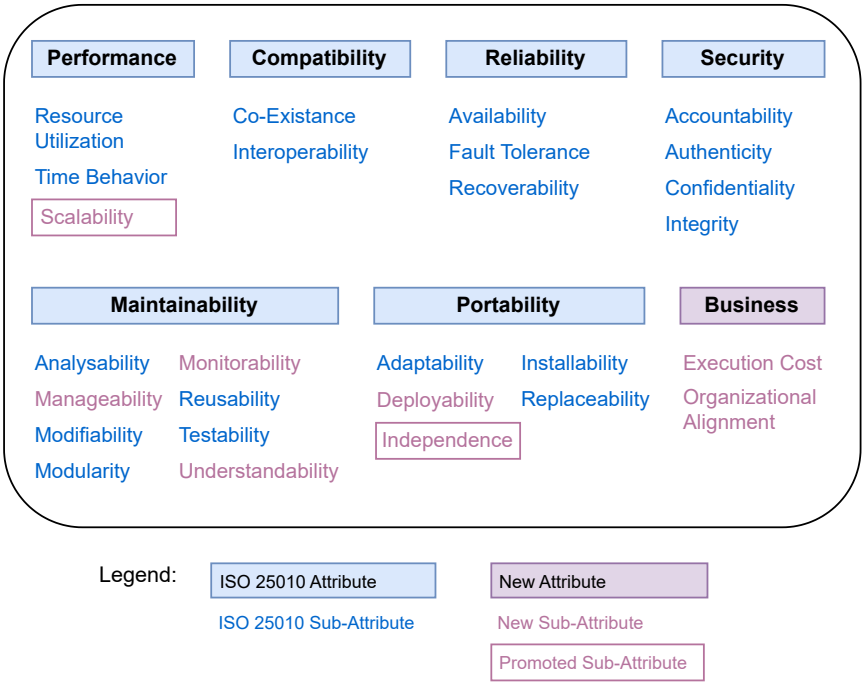



**Figure 2.5.:** Quality Attributes and Sub-Attributes in Microservices Architectures based on 13 Publications (Adapted From Koch [Koc22])

in the context of microservices architectures, by highlighting their influence on quality attributes.

**Granularity** of services, and thereby the appropriate size of a service [COU19; SCKP08], reflects a key question of a microservices design. Bogner et al. denote it as "*size and degree of decomposition*" [BWZ17b] in a service-based system. Finding the correct or optimal granularity is one of the most frequently discussed topics of this architectural style [COU19]. A finer granularity directly improves *Modularity*, and thereby promotes *Reusability*, *Understandability*, *Flexibility*, and *Maintainability* in general [COU19; SCKP08]. However, it can also increase the *Complexity* and impede *Performance*, as additional communication effort between microservices is required [ECA+16].

A finer granularity can reduce the *Resource Utilization* and a thereby decrease *Execution Cost*, as scaling small individual microservices would be more efficient [LZJ+21].

**Coupling** reflects the strength or number of dependencies between components, modules, or services [BWZ17b; COU19; ISO11a]. Hutapea et al. describe it as the *"power of interaction and dependency between services in a system"* [AWS18]. Coupling can be measured internal to a service by calculating "*the sum of the dependencies between the modules inside the microservice*" [YGS21] and, more common, through weighted coupling of interfaces which "*measures the degree of dependency between microservices*" [YGS21]. It is commonly desired to reduce the coupling between components or services to a minimum [AA23; AAC+17]. The concept of loose coupling dates back to organizational studies by Weick [Wei76], and is a considered a key principle of realizing SOAs based on web services [Kay03]. Loose coupling improves *Maintainability* by promoting *Manageability*, *Testability*, and *Reusability* [BWZ17a; BWZ17b; COU19; SCKP08]. As well, it promotes *Understandability* [SCKP08] and *Analyzability* [AAC+17], but can have a negative impact on *Performance* [DB20]. We found no documented associations with the quality attribute *Independence*, but a strong correlation seems obvious [COU19].

**Cohesion** describes "*the degree to which the elements of a component are interrelated*" [AAC+17]. In the context of microservices, it refers to the functionality that a microservice accommodates [BWZ17b; COU19; PRF07], i.e., the degree, to which individual functions within a microservice are interrelated [ISO11a]. Cohesion can be measured, e.g., through determining the semantic similarity of a service's interfaces in terms of parameter types and names [COU19]. There exists a strong interrelationship with coupling, as a high cohesion within a service would promote a low coupling with other services [AAC+17]. Like loose coupling, a high cohesion of services therefore promotes *Maintainability* [BWZ17a; BWZ17b; PRF07] in a microservices architecture. In particular, it can promote *Reusability* [DB20],

*Manageability* [COU19], *Analyzability*, *Testability* [PRF07] and *Understandability* [SCKP08].

**Technology Heterogeneity** as an additionally identified property describes the degree to which the choice of programming languages or technologies is unconstrained for the implementation of individual services [AAE16]. While commonly regarded as desirable and advantageous, it can impact the overall *Maintainability* of a system [COU19]. Such additional efforts for maintaining a heterogeneous technological ecosystem could arise from software updates, dependencies, or the acquisition and transfer of technology-related expertise.

**Isolation** encompasses the isolation principle, meaning that individual microservices are designed to be self-contained throughout development and deployment. It also refers to the mutual independence of development teams [COU19]. The **Autonomy** of microservices describes that they *"are responsible for the execution of their own underlying processes, which can be initiated, operated, and shut down independently"* [MFC+18]. The authors add that autonomy also refers to the existence of an own development lifecycle. Both properties foster the quality attribute *Independence*, and are sometimes used synonymously [AAE16].

**Complexity** describes *"the degree to which a system or component has a design or implementation that is difficult to understand and verify"* [ISO11b], while Shim et al. emphasize the relationship complexity between services in the context of service-oriented architectures [SCKP08]. The reasons can be manifold, with one of them being numerous components or relationships between components [ISO11b]. In the context of microservices, Cojocaru et al. relate it to "*the amount and variety of internal work carried out*" [COU19]. Yu et al. add the occurrence of ripple effects for changes, i.e., the number of necessary changes to other modules, when one module is changed [YGS21]. A high complexity in a microservices architecture primarily leads to a reduced *Maintainability* [BWZ17a] by hindering *Understandability* [SCKP08].

Based on the outlined interrelationships, Table 2.2 summarizes them by reflecting the influences of design properties on quality attributes.

**Table 2.2.:** Interplay Between Design Properties and Quality Attributes (Sub-Attributes in grey-blue), Based on 13 Publications [Koc22]

| Design Property | Influenced Quality (Sub-)Attribute |
|---|---|
| Granularity | Maintainability, Modularity, Performance, Resource Utilization, Execution Cost |
| Coupling | Maintainability, Analyzability, Testability, Reusability, Understandability, Manageability, Performance, Independence |
| Cohesion | Maintainability, Analyzability, Testability, Reusability, Manageability, Understandability |
| Complexity | Maintainability, Modularity, Understandability |
| Isolation and Autonomy | Independence |
| Technology Heterogeneity | Maintainability |

### 2.3.3. Quality Trade-Offs

The simultaneous targeting of several quality attributes can become a multi-faceted decision-making problem, which sometimes involves mutually exclusive goals (Section 2.1.1). In this regard, our review revealed several trade-offs between quality attributes in a microservices architecture. For instance, *Security* is often in conflict with *Maintainability* [MM22]. Li et al. further note that certain tactics for authentication and authorization lead to a trade-off between *Security* and *Performance* [LZJ+21], while *Scalability* can be hindered by securing the communication between services [AI16]. While *Performance* and *Scalability* are often considered in combination [DML17], trade-offs between both can be necessary: A finer granularity of services would increase the system's *Scalability*, while *Performance* may be degraded as a result of the communication overhead [KVGJ17; LZJ+21]. According to Auer et al. [ALFT21], the *Availability* in a microservices architecture is not in-

herently high, and it would be difficult to achieve, which can imply increased *Maintainability* efforts. Based on the reviewed literature, we identified the following quality trade-offs for microservices architectures:

- Security ⟷ (Maintainability ∨ Performance ∨ Scalability)
- Performance ⟷ Scalability
- Availability ⟷ Maintainability

In this section, we identified quality attributes that are commonly associated with a microservices architecture. They build the foundation for a quality-driven migration process, which we will detail in the following section.

## 2.4. Quality-Driven Migration to Microservices

While there can be various reasons for migrating a monolithic system towards microservices, improving one or more software product qualities is often a major intention, if not the main driver. In a survey conducted in 2018 by DZone[1], an online community for software professionals [Gle18], a total of 732 participants were asked about intentions for using microservices within their companies. As main reasons for adopting microservices, scalability (∼81%) and faster deployments (∼70%) were stated. The third and fourth most frequent responses expected an improvement of the overall quality by individually focusing on small parts of the application.

Academia similarly emphasizes the importance of non-functional requirements and quality attributes throughout a migration process [Cap21; SWG21]. In their systematic literature review, Li et al. state that rationales for a migration to microservices need to be clear upfront [LZJ+21]. Capuano and Muccini add that for assuring software quality standards, all phases of a migration must be examined in this regard [Cap21]. In a follow-up systematic review, Capuano and Muccini [CM22] inspected 58 publications regarding quality-driven migration approaches. They identified the involvement of

[1] https://dzone.com/

quality attributes throughout all four phases in a migration process, but to differing extents. While around 28% of the approaches consider quality attributes in the initial comprehension phase, nearly half of them consider them in the following microservice identification phase. Still, around one third of the approaches consider them during the third microservices assessment phase, and only ∼2.5% in the final packaging phase. The authors summarize that many approaches consider quality attributes throughout the migration process, but often do not explicitly state quality improvements as a migration goal.

We conclude that both academia and industry confirm the significance of quality aspects in a migration to microservices. Capuano and Muccini regard it as a growing trend and introduced the notion of a "*quality-driven migration*" as "*modernization process that allows to satisfy, and potentially improve with respect to the starting software system, a subset of quality attributes*" [CM22]. They emphasize that a predefined subset of quality attributes, originating from quality goals, should be considered at each stage. The definition or elicitation of such goals is commonly the result of an architecture evaluation. It should involve the various stakeholders of a system, such as requirements engineers and software architects [SWG21]. In the following section, we introduce an established method for performing such architectural evaluations. It thereby represents the first stage or "*comprehension phase*" [CM22] in a quality-driven migration to microservices.

### 2.4.1. Architecture Assessment and Migration Decision

Software can be assessed regarding its development process and artifacts, whereas the latter include requirements, source code, or architectures [Sta08]. Starke adds that architectures are particularly suitable for an assessment due to their design early in the development process, which leads to associated decisions of high relevance for the resulting system. In the context of an architectural refactoring, the necessity of an initial assessment arises from the intent itself. According to Zimmermann, a baseline should always be established, or goals should be set in the form of certain quality

attributes [Zim17]. Inspired from *agile user stories* [Coh04], he proposes to specify *quality stories* in this regard.

In the context of migrations from monoliths to microservices, Auer et al. [ALFT21] propose a decision support framework. From a series of expert interviews, they derived a set of relevant quality characteristics and metrics that should be collected from the monolithic system. An analysis of these characteristics should strengthen the migration justification, against potential alternatives like monolith refactoring or organizational restructurings [ALFT21]. Similarly, Cojocaru et al. provide a set of quality attributes as criteria for an initial quality assessment, extracted from migration tools and techniques [COU19]. While an ad-hoc assessment based on quality attributes can be suitable for smaller systems, more systematic architecture evaluation methods should be preferred for large-scale enterprise systems.

Mattsson et al. [MGM06] distinguish four categories of such evaluation methods, namely experience-based, simulation-based, mathematical modeling, and scenario-based. Experience-based methods leverage previous knowledge of architects and other stakeholders, while methods based on simulations use high-level implementations or prototypes to simulate the system's behavior, and thereby derive quality requirements. Mathematical modeling relies on formal techniques and proofs for evaluating quality attributes, which can, e.g., be combined with simulation-based methods for a detailed performance evaluation [MGM06]. Scenario-based methods, eventually, employ operational scenarios that reflect a use case or requirement of the system. Such scenarios combine both functional and non-functional aspects. A scenario aims for the evaluation of a particular quality attribute by "*creating a scenario profile that forces a very concrete description of the quality requirement*" [MGM06]. Architectural decisions are then reflected per scenario, to determine their influences or appropriateness for fulfilling the scenario. Bosch notes that a scenario-based assessment can be specifically used for comparing two architectures [Bos99].

Several scenario-based evaluation methods have been proposed, such as Software Architecture Analysis Method (SAAM) [KBWA94], Architecture Trade-off Analysis Method (ATAM) [KKC00], or Architecture Level Modifia-

bility Analysis (ALMA) [BLBV04]. Given their systematization, involvement of various stakeholders, and suitability for comparative assessments, we consider them most appropriate in the context of an architectural refactoring from a monolithic architecture to microservices. For our following considerations, we employ the established ATAM method [KKC00]. Reijonen et al. [RKH10] summarize the procedure of an ATAM evaluation, which passes through the four phases of preparation, evaluation, verification, and follow-up. In the preparation phase, a preliminary schedule is compiled, including identification of required stakeholders. The evaluation phase contains a presentation of business drivers and current architecture to stakeholders, followed by an identification of quality goals and elicitation of scenarios. Starke [Sta08] summarizes this evaluation phase as a brainstorming session by stakeholders, which results in a utility tree. Quality attributes form the basis of the tree, and are subsequently refined into concrete scenarios. For a common understanding of software product qualities, the process can be guided by an established quality model, like ISO/IEC 25010 [ISO11a].

After the scenario elicitation follows their prioritization according to business value, and difficulty to achieve or realize. For both ratings, that should be agreed on by all participating stakeholders, a simple High-Medium-Low (or A-B-C) scale can be used [RKH10; Sta08]. Hence, each scenario is assigned an importance and difficulty rating, with **A** = important/difficult, **B** = medium importance/difficulty, and **C** = less important/difficult. Starting with the scenario providing the highest business value, all scenarios are now analyzed [RKH10], wherein different architectural solutions can be discussed, involving further thoughts and ideas. During this analysis, "*a set of related architectural decisions and sensitivity points are selected*" [RKH10] for every scenario. Such sensitivity points would likely affect a scenario's associated quality attribute. In this process, the scenarios serve as vehicles that allow for probing the architecture [RKH10] or alterations to it. An exemplary utility tree with two scenarios is shown in Figure 2.6, in which the quality attribute *Performance* has scenarios assigned to its sub-attributes *Latency* and *Throughput*.

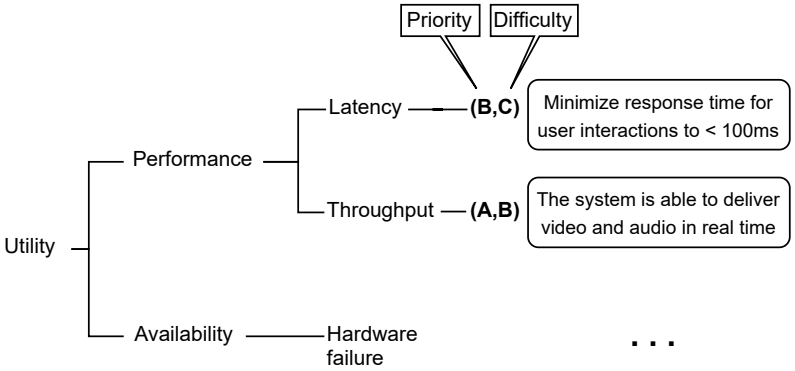



**Figure 2.6.:** ATAM Utility Tree (Adapted From [KKC00; Sta08])

Such a utility tree provides a concise and structured overview of the quality demands on the architecture, including a weighted prioritization [Sta08]. The elaborated evaluation phase is followed by the verification and follow-up phases [RKH10], which makes ATAM an altogether complex and time-consuming method that can span even several weeks [KKC00]. Thus, Svahnberg and Mårtensson proposed a more lightweight method based on ATAM and SAAM [SM07], which similarly generates the above described utility tree. However, their devised method can be conducted within one day and is therefore suitable for smaller systems and organizations with tight budget constraints. While originally developed and used in student projects, the authors also report on their industry experiences gathered over several years.

A systematic architecture evaluation serves as migration decision guidance first, and thereby constitutes a means to justify the cost and effort of such a transformation to stakeholders and management, based on cogent reasoning. Moreover, it provides predefined quality goals and tangible measures that help to select suitable approaches for the architectural refactoring. In the following, we will introduce the role of software quality during this second phase, the decomposition into services.

### 2.4.2. Quality-Oriented Decomposition

Academia proposed a variety of approaches and techniques for the service identification, i.e., decomposition of a monolith into microservices. In Section 2.2.1, we presented an overview of such architectural refactoring approaches. The before cited review by Capuano and Muccini [CM22] showed that nearly half of their 58 inspected migration approaches consider quality attributes in the microservices identification phase, meaning that they address one or more quality attributes while generating a decomposition. Hence, quality aspects represent one of the selection criteria (Section 2.2.2) for determining their suitability in a specific context: While one approach may aim at optimizing scalability [CCRZ23], another one would tailor the partitioning to address security aspects [AI16], and still others consider several qualities simultaneously [ACC+22a; GM20]. Such qualities can also include business-related aspects, like aiming for zero downtime or minimizing the cost of a functional redesign [FSC+21; SS22]. A quality-oriented selection of refactoring approaches can continue a quality-driven migration that started with a systematic architecture assessment, as discussed before in Section 2.4.1. The classification of such approaches in the microservice migration context will build a foundation hereto, and represents a main analysis contribution in this thesis (Chapter 5).

### 2.4.3. Quality-Oriented Design of Microservices

The process of designing an architecture is regarded a complex and challenging task, wherefore architects have been seeking "*for ways to capture and reuse hard-won architectural knowledge*" [BCK21]. Architectural patterns are a means to document and preserve such knowledge for later use. They represent "*compositions of architectural elements, which provide packaged strategies for solving some of the problems facing a system*" [BCK21]. According to the authors, a template for such patterns consists of context, problem, and solution. Several publications emphasize the importance of patterns and their role in satisfying desired quality goals of an architec-

tural design in general [AFS12; BCK21; FBB+14], and for microservices in particular [ODGG20; VCG+22; VLC19; VLL+20].

The principles and characteristics outlined in Section 2.1.3 represent the cornerstones for designing a microservices architecture. Patterns provide building blocks hereto, based on proven and reusable solutions on a lower level. Valdivia et al. identified 34 patterns for microservices design from scientific literature [VLC19], supplemented by 20 patterns from grey literature [VLL+20]. The authors remark on the differing notions *architectural pattern* and *design pattern* used in literature [VLC19]. They refer to Buschmann et al. [BMR+96], by stating that architectural patterns are *"fundamental structures of the system which maintain responsibilities and rules; whereas design patterns are described as subsystems or components added to the system"* [VLC19]. Moreover, design patterns would be characterized by a more detailed description and their application would not affect the fundamental structures of the architecture. Some authors use the terms "architectural style" and "architectural pattern" synonymously, while Zimmermann regards an architectural style as "*a set of architectural principles and patterns that are aligned with each other*" [Zim09]. Based on this view, we follow the terminology used by Valdivia et al. [VLC19], who use the term *pattern* interchangeably for referring to design and architectural patterns. We regard the latter as composite patterns that are composed of several, e.g., design patterns [Zim09], and eventually contribute to an architectural style.

In addition to patterns, *best practices* also seek to capture such knowledge and often complement pattern catalogs. While originating from industry practice, they are usually documented in a less structured manner [BBD+22; NZP+20]. Besides, their denotation differs in scientific publications: Lichtenthäler and Wirtz refer to them as "architectural characteristics" or "product factors" in the context of cloud-native software architectures [LFW23; LW22].

As further preparation for our contributions in this thesis, we seek evidence on the qualities that such patterns and best practices for microservices address. Their targeted selection would facilitate a quality-oriented architectural design, which represents the third phase in a migration process.

It completes a quality-driven migration to microservices that started with a systematic architecture assessment (Section 2.4.1) and went through a quality-oriented decomposition (Section 2.4.2). In the following, we therefore identify patterns and best practices for microservices designs, while collecting evidence on the quality attributes that they address.

## 2.5. Patterns and Best Practices for Microservices

Adhering to established patterns for the design of microservices can avoid common mistakes [SB21; TL18], while helping to build systems that exhibit desired software qualities. According to Zimmermann, patterns have a strongly educational character [Zim09], which may also explain their comprehensive coverage in scientific literature. However, their influence on various quality attributes is less well understood. In the following, we therefore attempt to link patterns and best practices to their associated quality goals. To do so, we review secondary research on patterns for microservices, which we later supplement by proven best practices.

Osses et al. [OMA19] systematically reviewed 1,067 studies and extracted 44 patterns and tactics for microservices. They identified the five quality attributes *Flexibility*, *Testability*, *Elasticity*, *Performance*, and *Scalability* as most commonly associated with these patterns, with the latter two being dominant. Based on the review results, the authors propose a basic pattern taxonomy for microservices.

Valdivia et al. [VLC19] contributed another systematic review on the mapping between quality attributes and patterns in microservices. They describe the patterns, their associations with quality attributes, and identify trade-offs in pattern selection. The authors emphasize potentially conflicting quality goals and add that their implications can not be generalized to all systems. Rosa et al. [ODGG20] confirm this difficulty and also claim that software architects lack expertise in patterns. They propose a method for architectural trade-off analysis based on patterns in the microservices context.

**Table 2.3.:** 51 Microservices Patterns From 7 Publications (From Koch [Koc22])

| | |
|---|---|
| **Communication** | |
| Asynchronous Messaging | REST Integration |
| Change Code Dependency to Service Call | Secure Channel |
| Competing Consumers | Service Discovery |
| Event Notification | Service Locator |
| Log Aggregator | Service Registry |
| Pipes and Filters | Service Registry Client |
| **Data Management** | |
| CQRS | Local Sharing-based Router |
| Database is the Service | Scalable Store |
| Event Sourcing | Shared Database Server |
| Local Database Proxy | Database per Service |
| **Deployment** | |
| Multiple Service per Host | Single Service per Host |
| **Distribution** | |
| Anti-Corruption Layer | Microservice DevOps |
| Container | Self-containment of Services |
| Deploy Cluster and Orchest. Containers | Strangler |
| Externalized Configuration | Enable Continuous Integration |
| **Entry Point** | |
| API Gateway | Backend for Frontend |
| Auth-Service | Gatekeeper |
| **Complementary** | |
| Ambassador | Health Check |
| Asynchronous Completion Token | Internal Load Balancer |
| Asynchronous Query | Key-Value Store |
| Circuit Breaker | Load Balancer/Load Balancing |
| Correlation ID | Page Cache |
| Edge Server | Priority Queue |
| External Load Balancer | Result Cache |
| Gateway Aggregation | Sidecar |
| Gateway Offloading | |

The discussed studies provide a first overview of the relationships between patterns and quality attributes, but lack empirical evidence from practice. Extending the efforts of Valdivia et al. [VLC19; VLL+20], we contributed an empirical study that reflects industry perceptions of 14 design patterns and uncovers their positive and negative impacts on seven quality attributes [VCG+22]. While our interviews confirmed the commonly documented trade-offs, we identified additional gains and pains that led to new insights about microservices patterns.

In addition [Koc22], we reviewed seven sources that document patterns for microservices along with their implications for software quality. Our review included works by Li et al. [LZJ+21], Márquez and Astudillo [MA19], Taibi et al. [TLP18], Valdivia et al. [VLC19; VLL+20], supplemented by our previous interview study [VCG+22]. The list of scientific publications was supplemented by a public web repository [Ric23]. Because the reviews use different categorization schemes, we merged closely related or synonymous terms, while trying to preserve existing groupings as much as possible. We distilled the six categories Distribution [VLC19], Communication [MA19; VLC19], Entry Point [VLL+20], Deployment [TLP18; VLL+20], Data Management [TLP18; VLL+20], and Complementary [Ric23; VLL+20]. Table 2.3 shows the 51 collected patterns that were mentioned at least twice in the reviewed publications.

### 2.5.1. Mapping Quality Attributes to Patterns

In the following, we associate quality attributes and sub-attributes with our 51 collected patterns in Table 2.3. An association reflects the pattern's positive influence on a given quality in the resulting architectural design. For this purpose, we use our previously identified set of quality attributes for microservices architectures (Figure 2.5). We documented a quality association for a pattern if it was mentioned at least once in any of the seven reviewed sources. Table 2.4 gives an overview of all identified quality attributes with their number of associated patterns.

**Table 2.4.:** Qualities Promoted by 51 Patterns (Based on Koch [Koc22])

| Quality Attribute | Number[a] of Patterns |
|---|---|
| Maintainability | 35 |
| Monitorability | 3 |
| Reusability | 1 |
| Understandability | 1 |
| Testability | 1 |
| Reliability | 24 |
| Availability | 8 |
| Performance | 14 |
| Security | 14 |
| Scalability | 13 |
| Compatibility | 6 |
| Portability | 1 |
| Independence | 1 |

[a] Number for Quality Attributes includes Sub-Attributes

With a total of 35 out of 51 patterns, the majority promotes the quality attribute *Maintainability*, with three explicit mentions for *Monitorability*, and one each for *Reusability*, *Understandability*, *Testability*. The second most promoted attribute is *Reliability* with 23 documented associations, while its sub-attribute *Availability* received 8 mentions. The attributes *Security* and *Performance* follow with 14 pattern associations, while *Scalability* is improved by 13 patterns. An improvement of *Compatibility* was still associated with six patterns, while improvements of *Portability* and *Independence* were only mentioned once in our reviewed sources. We note that *Maintainability* and *Reliability* were most prominent, while *Performance* combined with *Scalability* formed the third cluster. Quality sub-attributes were mentioned much less

often, with the exception of *Availability*. The complete list of patterns and their associated quality attributes can be found in the Appendix Table A.1.

### 2.5.2. Quality Trade-Offs for Patterns

As discussed in Section 2.3.3, optimizing for a particular quality goal may require trade-offs with other quality goals. In the context of patterns, such trade-offs have been reported by Taibi et al. [TLP18] and in our work [VCG+22]. In their systematic mapping, Taibi et al. [TLP18] point out advantages and disadvantages for each identified pattern. For example, they identify a trade-off for the Service Registry pattern with respect to *Reusability*, while Shared Database Server, API Gateway, and Single Service per Host may hinder *Scalability*.

In our study [VCG+22], we collected such quality trade-offs from expert interviews with practitioners. Here, we identified the Pipes and filters pattern to impact *Monitorability*. For the Gateway Aggregation pattern, *Scalability* and *Fault Tolerance* were identified as trade-offs, as applying the pattern can introduce a single point of failure. Similarly, applying the Externalized Configuration pattern may affect *Fault Tolerance*. We also found that *Performance* can be affected by the Anti-Corruption Layer pattern. Finally, the popular Strangler migration pattern may imply a trade-off with *Maintainability* and backwards *Compatibility*.

An increase in *Complexity* is mentioned in both studies for several patterns, such as Service Registry and API Gateway [TLP18], supplemented by Pipes and Filters, Ambassador, Sidecar, CQRS, Anti-Corruption Layer, Externalized Configuration, and Backend for Frontend [VCG+22]. The Gateway Aggregation was found to increase *Coupling* in the architecture [VCG+22], while applying the Shared Database Server pattern may reduce the degree of *Isolation* [TLP18]. Table 2.5 summarizes the 13 identified quality trade-offs for patterns in the context of microservices based on the two discussed publications.

**Table 2.5.:** Quality Trade-Offs for 13 Patterns (From Koch [Koc22])

| Pattern | Quality Trade-Off |
|---|---|
| Ambassador | Complexity [VCG+22] |
| Anti-Corruption Layer | Performance, Complexity [VCG+22] |
| API Gateway | Scalability, Complexity [TLP18] |
| Backend for Frontend | Complexity [VCG+22] |
| CQRS | Complexity [VCG+22] |
| Externalized Configuration | Fault Tolerance, Complexity [VCG+22] |
| Gateway Aggregation | Scalability, Fault Tolerance, Coupling [VCG+22] |
| Pipes and Filters | Monitorability, Complexity [VCG+22] |
| Service Registry | Reusability, Complexity [TLP18] |
| Shared Database Server | Scalability, Isolation [TLP18] |
| Sidecar | Complexity [VCG+22] |
| Single Service per Host | Scalability, Performance [TLP18] |
| Strangler | Maintainability, Compatibility [VCG+22] |

### 2.5.3. Best Practices

Best practices and high-level design principles can also contribute to certain quality goals in an architectural design [LW22]. Our collection of best practices for microservices design is based on the work of Lichtenthäler and Wirtz [LW22], which was validated in our follow-up study [LFW23]. The works by Pulnil and Senivongse [PS22], and Schirgi and Brenner [SB21] complement the above two publications. In them, the authors present collections of anti-patterns and the corresponding opposite best practices – which we use for confirmation. We documented all references mentioned at least once in any of the four publications. The resulting number of best practices influencing a quality attribute is summarized in Table 2.6, while the complete list of best practices and their associated quality attributes can be found in Appendix Table A.2.

**Table 2.6.:** Qualities Promoted by 43 Best Practices (Based on Koch [Koc22])

| Quality Attribute | Number[a] of Best Practices |
|---|---|
| Maintainability | 27 |
| Modifiability | 16 |
| Modularity | 11 |
| Understandability | 6 |
| Reusability | 5 |
| Analyzability | 1 |
| Testability | 1 |
| Reliability | 15 |
| Recoverability | 6 |
| Fault Tolerance | 5 |
| Availability | 4 |
| Portability | 8 |
| Adaptability | 3 |
| Installability | 2 |
| Replaceability | 2 |
| Deployability | 1 |
| Security | 8 |
| Confidentiality | 4 |
| Accountability | 2 |
| Integrity | 2 |
| Authenticity | 1 |
| Performance | 6 |
| Resource Utilization | 5 |
| Time Behavior | 1 |
| Scalability | 5 |
| Compatibility | 3 |
| Interoperability | 2 |
| Co-Existence | 1 |
| Independence | 2 |

[a] Number for Quality Attributes includes Sub-Attributes

Similar to the previously discussed patterns, *Maintainability* is promoted most by 27 of the found best practices. Its sub-attribute *Modifiability* was addressed by 16, and *Modularity* by 11 best practices, followed by *Understandability* (6) and *Reusability* (5). The attribute *Reliability* is the second most mentioned quality with 15 mentions, similar to the patterns (Table 2.5). Its three identified sub-attributes *Recoverability*, *Fault Tolerance*, and *Availability* are also similarly weighted. They are followed by *Portability* and *Security*, both with 8 occurrences. The attributes *Performance* and *Scalability* are much less common among the best practices, with only 6 and 5 mentions, respectively. Again, *Compatibility* and *Independence* were less prominent, as for the patterns analyzed before.

# Part II.

# STATE OF THE ART ANALYSIS

Chapter 3

# State of Practice: Microservices Migration in Industry

Our first contribution aims to take an inventory of the situation that industry is confronted with in the context of migrations to microservices. Hands-on experiences from practitioners provide in-depth qualitative data, but are still rare in scientific literature. Hence, we contribute a qualitative study on *intentions*, *strategies*, and *challenges* in the context of migrations to microservices. We investigate the migration process of 14 systems across different domains and sizes by conducting 16 in-depth interviews with software professionals from 10 companies. Along with a summary of the most important findings, we present a separate discussion of each case. Our subsequent work builds on these foundations that we establish by systematically analyzing the current state of practice.

This chapter extends the following publication:

> J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. "Microservices Migration in Industry: Intentions, Strategies, and Challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2019, pp. 481–490 [FBWZ19]

Next to the research objective targeted in this publication, our conducted interviews also covered related topics that can be found in two additional publications. Insights into microservices characteristics, technologies, and their impact on software quality are presented in [BFWZ19b], while practices and challenges for assuring the evolvability of microservices are presented in [BFWZ19a].

## 3.1. Context: Intentions, Strategies, and Challenges

Several publications discuss the benefits and challenges of adopting microservices [LAO+18; VTR+18], address design aspects [RSS18; SFP+19] or patterns [TLP18; VLC19]. Studies also state that industrial state-of-practice has already reached some degree of maturity, while academia is still at an early stage [PJ16; STV18]. According to Vural et al., it equally applies to empirical studies [VKG17]. Moreover, Jamshidi et al. note that recently published papers have had "*little if any impact on microservice practice*" [JPM+18], which might be caused by limited access to industry-scale applications. A recent mapping study again confirms a strong industry interest in migrating legacy systems to microservices [DLM19].

To fill this gap and to provide additional empirical and industry-focused research, we conducted 16 in-depth interviews with software professionals based in Germany from 10 different companies. We spoke with them about 14 microservices-based systems, nine being in the course of a migration, three having migrated recently, and one greenfield[1] development. Our goal was to

[1] a system developed for a new environment, without concerns for integrating with legacy systems (opposite is denoted as brownfield)

reveal intentions and strategies for such migrations, as well as challenges that companies were faced with. By focusing on the individual cases separately, we aim to also provide rationales and successfully applied practices.

Existing case studies report on microservices migrations in a single company [BDD+18] or domain [LAO+18]. Balalaie et al. extract common patterns from three industrial migration projects [BHJ+18], while Di Francesco et al. aimed to achieve more generalizable insights on migration activities and challenges by consulting 18 practitioners [DLM18]. The results focus on activities and faced challenges per migration phase. Taibi et al. went one step further by deriving a process framework out of 21 interviews with experienced practitioners, who already migrated their monoliths to microservices [TLP17]. They identified maintainability and scalability as main drivers, while decomposing the monolith and splitting the data remained major technical challenges. Similar aspects are covered by Knoche and Hasselbring [KH19]. In their survey among 71 participants, the authors analyze their results by industry sector and identify scalability as the main driver for early adopters, while traditional businesses rather aim for long-term maintainability. Likewise, the survey by Ghofrani and Lübke [GL18] focuses on challenges during a migration.

In summary, two of the studies [DLM18; TLP17] interviewed groups of similar size. With our study, we replicate this basic study design, but also present additional insights and experiences from practitioners through a discussion of each system. In contrast to Di Francesco et al. [DLM18], we concentrate on high-level intentions and strategies rather than activities performed during a migration. As a result, we can substantiate and enhance the aforementioned findings with a holistic view of 14 industrial microservices migration cases that involves organizational aspects as well.

## 3.2. Method: Semi-structured Interviews

We followed a qualitative approach by conducting semi-structured interviews [HA05; Sea08]. Qualitative methods aim at understanding phenom-

ena within their real-life context [Sea08], directly deal with existing complexity, and are useful for addressing the *how* question [BGM87]. This seems appropriate for intentions, strategies, and challenges of a migration scenario, as they are tightly linked to their context. Furthermore, interviews provide the flexibility to probe for further details on specific aspects, allowing us to focus on participants' rationales. We followed the guidelines proposed by Runeson and Höst [RH08], who advocate the execution of five steps: 1) Case study design, 2) Preparation for data collection, 3) Collecting evidence, 4) Analysis of collected data, and 5) Reporting.

**Study design:** Our overarching research objective is to analyze the migration process from monolithic architectures to microservices based on real-world industry systems. To get a holistic view, we aim to identify intentions, applied strategies, and challenges, as well as the interplay between these concepts. To increase the transparency of our findings, we describe the main aspects for each case individually. This objective is framed by the following three research questions:

**RQ1:** What are *intentions* for migrating existing systems to microservices?

**RQ2:** Which migration *strategies* and decomposition approaches do companies apply?

**RQ3:** What are the major technical and organizational *challenges* during a migration to microservices?

**Preparation and evidence collection:** An interview preamble [RH08] was created and sent to the participants beforehand to make them familiar with the study. It explained the interview process and topic, but also aimed to maintain ethical standards by ensuring confidentiality and asking for consent to create audio recordings. To ensure comparability of the interviews, an interview guide for the moderators was set up. It contained the interview questions along with remarks and provided a basic structure.

The first two authors conducted the interviews in the fourth quarter of 2018. Of the 16 individual interviews, six were conducted in person and 10

via remote communication software with screen sharing. Anonymity was ensured, as well as the deletion of the recordings after transcription. During the interviews of ∼45 to ∼75 minutes in length, the prepared guide was loosely followed and adapted based on participants' reactions. Shortly after each interview, the recording was transcribed and sent to the participant for review and final approval.

## 3.3. Interview Case Descriptions

The participants were software professionals in technical roles (developer or architect) with significant professional experience (minimum 5 years) and solid knowledge of service-oriented architectures. We required participation in the implementation of a microservices-based system, primarily in a migration context. The selection of participants was based on convenience sampling via inquiries to several companies in addition to existing contacts within the research group. All participants were based in Germany, even though some of the companies were active in several European countries or globally. Details of the participating companies (C1–C10) of various sizes and domains are shown in Table 3.1. Most of them were software & IT services companies that develop systems for external clients. Altogether, we investigated 14 systems (S1–S14, see Table 3.2), which we discuss in the following case descriptions based on our participants' explanations.

**C1-S1, Derivatives Management System:** C1, a small financial services company, offers a system for the management and search of derivatives as Software as a Service (SaaS). The single-tenant solution evolved to a complex monolith over time. According to P1, it was difficult to make changes in the code due to the high complexity and internal dependencies. The decreasing maintainability made it difficult to release new functionality in a timely manner. Another migration driver was the planned scaling of organizational units. The small team decided for a gradual migration without involving external resources. The team successively extracted functionality from the PHP monolith, starting with parts that did not necessarily need to run

**Table 3.1.:** Company and Participant Demographics

| C-ID[1] | Company Domain | # of Employees | Participant ID / Role | Exp.[2] | S-ID[3] |
|---|---|---|---|---|---|
| C1 | Financial Services | 1 - 25 | P1 Developer | 6 | S1 |
| C2 | Software & IT Services | >100,000 | P2 Lead Architect | 30 | S2 |
| | | | P3 Architect | 24 | S3 |
| | | | P4 Architect | 30 | S4 |
| C3 | Software & IT Services | 26 - 100 | P5 Architect | 20 | S5 |
| | | | P6 Lead Developer | 8 | |
| C4 | Software & IT Services | 101 - 1,000 | P7 Architect | 9 | S6 |
| | | | P8 Architect | 17 | S7 |
| C5 | Software & IT Services | >100,000 | P9 Lead Developer | 7 | S8 |
| C6 | Tourism & Travel | 1,001 - 5,000 | P10 Developer | 9 | S9 |
| | | | P11 Data Engineer | 7 | |
| | | | P12 Architect | 12 | S10 |
| C7 | Logistics & Public Transport | 101 - 1,000 | P13 DevOps Engineer | 5 | S11 |
| C8 | Retail | 5,001 - 10,000 | P14 Lead Architect | 9 | S12 |
| C9 | Software & IT Services | 101 - 1,000 | P15 Architect | 18 | S13 |
| C10 | Retail | 1,001 - 5,000 | P16 Architect | 22 | S14 |

[1] Company ID; [2] Experience in Years; [3] System ID

inside the central instance. This resulted in several smaller services grouping around the core service. No structured decomposition approach was followed. Challenges arose around finding a solution for service discovery, which was achieved by static configuration in the beginning. Eventually, the team found a reasonable solution with Eureka (Spring Cloud). Another challenge was to achieve a certain degree of fault tolerance by "*reacting in a graceful way*" (P1) to failing or missing services. P1 highlighted a significant reduction of time to market for new features as an intermediate migration result.

**C2-S2, Freeway Toll Management System:** The first of three systems developed by the large software consulting enterprise C2 was a management and payment system for freeway tolls. The initial decentralized solution was

**Table 3.2.:** System and Organization Overview

| ID | Purpose | Inception | Timeframe[1] | MS[1] | Ppl.[3] | Team[4] | Process Model |
|---|---|---|---|---|---|---|---|
| S1 | Derivatives mgmt. (banking) | Rewrite | 1.75 (ongoing) | 9 | 7 | 7 | Scrum |
| S2 | Freeway toll management system | Rewrite & Extension | 1.5 to 2 | 10 | 10 (only devs) | 5-10 (<40) | Individual (based on Scrum) |
| S3 | Automotive problem management system | Rewrite & Extension | 2 to 3 (ongoing) | 10 | 50 | 7-9 | Scrum (from Waterfall) |
| S4 | Public transport sales system | Rewrite & Extension | 2 (ongoing, expected 4) | ~100 | ~300 | 6-10 | Scrum, SAFe (from Waterfall) |
| S5 | Business analytics data integration system | Greenfield | 1.5 to 2 (ongoing) | 6 | 7 | 7-9 | Individual (based on Scrum, Kanban) |
| S6 | Automotive configuration management system | Rewrite | 0.5 (ongoing, expected 3) | 60 | 20 (w/o cust.) | 4 | Scrum (from Individual) |
| S7 | Retail online shop | Replace COTS | 2.5 (ongoing) | ~250 | ~200 | 6-8 | Scrum, Kanban |
| S8 | IT service monitoring platform | Cont. Evolution | 2 (ongoing, expected 3) | 9 | 15 | 6-10 | custom |
| S9 | Hotel search engine | Cont. Evolution | 1 to 1.5 (ongoing) | ~10 | ~50 | 3-6 | Scrum |
| S10 | Hotel management suite | Rewrite & Extension | 0.5 to 1 (ongoing, expected 2) | 20 | 50 | 1-5 | Scrum, Kanban |
| S11 | Public transport management suite | Cont. Evolution | 2 to 3 (ongoing) | 10 | ~175 | 5-8 | Scrum |
| S12 | Retail online shop | Replace COTS | 1.5 | ~45 | ~85 | 6-10 | Scrum, Kanban |
| S13 | Automotive end-user service management | Rewrite & Extension | - (ongoing) | 7 | 30 | 5-7 | Scrum |
| S14 | Retail online shop | Replace COTS | 2.5 | ~175 | ~350 | 6-10 | Scrum, Kanban |

[1] Migration Timeframe in Years; [2] Number of (Micro)Services; [3] Number of People Involved
[4] Team Size

replaced by a central microservices-based system with extended functionality. According to P2, the system suffered from degraded maintainability and did not meet the current requirements anymore. The increased amount of transmitted sensitive data led to additional security and privacy concerns. A major technical challenge was finding the right service cut. Here, P2 advocated for a greenfield-like approach to develop the new system: "*Most productive and successful microservices projects that I know are greenfield developments*". He was afraid of the complex monolith and the necessary effort to replace transactions by eventual consistency. Moreover, he was skeptical towards concepts like Domain-Driven Design in this regard. Other major challenges were the integration of services and communication with external parties, as well as test automation. The need to migrate the three core services within a short time resulted in slightly over-sized development teams, which had a negative impact on their efficiency. As a result of the completed migration at that time, the new system fulfilled all requirements.

**C2-S3, Automotive Problem Management System:** The second system from C2 was an automotive management system for the categorization and analysis of problems. The very traditional Java monolith with JSF frontend had many flaws. Users started building their own workarounds using, e.g., Excel. To increase maintainability, the customer decided to rewrite and extend the solution. According to P3, further expectations for the new architecture were scalability and the facilitation of a multi-vendor strategy. However, the customer's decision for microservices would not have been taken very rationally.

The migration was started by rebuilding a user-facing component using current technologies and integrating it with the legacy monolith. This component served as a foundation for successively building a service-based solution along the entire business workflow, with each bounded context forming a service. There was no particular method followed to determine the service cut or achieve appropriate granularity. P3 also notes that the decomposition partially fell short in terms of loose coupling. A change to one service sometimes involved adapting two or three others. Another challenge was

keeping both systems in sync during the migration phase. Here, the initially rebuilt landing page was used to route the traffic between both systems. P3 emphasized the vast amount of new technologies that had to be mastered in a short time, which implied learning efforts on the way. The customer originally followed a Waterfall model, which was gradually replaced by agile processes based on Scrum. The ongoing transition turned out to be a difficult process in this regard. Nonetheless, the agile principles were perceived positively by the customer: P3 found it interesting to see how people were more motivated and sometimes changed their characteristics.

**C2-S4, Public Transport Sales System:** The last system from C2 was a sales system for public transportation. The legacy system was a complex of four monoliths that grew over almost 40 years and provided an enormous set of functionality. The monoliths were not well integrated, which often required duplication of development efforts. Microservices seemed to be the fitting architecture to accommodate the system in manageable chunks. P4 described the chosen approach as something between green- and brownfield. In fact, the system was completely rebuilt, but adjacent systems remained unchanged. The service cutting was guided by the functional areas of the originating system, resulting in ∼100 microservices structured in 15 domains. Hereby, Domain-Driven Design was used without involving additional tools.

The biggest challenge for this large-sized system was the alignment of activities for the 300 developers in 30 teams, e.g., maintaining a common backlog for functionality that affects multiple teams. Likewise, orchestrating the resulting microservices was regarded challenging on the technical side. To mitigate the issues, C2 created "*many guidelines and rules for service creation*" (P4). Still, the resulting service granularity was very different. The process model had been adapted from classic Waterfall to SAFe (Scaled Agile Framework) and Kanban in some teams, which was rolled out one year after the migration started. P4 observed an efficiency increase after a settling-in period of around one more year. According to P4, the decreased number of defects found via manual testing proved the positive impact of test automation in the CI/CD pipelines.

**C3-S5, Business Analytics & Data Integration System:** The small software and consulting company C3 develops a business analytics system for big data that is focused on data integration and analysis via data mining or linguistic algorithms. We interviewed architect P5 and lead developer P6. The company decided to build the central backend in a greenfield approach based on microservices. Intentions were building a manageable, maintainable, and scalable platform, which was seen as a mandatory capability in the *big data* context. The composition of six microservices was designed by the lead architect.

Both interviewees saw building the necessary expertise as challenging for a small team. It would hardly be possible to have dedicated experts for all areas. At times, this situation led to compromises in architectural design for the sake of simplicity, e.g., a single service accommodating the entire data. Another challenge was achieving fault tolerance among services, which led to "*lengthy philosophical and controversial discussions*" (P6) on how reliable the solution needed to be. Furthermore, versioning and security aspects were brought up, whereas the latter ones were crucial for the product and got resolved in a time-devouring process. The team of C5 employed agile methods based on Scrum and Kanban, but got slightly off track due to the pressuring day-to-day business, which sometimes had to be managed in a "traditional way". Implementing a pipeline automation was also hindered by tasks of higher priorities and the lack of experienced personnel.

**C4-S6, Automotive Configuration Management System:** The software services company C4 was in the process of modernizing a very large monolithic configuration management system for an automotive customer. P7 was impressed by its sheer size, which likewise applied to the processes: "*A significant change takes ludicrous 6 months to go into production.*" The customer opted for a microservices architecture for a number of reasons, including extensibility, time to market, scalability of components and teams, avoiding maintenance-downtime, and various new functional requirements.

According to P7, a breakdown of the monolith was not an option due to its huge size, business criticality, and included proprietary components. The

only feasible option was moving the functionality to another platform. To deal with the peculiarities of the system and equally gain the trust of the customer's workforce, the architects chose a two-step approach. At first, all functionality based on read operations was moved. This tremendously reduced the load of the legacy system and would ease the more critical second step. P7 outlined the subsequent extraction of functionality as follows: 1) start at the outer layer with documented and external functionality, 2) gradually move to the inner structure and isolate, 3) coordinate with architect and business unit and 4) extract to a new service. The very comprehensive documentation of functionality was key for this approach. P7 was aware that the achieved cut will never fulfill all requirements, but the earned flexibility in merging/splitting later on would compensate for it. To avoid too fine-grained services, P7 followed the rule "*per microservice a 4-developers team and 3 months to production*".

A major technical challenge was providing 15 TB of data to all consumers in a timely manner and in this regard familiarizing stakeholders with the concept of eventual consistency. The customer also aimed to establish agile processes in conjunction with a mindset change towards a *fail fast* philosophy. Prototyping and rolling releases helped the team members to gain confidence and support the transition towards agile processes. Dissolving the strict separation between business and IT teams was another important step to gain momentum. After a "*bumpy start at day one*" (P7), the customer very much appreciated the tremendous improvements achieved so far, especially in time to market.

**C4-S7, Retail Online Shop:** The second system developed by C4 was an online shop for a large retailer. The existing commercial off-the-shelf (COTS) solution was incapable to fulfill the growing need for new functionality. Moreover, the customer aimed for scalability in terms of system users and development teams. Consequently, a new e-commerce platform based on microservices should replace the existing solution. For this, functionality was gradually re-implemented as services and new ones were added as required. So far, the nearly 200 involved people had created ∼250 services that were

structured by domains and products. Since only use cases were leveraged from the existing solution, P8 described it as a greenfield approach. Integrating the various 3rd-party systems with the new architecture was seen as a challenge. Before the migration, the customer had already established agile processes and lately tended more towards practicing Kanban. Organizationally, P8 saw the main challenge in building a future for existing teams in the new environment and empowering them to deal with new the technologies.

**C5-S8, IT Service Monitoring Platform:** The global IT services provider C5 developed an IT monitoring solution with an operations dashboard. Lead developer P9 described the initial architecture as a RESTful monolith that was currently migrated towards a more fine-grained architecture. Driver for breaking up the system was its quick growth over the last few years. It had a negative impact on maintainability and increased the on-boarding time for new developers. As the system had reached resource limits too, in a first step, a memory-intensive component was isolated and moved to a separate system. The approach of isolating, extracting and splitting up further was repeated for other components likewise. P9 stated that the aspect of service re-usability was considered for splitting, but in general, no systematic decomposition approach was followed. Over a time frame of almost two years, nine services had been extracted that way.

A major technical challenge was setting up the Kubernetes environment with its peculiarities in persistence, logging, and security aspects. P9 termed the applied process model as agile, without an underlying framework. The organization strove towards functional teams with a separation of development, operations, and support. This should help developers to concentrate on their core competencies across several services. External colleagues would compensate for the shortage of skilled personnel by contributing to single services. In this regard, P9 described the knowledge transfer between all contributors as one of the biggest challenges to date.

**C6-S9, Hotel Search Engine:** Developer P10 and data engineer P11 both spoke with us about S9, an online search platform for hotels. The decision to implement microservices resulted from a company-wide strategy. It involved

aiming for cloud readiness and scalability to deal with the increasing and dynamically changing traffic caused by search queries to the system. Apart from performance bottlenecks related to data storage, the monolithic PHP application suffered from a long time to market for new features. In the course of implementing the company's strategy, management approved to rebuild the existing system. The overall strategy followed a gradual rebuild and extension of the entire functionality, based on the 12-factor-app principles [Wig17]. According to P10, the entire system was newly built, except for some special logic. P10 emphasized the achieved flexibility with regard to splitting and merging services and saw a tendency towards more fine-granular services, whereas data engineer P11 was skeptical about this direction and the increased traffic overhead. The switch from a central database to a stream-based architecture was seen as one of the challenges, next to mastering new technologies (e.g. gRPC) and finding best practices for building microservices in general (e.g. deployment or version control). Running the legacy and the new system in parallel during the migration was seen as challenging due to the necessary synchronization.

Teams were realigned in the course of the migration and followed Scrum or Kanban process models. P10 emphasized a missing "*high-level scope*" during the transition, which manifested in frequent unforeseeable changes. In particular, realignments that led to discarding activities right before completion were frustrating for the teams. P11 expressed similar concerns and was also skeptical about politically motivated changes in general. Still, both interviewees rated the migration results, in the form of improved efficiency and collaboration within teams, as positive.

**C6-S10, Hotel Management Suite:** The second system from C6 was a management suite for hoteliers with several modular products. Intentions for migrating the five to six-year-old monolith were similar to S9, whereas P12 would add a tendency towards a modern and trending architectural style. The followed strategy was a functional decomposition with vertical isolation into 20 services, following the Self-Contained Systems (SCS) paradigm. The existing code was transferred to the new system in small chunks. P12 made

experiences with other microservices-based systems already and stated that the decomposition approach always depended on the business case and could not be automated in any way. He made good experiences with learning from the original creators of the system, but also added for consideration that "*it can be luck or a challenge, depending on the guy*" (P12). He explicitly neglected Domain-Driven Design due to its perceived complexity, given the various other challenges that will arise in a migration. It was also important for him to prevent "*hipsterization*": "*Most people that do new stuff, have a tendency to first choose the technology and then make a decision*" (P12).

**C7-S11, Public Transport Management Suite:** Company C7 develops a product suite for managing the whole process of public transportation (S11). DevOps engineer P13 described the ongoing modernization of the whole suite, which is accompanied by occasional team reorganizations and introduction of agile methods. The current modernization efforts of the nearly 30-year-old ecosystem aimed at standardizing and automating deployment processes and simultaneously achieving a more fine-grained architecture. The originally project-driven developed and highly customizable solution showed many symptoms of low maintainability. As a very small team drove the modernization, only new functionality was implemented as separate services by using new technologies: "*Migration without feature development does not happen due to time constraints*" (P13). The goal of the architect group was to let the monolith starve out in the long term, without needing to touch old code again. The integration of new services with the distributed legacy application proved difficult at times. In this regard, the increased testing effort was an issue. As the product was operated by partly "*IT-conservative*" customers, switching to containers was still not yet discussed. Building up the required expertise on a large scale in the established organization was also regarded a major challenge.

**C8-S12, Retail Online Shop:** Retailer C8 was operating a COTS online shop (SAP Hybris). The existing solution was limiting feature development and dynamic scaling. For the decision to develop a custom in-house eCommerce platform (versus updating Hybris), time to market was crucial. According

to P14, this would be an important aspect for online retailers who act in an extremely fluctuating market. While the existing solution remained in production, a greenfield microservices project was started. As *retail* would be a long-established profession, the domain cut was more or less predetermined. For that, the team relied on Domain-Driven Design and event storming techniques. Furthermore, each subdomain comprised several services to achieve an appropriate size for a development team. External experts were consulted to identify best practices and avoid the emergence of anti-patterns. Still, services got often merged or split up again, which, according to P14, improved the system's maturity in a continuous evolution process.

He also recognized drawbacks of the architecture, like low synergies between teams and costly operation. The rather small services also led to a high complexity at the macro level. Still, the improved time to market would compensate for all of these issues. To improve collaboration across teams, the idea of guilds [KI12] was successfully implemented. In this regard, P14 emphasized the importance of an ongoing coaching by external consultants for technical *and* process-related topics. In retrospect, he saw a big threat in taking processes and organizational changes too lightly in such a transition where "*teams were often left alone*" (P14). Despite these challenges, the culture changed towards more open and unconstrained interactions. The established DevOps practices had strengthened the team spirit and eased handling duties or absences of people.

**C9-S13, Automotive End-User Services Management System:** Software and IT services company C9 was in the process of migrating a traditional WebSphere system of an external automotive customer, which handles management and payment of end-user services in cars. As intentions for a microservices architecture, the customer stated cloud readiness and shorter, independent release cycles with zero downtime updates. Accordingly, the initial focus was on containerization and hosting the solution in a cloud environment. Service cutting was postponed and then performed applying the Strangler pattern [Fow04b] for ongoing development.

A major technical challenge was removing the application's session state to achieve statelessness [FFZ+18] and thus enable dynamic scaling. Mastering technologies took a significant amount of time too, according to P15, around 25% of the development efforts. To facilitate distribution of skills and domain knowledge, the cross-functional teams rotate their members from time to time. Decisions by the customer were sometimes politically motivated, like moving from Cloud Foundry to Kubernetes or switching cloud providers. P15 recommended to always be skeptical of technology hypes and criticized the sometimes missing awareness at management level: "*It's for free and from Google, so we go with it*", which would significantly affect the teams' productivity.

**C10-S14, Retail Online Shop:** Similarly to the other two online shops (S7, S12), retailer C10 operated a COTS solution (Intershop Enfinity). The system with 200 instances on 100 manually configured blade servers was technically at its limits: "*Response times were disastrous*" (P17). The heavily customized solution also made it difficult to apply regular updates. A glitch of a wrongly priced offering, which could not be promptly removed from the shop, was the trigger for a modernization. The architects decided to develop a custom in-house solution in a greenfield approach. The architecture was initially based on the concept of Self-Contained Systems (SCS). Over the years, the architecture gradually evolved towards microservices, as P17 put it in a nutshell: "*The more horizontal aspects you remove and the more experience you have, the smaller the services you can build.*"

Referring to the rich body of available literature, P17 saw not too many obstacles regarding technologies and development of microservices, just operating them would become more complex. He regarded organizational challenges in a migration process much more crucial. There was a considerable need for workshops, coaching, and retrospectives to gain the trust of people originally operating the COTS solution: "*A single person can be easily convinced, but a group develops its own dynamics*" (P17). Especially, the move from a project-driven to a product-driven development mode was troublesome. Here, the Scrum framework served well in providing orientation to

teams and management, likewise. P17 emphasized that agility puts people before processes. Accordingly, retrospectives were taken seriously in their teams, with the consequence that most teams shifted from Scrum to Kanban or adapted it to their own needs. In retrospect, the efficiency of the teams and quality of the solution improved dramatically with much shorter release cycles and time to market.

## 3.4. Results and Discussion

In the following subsections, we present the aggregated results for each research question. Table 3.3 summarizes them for all aspects at a glance. In each section (*Intentions*, *Strategies*, *Challenges*), several characteristics were identified on a per-system basis. A cross mark indicates the presence of a certain aspect for the system. Our overview includes aspects that have been identified for at least two systems. Accordingly, we ordered them descending by their overall number of mentions.

### 3.4.1. Migration Intentions (RQ1)

The first section of Table 3.3 reflects the participants' intentions for migrating their systems. It commonly combines reasons for replacing the legacy system and reasons for adopting microservices, which we discuss separately in the following.

Main **drivers for replacing the legacy system** were a lack of maintainability with different symptoms: overview got lost (S1, S10), changes are too costly (S4) or are prone to cause side effects (S11), and bad analyzability (S11). Furthermore, the investigated systems suffered from operability issues like missing traceability (S11), long startup times (S9), downtime during updates (S6), or difficulties to apply updates in general (S14). P2 referred to their system as having reached the end of its lifetime, it just "*didn't work anymore*". Performance aspects including memory consumption and bandwidth bottlenecks were issues for three systems (S6, S9, S14). Several participants stated that current functional requirements could not

**Table 3.3.:** Intentions, Strategies, and Challenges per System

| | | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 | S10 | S11 | S12 | S13 | S14 | Sum[e] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Intentions | Maintainability[a] | × | × | | × | × | × | | × | × | × | × | | | × | 10 |
| | Scalability[b] | | × | | | × | × | × | | × | | | × | | × | 7 |
| | Functional Requirements | | | × | | | × | × | | | × | | × | | | 5 |
| | Operability | | × | | | × | | | | × | | × | | | × | 5 |
| | Company Strategy[c] | | | × | | | | | | × | × | | | × | | 4 |
| | Team Scalability | × | | × | | | × | × | | | | | | | | 4 |
| | Time to Market | | | | | | × | | | × | | | × | × | | 4 |
| | Interoperabiliy | | | | × | | × | | | | | | | | | 2 |
| | Reliability | | | | | | × | | | | | | | × | | 2 |
| | **Process** | **S1** | **S2** | **S3** | **S4** | **S5** | **S6** | **S7** | **S8** | **S9** | **S10** | **S11** | **S12** | **S13** | **S14** | **Sum[e]** |
| Strategies | Rewrite | × | × | × | × | | × | × | | × | × | | | × | | 9 |
| | Strangler Pattern | × | | × | | | × | × | × | | | × | | × | | 7 |
| | Extension | | × | × | × | | × | × | | | × | | | × | | 7 |
| | Parallel Operation | | | × | | | | | | × | | | × | × | × | 5 |
| | Greenfield | | | | | × | | × | | | | | × | | × | 4 |
| | COTS Replacement | | | | | | | × | | | | | × | | × | 3 |
| | Continuous Evolution | | | | | | | | × | | | × | | | | 2 |
| | **Decomposition** | | | | | | | | | | | | | | | |
| | Other (or non-systematic) | × | × | × | | × | × | | × | × | × | × | | | | 9 |
| | Functional Decomposition | | | | × | | × | × | | | × | | × | × | × | 7 |
| | Existing System's Structure | | × | × | | | | | × | × | | × | | | | 5 |
| | Domain-Driven Design | | | | × | | | | | | | | × | | × | 3 |
| | **Technical** | **S1** | **S2** | **S3** | **S4** | **S5** | **S6** | **S7** | **S8** | **S9** | **S10** | **S11** | **S12** | **S13** | **S14** | **Sum[e]** |
| Challenges | Decomposition | | × | × | × | | × | | × | × | × | | × | | | 8 |
| | Lack of Expertise | | | × | | × | | | × | × | × | × | × | × | | 8 |
| | DevOps and Automation | | × | | | × | | | × | × | × | × | | | | 6 |
| | Integration of Services | × | × | | | | | × | | | | × | | | | 4 |
| | Legacy System[d] | | | × | | | × | | | | | × | | × | | 4 |
| | Security | | × | | | × | | × | | | | | | | × | 4 |
| | Fault Tolerance | × | | | | × | | | | | | | | | | 2 |
| | **Organizational** | | | | | | | | | | | | | | | |
| | Mindset Change | | | × | | | × | × | | × | × | × | × | × | × | 9 |
| | Collaboration btw. Teams | | | × | × | | × | | × | × | | × | × | | | 7 |
| | Justification to Mgmt./Cust. | | | × | | × | × | | | | | × | | × | × | 6 |
| | Recruiting Personnel | × | | × | | × | | | × | | × | | | | × | 6 |
| | Central Governance | | | | × | | | | | × | | | | × | | 3 |
| | Volatile Requirements | | | | | | × | | | × | | | | × | | 3 |

[a] incl. Analyzability and Modifiability (7), Size and Complexity of Ex. System (6), Small Units (5)
[b] incl. Traffic Bottlenecks (3)
[c] incl. Cloud Readiness (2)
[d] Complexity, Peculiarities, Deprecated Technologies
[e] Total Number of Mentions per Aspect

be addressed due to deprecated technologies (S6, S11) or could not be implemented within the existing solution by design (S3, S7, S12). Time to market for new features was a driver for several systems (S6, S9, S12, S13). In the special case of S6, the feature lead time for a major change was reported to be up to six months.

The main **drivers for choosing microservices** were the system's scalability (S2, S5, S6, S7, S9, S12, S14) as well as for development teams (S1, S3, S6, S7) which for S3 would also facilitate a multi-vendor strategy. Aiming for smaller, and better manageable and maintainable units was explicitly mentioned for S4, S5 and S11. In some cases, a company-wide strategy prescribed implementing microservices (S3, S9, S10), which was not always perceived as justified: "*It was hip, and so they went for it*" (P3). Achieving cloud readiness was a driver for two systems (S9, S13), as well as for the quality attributes interoperability (S4, S6) and reliability (S6, S13). As just three out of the 14 systems had finished their migration activities (see Table 3.2), results were only partly obtained to that effect. P1, P7, and P16 confirmed faster decisions and a significantly shorter time to market. The number of defects had decreased for P4, while P2 was confident for the system's long-term maintainability.

### 3.4.2. Migration Strategies and Decomposition (RQ2)

The prevalent **migration strategy** for our investigated systems was rewriting the existing application (nine cases) combined with extending functionality (seven cases). The *Strangler* pattern [Fow04b] was used in seven cases to gradually replace the existing system by microservices. P2 experienced that splitting up a complex system would often be too difficult, similarly P7 considered their system as too big for a brownfield migration: "*The best consultant on earth can't grasp what they have built over 10 years.*" Systems S5, S7, S12, and S14 started as a microservices greenfield development, while the latter three also replaced a COTS solution. In five cases, old and new system were operated in parallel, at times with varying load balancing

(S3, S9, S12, S13, S14). Only two systems (S8, S11) were modernized in a continuous evolution process with moderate to low efforts.

The **time frames** for the investigated migrations range from 1.5 to over 3 years, based on the already finished migrations or expected dates (see Table 3.2). In several cases, a large team was assembled for the initial development and later reduced, e.g., S2 formed temporary teams of up to 40 people to shorten the duration. For some larger projects, several contractors were involved, or the system passed through several migration project phases until transitioning into a continuous product development mode.

Outgoing from the question "*What is a service, how big is it and what should it contain?*", a long discussion arose in P12's team. **Decomposition** is the process of splitting up a system or problem space into smaller parts. Seven participants used a functional decomposition approach (S4, S6, S7, S10, S12, S13, S14), as postulated by microservices design principles [LF14]. Even though Domain-Driven Design is often cited in literature to achieve such service cuts [RSS18], only three of them reported its explicit usage (S4, S12, S14). For the remaining systems, the use of a different or non-systematic approach was reported. Here, it was often described as the architect's task or the result of architecture group meetings. For these systems, we can observe a significant overlap with using the existing system's structure as a basis (Table 3.3). In the case of S6, the very extensive documentation of the legacy system was crucial for steering the decomposition, while P12 made positive experiences with learning from the original creators. Microservices-specific refactoring approaches described in academic literature, that partly offer tool support [FBWZ19], were not considered by any of our interviewees. When asked directly, participants were not aware of any such tools (P6, P14) or convinced that there is no way to automate it (P2, P12). Consequently, we could identify the *Wrong Cuts* anti-pattern [TLP18] in four cases (S3, S4, S9, S12). Two participants commented as follows: "*Our service cut was not good, we sometimes needed to touch two or three services for a change*" (P3), "*The code was not split up appropriately, traffic overhead and data volume increased unnecessarily*" (P11).

Furthermore, the *Shared Persistency* (S5) and *Inappropriate Service Intimacy* anti-patterns (S3, S5) were observed as a result of inappropriate cuts. In contrast, other participants appreciated the earned flexibility in merging and splitting services later on (P4, P14, P16), which would be less expensive for fine-grained services. However, our participants consciously aimed for more coarse-grained services in many cases. Reasons might be the mentioned difficulty in finding the right service cut and a tendency to avoid too complex macro architectures, as the case for S6.

### 3.4.3. Technical Challenges (RQ3)

Finding the right decomposition approach and the lack of expertise in the field of building a microservices architecture were reported equally often (eight cases). Here, the main issue was familiarizing with the variety of technologies and tools in a timely manner. The often reported difficulties in recruiting skilled personnel (P1, P3, P6, P9, P12, P16) reinforced the problem. To fully benefit from microservices, DevOps practices like build and test automation need to be established. We observed that many organizations faced challenges in this regard or postponed such activities (S2, S5, S8, S9, S10, S11). While most participants reported a decent degree of automation and the usage of CI/CD pipelines, fully automated continuous deployment existed in only three cases (S9, S12, S14).

For four systems, integrating the services was a challenge (S1, S2, S7, S11). This also involved interoperability with 3$^{rd}$-party software or the existing monolith (S11). Size, complexity, or deprecated technologies of the legacy system were obstacles for S3, S6, S11, and S13. Increased effort for assuring the system's security was reported for four systems (S2, S5, S7, S14). For P1 and P5, building a resilient architecture with fault-tolerant services was seen as challenging.

However, several interviewees (P4, P5, P8, P15, P16) regarded technical aspects not as the main challenge: "*There are always people, maybe external ones, who can master it*" (P8). P4 commented similarly: "*From 30 years of work experience, I can assure that you get the technology right, but you have to*

*bring the people on board*", which brings us to challenges around people and organization.

### 3.4.4. Organizational Challenges (RQ3)

Our investigated migrations in some cases involved a transition from traditional process models towards agile methodologies. A typical example is S6, where the customer's teams needed to undergo a mindset change from Waterfall to Agile. For architect P7, it was important to cautiously introduce people to the new process and technologies: "*People who realize that their constructed systems and achievements will become obsolete, fear losing their position in the new world [...] the most difficult thing is to take along everybody and convince them of the new strategy.*" P16 made similar experiences, while P14 emphasized taking processes and organization not too lightly in such a transition. P11 expressed his perceptions from a developer's point of view: "*Politically motivated changes are always difficult because an organization changes quicker than the software.*" For his team, the progressing restructuring led to a missing high-level scope, volatile requirements, and incoherent instructions.

Next to initiating a mindset change towards agile methodologies, collaboration between autonomous teams was seen as challenging in seven cases. P6 experienced that the vertical cuts caused pain points in horizontal collaboration. For several organizations, the initial months of restructuring caused constant frictions in collaboration (P3, P4, P10). Companies C6, C8, and C10 tried to mitigate the problem by regular coaching.

According to Table 3.2, the vast majority of organizations implemented the Scrum framework. P16 was convinced that it would likely evolve to, e.g., Kanban, if Scrum retrospectives were taken seriously. P8 experienced a similar move towards the *pull principle*. Furthermore, several interviewees emphasized the importance of putting people over processes [BBB+01] (P7, P8, P11): "*Be it Scrum, Kanban, or even Waterfall: the process doesn't matter with the right people and teams*" (P8). In this regard, P7 appreciated a culture where the ability to experiment and fail was valued.

The additional efforts that come along with a migration phase need to be justified to management, customers, or even investors. In six cases, allocating the necessary resources was reported as challenging at times (S3, S5, S6, S11, S13, S14). As microservices rely on decentralization, central governance and prescribed technological or architectural decisions were sometimes seen as problematic, too (S4, S9, S13).

## 3.5. Threats to Validity

Similar to other empirical studies, several limitations have to be considered. One such threat is the sampling method, in our case the recruiting of participants. We paid attention to achieving a variance with regard to industry branches as well as company and system sizes. We believe that all interviewees had sufficient expertise and experience in the field. However, not every interviewee had the same level of involvement in all analyzed areas. Our participants were not afraid to also talk about negative sides, albeit, in some cases, they may not have revealed their true opinion. We tried to mitigate this threat by guaranteeing confidentiality and anonymity. Likewise, interviewees may not have remembered all the details or could have misunderstood our questions. To address this, we sent the transcripts to each participant for review so that they could correct their answers. To alleviate researcher bias and threats to interpretation validity, all interviews were moderated by the first two authors, who also proofread every transcript. In case of any doubt and to avoid potential misinterpretations, the subsequent research progress was discussed in regular meetings. Even though we aimed for objectivity in reporting, some parts may be subjective to our interpretation.

Regarding generalizability, we cannot extrapolate from 14 cases to common patterns, e.g., the use of decomposition approaches. In this qualitative study, we thus focus on participants' rationales and the relations within individual cases. Another limitation could be the geographic location of participants in Germany. Furthermore, the majority of the participants worked

for software and IT services companies. As all of those companies migrated systems for different external customers, we are still confident to comprise a diverse sample in terms of sizes, domains, and cultures.

## 3.6. Conclusion

In 16 semi-structured interviews with German-based participants, we talked about 14 migration scenarios towards a microservices architecture. Investigated aspects were intentions, strategies, and challenges that participants faced in this process. Similar to related studies [KH19; TLP17], maintainability and scalability were identified as main drivers for a migration. Microservices were seen as the appropriate architectural style to address these issues, but were sometimes also prescribed as a company-wide strategy. The unfeasibility to realize new features with the existing system was a driver in five cases.

The majority of systems were rewritten using current technologies, while existing code bases were refactored in just two cases. This confirms a tendency also discovered by Di Francesco et al. [DLM18]. During the migration phase, the old and new system were either operated in parallel, or the *Strangler* pattern was used to successively replace the monolith. For determining the service cut, most companies chose a non-systematic or individual approach. Only three of our interviewees explicitly confirmed the use of Domain-Driven Design, which was not a guarantor for success in other migration cases as well [DLM18]. Our participants did not consider or were not yet aware of specifically tailored and partly automated approaches recently proposed by academia, which are, e.g., based on static code analysis [MCL17], process mining [TS19], or execution traces [JLC+19].

As a second major challenge, we identified the lack of experience with microservices-related technologies and concepts, similar to the findings by Knoche and Hasselbring [KH19]. This was reinforced by the difficulty to recruiting skilled developers, as reported in six cases. We equally focused on organizational challenges, which were considered partly more important by

our interviewees. While technical difficulties were reported across organizations of all sizes, the larger systems predominantly had issues managing the large number of teams and facilitating the collaboration between them. In particular, mature organizations had to cope with initiating a mindset change when simultaneously establishing agile processes.

## 3.7. Subsequent Work

Hasan et al. conducted a systematic review of 47 studies and closely inspected the pre-migration, migration, and post-migration stages in the timeframe of 2015 to 2022 [HOAM23]. The authors confirm our findings of intentions and challenges for migrations to microservices architectures. Among all publications, maintainability was mentioned 28 times as the main driver for migrating legacy applications to microservices, followed by scalability (24), flexibility (12), and availability (10). As the most prevalent migration challenges, the authors identified service (boundary) identification with 20 mentions, followed by lack of migration guidelines (13), refactoring/rearchitect (11), legacy complexity (9), and artifacts/documentation (6).

Chapter 4

# State of Practice: A Case Study in the CPS Domain

Our second contribution, a mixed-method empirical study on the state of practice, aims at evaluating the state of microservices migrations in the domain of cyber-physical systems (CPS). This domain has recently seen strong growth due to the rise of the Internet of Things (IoT). The process of digitization in industrial production environments is also referred to as "Smart Manufacturing" [LJ17] or "Industry 4.0" [TVS18].

To summarize the current state of research, we conducted a rapid review based on 146 scientific papers on microservices and DevOps in the CPS domain. We subsequently validated our findings in an interview-based case study with 9 CPS professionals from different business units at Siemens AG in the course of an industry-academia collaboration. While several differences in this domain may require adapted methods, most of the challenges and practices are common to typical enterprise applications. By focusing on challenges, applied practices, and migration rationales, the contribution expands and strengthens our foundation established in the previous chapter.

From the interviews, we further derive requirements for systematizing and automating the migration process. In addition, we examine applied DevOps practices that typically accompany the adoption of microservices.

This chapter extends the following publication, to which the two first authors contributed equally:

> J. Fritzsch, J. Bogner, M. Haug, A. C. Franco da Silva, C. Rubner, M. Saft, H. Sauer, and S. Wagner. "Adopting microservices and DevOps in the cyber-physical systems domain: A rapid review and case study". In: *Software: Practice and Experience* (Nov. 2022), pp. 1–20 [FBH+22]

## 4.1. Context and Goals: Microservices in the CPS Domain

A major technological foundation for the ongoing Industry 4.0 transformation in the CPS domain is the Internet of Things (IoT) [MRM21], or Industrial Internet of Things (IIoT). Such systems are characterized by the integration of mechanical and electrical devices that represent smart endpoints. Recent developments also utilize digital representations (digital twins), in particular for simulations in manufacturing systems [CMR+20]. Moreover, the digital representation and management of involved industrial processes aims at increasing flexibility and evolvability [IGP+21]. Gartner estimates the growth of IoT endpoints from 331.5 million in 2018 to 1.9 billion in 2028 [Gar20]. This massive growth of involved devices poses new challenges for their efficient management. Mass-customization and unpredictable workloads require re-configurable and highly automated platforms [CMR+20], which in turn has consequences for software architectures and development processes. Combemale and Wimmer express the need for more comprehensive views on the development and operation of such environments [CW20]. Similarly, Broy and Schmidt state a demand for industrial software engineering practices, i.e., the systematic development of high-quality cyber-physical systems (CPS) [BS14].

Even though the concept of microservices and DevOps originated in the context of enterprise applications, it is also regarded as promising approaches for the IoT and CPS domain [BPRS18]. Microservices and DevOps have similar objectives regarding the aforementioned qualities [WLS20]. Within an agile development process, the combination of microservices with DevOps practices promises higher flexibility, shorter release cycles, and better maintainability, which are highly desirable for CPS manufacturers as well. However, cyber-physical systems and the IoT domain differ from common enterprise systems in certain aspects:

- Strong focus on hardware with a virtual representation (digital twin) [BS14]
- Distributed systems with heterogeneous communication mechanisms [TS18]
- Large amounts of generated data with real-time analysis requirements [JHT18] that demand for computational resources like processing power, storage, and network bandwidth [AGCC21]
- Strict regulations and requirements on safety and compliance [LG18]

In their general CPS research roadmap, Törngren and Grogan [TG18] discuss the inherent complexities of these systems, including causes, effects, and limitations of existing methodologies. Similarly, Fahmideh et al. [FAB+21] acknowledge the lack of research on the development lifecycle for IoT systems and conducted a mixed-method study to derive a process framework. While it includes essential tasks from a software engineering perspective, it does not cover DevOps and microservices.

Campeanu contributed an early mapping study on microservices architectures in the context of IoT and cloud computing [Cam18]. He cataloged 364 primary studies from 2015 to 2018, visualizing the rapid emergence of this technological combination. In a more holistic review, Joseph et al. discuss the use of microservices in specific domains [JC19]. They found 12 studies proposing specific solutions for the IoT domain, including related fields like healthcare or smart living. The authors emphasize that "*the major challenge*

*faced in developing IoT solutions is having to deal with a plethora of devices that are heterogeneous in nature*" [JC19]. Similar findings were revealed in the review of 26 primary studies by Pereira et al. on the use of DevOps in IoT systems [PCF21].

Hasselbring et al. aim to mitigate lacking expertise in the adoption of Industry 4.0 solutions by proposing an approach to "*introduce methods and culture of DevOps into industrial production environments*" [HHL+19], especially among small and medium-sized enterprises. Taibi et al. focus on the migration aspect towards microservices and DevOps in general in their systematic mapping [TLP19], but cover only one case in the IoT domain.

In summary, several studies exist on either microservices and DevOps *or* the CPS domain. However, a comprehensive analysis of challenges and practices at the intersection of these three concepts is still missing. We aim to close this gap by providing foundations for modernizing existing CPS. While existing microservices and DevOps practices for enterprise applications might be applicable for CPS systems to a great extent, their weighting may differ, besides the necessity for additional variations and adaptions. Hence, in this study, we aim to analyze the use of microservices and DevOps in the CPS domain, with particular emphasis on the migration of existing legacy systems. The following research questions frame this objective:

**RQ1:** What are *challenges* related to microservices and DevOps in the context of cyber-physical systems?

**RQ2:** What are *practices* related to microservices and DevOps in the context of cyber-physical systems?

**RQ3:** What are *rationales* why CPS professionals want to migrate their systems towards microservices and DevOps?

**RQ4:** What are *requirements for tool support* to facilitate the migration to microservices and DevOps in the context of cyber-physical systems?

As the first part of our mixed-method empirical study, we systematically analyzed scientific literature. Second, we validated and extended the results

through an interview-based case study with industry experts from different units at Siemens AG[1] — a global technology enterprise with extensive CPS expertise in several domains, ranging from manufacturing, healthcare, or mobility to energy.

## 4.2. Method: Rapid Review and Interview-Based Case Study

To answer the RQs, we formed a team of academic software engineering researchers and industry researchers from Siemens AG, who are experts on the CPS domain. We decided to first conduct a literature review to analyze the scientific state of the art for microservices and DevOps in the context of CPS (see Section 4.2.1). The results were then compared to similar publications on microservices in the enterprise application domain. Afterwards, we conducted an interview-based case study at Siemens AG to compare the literature findings with industry experiences (see Section 4.2.2). The results from both studies were holistically analyzed, discussed within the research team, and presented to a larger audience at Siemens AG for additional feedback. This research process is visualized in Fig. 4.1. We share the most important study artifacts as a digital appendix via Zenodo[2].

### 4.2.1. Rapid Review

Structured approaches to literature surveys like systematic literature reviews [KC07] or systematic mapping studies [PFMM08] provide a rigorous and reproducible process, but also require considerable effort. Additionally, their results can be hard to integrate into industry practice. We therefore decided to conduct a *Rapid Review* [CPS20], a lightweight secondary study performed in close collaboration with industry and focused on providing timely and easily consumable results. The protocol of a rapid review is still systematic, but may consciously sacrifice rigor and extensiveness for industry

[1] https://www.siemens.com

[2] https://doi.org/10.5281/zenodo.6536538

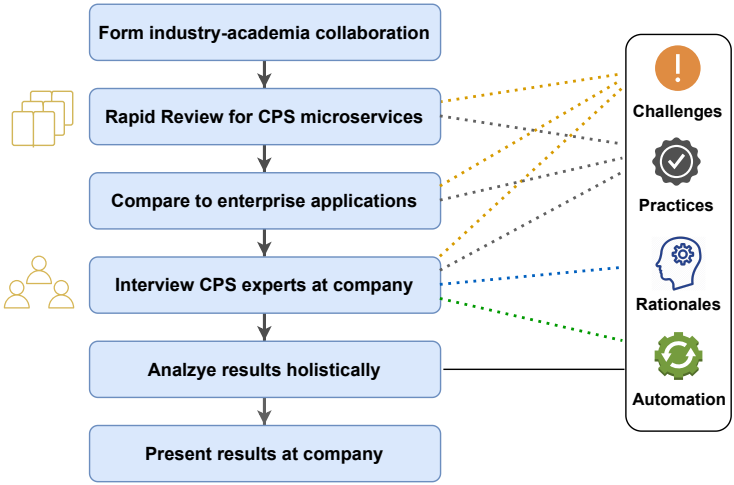



**Figure 4.1.:** General Research Process

relevance and efficiency. The first four authors of [FBH+22] conducted the review, with the rest engaging in discussions.

**Search Strategy** By analyzing known relevant literature, discussing important keywords, and piloting alternative terms, we eventually decided to use the following two search strings, one for "microservices" and one for "DevOps":

1. `microservices ∧ (cps ∨ cyber-physical ∨ manufacturing ∨ industrial ∨ edge)`
2. `devops ∧ (cps ∨ cyber-physical ∨ manufacturing ∨ industrial ∨ edge)`

An `AND` relation was used to combine each main term with CPS keywords, of which one had to match as well. Both search strings were executed independently. To reduce false positives, the query was only executed for the title field. We selected the following five common digital libraries / academic search engines:

- Google Scholar
- ACM Digital Library
- IEEE Xplore
- ScienceDirect
- SpringerLink

To ensure a manageable number of results, we defined a stopping criterion: only the first 50 publications per search string and source would be considered, i.e., a maximum of 500 publications (2 search strings × 5 sources × 50 hits). After data extraction, one round of backward and forward snowballing [Woh14] (reference and citation search) complemented this initial search. The rationale for this strategy was to combine sufficiently broad coverage with manageable effort.

**Study Selection** We used the web-based tool Rayyan[1], which offers convenient features to include or exclude studies in a multi-user mode. Each paper was assigned to two researchers, who performed the selection independently and compared the results. Disagreements were discussed within the whole group until consensus was reached. Papers were **included** based on covering at least one of the following criteria:

- CPS challenges associated with microservices or DevOps
- A practice to develop CPS using microservices or DevOps
- A practice to migrate CPS towards microservices or DevOps

Practices include, e.g., processes, techniques (e.g., for system migration), platforms, reference architectures, algorithms, quality assurance practices, etc. Additionally, a paper could also be **excluded** for one of the following reasons:

- Short paper with four pages or fewer
- Full-text not available to us
- Not published in English
- Published before 2015

[1] https://rayyan.ai

- Not peer-reviewed scientific literature (e.g., books, student theses, etc.)
- Duplicate or extension (the more recent paper is kept)

**Data Extraction** Included papers were exported into a spreadsheet with columns for challenges and practices plus additional columns for categorization, e.g., what part of the CPS domain the paper focused on. To validate the sheet, all four researchers independently extracted the first five papers. Afterwards, results were compared, which further improved the spreadsheet and led to more concrete extraction guidelines. We repeated this process a second time, after which we felt confident that our shared understanding led to reasonably consistent extractions. For efficiency, we therefore assigned each remaining paper to a single researcher for extraction. If it was discovered during extraction that a paper did not completely fulfill the selection criteria, it was excluded.

**Snowballing** All successfully extracted papers were used in one round of snowballing. For backwards snowballing, the references of all papers were automatically extracted from the PDF via a tool[1]. The references were deduplicated, and equally distributed among the four researchers for selection. For forward snowballing, the included papers were again split up, and each researcher examined their citations via Google Scholar up to a maximum of 50 citations per paper. All newly proposed papers were examined by the other researchers and only kept if nobody objected. Afterwards, we extracted the snowballed papers following the same process as described above.

**Data Synthesis** The final extractions were then holistically analyzed to synthesize answers to our RQs. For basic columns like the CPS domain or targeted software engineering activities, little effort was required to consistently aggregate them. However, since challenges and practices were extracted via direct paper quotes, we had to apply thematic synthesis [CD11]. We first assigned codes to each individual extraction (*open coding*), and afterwards synthesized these codes into higher-level themes (*axial coding*).

[1] https://github.com/helenocampos/PDFReferencesExtractor

This was a very iterative process, with lots of discussion and the frequent merging or splitting of codes and themes. The final codes and themes were aggregated and visualized using charts.

### 4.2.2. Interview-based Case Study

To validate and enrich the findings from the rapid review, we conducted an interview-based case study at Siemens AG. A case study [RH08] is a research method to deeply analyze something in a real-world setting (a *case*), usually following a qualitative and exploratory methodology. While a case study can provide rich insights into a topic, the generalizability of the produced results may be limited. We conducted an *embedded case study* [RH08], i.e., our *case* was Siemens AG as a CPS manufacturer and our *units of analysis* were individual cyber-physical systems in different organizations at Siemens AG. These systems were studied through the experiences and opinions of their stakeholders, i.e., interview participants in various roles with sufficient industry experience (at least 3 years). Ideally, their systems should also be in the process of migrating towards microservices and DevOps, or such a migration is considered for the future. We used convenience sampling via our personal networks at Siemens AG to find matching participants. For flexibility, we chose *semi-structured interviews* [Sea08], which were conducted by the first three authors of [FBH+22].

**Study Preparation** We created several artifacts for the interviews. An *interview preamble* [RH08] explaining the process was used to recruit participants and to make them familiar with the study. The preamble also described ethical considerations and requested consent for audio recordings. Additionally, we created an *interview guide* [Sea08] with the most important questions. This guide was used to loosely structure the interviews. We also prepared a small slide set with additional information for certain questions. Lastly, we created a *case characterization matrix* [Sea08] with basic attributes for the analysis.

**Evidence Collection** We conducted nine individual interviews, all of them via remote communication software. Three interviews were held in English and six in German. Interviews took 45 to 70 min, and all participants agreed to a recording. At least two researchers were present during each interview, and swapped between moderator and note writer. After a brief introduction, we showed a slide with microservices characteristics to ensure the same basic understanding. We then inquired basic demographics, and let participants describe their system from the CPS domain. Afterward, we asked about challenges and practices related to microservices and DevOps, specifically in a migration context. Additionally, we asked participants about their rationales for migrating their CPS towards microservices and DevOps, and requirements for tool support in the process. To support participant reflections, we briefly introduced the draft of a migration framework created by us (see Chapter 7 and [FBH+24]), where participants could describe their experience in each step.

We manually transcribed each recording into a text document. These documents were then sent to interviewees for review and approval. Participants could delete sensitive content or adapt unclear statements. We then used the approved transcripts for qualitative analysis.

**Data Analysis** To synthesize the interview findings, we used card-sorting [Zim16b], a lightweight, collaborative analysis technique. Instead of paper cards, we used a web-based tool with feedback cards for agile retrospectives[1]. Each interview block was assigned to one researcher, who then extracted relevant quotes per transcript into individual cards. In an iterative process, these cards were then sorted according to their similarity, and preliminary labels were assigned. Lastly, the results were presented to the other researchers. Together, we refined the labels and formed higher-level categories. Finally, important findings were extracted into spreadsheets for aggregation, and visualized.

[1] https://metroretro.io

## 4.3. Rapid Review Results

In this section, we present the results of the rapid review (challenges and practices). In total, we included 146 primary studies in the review, 87 of which were discovered in the initial search and another 59 during snowballing (see Fig. 4.2). The papers were published between 2015 and 2021, and covered all phases of the software engineering lifecycle, with *design* being the major focus (mentions in 127 papers), followed by *deployment* (44) and *operation* (31).

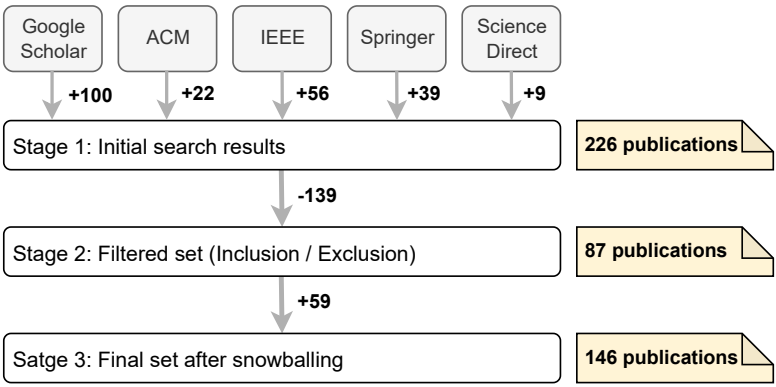



**Figure 4.2.:** Rapid Review Process: Number of Publications per Stage

### 4.3.1. Challenges (RQ1)

From all 146 primary studies, we extracted 405 statements, which were converted into 46 challenge labels in 11 categories during the analysis. The 46 challenges were mentioned a total of 490 times among all statements. A challenge was counted at most once per paper. Fig. 4.3 shows the 11 challenge categories with their number of mentions as orange bars, while Table 4.1 shows the 10 most frequent challenge labels.

The dominant category was *heterogeneity*, with *technological heterogeneity of devices* (48) and *variety of data formats and communication protocols* (24)

being the most mentioned challenges in it. Numerous devices also pose challenges for *operations & deployment*. In this category, the management of components like the *efficient placement, scheduling, and load balancing of microservices* (18) were prevalent issues. The third most popular category was *performance efficiency*, which contained the second and third most mentioned challenges overall: *realtime processing requirements* (31) and *limited hardware resources of devices and edge nodes* (29).

The *large amount of data to process* (25) together with *complex data storing and processing requirements* (18) highlight prevalent challenges for *data management & processing*. In general, we observed a strong tendency towards hardware-related aspects, and the management and operation of a multitude of distributed components. One of the major aspects here was *establishing connectivity and interoperability* (19), which was the most mentioned challenge in the *integration & communication* category. The *architecture & design* category was mostly driven by the *limited evolvability of traditional architectures like monoliths and SOA* (21), which expresses the need for more suitable architectural patterns.

The *security* category, with issues around *data privacy and authorization for a multitude of components* (12) and *strong requirements for security* (9), also received considerable attention. *Agility & business requirements* pose challenges on CPS as well, as the two most frequent challenges in this category reflect: *highly dynamic and uncertain environments* (10) and *quickly emerging technological advancements require more flexibility* (8). The *dependability* category was dominated by challenges around *strict safety and compliance regulations* (10), and *strong requirements for reliability* (8). Organizational and process-related challenges, however, played a minor role. The most mentioned challenge in this category was *adapting IoT/CPS processes and philosophy to DevOps* (6).

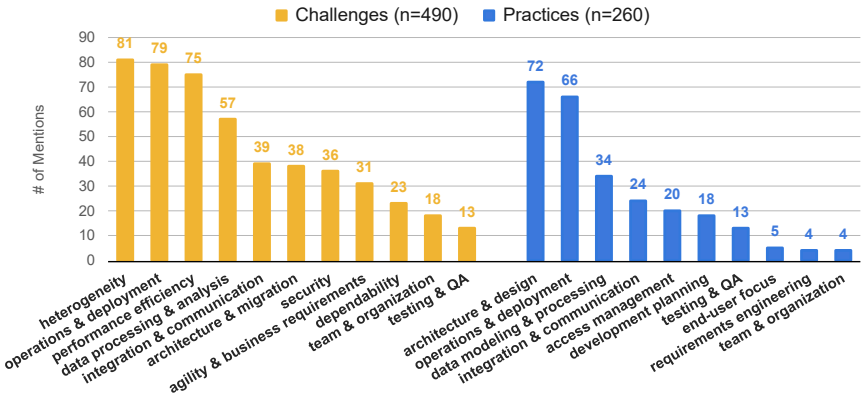



**Figure 4.3.:** Rapid Review: Challenge and Practice Categories

### 4.3.2. Practices (RQ2)

For practices, we extracted 283 statements, which were condensed into 32 labels in 10 categories. The 32 practices were mentioned a total of 260 times among all statements.

As with challenges, a practice was counted at most once per paper. The first of two dominant categories was *architecture & design*, with specific *architecture proposals for various application contexts* (55) being the most frequent label, e.g., microservice architectures for industrial control systems (ICS) or cloud robotics. Other practices in this category were rather insignificant. The second dominant practice category was *operations & deployment*, with the top label being *techniques for placement, scheduling, allocation, deployment in various domains* (35), often for edge-cloud continuums. Other labels received considerably fewer mentions: the second most addressed practice was targeting *techniques or patterns for monitoring of microservice-based systems* (12).

All other categories received notably less attention. *Data modelling & processing* was the third most popular category, with practices centered around *microservices for data processing and analytics on IoT/edge nodes* (18) or *conceptual modeling & model-driven engineering* (16). Several papers

also proposed practices for *integration & communication* aspects, such as *techniques and patterns for microservice registration and discovery in IoT/IIoT environments* (7), followed by techniques for *autonomous/automated device management and reconfiguration* (5). *Access management* centered around various *security techniques for IoT/edge systems* (16). *Development planning* was dominated by practices for *DevOps adoption in various application contexts* (15), such as ICS or IoT. While several DevOps-related aspects are also covered by other categories and labels, this label focuses on more general, process-related facets. Finally, *testing & QA* solutions received relatively little attention (13), similar as with challenges in this area. Most practices in this category were targeting simulation and performance testing.

In general, practices were more condensed within a few categories, dominated by concrete architecture proposals and techniques for the dynamic scheduling of microservices within edge/fog/cloud computing. Organizational and process-related solutions received noticeably little attention in our review.

**Table 4.1.:** Most Mentioned Challenges and Practices (by Number of Papers)

| **Challenge** | # | **Category** |
|---|---|---|
| technological heterogeneity of devices | 48 | heterogeneity |
| realtime processing requirements (response time of cloud microservices too high) | 31 | performance efficiency |
| limited hardware resources of devices and edge nodes | 29 | performance efficiency |
| large amount of data to process | 25 | data processing & analysis |
| variety of data formats and communication protocols | 24 | heterogeneity |
| evolvability is limited by traditional architectures (monoliths, SOA) | 21 | architecture & design |
| efficient management of a large number of components (microservices, devices, etc.) | 20 | operations & deployment |
| establishing connectivity and interoperability | 19 | integration & communication |
| complex data storing and processing requirements | 18 | data processing & analysis |
| efficient microservices placement, scheduling, and load balancing | 18 | operations & deployment |
| **Practice** | | |
| architecture proposals for various application contexts (ICS, IIoT, cloud robotics, healthcare, manufacturing, etc.) | 55 | architecture & design |
| techniques for placement, scheduling, allocation, deployment in various domains | 35 | operations & deployment |
| microservices for data processing and analytics on IoT/edge nodes | 18 | data modelling & processing |
| conceptual modeling and model-driven engineering | 16 | data modelling & processing |
| security techniques for IoT/edge systems | 16 | access management |
| DevOps adoption in various application contexts (CPS, ICS, IoT) | 15 | DevOps process |
| techniques & patterns for monitoring of microservice-based systems | 12 | operations & deployment |
| techniques, tools, patterns for efficient cloud/edge deployment | 8 | operations & deployment |
| techniques & patterns for microservice registration and discovery | 7 | integration & communication |
| guidelines for determining microservices boundaries | 6 | architecture & design |
| microservices architectural style to meet requirements of IoT/edge gateways | 6 | architecture & design |
| orchestration of services / containers for automated deployment in IoT/edge solutions | 6 | operations & deployment |
| simulation of IoT environments / applications to facilitate testing and evaluation | 6 | testing & QA |

## 4.4. Interview Results

In this section, we present the results of the expert interviews (challenges, practices, rationales, and automation requirements). In total, we conducted nine interviews with participants from different business units across Siemens AG. Table 4.2 lists their demographics. The experience column gives their professional experience in years, with the amount of microservice experience in parentheses. Except for P7, all participants had more than 10 years of professional experience. However, their prior experience with microservices and DevOps varies considerably, with two participants stating no prior experience, and two to six years for the rest. The interviewees described seven different systems across a variety of fields. Table 4.3 shows their purpose, size, staffing, and the projected migration timeframe. As we can see, even the "*smallest*" system (S6) has still a substantial size with over 100,000 lines of source code.

**Table 4.2.:** Interview Participant Demographics

| ID | Role | Experience (Microservices) | System |
|---|---|---|---|
| P1 | Architect, Researcher | 18 (3) | S1 |
| P2 | Architect | 19 (3) | S2 |
| P3 | Software Development Lead | 20 (6) | S3 |
| P4 | Architect | 15 (2) | S4 |
| P5 | Architect | 25 (0) | S5 |
| P6 | Architect | 16 (0) | S4 |
| P7 | Researcher | 6 (5) | S6 |
| P8 | Project Manager | 14 (4) | S6 |
| P9 | Project Manager | 15 (5) | S7 |

**Table 4.3.:** System and Migration Properties

| ID | Purpose | Size | Staffing | Timeframe | Targeted Architecture |
|---|---|---|---|---|---|
| S1 | Manufacturing Management System | n/a | 50 (5 teams) | 2020– | aspects of microservices |
| S2 | Engineering of Automation Systems | 70 MLOC | 300 | 2018–2026 | aspects of microservices |
| S3 | Digital Healthcare Services Platform | 100 MS | 250 (25 teams) | 2017–2023 | full microservice architecture |
| S4 | Digital Healthcare Services Platform | 15 MLOC | 300 (30 teams) | 2019– | full microservice architecture |
| S5 | Control Software for Electric Motors | 5 MLOC | 100 | 2011– | modularization |
| S6 | Digital Twin Platform for Power Distribution Equipment | 150 kLOC | 10 | 2021– | full microservice architecture |
| S7 | Process Control Engineering | 600 kLOC | 250 (12 teams) | 2021–2023 | aspects of microservices |

All systems were in the process of a migration. However, the expected migration timeframes differed considerably, ranging from two years (S7) to over a decade (S5). Often, no distinct end date was mentioned, e.g., for S1, S4, or S6. For some systems, the reason was an ongoing evaluation of the feasibility of a full migration. In other cases, the migration followed a continuous evolution strategy, where components were slowly migrated towards microservices when they had to be changed in the course of day-to-day development. We also discussed the architecture that should be achieved with the migration, which varied regarding the commonly understood characteristics of microservices [LF14], i.e., the characteristics we introduced at the beginning of the interviews. For three systems, the targeted architecture was a full microservice architecture that relied on almost all characteristics. For another three systems (denoted with *aspects of microservices* in Table 4.3), the targeted architecture fulfilled some microservices characteristics, but not enough to be considered a full microservice architecture. That means, S5

was the only system where the team mainly aimed to achieve a higher modularization. All other systems set out for a partial (i.e., some characteristics) or full microservices architecture (i.e., nearly all characteristics).

### 4.4.1. Challenges (RQ1)

Our interview analysis revealed 26 unique challenge labels, from which we could reuse 16 from the literature review, i.e., 10 were newly created (38%). These individual challenges were grouped into 10 categories, all of them reused from the review. The most prominent categories were *architecture & design* (9 mentions), *team & organization* (8), *integration & communication* (5), and *performance efficiency* (5). Categories like *operations & deployment* (3), *MDE & digital twins* (2), or *security* (1) were less prominent. Furthermore, individual labels were quite spread out, with 13 challenges being mentioned only once and no challenge being mentioned for more than 3 of the 7 systems, i.e., there were no "top" challenges affecting most CPS in our sample. The category distributions are shown in Fig. 4.4, while the most mentioned challenges are presented in Table 4.4. For the top categories, we present further details below.

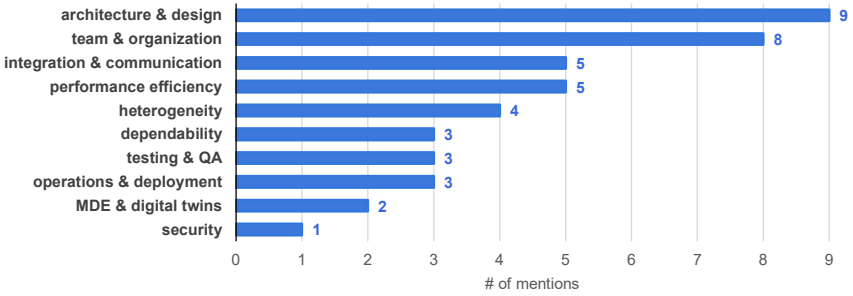



**Figure 4.4.:** CPS Interviews: Distribution of Challenge Categories

**Architecture & Design** All systems except S1 reported at least one challenge from this category. The most mentioned label was *reducing coupling or*

*hidden dependencies* (S3, S4, S6), i.e., participants found it challenging to untangle elements in preparation for the migration or service decomposition. Another challenge in this area was *technical debt of legacy systems* (S2, S5), which would slow down a migration considerably. All other challenges like *lack of architecture documentation* (S6), *massive system size* (S2), or *incorporating legacy components* (S6) were only mentioned once. Lastly, only one label (*limited evolvability due to traditional architectures like monoliths or SOA* from S7) was reused from the review, as most mentioned challenges here were not strongly specific to CPS.

**Team & Organization** Challenges related to human factors were also very prominent. For S4, S5, and S7, the *lack of personnel with modern technology experience* was problematic, i.e., many CPS engineers would know little about microservices or DevOps. Project manager P9 remarked that "*architects with the required knowledge are currently most wanted in the project and are therefore somewhat of a bottleneck*". Related to that, *building developer confidence in microservices and DevOps* was sometimes challenging (S5, S6), and could lead to "*trust issues*" (researcher P7) if not properly managed. For very large systems, a challenge not mentioned in the literature was *creating a shared system understanding with all teams* (S2, S4). Lastly, *adapting the CPS processes and philosophy to microservices and DevOps* was problematic for S7 — P9 saw the need for a cultural change while moving to DevOps and cross-functional teams in his department.

**Integration & Communication** This category was only comprised of two labels. However, they were mentioned for two and three systems respectively, and both appeared in the literature review. For S1, S4, and S6, *establishing connectivity and interoperability* between the newly distributed services was difficult. Architect P1 saw "*the challenging aspects of communication via message-passing*" as the main problem in the migration. Related to this was the challenge to *maintain predictability and data consistency within distributed processes* (S1, S4). Eventual consistency was not acceptable in some use cases, so creating "*several services that are independent yet for which*

*we also need to guarantee data consistency*" (P6) was difficult and therefore sometimes completely avoided.

**Table 4.4.:** Challenges Affecting Two or More Systems

| Challenge | # Systems | Category | in RR |
|---|---|---|---|
| reducing coupling / hidden dependencies | 3 | architecture & design | **no** |
| establishing connectivity and interoperability | 3 | integration & communication | yes |
| realtime processing requirements | 3 | performance efficiency | yes |
| lack of personnel with modern technology experience | 3 | team & organization | yes |
| technical debt of legacy system | 2 | architecture & design | **no** |
| strong requirements for reliability | 2 | dependability | yes |
| technological heterogeneity of devices | 2 | heterogeneity | yes |
| variety of data formats and communication protocols | 2 | heterogeneity | yes |
| predictability and consistency in distributed processes | 2 | integration & communication | yes |
| limited hardware resources of devices and edge nodes | 2 | performance efficiency | yes |
| building developer confidence in microservices & DevOps | 2 | team & organization | yes |
| creating shared system understanding with all teams | 2 | team & organization | **no** |
| creating new decoupled test suites | 2 | testing & QA | **no** |

**Performance Efficiency** In this category, two challenges were mentioned, both of which could be reused from the literature review. As expected, several CPS (S1, S4, S5) had strong *real-time processing requirements*, which needed to be kept intact during and after the migration. This meant that the response time of cloud-based microservices was often not sufficient, e.g., for "*functionalities that need to be close to the production line*" (P1). For some use cases, interprocess communication between microservices was not feasible at all, or as described by architect P5: "*we need to calculate cycles for*

*electric current regulators within a two-digit microsecond window.*" A related challenge were the *limited hardware resources of devices and edge nodes* (S4, S5). This could also impact latency or scalability, but mainly limited using resource-intensive technologies like containerization.

### 4.4.2. Practices (RQ2)

We identified practices in the following categories, each listed with the number of mentioned practices: *team organization* (12 mentions), *process strategy* (12), *tool support* (11), *service identification* (10), *DevOps* (8), and *architecture & infrastructure* (7). As we provided the interviewees with a stepwise migration methodology created by us, the practice categories were prescribed to some extent. Fig. 4.5 shows the categories we queried about with the number of mentions respectively. We discuss them in detail below.

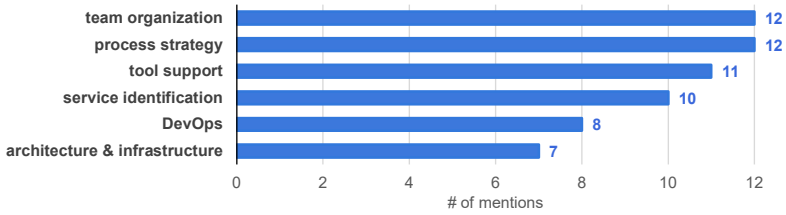



**Figure 4.5.:** CPS Interviews: Distribution of Practice Categories

**Team Organization & DevOps** Six participants reported an associated team re-organization during their migration. The prevalent motivation was promoting team independence and fostering collaboration between teams, mentioned four times. Two participants reported no organizational changes (P5, P6), and also stated that establishing DevOps practices was not a priority yet. Reported DevOps practices were establishing CI/CD pipelines (S1, S2, S3, S6) and using cloud platform services (S1, S3). Four interviewees had already established a DevOps culture. Two aimed to do so (S4, S7), but were hindered by technical or organizational obstacles. One participant remarked

that team re-organization was a sensitive topic in Germany, and related it to the difficulty of DevOps adoption in large enterprises.

**Process Strategy** The migration of a legacy system can follow different strategies. For brownfield developments, we distinguish between a *re-build* and *re-factor* type. The former is often used when the system is rebuilt using newer technologies and was applied for systems S1, S2, and S4. Sometimes, even a mix of several strategies can be used for different parts or subsystems, as was the case for S2, S4, and S6. We can further distinguish between a *big bang* migration that aims to minimize the duration, and a continuous evolution approach that tries to minimize the needed resources. The latter was used for four systems (S4, S5, S6, S7), and often based on the popular *Strangler* pattern [Fow04b] (S2, S3, S4). Gradually building the new system with parallel operation next to the existing one was reported for S4 and S6. As rationales, participants stated limited resources, time pressure, and strong requirements for stability and compliance.

**Service Identification & Tool Support** Existing literature identifies the decomposition task as the most complex one in a migration scenario [FBWZ19]. In a similar notion, P1 stated that "*success or failure will depend on the services having good granularity or not*". Five of our participants described service decomposition as a manual task. P1 and P3 reported Domain-Driven Design as the used technique, others mentioned static code and metadata analysis (P2, P9), as well as dynamic analysis (P5) of the existing system. Our participants used a variety of mostly commercial tools that support the system comprehension via static code analysis (P2, P3, P6, P8). However, a subsequent automatic decomposition is often beyond their scope. For this purpose, P5, P6, and P9 relied on self-developed tools, two involved external consultants to compensate for missing in-house expertise (P1, P5), and another two did not use any tools for this purpose (P4, P7).

**Architecture & Infrastructure** Three participants mentioned workshops as a practice to support the architecture design and infrastructure choices (P2, P3, P4). Conducted cross-team and involving all system stakeholders,

they would facilitate the build-up of a common understanding and the identification of requirements. P6 expressed his interpretation of agility in the following way: "*Big upfront design is a no-go. Just as much architecture as necessary.*" For the preferred cloud model, platform as a service (PaaS) hosting was mentioned twice. On-premise hosting was a strong requirement for S7 due to its tight integration with physical devices.

### 4.4.3. Rationales (RQ3)

When asked for their migration rationales, i.e., why they modernized their systems towards microservices and DevOps, our participants mentioned a total of 10 different reasons, most of them quality attribute goals (see Fig. 4.6). For six of the seven systems, increased *development productivity* was a main driver (all systems except S1). Decreasing cycle time, establishing faster feedback loops, or providing new functionality faster were prevalent themes here. Related to this was the goal to improve *maintainability*, which was mentioned for five systems (S2, S4, S5, S6, S7). While better maintainability ultimately should lead to better productivity, the mentioned reasons here were more technical and structural, e.g., improving modularity, reducing coupling, improving code quality, reducing complexity, or improving reusability. Five systems (S2, S3, S4, S5, S7) also had the goal to improve *testability*, i.e., to be able to test smaller components in isolation and in less time.

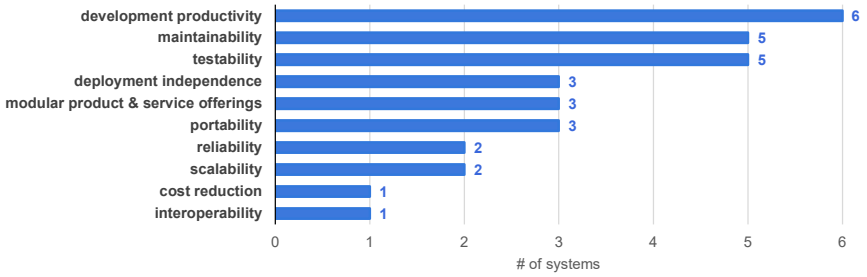



**Figure 4.6.:** CPS Interviews: Distribution of Migration Rationales

Other rationales occurred less frequently. For three systems (S3, S4, S6), *deployment independence* was explicitly mentioned, i.e., each team should be able to release and deploy their services independently of all others. More a business than a technical driver was the need for *modular product & service offerings*. This was named for S1, S2, and S4, where fine-grained and more specialized offerings to customers and avoiding redundant offerings were very important. In three cases (S1, S2, S6), *portability* was mentioned as a migration rationale, i.e., supporting various deployment environments like cloud vs. edge, Windows vs. Linux, digital twin platform vs. real hardware, etc. Less prevalent rationales were improving *reliability* (S4, S6) or *scalability* (S4, S6), even though these two are typical quality attributes associated with microservices. Lastly, *cost reduction* (S6) and *interoperability* (S2) were each mentioned only once.

#### 4.4.4. Requirements for Tool Support (RQ4)

In this section, we present interviewees' reflections on migration guidance and automation potential during a migration to microservices. The following subsections thereby extend our publication [FBH+22] by an analysis and synthesis of additional content collected during the interviews.

##### 4.4.4.1. Feedback on Migration Guidance

The nine interviewees were provided with a framework showing the major migration activities in a migration process (draft of Figure 7.1 in Chapter 7). In a first step, we collected general feedback on the depicted migration process, its phases, and activities.

The participants responded with predominantly positive feedback. Development Lead P3 regarded the inherent aspect of knowledge accumulation regarding microservices migrations important. Architect P4 confirmed that the depicted steps would very well reflect their system, especially in the platform area. Also, Architect P6 had an overall positive impression and stated that "*it is clear and covers the most important points*". In the same way,

Architect P5 responded that the diagram reflects what their team would need to tackle. He was also interested in experiences from others who followed this guideline already. In this regard, project manager P9 stated that "*this depicts very precisely what we did*".
When more directly asked if interviewees would consider adopting the framework for their own migration process, six participants responded positively. Three of them responded very optimistic with "*Definitely.*" (P8) and "*your framework is also something we are considering and that we really need*" (P1). P3 saw building up such internal knowledge as a big advantage, as consulting companies, in the end, would not have to bear the responsibility. P7, whose migration activities already went beyond the planning phase, remarked that such a guidance would have helped them earlier in the process to better structure their activities. Two more participants responded positive, but less enthusiastic. Architect P4 said that it would of course help, while P5 expressed openness to examining new ideas as long as the effort is limited. Architect P6 took a neutral stance and was revealing about his own activities: "*I am currently building something in this direction [...] modeling from the requirements to an architecture*", but put less importance on the results adhering to all microservices characteristics.

Two participants, P2 and P9, were concerned about the suggested process. Architect P2 would not use it and reasoned "*I don't go through the process in a structured way as shown here*". Project manager P9 was skeptical regarding the potential and maturity of the suggested process, given their already progressed, complex project. He added that "*if you had asked us 8 months ago, I would liked to have something [tried] which is still in progress, in a maturation phase*" (P9). Figure 4.7 summarizes the sentiments on the adoption of our suggested process framework.
Our interviewees provided rich feedback in the form of aspects for consideration and suggestions for improvement. P3 and P6 emphasized the importance of the system comprehension phase with regard to a grounded migration decision, especially in the case of a "*really historical*" (P3) system. In this regard, project manager P8 expressed a wish for rationals to justify technical decisions towards the management. Lastly, P2 found it an

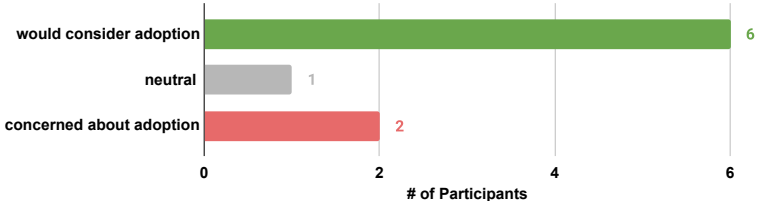



**Figure 4.7.:** Interviewees' Sentiment on Framework Adoption

interesting extension to consider also alternative architectural styles beyond microservices.

Concerning the second strategy definition phase, P3 remarked to clearly distinguish between different development strategies, such as *Strangler* and *Continuous Evolution*. Architect P6 questioned, if first defining a migration strategy and afterwards defining the target architecture would be the correct order. In this regard, an imprecise use of the notions *architecture definition* and *architecture design* might have caused some ambiguity. Eventually, P2 suggested considering even multiple migration strategies for the individual subsystems.

For the third implementation phase, hands-on guidance was requested by several participants. Architect P2 suggested providing a documentation or even explanatory videos that detail the options recommended for a specific system. He emphasized the explainability aspect by stating: "*What is important to me if you suggest something: a) I understand why you suggested this to me, and b) I understand the approach.*" (P2). Architect P4 was looking for tooling that would visualize the data flow through a system on a higher level of abstraction. Eventually, P1 and P7 would appreciate a systematic method for guiding the migration process, as it "*would save a lot of time and give you a direction or base for discourse within the team.*" (P7). Regarding the general orientation and scope of providing guidance and tool support in a migration, P4 noted that "*a divide & conquer approach could be better suited because a super generic tool is always difficult*" (P4).

#### 4.4.4.2. Feedback on Automation Potential

In a second step, we asked for automation potential regarding the various activities depicted in the framework, and participants' requirements and concerns hereto. The results in Figure 4.8 indicate a high demand for tools helping to understand the legacy system and identifying services, as confirmed by five participants. Further areas like defining the migration process, and ensuring governance and quality assurance (QA) received two mentions each. When asked for requirements on such tool support, usability and applicability were considered most important. Offered maintenance for tools was required by two participants, which was also the case for flexibility regarding its adaption to a custom system. Two participants were skeptical that tools could offer such flexibility in a migration to microservices, while three times, concerns were expressed on automation capabilities of migration activities in general.

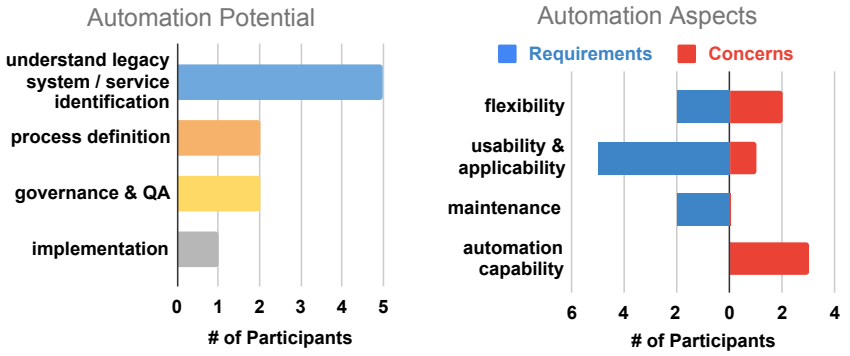



**Figure 4.8.:** Automation Potential and Aspects for Tools in Migrations

When asked for rationales regarding automation potential, architect and researcher P1 stated that "*tool support in all phases is valuable*", e.g., for analyzing the legacy system or repository mining. Architect P2 added that identifying "*parts that belong together, shared responsibilities*" would be predestined for automation, which was confirmed by P6: "*In general, it would*

*be great to show dependencies, independent of the used technology*". Architect P5 sought for automating change detection on an abstract syntax-tree level. The aspect of defining migration strategies and organizing development teams was emphasized by researcher P7. Architect P2 added that the related automation of governance, where his team already "*built a lot of tooling on [their] own*". Eventually, P3 saw potential for tools supporting verification and validation processes, particularly in the regulatory area.

When asked about rationales for automation requirements and concerns, architect P5 expressed his demand for flexibility over an "all-in-one" solution. He added the aspect of long-term maintenance, unless the tool would not be used continuously. P1 and P9 expressed usability aspects, which includes a variety of aspects next to a solid documentation: "*We need tools usable in practice – this is the challenge*" (P1). In this context, P9 stated that every software project is different, and a single tool could hardly account for this. P6 and P7 had similar concerns for efficient tool support, given the individuality and complexity of a system's architectural refactoring and migration.

## 4.5. Discussion

The review and interview results show many similarities, but also reveal certain discrepancies. In the following, we first compare the CPS results of our rapid review to similar literature studies or surveys for enterprise applications. Afterwards, we contrast the rapid review results with the interview results and reflect on guidance and automation in the migration process.

### 4.5.1. Comparing CPS with Enterprise Applications

To highlight the peculiarities of the CPS domain and show differences to common enterprise applications, we extracted challenges and practices from existing studies on enterprise application microservices. For this purpose, we relied on our extensive experience with enterprise application microservices

in industry [BFWZ19b; BFWZ21; FBWZ19; MCF+20; VCG+22], to select and analyze nine publications, primarily recent literature studies and two interview studies, that focus on general practices and challenges in the microservices and DevOps migration context. Table 4.5 lists these eight publications along with the number of included primary sources or study participants. The last two columns show the number of extracted challenges and practices per paper. Altogether, we extracted 513 challenges and 114 practices from all sources. For reviews, we counted a challenge or practice per primary source that stated it, for the interview and survey paper per participant that stated it. To ease comparability, the labeling was harmonized with our preceding literature review by using the same categories, as can be seen in Figs. 4.9 and 4.10.

**Table 4.5.:** Empirical Studies on Enterprise Application Microservices

| ID | Title | Year | Type | # Sources | # Chal. | # Prac. |
|---|---|---|---|---|---|---|
| 1 | Deployment and communication patterns in microservice architectures: A systematic literature review | 2021 | SLR | 38 | 57 | 14 |
| 2 | Monoliths to microservices - Migration Problems and Challenges: A SMS | 2021 | SMS | 37 | 63 | 2 |
| 3 | Understanding and addressing quality attributes of microservices architecture: A Systematic literature review | 2021 | SLR | 72 | 0 | 20 |
| 4 | A Systematic Mapping Study on Microservices Architecture in DevOps | 2020 | SMS | 47 | 72 | 37 |
| 5 | Promises and challenges of microservices: an exploratory study | 2020 | Interview + Survey | 21+37 | 111 | 21 |
| 6 | Architecting with microservices: A systematic mapping study | 2019 | SMS | 103 | 0 | 7 |
| 7 | Microservices Migration in Industry: Intentions, Strategies, and Challenges | 2019 | Interview | 16 | 70 | 0 |
| 8 | Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices | 2017 | SLR | 69 | 140 | 13 |
| Sum | | | | 440 | 513 | 114 |

**Challenges** *Heterogeneity* reflects the most mentioned challenge with CPS systems. It goes along with achieving sufficient system *performance* as well as efficient *data processing* in such environments, which are the third and forth most mentioned challenges for CPS. We attribute these aspects to typical IoT/IIoT sytems, which are characterized by operating a large number of devices with limited hardware resources. Compared to enterprise applications, these three challenge categories, plus *integration & communication* aspects, are seen as more important by the CPS literature. Challenges with enterprise systems, on the other hand, are strongly centered around *organizational aspects*. This category seems to have received very little attention in the CPS literature, as the big gap in mentions reveals. Difficulties with efficient *operations & deployment* follow, which is a very relevant category also for CPS systems. When it comes to *architectural aspects*, the gap again reveals a higher importance for typical enterprise systems. Here, challenges are mainly centered around finding a suitable decomposition into microservices. As many CPS are naturally designed as distributed systems, the modularization aspect might therefore be perceived as less challenging compared to other design aspects. *Security concerns* showed a little higher relevance for CPS systems, while the remaining categories draw a largely balanced picture. Interestingly, *dependability-related challenges*, which one would intuitively attribute to CPS microservices, played no major role in both domains, and received even slightly more mentions for enterprise application microservices.

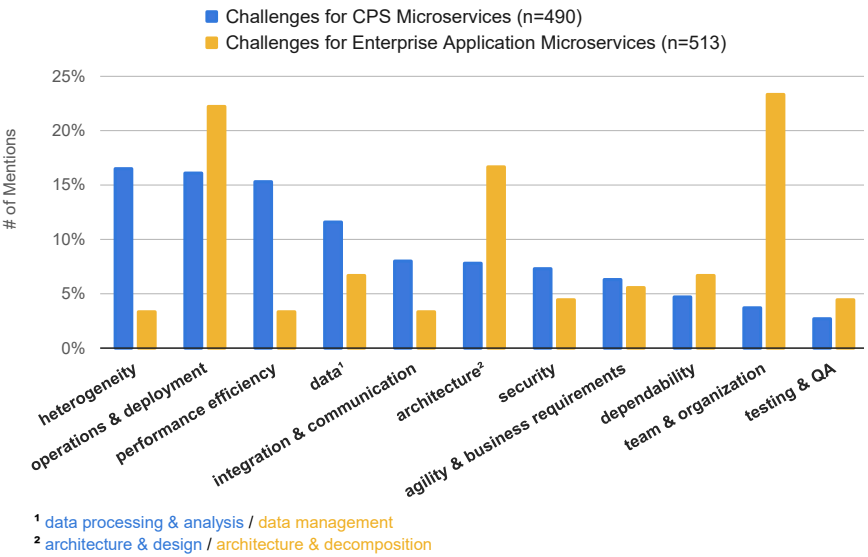



**Figure 4.9.:** Challenges: Comparing CPS and Enterprise Application Domains

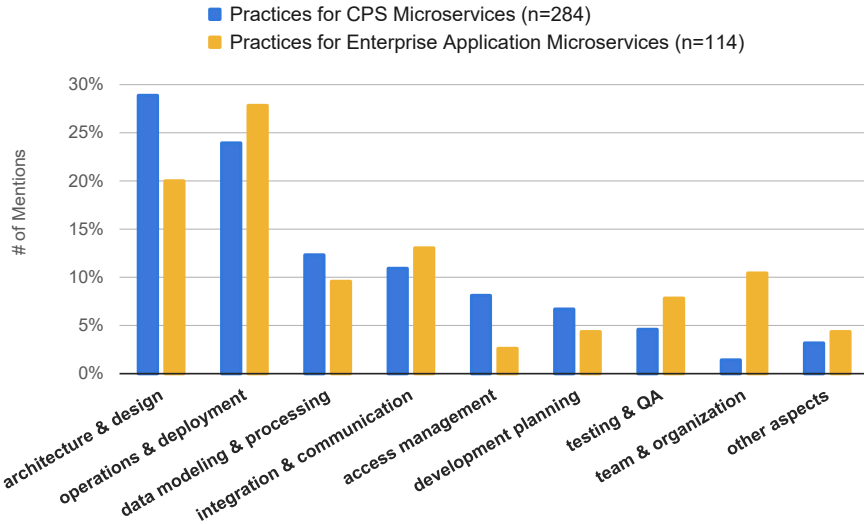



**Figure 4.10.:** Practices: Comparing CPS and Enterprise Application Domains

**Practices** The comparison of practices shows a more balanced picture. We again see a big gap related to *organizational matters* and for *architecture & design* practices. A closer look into the latter reveals that architectural solutions proposed for CPS account for distributed infrastructure, heterogeneity, and performance aspects. Solutions for the enterprise domain, on the contrary, focus mainly on decomposing monolithic architectures into a suitable set of services, which correlates with the identified challenges in this category. A similar difference shows up for the *operations & deployment* category. Deployment solutions for CPS mainly address challenges of automation, placement of services, load balancing and monitoring. Solutions for enterprise systems prioritize the automation aspect as well, but focus less on performance-related aspects than on integrated CI/CD pipelines that foster agility and short release cycles. The inherently more challenging *security* aspect for CPS systems is reflected by a higher number of practices around necessary access management. *Other aspects* include techniques and tools to improve autonomy or automation, managing product variants, and customer involvement.

### 4.5.2. Comparing Rapid Review and Interview Results

The primary studies in our rapid review reflect a strong focus on hardware-related aspects around the management and operation of heterogeneous environments, especially the identified challenges. This is augmented with the complexity of big data processing and real-time conditions. To address unsuitable CPS architectures that fail to address evolvability, operability, or performance efficiency, researchers proposed many concrete microservices-based architectures for various CPS contexts. They predominantly offer solutions for efficient operations & deployment in the cloud-edge continuum, but also for data processing & modeling.

However, our interviewees from Siemens AG perceived most challenges with different importance. The top two review categories (*heterogeneity* and *operations & deployment*) played no major role for most analyzed systems, with only a few mentions. Standardization in domains like healthcare

(partly) addressed the former, while the latter was more relaxed because none of the analyzed systems needed dynamic edge-cloud deployment and scheduling at runtime. Another reason for the lack of operational challenges might be that most systems were still early in the migration. Instead, *architecture & design* was the dominant challenge category, with a focus on the difficulty of modularizing and decomposing monolithic systems. Mentioned issues were commonly associated with legacy systems, e.g., huge size, hidden dependencies, or accumulated technical debt. Consequently, the decomposition into services was a major challenge among our interviewees that was mostly described as a manual task with a lack of adequate tool support.

Another difference was the importance of *team & organizational challenges* in the interviews, while these seemed neglected in the review. Establishing a DevOps culture and building developer confidence were seen as challenging, and the lack of employees experienced with current technologies could present a bottleneck. Consequently, our interviewees reported compensatory practices around promoting team independence and collaboration, e.g., through conducting workshops. Moreover, we also saw a preference for continuously evolving a system towards microservices and DevOps rather than a swift shift, which was, however, also due to strict quality and compliance requirements.

Contrary to the review, the interviews mostly reflected typical challenges and practices of enterprise microservices, as described by, e.g., Baškarada et al. [BNK18] and our earlier interview study [FBWZ19]. Except for performance challenges (e.g., real-time processing requirements), typical CPS challenges like heterogeneity, dependability, or security were not seen as problematic for using microservices. For several systems, that meant that not all microservices characteristics were desired for every part of the system. However, this tendency has also been observed for enterprise applications [BFWZ19b; ZLJ+19]. The good news about these findings is that many general approaches for microservices and DevOps are also valuable for the CPS domain. Nonetheless, slight adaptations or trade-offs are needed

in some cases, as several CPS characteristics still seem to have an influence, albeit not a major one.

### 4.5.3. Migration Guidance and Automation

Our presented draft of a framework depicting the migration process was perceived positively by the majority of our interviewees. A systematic method for guiding such a process was generally appreciated. The participants stated that the depicted phases and activities would reflect their understanding appropriately, confirmed by one participant's past migration experiences. Consequently, the majority expressed a positive stance on its potential adoption for their own systems. It would help to build up internal knowledge over leveraging external consulting, and thereby support architects who are less experienced with microservices migrations. Our participants emphasized the importance of a grounded migration decision and its justification to the management. The flexibility to treat individual subsystems differently was seen as important, as well as a solid documentation and explanation of automatic recommendations, e.g., though video tutorials. However, one participant also expressed concerns regarding its potential and maturity, while another participant preferred a generally less structured migration approach.

Our exploration of automation potential for the various migration activities caused a similarly positive echo. Two participants mentioned that they or their team had already spent significant efforts on creating their own tools. The dominant area for automation potential was understanding the legacy system, e.g., through static analysis, and a derived service identification. Automating the process itself, and tools for governance and quality assurance in the migration were in moderate demand, while dedicated tooling for implementation activities was only mentioned once. Such tools should primarily demonstrate usability and practical applicability, but also their maintenance and flexibility were mentioned twice each as a requirement. However, the complexity and individuality of a system migration was also regarded as a particular challenge for providing effective tool support.

### 4.5.4. Main Takeaways

In the following paragraphs, we briefly synthesize the main takeaways of the combined study results and their implications.

**The CPS and enterprise application literature differ in the importance of several microservice-related challenges.** For example, *heterogeneity* and *performance efficiency* were perceived as much more challenging for CPS microservices than for enterprise application microservices. Conversely, challenges related to *team & organization* were barely discussed in the CPS literature, while this was one of the top challenge categories for enterprise application microservices. This may imply that the different usage scenarios and quality requirement prioritization between these two types of systems lead to a slightly different research focus. Since the introduction of microservices in the CPS domain happened later than for enterprise applications, it might also mean that some industry-relevant challenges are not yet discussed frequently, but will appear more in future publications, once the field matures a bit more.

**Osmotic computing [VFD+16] was a prevalent microservices theme in the CPS literature, but not at Siemens AG.** As a concrete example of the different research focus, many proposed CPS practices in the literature centered around the dynamic scheduling and deployment of microservices at runtime within the cloud-fog-edge continuum, e.g., based on the origin of the request or the current load of the system. However, not a single one of the studied CPS at Siemens AG had this requirement. It may be that osmotic computing is only relevant for CPS outside of Siemens AG 's scope, e.g., very large, geographically distributed IoT environments, or that research in this space is still too early to broadly carry over to industry.

**Improved development productivity and maintainability were the primary migration rationales at Siemens AG.** CPS stakeholders predominantly wanted an architecture that would enable them to deliver features faster and more sustainably in the long run. While this is also one of the main

drivers in the enterprise application literature [BFWZ19b], other prominent microservices quality attributes like scalability, elasticity, or interoperability were rarely mentioned at Siemens AG. A potential reason could be that the average CPS may not experience as frequent or rapid changes in the required throughput as typical internet-facing applications.

**Typical CPS requirements like performance, reliability, security, or privacy were not perceived as major challenges regarding microservice migration at Siemens AG, but were mentioned for selected systems.** Three systems named performance as a challenge with microservices (S1, S4, S5), two systems reliability (S1, S4), and one system each did the same for compliance (S3) and security (S6). When explicitly queried for such quality attributes, other participants described that, while these requirements existed to a certain degree, the teams had experience in assuring these qualities, as they had to take care of them anyway in their domains. Using microservices would not fundamentally change this, i.e., it would not be a migration-related challenge. This implies that it is highly situational if assuring these quality attributes is challenging with CPS microservices and if it requires special methods or tool support.

**Several studied CPS at Siemens AG did not target a full microservice architecture with all characteristics.** Apart from S3, S4, and S6, the systems in our sample consciously decided to only aim for certain microservice characteristics and to leave out the rest, which was sometimes motivated by special CPS requirements. For example, the "no shared databases" principle was consciously ignored for S4 because data consistency was just too important for their regulated domain. This is important to keep in mind for researchers that work on methods or tool support for microservices in the CPS domain.

**Most frequently mentioned challenges and practices at Siemens AG were not highly specific to CPS.** Similar to the disconnect between the literature on CPS vs. enterprise application microservices, there are also several differences between the CPS literature and our case study findings

at Siemens AG. While the above sections mentioned several facets where the CPS domain had an influence on the usage of microservices, the overall sentiment regarding microservices challenges and practices was much closer to the enterprise application literature than to the CPS literature. A potential reason could be that our rapid review contained many papers on large, decentralized IoT environments, which have some differences to the typical CPS at Siemens AG. Additionally, it could be argued that a few of our studied systems are more platforms or applications from the CPS domain rather than actual CPS. Nonetheless, it seems plausible that a large proportion of methods and tools for enterprise application microservices will at least be partly of value for CPS microservices. Some flexibility for the pointed out differences will be helpful, but the extensiveness of the required adaptations should be subject to future research.

**A demand for systematic migration guidance and tool support was confirmed for the majority of investigated systems at Siemens AG.** We presented a draft of our migration framework presented in Chapter 7 to collect early feedback on its design, suitability, and practical relevance. The majority of our participants confirmed its appropriateness and were open to applying it in their environments. The accumulation of internal knowledge and offering decision support in the migration process were seen as major benefits. Similarly, the interviews revealed a significant potential for automation in the migration process at Siemens AG. While tools for legacy system comprehension and service identification were in high demand, also general concerns regarding effective automation in a complex and highly individual migration process were expressed. Particular emphasis should be put on documentation and comprehensibility of automatically generated recommendations. Tool support should thus focus on usability and practical applicability, but also address flexibility and maintenance to support practitioners adequately.

## 4.6. Threats to Validity

We generally aimed for a rigorous study design, but also made some tradeoffs regarding efficiency, as is common in rapid reviews. The mixed-method approach and the collaboration between industry and academia could have partially mitigated this. Still, several limitations need to be mentioned for our results, which we structure according to three common notions of validity described by Wohlin et al. [WRH+12].

*Construct validity* refers to a potential gap between operationalized measurements and the constructs targeted in the study, i.e., if the collected data is suitable for the intended analysis purpose. In our study, data collection was text-based and qualitative, namely extractions from papers and interview transcripts with subsequent thematic analysis. To combat potential threats in this area, we spent great care to arrive at a unified understanding of our central constructs in the research team, e.g., microservices, practices, challenges, rationales, etc. Additionally, we introduced important constructs to interview participants, posed clarifying questions when encountering ambiguous terms, and sometimes briefly reflected our summarized understanding back to interviewees. Nonetheless, there is a slight possibility that we interpreted a few instances differently than the authors of our primary studies or our participants. Furthermore, interviews rely on the described experiences and opinions of participants, which are subjective. A few interview statements could even be incorrect without us realizing. In general, however, we estimate the impact of threats in this area to be very low.

*Internal validity* is concerned with potential confounding factors and the rigor and consistency of the study design. While we consciously chose a more lightweight research method for the literature review, we still followed a very systematic process. At least two researchers decided if a study should be included or excluded. While only a single researcher performed the data extraction for most papers, we only split up the work after having piloted the extraction sheet and improved our shared understanding of it with 10 papers. Additionally, all important findings were discussed among several researchers to counteract potential subjective bias. For the interviews,

confidentiality and anonymity were provided to enable participants to talk freely about their experiences. Participants also seemed not to be afraid to describe negative or more sensitive areas of their systems. We also tried to limit our influence as interviewers on our participants, e.g., by avoiding leading questions or any judgmental statements. However, despite these countermeasures, there is still a chance for non-mitigated influences on data collection, which we nonetheless perceive as non-critical.

Lastly, *external validity* is concerned with how well the findings can be generalized to other settings or populations. From the discussed facets of validity, this is the weakest one in our study. A rapid review usually covers a lower percentage of the relevant literature, in our case via a stopping criterion (a maximum of 50 hits per search string and source). We counteracted this via five different data sources and one round of snowballing, which should have led to sufficient study diversity. Nonetheless, it is possible that the distributions for extractions could be different with a larger number of primary studies. We also need to be careful to generalize the review findings to industry settings. However, the more important limitation in this area is that we conducted a case study, a method that trades off external validity for result richness and depth. Our interview-based findings at Siemens AG are valuable, but not all of them may be generalizable to other companies. While Siemens AG is a global technology enterprise with diverse business units and projects in various domains, a certain degree of homogeneity between our units of analysis is to be expected. Parts of our case study findings may therefore not be transferable to other CPS companies, especially to smaller or less experienced ones. Even though we touched several CPS domains, our sample only contains seven systems that were described by nine CPS professionals. Additional studies with other companies are therefore necessary to broaden the understanding of microservices and DevOps for CPS, which we try to enable by sharing our study artifacts for replications.

## 4.7. Conclusion

While challenges and practices regarding microservices and DevOps are well studied for enterprise applications, an overview of the cyber-physical systems domain is missing in this regard. To address this, we formed an industry-academia collaboration, and conducted a rapid review (146 papers) followed by an interview-based case study with 9 different CPS experts at Siemens AG. We found that, with a few exceptions, our interviewees' perception of challenges and their applied practices were much closer to the enterprise application domain than to the CPS literature. Researchers and practitioners should take these findings into account when studying, developing, or modernizing CPS with microservices and DevOps. As well, additional research would be necessary to assess the generalizability of our findings at Siemens AG to other CPS companies.

From our interviews, we identified a significant demand for systematic migration guidance and tool support. Our draft of the framework was confirmed to be substantial and applicable to most of the involved systems. The by far highest potential for automation was seen in the activities around system analysis and service identification. Both findings confirm our identified research gap and encourage our efforts in providing migration guidance and tool support for the involved architectural refactoring. The detected considerable congruence of cyber-physical systems at Siemens AG with common enterprise applications confirms the significance and relevance of our findings also beyond the CPS domain. In this regard, the study results provide another solid foundation for the following contributions in this thesis.

Chapter 5

# State of Research: A Classification of Refactoring Approaches

Our third analysis contribution examines of the state of scientific research in the area of migrations from monolithic architectures to microservices. Specifically, we take inventory of the individual approaches the academia proposes for architectural refactoring to microservices. Motivated by the challenges identified in our two previously presented interview studies (Chapters 3 and 4), we seek answers and solutions that academia can offer to practitioners. Throughout four iterations, we reviewed refactoring and migration approaches proposed by researchers, while classifying them based on several criteria, such as their underlying techniques and strategy followed, required input and generated output, usability, and tool support. Besides, we document their targeted product quality attributes and design properties that they aim to optimize during a refactoring. This chapter extends the groundwork we built within the following publication:

> J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner. "From Monolith to Microservices: A Classification of Refactoring Approaches". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. DEVOPS 2018*. Ed. by J.-M. Bruel, M. Mazzara, and B. Meyer. Cham: Springer International Publishing, 2019, pp. 128–141 [FBZW19]

The publication constitutes one of the first scientific publications reviewing architectural refactoring approaches for migrations to microservices. The 10 therein inspected approaches are classified by the underlying decomposition technique, and visually presented in the form of a decision guide for practitioners. Over the following years, three additional iterations of this review have been conducted that yielded a multitude of new publications and approaches. Over these iterations, we extended and refined our classification scheme, resulting in a body of knowledge that covers 110 refactoring approaches proposed by academia.

## 5.1. Context: Refactoring Approaches

Software architects confronted with the architectural refactoring of a legacy system are in need of an appropriate migration strategy and approach for the architectural refactoring. One of the most frequently discussed challenges is finding the right service cut and granularity to fully leverage the advantages of the microservices architectural style. We aim to review the current state of research in the context of refactoring approaches that researchers have proposed for migrating a monolithic system towards a microservices-based architecture. This study aims to fill the gap in scientific research by comparing and classifying refactoring approaches proposed in academic literature. The results provide architects and developers with an overview of currently available approaches for refactoring monolithic systems into microservices architectures. We conducted our initial review in 2018, resulting in a first classification of such approaches. In the following years, we replicated and

refined our review in three additional rounds, yielding in 110 documented approaches in the timeframe 2016 to 2023. We identified a variety of strategies and techniques to break down a monolithic application into loosely coupled and highly cohesive services. To classify such refactoring approaches, we derived a taxonomy that represents an additional contribution of this study. This taxonomy covers process strategy, decomposition techniques, required inputs, generated outputs, quality goals, and usability aspects like validation, accuracy, and tool support.
The research objective of our study is formulated by the following two research questions:

**RQ1:** What are existing architectural refactoring approaches in the context of decomposing a monolithic application architecture into microservices?

**RQ2:** How can such approaches be classified with regard to their strategies, underlying techniques, usability, and quality goals?

The goal of this study is to provide a structured overview of currently available architectural refactoring approaches for migrations to microservices, that have been published in scientific literature. Besides, we derive a taxonomy for classifying such approaches. The results can support software architects and developers in finding and selecting an appropriate refactoring approach applicable to their system. Researchers profit from a comprehensive inventory reflecting the current state of research. It further helps in identifying current shortcomings and challenges, underrepresented aspects in such approaches, or research gaps. Moreover, the derived taxonomy can facilitate the cataloging of newly emerging approaches and helps to make such research contributions easier accessible for both academia and industry. In this context, and in the course of this thesis, our review study provides the basis for building a structured and continuously evolving repository for such refactoring approaches. This repository represents a core component for the realization of our envisioned quality-driven migration methodology.

## 5.2. Method: Literature Review

To answer the above formulated research questions, we conducted a literature review of scientific publications. Brereton et al. propose a three-step review process that serves as a basic structure: planning, conducting, and documenting [BKB+07]. We additionally consulted the guidelines for systematic literature reviews proposed by Kitchenham and Charters [KC07] who suggest an explicit and formal processes to ensure reproducibility and consistency of the acquired results which are less likely biased. Following a methodology ensures a structured process, summarizes the empirical evidence adequately, and reveals gaps in current research that enables researchers to conduct follow-up research [KC07; PT19]. However, systematic reviews require a considerable effort compared to less strict methods. We thus followed a less systematic method, while nonetheless aiming for scientific rigor in conducting the review and documenting our results. The threefold replication of our review in differing personnel constellations, followed by an aggregation of all results, will mitigate the risk of a biased research process and potentially missed sources. We summarize our results in textual and visual form when applicable.

### 5.2.1. Search Databases and Keywords

Fundamental constraints of a literature review are the databases to query and the associated search strings. Four of the most frequented scientific libraries and indexing systems in computer science are ACM Digital Library [1], IEEE Xplore[2], Springer Link[3], and Google Scholar[4]. According to Petersen et al., these databases have been proven most relevant for conducting systematic reviews in the software engineering field too [PVK15]. Google Scholar as a supplementary meta search engine can yield additional contributions and

[1] ACM Digital Library. https://dl.acm.org/about

[2] IEEE. https://www.ieee.org/about/index.html

[3] Springer. https://www.springer.com/gp/about-springer.

[4] Google Scholar. https://scholar.google.de/

thereby complement the three other scientific databases. Their combined application will ensure comprehensive coverage of relevant and peer-reviewed publications.

In a second step, we defined a set of search keywords that condense our study objective and research questions. We focus on the architectural styles *microservices* and *monolith* while our activity under investigation is the *architectural refactoring* in the context of a *migration*. We considered alternative notions, synonyms like *transformation*, *adoption*, *partition*, and *decomposition*. Several related aspects of our interest in the context were added, i.e., *granularity*, *quality*, *metric*, or *tool* support, that reflect our intentions for a subsequent classification of such approaches. By combining the keywords, the following generic search string was composed and further adapted to the requirements of each scientific database:

```
(microservice ∨ micro-service ∨ micro service)
[∧ monolith]
[∧ (refactor ∨ transform ∨ migrat ∨ decompos ∨ adopt
 ∨ partition ∨ granular ∨ tool ∨ quality ∨ metric)]
```

### 5.2.2. Inclusion/Exclusion and Data Extraction

Based on publication title and abstract, the following explicit inclusion and exclusion criteria have been applied:

**Include:** The publication describes an approach in the context of a migration from monolithic applications to microservices. A qualitative assessment ensures relevance of the content and a sufficient degree of technical depth.

**Include:** The publication must be published in 2016 or later.

**Exclude:** The publication's full-text is not available to us.

**Exclude:** The publication is not written in English.

**Exclude:** The publication is not peer-reviewed.

Guidelines recommend backward/forward snowballing on the resulting list of included publications [Woh14], which we applied after applying the inclusion/exclusion criteria. Figure 5.1 illustrates the essential steps of our search strategy.

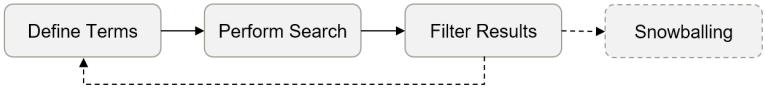



**Figure 5.1.:** Search Strategy for Literature Review

A unified data extraction form helps to collect all relevant information from the publications and allows "*to highlight similarities and differences between study outcomes*" [KC07]. Our form comprised three sections: "Paper Context Data", "Empirical Data", and "Quality Assessment". The first section aimed to capture basic information such as title, authors, publication year, and DOI. The second section "Empirical Data" aimed to catalog the details of the presented approach, including strategy, techniques, input, output, usability, and quality aspects. Eventually, the "Quality Assessment" intends to document feasibility, comprehensiveness, and other notes like the leveraged sample/reference applications used for evaluation. We used spreadsheets for data extraction and populated the tables while full-text reading each paper. By referring to the inclusion and exclusion criteria again, additional publications were excluded when applicable, e.g., literature reviews or conceptual solution proposals without tangible migration process description. In the last step, data synthesis, the extracted data and findings of the individual studies are combined and evaluated, thereby extending the existing body of scientific knowledge. As an additional result, we derive a taxonomy for classifying refactoring approaches in the context of migrations to microservices.

### 5.2.3. Review Iterations

We conducted the first iteration of this review in 2018, followed by three replications, in which we both added new publications and refined our

methodology and classification scheme. Table 5.1 shows the coverage and yielded refactoring approaches for all iterations.

**Table 5.1.:** Iterations of This Review Between 2018 and 2023

| Iteration | Year | Coverage | # Publications | Reference |
|---|---|---|---|---|
| 1 | 2018 | 2016–2017 | 10 | [FBZW19] |
| 2 | 2020 | 2018–2020 | 31 | [Gu20] |
| 3 | 2022 | 2016–2022 | 90 | [Koc22] |
| **4** | **2023** | **2016–2023**[1] | **110** | **this work** |

[1] till July 2023

In the following, we present the review results based on the fourth iteration conducted in 2023, covering approaches published from 2016 to July 2023.

## 5.3. Results

In this section, we present the results of our reviewed publications on refactoring approaches to microservices. We start with an overview of the collected publications, discuss the resulting classification scheme, and separately investigate quality goals that the approaches aim for. We conclude with our proposed taxonomy for such approaches.

Figure 5.2 shows all reviewed refactoring approaches by their year of publication. Our literature review yielded in total 110 approaches, stemming from the same number of publications. Hence, we use the terms *publication* and *approach* interchangeably throughout this review. The majority of approaches were published between 2020 and 2022, while 2023 suggests a similarly high publication frequency. The results indicate its high relevance in academia, even several years after the emergence of microservices. The

complete list of our reviewed publications can be found in Appendix Table B.1, while we share the raw and aggregated data as a digital appendix[1].

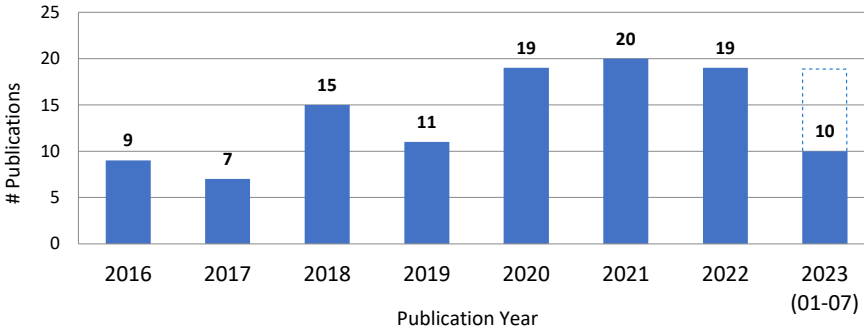



**Figure 5.2.:** Refactoring Approaches by Year of Publication (N=110)

The reviewed approaches are presented with a varying orientation, coverage, technical depth, evaluation, and tool support. In Section 2.2.2, we discussed a classification scheme that adapts and enhances the taxonomy of Service Identification Approaches (SIA) in general by Abdellatif et al. [ASM+21]. In this review, we use a derived classification scheme for characterizing the herein reviewed refactoring approaches in the microservices context. Tables 5.2 and 5.3 present an overview of all reviewed approaches based on this scheme. Each identified characteristic is assigned the number of its occurrences in all publications, i.e., refactoring approaches. An approach can be associated with a multiplicity of characteristics, even within the same subcategory. For instance, an approach can require the inputs *Human Expertise* **and** *Documentation* at the same time, and hence would be represented twice in the subcategory *Domain artifacts* of the category *Input*. In the following, we discuss each of the categories and provide examples when appropriate.

[1] https://doi.org/10.6084/m9.figshare.25856002

### 5.3.1. Required Input

Input artifacts, as shown on the upper left in Table 5.2, set the starting point for a refactoring activity. Their availability for a given monolithic system determines the possible migration strategies, techniques as well as the achievable accuracy of a decomposition [ASM+21]. According to the authors, a combination of different input types can positively influence the decomposition process and result quality. According to our discussion in Section 2.2.2, we distinguish four categories of input artifacts, namely *Domain Artifacts*, *Runtime Artifacts*, *Model Artifacts*, and *Executable*. Abgaz et al. use a similar scheme in their recent review, and differentiate between static, dynamic, model-based, and version data for inputs [AME+23].

**Domain Artifacts** In the first category of Domain Artifacts, we locate *Human Expertise* with 26 findings. Domain knowledge and technical expertise are essential for many approaches due to the inherent complexity of an architectural refactoring. We assigned approaches to this input type, when such domain expertise was explicitly mentioned or the presented level of abstraction implied substantial manual efforts for its application. A system's *Documentation* was mentioned for 9 approaches. Such documents, including requirements, specifications, diagrams, and similar artifacts, ideally complements other input artifacts and can compensate a lack of human expertise, allowing for better replicability. Historic data in the form of a *VCS History* data leverages information that a version control system accumulates over time. It was used by five approaches and allows for applying a dedicated analysis technique for generating a decomposition. An *Ontology* serves for two approaches as input. Abdellatif et al. describe it as a "*structured set of terms representing the semantics of a domain*" [ASM+21]. As their orientation and level of abstraction can vary significantly, the major challenge would be defining an ontology that allows for the identification of services. Hence, the authors regard generating an ontology for services identification as complex and expensive due to the required substantial human expertise.

**Table 5.2.:** Classification of 110 Approaches by Input, Output, and Process

**Input**

| | Total |
|---|---|
| **Domain artifacts** | |
| Human Expertise | 26 |
| Documentation | 9 |
| VCS History | 5 |
| Ontology | 2 |
| **Runtime artifacts** | |
| Log Traces | 18 |
| User-Application Interactions | 7 |
| **Model artifacts** | |
| Custom Model | 20 |
| Use Case Model | 18 |
| Business Process Model | 11 |
| Class Diagram | 6 |
| Data-Flow Diagram | 3 |
| Entity Model | 3 |
| Activity/Sequence Diagram | 1 |
| **Executable** | |
| Source Code (Java: 16, Python: 1) | 55 |
| Database | 15 |
| API / Interface | 4 |
| Test Cases | 1 |

**Output**

| | Total |
|---|---|
| **Representation** | |
| List of Services | 73 |
| Splitting Recommendations | 70 |
| Guideline / Workflow | 27 |
| Visualization | 7 |
| Source Code | 5 |

**Process**

| | Total |
|---|---|
| **Process Strategy** | |
| Refactor | 102 |
| Rewrite / Rebuild | 78 |
| Greenfield | 15 |
| Continuous Evolution | 7 |
| Strangler | 4 |
| **Direction** | |
| Bottom-Up | 56 |
| Mixed | 21 |
| Top-Down | 19 |
| **Automation** | |
| Manual | 44 |
| Semi-Automatic | 33 |
| Automatic | 16 |
| **Analysis Type** | |
| Static | 89 |
| Dynamic | 24 |
| Lexical | 13 |
| Historic | 5 |
| **Technique** | |
| Clustering | 59 |
| Custom Heuristics | 40 |
| Graph-Based | 27 |
| Genetic Algorithm | 13 |
| Execution-Trace Modeling | 10 |
| Domain-Driven Design | 8 |
| Machine Learning | 7 |
| Data-Flow driven | 6 |
| Scenario Analysis | 6 |
| General Guidelines | 4 |
| Wrapping / Black Box | 3 |
| Multi-Tenancy Orientation | 1 |
| Performance Modeling | 1 |
| **Atomic Unit** | |
| Functionality | 52 |
| Business Capability | 32 |
| Class | 23 |
| Entity | 18 |
| Function | 12 |
| Interface | 9 |
| Other | 6 |

**Table 5.3.:** Classification of 110 approaches by Usability and Quality

**Usability**

| | Total |
|---|---|
| **Validation** | |
| Experiment | 47 |
| Case Study | 40 |
| No Validation | 18 |
| Industry | 5 |
| **Accuracy** | |
| Not available | 100 |
| Medium | 5 |
| High | 3 |
| Low | 2 |
| **Tool Support**[1] | |
| Tool available | 36 |
| Static Analysis | 30 (23) |
| Decomposition | 29 (20) |
| Open Source | 19 (15) |
| Visualization | 10 (8) |
| Dynamic Analysis | 6 (6) |
| Reverse Engineering | 2 (1) |
| Database | 1 (1) |

**Quality**

| | Total |
|---|---|
| **Quality Goals** | |
| Maintainability | 31 |
| Performance | 29 |
| Scalability | 20 |
| Portability | 17 |
| Independence | 13 |
| Reliability | 12 |
| Business-oriented | 11 |
| Security | 4 |
| Compatibility | 3 |
| **Design Properties** | |
| Cohesion | 44 |
| Coupling | 43 |
| Granularity | 29 |
| Complexity | 6 |
| Techn. Heterogeneity | 6 |
| Autonomy | 5 |
| Isolation | 1 |

**Quality Sub-Attributes**

| | Total |
|---|---|
| **Maintainability** | |
| Modularity | 16 |
| Reusability | 8 |
| Monitorability | 3 |
| Understandability | 3 |
| Manageability | 2 |
| Modifiability | 2 |
| Testability | 1 |
| **Portability** | |
| Deployability | 11 |
| Replaceability | 1 |
| **Performance** | |
| Time Behavior | 7 |
| Resource Utilization | 4 |
| **Reliability** | |
| Availability | 7 |
| Fault Tolerance | 2 |
| **Business** | |
| Execution Cost | 4 |
| Organizational Alignment | 1 |
| **Compatibility** | |
| Co-Existence | 1 |
| Interoperability | 1 |
| **Security** | |
| Confidentiality | 1 |

[1] Number of individual tools in brackets, from 31 tools altogether

**Runtime Artifacts** The second category of Runtime Artifacts includes *Log Traces* and *User-Application Interactions* that a software system produces during is operation. *Log Traces* have been found 18 times among our approaches. Abgaz et al. refer to them as dynamic data or "*Log-based Input Collection*" [AME+23] and give performance or web access logs as examples. They protocol the dynamic behavior of a system at an arbitrary level of detail, usually controlled via configuration files. Of particular interest are sequences of function or method calls that reflect a specific execution scenario, e.g., a business process or use case [ASM+21]. The authors add that log traces are often combined with other input types like business process models and use case models, complemented by human expertise. *User-Application Interactions* reflect a certain workflow that a user is required to follow for performing a certain task [ASM+21]. They have been found seven times among our approaches. Commonly, the user interface reflects such a workflow and their underlying business logic [ASM+21]. The authors add that such interactions would allow for identifying related sequences and reusable tasks in the context of service identification. However, this type of input would hardly suit an approach aiming for a high degree of automation.

**Model Artifacts** Input of the third category, Model Artifacts, provide an abstraction of the system with a specific orientation. The category includes the second most frequently used types of input, after Executable. A *Custom Model* is required by 20 approaches, followed by *Use Case Model* (18) and *Business Process Model* (11). Less frequently used were other standardized model types like *Class Diagram* (6), *Data-Flow Diagram* (3), *Entity Model* (3), and *Activity/Sequence Diagram* (1). While custom models are tailored to the specifics of the underlying analysis technique, their generation can require significant efforts. A prominent example is the well-known Service Cutter approach by Gysel et al. [GKGZ16], whose applicability for large-scale applications is limited by a lack of automation for generating the required custom input models.

A *Use Case Model* reflects the interactions between a system and its user at a higher level of abstraction and is commonly used for documenting

functional requirements [ASM+21]. Gemino and Parker describe use cases as "*simplified and graphical representation of systems*" [GP09] that serve as a "blueprint" of the system, illustrating the users' interaction with it. Such isolated use cases are an appropriate means to identify services that strive for a single responsibility. However, approaches relying on use cases are difficult to automate, as they often require substantial human expertise [ASM+21].

A *Business Process Model* documents the business logic of a system and thereby provides a means to derive subsets of business functionalities that form a microservice [Ami18]. While they have been widely used among the mostly SOA-oriented approaches reviewed by Abdellatif et al., we found them being less prominent for microservices. However, the established and widely used Business Process Model and Notation (BPMN) allows for an automation when translated to the associated Business Process Execution Language (BPEL) [ASM+21]. However, the authors add for consideration that such models are rarely available for legacy systems, which may explain the limited number of approaches leveraging them.

While a *Class Diagram* is a common artifact in object-oriented programming, it was used only six times by our reviewed approaches. As such diagrams commonly imply the existence of source code, the latter offers a more reliable way of extracting such information using static analysis techniques. A *Data-Flow Diagram* represents the functional dependencies originating from data flow analysis and is another means to reflect the business logic [ASM+21]. Only three approaches use them as input, which may also be caused by their origin in the application's source code, as the authors note. According to Daghaghzadeh and Babamir, entity models reflect entities and their interrelationship, while entities are the "*most important domain abstractions of a business*" [DB20]. The authors combine entity models and business process models in their proposed service identification approach.

**Executable** The existence of source code is essential for *brownfield* developments that follow a refactor strategy. By quoting a "Swiss Army Aphorism": "*If the map and the terrain disagree, trust the terrain*" [ASM+21], Abdellatif et al. emphasize that the source code would often be the only reliable source

of information about a legacy system. It would allow for the extracting a wealth of information using static analysis and reverse engineering techniques. Intermediate stages could represent the system's dependencies and behavior as, e.g., the above-mentioned data flow diagrams. Supplementary artifacts that are collected during application runtime can enrich the generated abstraction and optimize the subsequent decomposition further. Its high automation potential during for the processing makes source code the most often used input artifact, used by exactly half of all reviewed approaches (55). While most approaches do not state a limitation to a certain programming language, Java is the most popular language among the ones who do.

Another 15 approaches base their technique on an existing *Database*, which includes database schemas and transactions, next to the persisted contents. Leveraging the database structure for a decomposition is often combined with a static code analysis, as applied in the approach by De Alwis et al. [DBFP19]. Four approaches use *API/Interface* descriptions or specifications for steering the decomposition. Baresi et al. suggest such an approach that leverages the semantic similarity between functionality exposed through the API [BGR+17]. Eventually, the approach suggested by Jin et al. [JLZ+18] is the only one that exploits the monolithic application's test cases by executing them and inspecting the resulting logs for further analysis.

### 5.3.2. Generated Output

The manifestation and quality of the generated microservices is likely the most important characteristic of an approach. Abdellatif et al. differentiate the generated output services by service type, which reflects their adherence to service-oriented architectures in general [ASM+21]. Microservices, however, do not distinguish between enterprise, application, or infrastructure services. Instead, they reflect functional services in a business logic context, and put a higher emphasis on an appropriate granularity. Ideally, a fully functional microservices application is generated in the course of an auto-

mated refactoring. However, none of the reviewed approaches achieve this goal adequately. Still, five of the approaches generate *Source Code* like the approach contributed by Kalia et al. [KXK+21] which produces a skeleton for a subsequent service implementation.

Our review showed that 73 approaches generate a candidate *List of Services* and a similar number of 70 approaches provide *Splitting Recommendations*. The latter involve several potential service cuts, while a list of services is not necessarily parameterizable. Architects and developers can then incorporate such generated outputs when creating a partitioning and designing a microservices-based system. A drawback of both generated output types is that relationships and interactions between suggested microservices are often omitted or not reflected. This can be mitigated by a *Visualization* of the resulting services or clusters of classes, methods, interfaces, data entities, or other atomic units. Seven approaches provide such a graphical representation for their generated output, with one prominent example being the Service Cutter tool by Gysel et al. [GKGZ16]. Another 30 approaches do not produce tangible output artifacts, but rather serve as *Guideline/Workflow* in the decomposition process. Such approaches typically require substantial human expertise and offer a low degree of automation.

### 5.3.3. Process

The *Process* section of refactoring approaches, shown on the lower left and right in Table 5.2, includes the categories *Process Strategy*, *Direction*, *Analysis Type*, *Technique*, and *Atomic Unit*. While the first category expresses the approach's suitability for an overarching migration strategy, the latter four categories characterize the decomposition process by the underlying techniques.

**Process Strategy** As outlined in Section 2.1.4, the followed process of a migration to microservices is influenced by several factors beyond technical aspects like available inputs or used techniques. Available resources, the personnel's expertise, envisioned timeframe, or a company strategy can be

such boundary conditions that prescribe a specific migration strategy. It may also be appropriate to follow different strategies for individual subsystems of a larger system.

Among our reviewed approaches, *Refactor* is the most widely applicable strategy, supported by 102 approaches. When applying this strategy, the existing monolithic code base is split up adequately into smaller parts, eventually forming several microservices. In the process of a refactoring, the currently employed technologies like programming languages, libraries, or packages can be retained. Moving to new technologies is not a necessity and thus reduces the overall cost and effort, allowing to shorten the timeframe for the entire migration process.

A total of 78 approaches allow for a *Rewrite/Rebuild* process strategy. Compared to the Refactor strategy, rewriting or rebuilding the application's code eases the introduction of new technologies. Individual services can be implemented using, e.g., the most suitable programming language, framework, or database technology. They can resolve current technical limitations and thereby improve several qualities of the application like performance or maintainability. As well, this migration strategy offers more flexibility in generating an appropriate service cut, as the existing code structure and dependencies would not complicate the decomposition. This strategy may also be preferable when the existing legacy code reached an unmaintainable state, e.g., due to accumulated technical debt [LLST20; SS13] or the absence of its original creators. However, rewriting an application can involve significantly higher efforts and requires building up expertise with new technologies, as we found from interviewing industry experts (see Chapter 3).

When architects are faced with several of these limitations, rewriting the entire application or a greenfield project may sometimes be the only feasible solution. The *Greenfield* strategy aims for building an entirely new system without restrictions or dependencies on the legacy application. This strategy is supported by 15 of our reviewed approaches. While it allows for building, testing, and operating the system independently and in parallel to an existing system, the cost and effort can be significantly higher, e.g., due to a steeper learning curve for building up the necessary expertise. Greenfield projects are

particularly suitable when the functional requirements change significantly, or a COTS solution is replaced in the course of a modernization [FBWZ19].

Contrary to a greenfield approach, following a *Continuous Evolution* strategy aims at minimizing the effort and maintaining the system in its current state as far as possible. We consider it a variation of the *Refactor* strategy, describing a gradual refactoring of a system towards a microservices architecture. When available resources are short and the migration timeframe is not a limiting factor, this strategy can be favorable. Compared to the refactor strategy, changes can be done in a more targeted manner, e.g., combined with the implementation of new features, or based on certain qualities that should be improved. While most of the approaches assigned to the other categories are suitable for a continuous evolution too, we found six approaches that specifically target this strategy type.

The in this context popular *Strangler* pattern was mentioned by four approaches only. It similarly allows for an iterative replacement of system functionalities with newly implemented microservices Existing functionalities will be implemented as services that are then attached to the existing system, while at the same time, this functionality in the existing system is disabled. The process is completed when the newly emerged microservices application replaces the legacy system entirely [Mic23c]. The principle behind, called "StranglerFigApplication", was before described by Fowler in 2004 [Fow04b]. Its application is recommended when, e.g., resources are limited, the legacy application is too complex for a fast restructuring, or cannot be easily duplicated for development purposes. In such cases, both the Strangler and Continuous Evolution strategies allow for keeping the existing system in production throughout a migration process.

**Direction** According to Abdellatif et al., an approach's service identification can follow three directions [ASM+21]. *Top-Down* approaches start with high-level artifacts such as an analysis of the domain or requirements, while low-level artifacts such as source code or database schemas are not considered [ASM+21]. We identified 19 approaches that can be characterized as top-down-oriented. On the contrary, *Bottom-Up* approaches start with

"*low-level artifacts to maximize code reuse and minimize changes*" [ASM+21]. Such approaches typically employ static code analysis of the monolithic system. We identified a bottom-up direction for around half of the approaches (56). The authors note that bottom-up approaches can yield fast results, but the reuse of such services would be limited. This can be complemented by combining both directions in the service identification, which we found in 21 of the reviewed approaches.

**Automation** For the automation capabilities of approaches, we as well adhere to the basic categorization by Abdellatif et al., who distinguish *Manual*, *Semi-Automatic*, and *Automatic* approaches [ASM+21]. Manual approaches rely solely on human expertise and provide guidance in the form of instructions and procedures to follow. Semi-automatic approaches allow for an automation of single steps or parts in the service identification, while a fully automated approach offers comprehensive tool support. We identified 44 manual and 33 semi-automatic approaches, while 16 approaches stated to offer an automatic refactoring. Here, we relied on the statements made by the authors in the respective publications.

**Analysis Type** The analysis type is commonly associated with the technique used, which we will discuss in the next paragraph. In Section 2.2.1, we discussed the different analysis types that refactoring approaches employ. *Static* analysis is used by the vast majority of 89 approaches and represents a common method to analyze the existing application's source code without executing it. Its counterpart, *Dynamic* analysis, is used by 24 approaches and relies on observing the application at runtime. Commonly, the execution of certain use cases or scenarios is required to simulate the application's typical behavior [ASM+21]. Alternatively, a production system can be instrumented and monitored over a certain timeframe. It is often combined with static analysis techniques to achieve a more profound characterization of the system.

While *Lexical* (13) and *Historic* (5) analysis types are subordinated to static analysis, we reflect them separately to point out their specifics. Lexical

analysis leverages textual similarities between identifiers of classes, methods, or interface descriptions, while historic analysis employs meta-data from a version control system. Both analysis types are usually combined with a subsequent clustering to form cohesive services. While many approaches employ one analysis type, we find for 14 approaches a combination of dynamic and static analysis.

**Technique** A refactoring approach can employ one or more techniques to generate a decomposition into microservices. The technique is closely related to the other process aspects direction, automation, analysis type, and atomic unit. *Clustering* is a common technique to group elements based on their internal homogeneity. Abdellatif et al. refer to Xu and Wunsch [XW05], who state that "*elements in the same cluster should be similar to each other while elements in different clusters should not*" [ASM+21]. The authors add that such clusters are often referred to as groups, categories, or partitions. In the context of generating highly cohesive and loosely coupled services, clustering can be considered an essential technique. It typically follows a prior analysis stage with identification of atomic units, which are then clustered to form microservices. In this context, Abgaz et al. likewise distinguish cluster-based and rule-based methods [AME+23]. We found clustering techniques in 59 of our reviewed approaches.

*Graph-Based* techniques were employed by 27 approaches and are common in the context of static code analysis, e.g., to represent source code on a higher level of abstraction. *Genetic Algorithms* are defined as "*a meta-heuristic for solving optimization problems that is based on 'natural selection' [... which] relies on the calculation of a fitness function to reach an optimal (or near-optimal) solution*" [ASM+21]. However, they add for consideration that the yielded optimization and service cut may not necessarily be architecturally relevant. Abgaz et al. similarly state that genetic algorithms can be used to find an optimal combination of packages, classes, or methods *"based on constructs such as cohesion, coupling, and semantic similarity*" [AME+23]. The authors add that they can complement a clustering technique during

decomposition. Such genetic algorithms could be identified in 13 of our reviewed approaches.

*Execution-Trace Modeling* as a technique denomination was introduced by Schröer et al. [SKM20] and is associated with a dynamic analysis type. We found this technique in 10 approaches. *Domain-Driven Design* (DDD) [Eva04] is a popular technique in the context of microservices design [GB19; MY20; PJ16; SKM20; ZLJ+19] and is considered *the* standard for achieving the postulated microservices characteristics. However, its complexity and involvement of domain expertise makes its automation difficult, which may account for the fact that we found an explicit mention of DDD in only 8 approaches. Still, many other techniques are based on the underlying concept of domain modeling [AME+23] and adhere to DDD for the identification of bounded contexts [Bog20]. *Machine Learning* techniques were reported by seven approaches, followed by "Data-Flow driven" techniques in six approaches. The latter focus on the flow and processing of data throughout the application by respective diagrams [SKM20]. *Scenario Analysis*, as introduced in [SKM20], was also found six times among our reviewed approaches and is usually associated with a top-down direction and the utilization of, e.g., use-case models as input, but can also be used with dynamic analysis techniques.

The *Wrapping/Black Box* technique type was described by Abdellatif et al. [ASM+21] and Schröer et al. [SKM20] and is characterized by leaving an existing implementation unchanged, but wrap it by an additional service layer that exposes only the needed functionality. We found such a technique in three approaches. *Multi-Tenancy Orientation* describes more a behavior or capability of the resulting services and ensures concurrently serving multiple users, which we found in the approach by Haugeland et al. [HNSC21]. Camilli et al. employ a *Performance Modeling* technique "*to drive the decomposition of existing applications by taking into account performance and scalability*" [CCRZ23].

*Custom Heuristics* comprise techniques that have been specifically developed for the decomposition task. For them, we could not clearly identify one of the above-mentioned techniques forming a basis. It constitutes the second-

largest group of techniques, with altogether 40 associated approaches. When no specific technique was stated, we classified the approach as *General Guidelines* in this category. Such approaches usually describe the decomposition on a higher level of abstraction, which we found for four approaches.

**Atomic Unit** The atomic unit describes the smallest entity or piece that an approach can segregate in the course of a decomposition. It thereby determines the smallest possible size of a microservice, given that it contains one instance of such an atomic unit. For around half of the approaches (52), the atomic unit was identified as a specific *Functionality* of the application. A *Business Capability* could be identified in this regard for 32 approaches. Common atomic units that are used by approaches employing static code analysis are *Class* (23) and *Function* (12), while the latter includes the synonymously used terms method and procedure. An *Entity* as atomic unit was mentioned 18 times and is often associated with a data-oriented analysis. The *Interface* specifications of an application served for nine approaches as an atomic unit, while such a unit could not be identified for six approaches.

### 5.3.4. Usability

The *Usability* aspects of our reviewed approaches, shown on the upper right of Table 5.3, are particularly useful for practitioners who consider applying such an approach in an industrial environment. The first category, "Validation" collects empirical evidence on the approach's practical applicability, while "Accuracy" aims to assess the generated output. The category "Tool Support" contains essential details on the approach's automation capabilities.

**Validation** As it is common practice for scientific research, the vast majority of or reviewed publications document an evaluation of the therein presented approach. A validation demonstrates the effectiveness of the suggested method and proofs the asserted result quality. Authors either provided experimental evidence gathered from fictional examples that illustrate the approach's application, while others conducted case studies, sometimes involving industry-scale systems. Abdellatif et al. distinguish industrial

and experimental validations [ASM+21] similar to Schröer et al., who differentiate case studies, experiments, and experience reports [SKM20].

An *Experiment* was documented for 47 of our approaches. It usually involves one or several smaller exemplary or fictional monolithic applications. Such applications have a clear structure without complex interrelationships, and hence allow for explaining the approach step-wise in an easily comprehensible manner. Such experiments require less effort, but risk a lack of empirical evidence on their practical applicability. However, several of such reference applications are available and allow for easy replication and result comparison. Among our reviewed approaches, we found, e.g., *JPetStore*, *AcmeAir*, *DayTrader*, and *Cargo Tracking System* used by several approaches. As an example, Tyszberowicz et al. evaluated their approach using a well-known CoCoME (Common Component Modeling Example) sample trading system application [THLL18] that often serves as a case study for software architecture modeling. To validate the consistency and correctness of the resulting decomposition, they compared their result with a manual decomposition performed by three groups of students. The validation was aided using the KAMP approach [RSHR15] to calculate the differences between the two resulting architectural models. It could be shown that their approach yields a result that is comparable to a manual decomposition.

A *Case Study* usually documents the application of the approach in a real environment, which involves external participants and exposes the study to unexpected influences and risks. The results gained from case studies can therefore be more rich and of greater significance for assessing the approach's practicability. However, due to the individual setting, comparability with other approaches is complicated. In our review, we found case studies reported for 40 approaches. In five cases, the approaches have been applied to production environments used in *Industry*, typically in the course of such a case study. For altogether 18 approaches, we could not find a documented validation activity.

**Accuracy** The accuracy of a generated decomposition is an important usability criterion. However, comparing the accuracy of refactoring approaches

with varying orientation and level of abstraction would require a well-defined measure and representative sample decompositions. As the effort of applying each approach would be immense, we can only rely on the authors' judgments. Hence, we documented an accuracy rating when the publication describing the approach made an explicit statement on it, resulting in a rating for 10 of the reviewed approaches. Abdellatif et al. [ASM+21] define percentage ranges for a high, medium, or low accuracy. We think, using the mentioned three values as an ordinal scale would be sufficiently accurate, as we rely on the authors' assessments only. Eventually, we found three approaches reporting a high accuracy, five a medium, and two a low accuracy.

**Tool Support** For altogether 36 approaches, we identified supporting tools that researchers and practitioners can leverage to automate the approach or parts of it. A tool was only considered when its use for an approach was explicitly mentioned, mere references and links to tools were ignored. Many of these tools have been developed by the authors themselves and automate the decomposition according to their developed approach. Others are third-party products that assist in a specific phase. Some approaches combine several such tools in a custom workflow that covers, e.g., instrumentation, analysis, decomposition, clustering, and visualization. While few tools are used by several of the reviewed approaches, we report the following numbers on a per-tool-basis. As such, we provide a basic classification of the 31 identified tools, reflecting their purpose and availability (see Figure 5.3).

We identified 23 *Static Analysis* tools among all approaches. They typically inspect the source code, but also models, database schemes, or other input artifacts. A typical example is the tool Structure101[1] for static code analysis and visualization. Tools for lexical analysis calculate semantic similarities between identifiers, interfaces, or notions in system specifications. Tools that actually perform the *Decomposition*, i.e., the generation of one or more

[1] https://structure101.com/

candidate service cuts, have been found 20 times. One example is the well-known Service Cutter[1] tool.

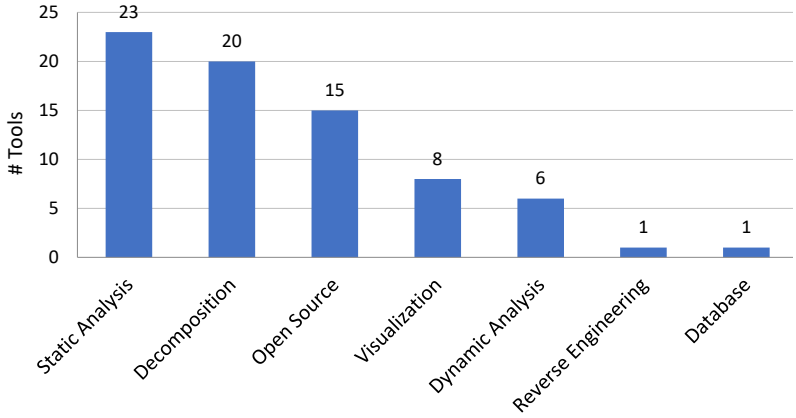



**Figure 5.3.:** Characteristics of Tools Used in Refactoring Approaches (N=31)

A *Visualization* of the existing code base or the generated service cuts is an important usability aspect and was found for eight tools. Another six tools perform a *Dynamic Analysis* by inspecting the application's runtime behavior and collect logs or execution traces, which sometimes requires instrumenting the application or performing certain scenarios or user interactions. For instance, the tool Mono2Micro "*collects runtime traces under the execution of specific business use cases of the application*" [KXK+21]. We found that dynamic analysis tools are often combined (four times) with static analysis to inspect the monolithic application holistically and generate a more appropriate service cut. Besides the mentioned tool types, Dehghani et al. [DKTT22] employ MoDisco[2], a model-driven reverse engineering framework that provides a set of tools for *Reverse Engineering*. Krause et al. [KZH+20] employ the *Database* tool DBeaver[3] that helps in identifying tables from SQL-based

[1] https://servicecutter.github.io/

[2] https://eclipse.dev/MoDisco/

[3] https://dbeaver.io/

databases. Out of the tools used in our reviewed approaches, 15 tools are developed as *Open Source* software.

### 5.3.5. Quality Goals

The vast majority of the reviewed approaches states to optimize one or more software product qualities during the architectural refactoring. Some approaches provide evidence in the form of a result comparison supported by metrics or provide qualitative evidence collected from, e.g., interviews. Others describe a general or informal improvement of certain qualities when applying the approach. We map these qualities to our identified quality attributes relevant for microservices architectures as outlined in Section 2.3. In this context, we distinguish between quality attributes and design properties, which we discussed in Section 2.3.2.

For mapping the quality attributes, sub-attributes, and design properties to our reviewed refactoring approaches, we performed a full-text search on them in each corresponding publication. For each finding, we looked for a proof, assertion, or at least an attempt which suggests that the approach improves the quality in question towards the target architecture. A mere mention of the attribute without a suggested optimization, e.g., in the context of reflecting on typical microservices characteristics, was ignored. As well, only qualities that have been explicitly named, were included in the results. We made no assumptions or conclusions based on the context, unless commonly accepted synonyms were detected. For example, “Robustness” was mentioned in one of the refactoring approaches and is defined in the IEEE Standard Glossary of Software Engineering Terminology as the “degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [ISO11a]. According to the cited standard, it directly relates to *Fault Tolerance*, while we regard it as a synonym for the same within our classification. In the same way, we treated “Flexibility” as a synonym for *Adaptability* [ISO11a]. Lastly, we found approaches that aim at reducing “Network Overhead”. When referring to the ISO Standard [ISO11a], we find *Overhead Time* subordinated to

*Performance*. Consequently, we treated "Network Overhead" as a synonym for *Time Behavior*.

For design properties, we made an exception for the property *Technology Heterogeneity* which was sometimes paraphrased. According to Alshuqayran et al. [AAE16], *Technology Heterogeneity* is equivalent to the ability to freely choose different programming languages or technologies for microservices. Accordingly, we accepted statements expressing a "technological freedom of choice" towards the target architecture. We similarly interpreted the counterpart "simplicity" of design property *Complexity* as a reduction of the same. Besides, we documented the increase/promotion of a design property, or its decrease/impediment by the approach during a decomposition. After identification of an attribute or property, we documented all findings in tabular form. To illustrate our extraction method, we inspect the refactoring approach proposed by Abdullah et al. [AIE19]. The authors conducted an experiment in which they compared a manual decomposition with their developed automatic approach, based on the quality attributes *Scalability* and *Performance*. They conclude that their approach*"yields a significant improvement of the application performance and scalability compared to the manually created microservices while reducing the overall operational cost"* [AIE19]. Based on these statements, we document the addressing of *Performance*, *Scalability*, and *Execution Cost* for this approach.

Table 5.4 shows the total of 140 assigned quality attributes and 134 assigned design properties to all approaches. For quality sub-attributes, we still found 75 assignments. For the cases, where only a sub-attribute improvement was mentioned, we counted its top-level attribute as well. The majority of 74 refactoring approaches state to optimize for one or more quality attributes, while a similar number of 70 approaches state to optimize for one or more design properties. In the following, we highlight the most often addressed qualities by discussing both groups separately.

An overview of all addressed qualities by our reviewed approaches is shown in Table 5.3. Each identified quality characteristic is assigned the number of refactoring approaches that aim for its optimization. On the lower left, top-level quality goals and design properties are listed, while on the right

**Table 5.4.:** Quality Goals of Refactoring Approaches (N=110)

| | Quality Attributes | Sub-Attributes | Design Properties |
|---|---|---|---|
| Total Assignments | 140 | 75 | 134 |
| Approaches with Assignments | | | |
| 1 or more | 74 | 51 | 70 |
| 2 or more | 40 | 20 | 42 |
| 3 or more | 18 | 4 | 16 |
| 4 or more | 6 | 0 | 5 |

side, each of the top-level attributes is broken down to its sub-attributes. An assignment indicates that the attribute or property is either improved, enabled, or optimized by applying the approach. While for the vast majority of qualities, its increase usually corresponds to an improvement, the opposite holds for *Coupling*, *Complexity*, and *Execution Cost*. A special case represents the design property *Granularity*, which indicates that an approach allows for its adjustment.

**Quality Attributes** Figure 5.4 summarizes the addressed quality attributes of all reviewed approaches. Our results show that the top-level attributes *Maintainability* (31) and *Performance* (29) are optimized most often.

The attributes *Scalability* (20), *Portability* (17), *Independence* (13), and *Reliability* (12) follow, while *Business-oriented* goals are still addressed by 11 refactoring approaches. Little attention received the attributes *Security* and *Compatibility* attribute, with only four and three approaches aiming for their improvement. When looking at the sub-attributes, we can spot *Modularity* and *Reusability* being the most prominent sub-attributes of *Maintainability*, with 16 and 8 approaches respectively. For the quality goal *Portability*, *Deployability* (11) is the dominant sub-characteristic, followed by *Replaceability* with only one mention. The *Performance* sub-attributes *Time Behavior* and

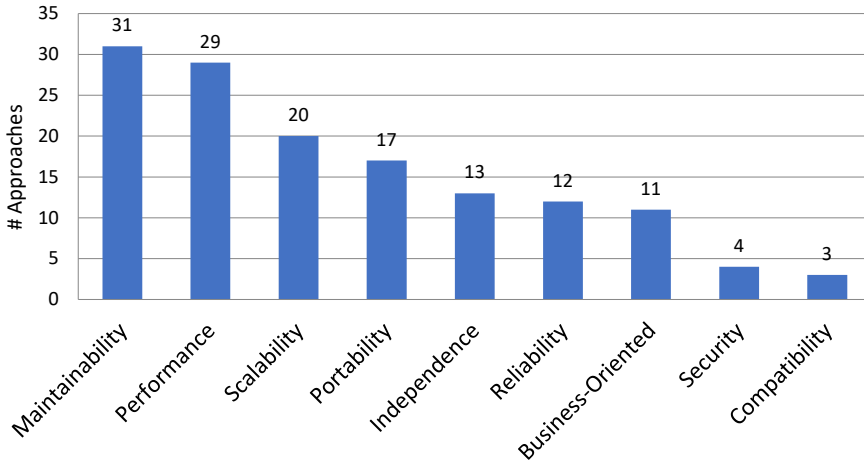



**Figure 5.4.:** Refactoring Approaches by Quality Attributes (N=110)

*Resource Utilization* are both less often addressed by the approaches, with 7 and 4 mentions, respectively. Approaches optimizing *Reliability* target the *Availability* and *Fault Tolerance* sub-attributes with 7 and 2 mentions. In the category of *Business* goals, *Execution Cost* is addressed by 4 approaches, while *Organizational Alignment* was stated for one approach. The quality sub-attributes for *Compatibility* and *Security* were less prominent: *Co-Existence*, *Interoperability*, and *Confidentiality* received one optimization each.

Not all quality sub-attributes that we identified in Section 2.3 are explicitly addressed by our reviewed refactoring approaches. No matches could be found for *Recoverability* (Reliability), *Accountability*, *Authenticity*, *Integrity* (all Security), *Installability*, *Adaptability*, *Agility* (all Portability), and *Analyzability* (Maintainability).

**Design Properties** The design property *Cohesion*, a main characteristic of individual microservices, is addressed by a total of 44 approaches. Almost half of them (20) explicitly state an increase, while the remaining approaches do not make a directional statement. A total of 43 approaches state to address the equally prominent design property *Coupling*. An explicit reduction of

coupling in the course of the decomposition is also stated for 20 approaches. Another 29 approaches address the *Granularity* aspect in the course of a decomposition, indicating its adjustability. Two approaches explicitly aim for an increase towards more fine granular services, and one approach aims for more coarse granular services. Another six approaches address the *Complexity* of the architecture during the decomposition, with one of them explicitly stating an increase, while another one states its reduction. The aspect of *Technology Heterogeneity* is considered by a total of also six approaches, with four of them aiming for an increase, while the remaining ones do state a direction. The design property *Autonomy* was detected in five approaches, with two of them suggesting an increase. Lastly, the *Isolation* property was addressed by one approach.

## 5.4. A Taxonomy for Refactoring Approaches

Based on the above outlined results, we derive a taxonomy for refactoring approaches in the context of migrating monolithic systems to microservices architectures. Table 5.5 shows the taxonomy consisting of five categories and 16 subcategories, with the values reflecting the findings of our 110 reviewed approaches. We consider our taxonomy an evolution and refinement of the taxonomy for Service Identification Approaches (SIA) proposed by Abdellatif et al. which we presented in Section 2.2.2. Our introduced extensions and adaptions reflect the specifics of the microservices architectural style, which Abdellatif et al. consider a "*variant of the service-oriented architecture style*" [ASM+21]. In particular, we regard our detailed reflection of quality goals in the migration process as a substantial improvement over their taxonomy. In Part IV of this thesis, we present a series of evaluation activities which serve as a basis for the validation of this taxonomy. Given the inherent difficulties of a thorough validation [ASM+21], a more detailed investigation of its strengths and limitations should be considered as subject to future research.

**Table 5.5.:** Taxonomy for Refactoring Approaches Towards Microservices

| Category | Subcategory | Values |
|---|---|---|
| **Input** | Domain Artifacts | Documentation, Human Expertise, Ontology, Version Control History |
| | Runtime Artifacts | Log Traces, User-Application Interactions |
| | Model Artifacts | Business Process Model, Class Diagram, Custom Model, Data-Flow Diagram, Entity Model, Activity Diagram, Use Case Model |
| | Executables | API/Interface, Database, Source Code, Test Cases |
| **Process** | Process Strategy | Continuous Evolution, Greenfield, Refactor, Rewrite/Rebuild, Strangler |
| | Direction | Bottom-Up, Mixed, Top-Down |
| | Automation | Automatic, Manual, Semi-Automatic |
| | Analysis Type | Dynamic, Historic, Lexical, Static |
| | Technique | Clustering, Custom Heuristic, Data-Flow Driven, Domain-Driven Design, Execution-Trace Modeling, General Guidelines, Genetic Algorithm, Graph-Based, Machine Learning, Performance Modeling, Scenario Analysis, Wrapping/Black-Box |
| | Atomic Unit | Business Capability, Class, Entity, Function, Functionality, Interface |
| **Output** | Representation | Guideline/Workflow, List of Services, Source Code, Splitting Recommendations, Visualization |
| **Usability** | Validation | Case Study, Experiment, Industry |
| | Accuracy | High, Medium, Low |
| | Tool Support | Database, Decomposition, Dynamic Analysis, Open Source, Reverse Engineering, Static Analysis, Visualization |
| **Quality** | Quality Goals | Business-Oriented, Compatibility, Independence, Maintainability, Performance, Portability, Reliability, Scalability, Security |
| | Design Properties | Autonomy, Cohesion, Coupling, Complexity, Granularity, Isolation, Technology Heterogeneity |

## 5.5. Limitations

There are several threats to the validity of our review. We did not strictly follow the guidelines of a systematic literature review [KC07], which may affect its repeatability and reproducibility. The publications for this review were obtained from four scientific databases and search engines. Although we may have missed relevant publications, we are confident that our snowballing activities mitigated this risk. In addition, we believe that both threats to validity are sufficiently addressed by conducting multiple review iterations with partially overlapping scopes.

All refactoring approaches were only theoretically investigated based on the available publications and provided supplementary material. Thus, all results are based on assertions of the authors and may be biased by our interpretations. A thorough investigation and evaluation of such approaches would have required their execution involving one or more sample applications, or even better, real-world systems. While our resource constraints did not allow for such an in-depth inspection, we also consulted other secondary studies and categorization schemes to minimize researcher bias. The approaches reviewed emphasize quality optimizations or improvements, with less emphasis on potential drawbacks. An approach that claims to improve one quality may result in the degradation of another quality, as described in Section 2.3.3. We did not specifically screen approaches for such trade-offs, but we believe that this aspect poses a threat to the validity of our results.

We also did not examine the extent to which a particular quality goal is influenced by an approach. Because of our evidence-gathering method, which relies on authors' claims, we believe that a comparison in this regard would not yield reliable results. However, the magnitude of such improvements is also highly dependent on the system under investigation and would be difficult to generalize. In this regard, we consider our categorization of quality goals as another reference point for architects in planning their migration. Finally, our derived taxonomy has not yet been validated by architects.

## 5.6. Conclusion

By means of a literature review, we identified and categorized 110 publications that describe an approach for the architectural refactoring and migration of monolithic software systems to microservices. All reviewed papers were published between 2016 and 2023 and have been categorized according to our extended classification scheme that builds on the groundwork by Abdellatif et al. [ASM+21]. Our classification extends the existing research with a strong focus on practical applicability. We found that almost all approaches are not universally applicable, but rely on different input data, employ different decomposition techniques, and operate at different levels of abstraction. Throughput the classification, we focused on the underlying techniques that steer a decomposition, and the approaches' automation capability. Hence, we thoroughly investigated each approach regarding its tool support and thereby identified a variety of solutions. Moreover, we identified the quality goals associated with each refactoring approach. From a practitioner's perspective, this is an essential aspect in aligning the architectural refactoring with predefined quality goals. In this regard, our classification lays the foundation for a quality-driven migration to microservices, as introduced in Section 2.4.

With the presentation of our results, we have answered the research questions on refactoring approaches and their classification. Given the prevalence of microservices in modern web-based applications, but not limited to them, it can be expected that the migration from monolithic systems will remain relevant, as confirmed by the ongoing emergence of new refactoring approaches (Figure 5.2). To allow for a systematic classification of such approaches, we propose a taxonomy consisting of five categories: input, process, output, usability, and quality. While we derived this taxonomy from our reviewed approaches, its empirical validation should be considered in subsequent research. Potential future research could, moreover, focus on comparing different approaches using adequate example or real-world applications. In this context, the quality goals claimed by authors and tool developers could also be validated.

Chapter 6

# State of Research: Automated Refactoring to Microservices

The cost-efficient and targeted architectural refactoring of monolithic systems towards microservices is still a major challenge in industry. In Chapter 5, we presented strategies and approaches that aim to partly automate the process of decomposing monolithic structures into highly cohesive and loosely coupled functional units. In this fourth contribution, we inspect a subset of the identified tools to assess their Industrial suitability. From 91 of the publications reviewed in the last chapter, we extracted 22 tools, seven of which address service decomposition. For five of them, we generated service cuts using reference applications, while specifically focusing on practical applicability. We report our experiences on installation, documentation, usability, and support, next to examining the tools' underlying techniques.

This chapter extends the following publication:

> J. Fritzsch, F. Correia, J. Bogner, and S. Wagner. "Tools for Refactoring to Microservices: A Preliminary Usability Report". In: *5th International Conference on Microservices (Microservices 2023), Pisa, Italy*. 2023 [FCBW23]

## 6.1. Context and Goal: Automating the Decomposition

While a variety of design guidelines, pattern languages, and best practices are available for greenfield developments [New21; VCG+22], the process of migrating an existing system is not easily generalized. Various methods and approaches have been suggested that aim to systematize and automate this process, which we reviewed and classified in the last chapter. Based on our findings and supported by other secondary studies [ASM+21; BBGG21; FBZW19; KBAR21; PMA19], summarized in Table 2.1, we discern a dominance of static analysis techniques operating on models, requirements, and predominantly source code. Dynamic analysis techniques are less common and mostly complements the former. Some of our consulted reviews [ASM+21; BBGG21; LCA21] also report specific tools like *Structure101, JPROF, SonarGraph Architect, DBeaver* for static analysis, and *Kieker, ExplorViz, JProfiler* for dynamic analysis. However, these tools only address the analysis part but do not generate decompositions into microservices. We detect a lacking coverage of tools that specifically address the decomposition task. In particular, we are not aware of existing studies that inspect such tools from a user perspective to understand how well they are prepared for industry adoption.

The applicability of such tools is also heavily dependent on several aspects of the system under investigation, e.g., the availability and needed effort for preparing input artifacts, or targeted quality attributes. Moreover, practitioners searching for such approaches and tools are also interested in aspects like flexibility, usability, maturity, and support, as our interview study in Chapter 4 revealed. However, our reviewed publications in Chapter 5 rarely

cover all of these aspects. Other secondary studies offer some perspective, but fall short of providing practical insights, too. They predominantly rely on specifications stated in the primary sources, without inspecting or testing the accompanying tools. Hence, in this contribution, our goal is to summarize the current state of research on automating the architectural refactoring towards microservices by reviewing available tools from the viewpoint of potential end users – software architects and developers. To refine our objective and to set the scope for our study, we formulate the following research question:

**RQ:** How practically usable are existing tools for automating the refactoring from monoliths to microservices?

We first conducted a literature review to summarize the current state of research on tool support for architectural refactoring to microservices. The retrieved publications were filtered for tools automating the decomposition into microservices. Based on their availability and operability, we subsequently installed these tools, and performed decompositions using common reference applications. During this process, we documented our experiences with their installation, documentation, usability, and support. Eventually, we combine our observations with a basic characterization of each tool, that covers its underlying techniques.

## 6.2. Method: Rapid Review

We employed the method of a *Rapid Review* [CPS20], which represents a lightweight literature review to collect evidence in a timely manner. For the scientific databases to query, we relied on the selection and keywords used in Section 5.2.1, namely ACM Digital Library, IEEE Xplore, Springer Link, and Google Scholar. As well, we applied the therein stated criteria for inclusion/exclusion, followed by backward and forward snowballing to discover potentially missed publications. Our in July 2022 conducted review yielded 91 initial publications for further consideration, shown in Figure 6.1.

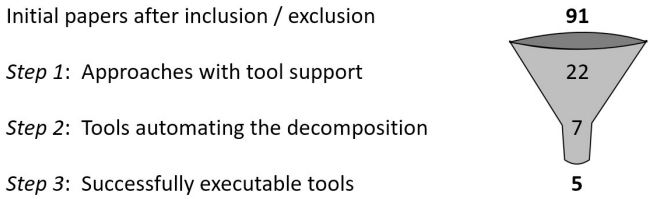



**Figure 6.1.:** Review Process: Filtering Tools Addressing the Decomposition

Based on this literature corpus, we applied subsequent filtering to identify tools addressing the decomposition. To do so, we screened the publications for typical keywords, tool names, and referenced web pages or repositories. If appropriate, we conducted a web search or contacted the authors.

This *Step 1* yielded 22 papers that report on tools for automating a part of the migration process. Auxiliary tools, e.g., tools for solely generating graphical representations, have been excluded in this step already.

In *Step 2*, we filtered the remaining publications for tools that specifically address the decomposition task. The remaining seven publications and tools have been inspected in-depth.

In *Step 3*, we acquired these tools and installed them, according to their documentation, on a dedicated system. This activity resulted in five successfully executable tools that we set up for testing with suitable reference applications. In the 91 publications, we identified several such, mostly Java-based, applications that have been used frequently for evaluations and comparisons, e.g., *JPetStore* (7), *AcmeAir* (6), *DayTrader* (6), and *Cargo Tracking System* (5). The number in brackets reflects the number of references among all publications. We chose such sample applications according to their suitability for the tools' input data format. To limit efforts, we did not use the same application for all tools, partly because result comparison was not our intended focus. Each tool was inspected and assessed regarding installation, documentation, usability, and support. Three software engineering master's students separately inspected the tools, discussed their experiences, and formed a consensus using a three-point ordinal scale (A-C),

with A reflecting a *good*, B a *medium*, and C a *poor* experience. The details of our rapid review and tool inspections can be found online[1].

## 6.3. Results

In the following, we characterize the five tools that we were able to successfully execute (Table 6.1), namely *Mono2Micro (socialsoftware)*, *Service Cutter*, *Mono2Micro (IBM)*, *Microservices Identification*, and *MonoBreaker*. While describing the concept and basic **usage**, we take the practitioner's perspective and shed light as well on **installation**, **documentation**, **usability**, and **support**.

### 6.3.1. Mono2Micro (socialsoftware)

The tool *Mono2Micro* contributed by the GitHub group *socialsoftware* which is led by Silva [CR22; NSR19; SS22]. Published in 2019 under the MIT License, the tool is currently under active development. The creators' goal was to provide an automated workflow for the identification of microservices considering the trade-off between consistency and partition tolerance in the sense of the *CAP theorem*. In this regard, the approach aims to minimize the number of distributed transactions in the context of a migration [CR22]. The **installation** based on the provided README file revealed smaller inconsistencies and errors, but led to a working instance. For maximum portability, the tool can be set up using Docker. The **usage** follows a workflow consisting of four phases: static analysis, clustering, visualization, and analysis. Unfortunately, the static analysis phase limits *Mono2Micro's* overall generalizability due to its adherence to the MVC-pattern and certain technologies for the source monolith. The following clustering step results in a dendrogram which allows for generating different clusters and decompositions. Several metrics like coupling, cohesion, and number of services characterize the generated service cut. The type of output service can be termed *entity services*, since

[1] https://doi.org/10.5281/zenodo.7949154

the tool groups database entities. We could not find a detailed **documentation** on functionality and tool usage. However, its **usability** profits from a web-based UI that visualizes the decomposition as a graph. The interface allows for several adjustments, even if functionalities are not always easy to locate and interpret. Metrics like coupling, cohesion, and number of services are generated per service cut. In [SS22], the authors performed an extensive analysis by generating decompositions for 121 monoliths, involving the tool. The authors in [NSR19] report on evaluations within two case studies that involved a comparison with another tool and an expert decomposition. The developers provided **support** by promptly answering our questions and handling our submitted GitHub issue. As the tool is in active development, we expect future updates and efforts addressing its limited applicability.

### 6.3.2. Service Cutter

The tool *Service Cutter* was contributed by Gysel et al. [GKGZ16] in 2016 and is freely available under the Apache License 2.0 License. The web-based tool follows a model-driven approach and thereby achieves a high degree of general applicability. The **installation** is straight-forward using the already prepared Docker image. The tool's **usage** requires providing the monolithic application as representations of entities, use cases, and various coupling criteria in a custom JSON format. It represents an Entity Relationship Model (ERM) and contains so called *nanoentities* (data, operations, or artifacts) and their relations. To fully leverage all capabilities, use cases and coupling criteria can be provided in addition. Once imported, different clustering algorithms and various settings allow for configurable service cuts, which are promptly generated for small applications. The effective visualization allows for detailed inspection of the decompositions which can be exported to JSON files.

The provided **documentation** comprehensively describes the concept, installation, and use of the tool, but the same level of detail is missing for the JSON format specification. However, two provided sample applications demonstrate the creation. While *Service Cutter's* **usability** profits from

the clean UI and its wide applicability, generating the necessary input for industry-scale applications is not feasible without additional tooling. Recent efforts to address this issue have been presented by the contributors of ContextMapper[1], an open-source project that aims for providing additional resources for service decompositions based on domain-driven design. The therein described *Service Cutter Input Generators* allow for generating such input files from a domain-specific language that was specifically designed for this purpose. The tool authors evaluated their tool based on two sample applications, a fictitious trading system and the DDD *Cargo Tracking System* [GKGZ16]. Besides, *Service Cutter* was used in several other publications for result evaluations and comparisons with the therein described approaches [BGR+17; CLL17; LZJ+19; SBH+22; VPAG20; VPAG21]. However, *Service Cutter* as such is no longer maintained as of March 2021, and hence developer **support** cannot be expected.

### 6.3.3. Mono2Micro (IBM)

IBM offers *Mono2Micro*, an AI-based and semi-automated tool for refactoring monolithic Java applications into microservices. Originally introduced in [KXK+21; KXL+20], the tool is under active development and can be used under a paid license with optional professional **support**. The **installation** as a portable application only requires the presence of Podman or Docker. The tool's **usage** mainly relies on a command line interface and passes four phases: data collection, data analysis, inspection of partitioning suggestions, and generation of source code. Input and output are Java source code, while intermediate results are stored in JSON files. In addition, runtime trace files of the monolith need to be generated by the provided instrumenter. A recorder then captures the execution of use cases. The now following data analysis step is performed by a cloud-based AI which generates business logic-based class groupings. The resulting partitioning suggestions can be inspected in a web UI called *workbench*. Here, the user can make adjustments

[1] https://contextmapper.org/docs/service-cutter/

**Table 6.1.:** Characterization of Executable Tools for Decomposition

| ID | Tool Name | Technique | Installation | Documentation | Usability | Support |
|---|---|---|---|---|---|---|
| 1 | Mono2Micro[a] (socialsoftware) | static code | B | - | B | B |
| 2 | Service Cutter[b] | static model | A | B | B | C |
| 3 | Mono2Micro[c] (IBM) | static & dynamic | B | A-B | B | A |
| 4 | microservices Identification[d] | static lexical | A | - | C | C |
| 5 | MonoBreaker[e] | static & dynamic | B | B-C | B | B-C |

Ratings: A = easy/good, B = fair/sufficient, C = poor/insufficient

[a] https://github.com/socialsoftware/mono2micro
[b] https://servicecutter.github.io
[c] https://www.ibm.com/cloud/mono2micro
[d] https://github.com/miguelfbrito/microservice-identification
[e] https://github.com/tiagoCMatias/monoBreaker

and inspect various metrics about the partitioning. IBM's *Mono2Micro* is the only tool in our review that automatically generates executable Java microservices, which can then be deployed as a container. However, practically, they serve merely as code skeletons due to several limitations, e.g., the exclusion of processing configuration files. IBM's *Mono2Micro* is an industry-ready tool with solid command-line **usability**, and combines static and dynamic analysis for optimal decomposition results. Its applicability, however, is limited to applications running on WebSphere Application Server Liberty or Open Liberty. The authors evaluated *Mono2Micro* against four other techniques based on a set of open-source and proprietary Java applications, involving five metrics for assessing the decomposition quality and tool efficiency [KXK+21]. The comprehensive **documentation** covers all

aspects of the tool's capabilities, installation, and usage based on a sample project. In [KXL+20], the authors document the tool's workflow based on the popular open source JEE application *DayTrader*.

### 6.3.4. Microservices Identification

*Microservices Identification* was published in 2019 on GitHub by Brito et al. [BCS21] and aims to identify plausible decompositions of a Java-based monoliths into microservices. For that, the tool relies on a semantic technique based on lexical analysis. It searches for semantic similarity of identifiers in Java source files, along with static relationships between them. According to the provided README file, the **installation** requires a Linux system with Python 3.8. Its **usage** is also based on a command line interface, but intermediate steps can be inspected in a web UI. The automated static and lexical analysis results in an Abstract Syntax Tree (AST) of the Java monolith. A lexical extraction collects all relevant identifiers from the source code. It follows a NLP typical pre-processing, including stop word removal and stemming. The resulting undirected graph represents the monolith's classes and their relationships, i.e., semantic similarities which are calculated via Latent Dirichlet Allocation (LDA) topic modeling [BNJ03]. Eventually, applying the Louvain algorithm [BGLL08] suggests a decomposition into microservices with controllable granularity. A resolution parameter allows controlling the granularity and size of resulting services, which group classes of similar semantic responsibilities. The **documentation** is brief: calculated metrics as well as command lines options are not sufficiently explained. Likewise, the solely textual output with no functionality to comfortably inspect the resulting service cuts affects the **usability**. While the tool's applicability is limited to Java-based monoliths, an adherence to certain patterns or frameworks is not required. The authors evaluated their tool against 200 Java Spring applications that are available as open source on GitHub [BCS21]. Lastly, developer **support** cannot be expected as the project is no longer maintained.

### 6.3.5. MonoBreaker

The tool *MonoBreaker* by Matias et al. [MCF+20] is the second of our inspected tools that combines static and dynamic analysis. It is available under the MIT license, and can operate on applications developed using Python with the *Django web framework* and the *Django REST framework*. This limits its applicability but makes for a refreshing exception to the prevalent Java focus of the other analyzed tools. The command-line-based **installation** and **usage** are briefly described on the GitHub page. It receives a project directory as input for static analysis, complemented by usage logs that contain information on HTTP requests and database queries observed in production over a custom period. These logs are generated using the *Django Silk* profiler. The detected dependencies during static and dynamic analysis are used to create an undirected weighted graph, which serves as input for the Girvan-Newman clustering algorithm [NG04]. The output of the tool enumerates the source code files that will belong to each service and suggests how the resulting services should communicate. Our **usability** experiences are based on running the tool over the command line in an industrial context and revealed a smooth installation and use. Its results, however, cannot be visually inspected. The tool's **documentation** is very brief, which is compensated to some extent by the Django Silk[1] profiler's comprehensive documentation on the complex dynamic analysis part. Developer **support** can not be expected either, however, as the project is no longer maintained.

## 6.4. Limitations

Our employed rapid review method bears the risk of missing out on important publications due to the less rigorous procedure compared to systematic reviews. We tried to mitigate this risk by relying on several existing reviews and performing additional snowballing. As well, the selection of tools for inspection was influenced by the researchers' capabilities in successfully

[1] https://github.com/jazzband/django-silk

installing and running them. To compensate for this threat, we relied on commonly used reference applications that have been used in evaluations of other approaches already. Our tool inspection is thus subjective to some extent, which we tried to mitigate through discussions in the group of researchers and students. Moreover, we did explicitly not assess the decomposition results of these tools, i.e., the plausibility and soundness of the generated service cuts. It would require to more carefully select appropriate reference applications, prepare inputs for the respective tools, and define measures for a meaningful result comparison. In this regard, comparing all five tools with each other was mainly hindered by the different input data formats and our limited resources for generating them. We consider this a next important step in evaluating tool support for decomposing monolithic applications into microservices.

## 6.5. Discussion and Conclusion

In this work, we reviewed the tool support for the architectural refactoring of monolithic systems to microservices, targeting the decomposition task. Generating an appropriate service cut is regarded one of the most challenging tasks, which commonly involves significant manual efforts. Automating this task can improve its efficiency, accuracy, and allows for an easier comparison of different candidate service cuts based on metrics. To gain an overview of the current tool landscape in the field of academia, we conducted a rapid review and filtered the results for executable tools targeting the decomposition.

From initially 91 approaches, we filtered out five executable and practically usable tools, which we were able to successfully set up for generating sample decompositions. Most of the remaining tools, reported in altogether 22 approaches, address only sub-tasks like analysis and visualization, were not freely available, or could not be executed successfully in our environment. All of our five inspected tools use static analysis techniques, while IBM's *Mono2Micro* and *MonoBreaker* apply dynamic analysis of the monolith

in addition. Two tools apply additional lexical analysis to detect semantic similarities based on identifiers. Three out of five tools are limited to Java-based source code, one requires Python input, while *Service Cutter* is the only language-agnostic tool. The latter has a potential to represent a general purpose tool for automating the decomposition. Unfortunately, its required custom input format makes the tool inapplicable for industry-scale applications. Efforts have been made to ease the generation of such input data, but an automated preceding static code analysis is not (yet) available. We locate the technology readiness level [Man+95] of the inspected tools between 4 and 6 with IBM's *Mono2Micro* being the most mature tool.

Despite the outlined limitations, our work can provide a first guidance to professionals seeking a tool for adoption. The implemented concepts are promising and offer a foundation for further adaption and customization for a system at hand. We encourage tool builders from inside and outside academia to follow up on these promising examples. For that, the numerous existing approaches we reviewed in Chapter 5 can serve as a valuable groundwork. To bridge the gap between academia and practice, researchers should ease the leveraging of their work by making tool prototypes publicly available. Based on the reviewed approaches and tools, we deem the following aspects essentials for practical applicability: flexibility (e.g. *Mono2Micro by socialsoftware*), generalizability and meaningful visualizations (e.g. *Service Cutter*), a sound documentation accompanied by a use case example (e.g. *Mono2Micro by IBM*), and developer support.

Our review can provide only a snapshot of the tool landscape, based on currently available scientific publications. The continuing high relevance and interest in microservices migrations might yield better solutions soon. For microservices as a field rooted in practice [DML17], involving grey literature might bring to light additional solutions that we did not cover within our scope. As important future work, we regard comparatively assessing the quality of generated service cuts, particularly involving industry-scale systems.

# Part III.

# MIGRATION METHODOLOGY

Chapter 7

# Conceptualizing a Microservices Migration Framework

Our state of practice analysis in Chapters 3 and 4 revealed prevalent challenges in the context of migrating monolithic systems towards microservices. Difficulties in the associated architectural refactoring center around a targeted and automated decomposition into services. Our subsequent state of research analysis (Chapter 5) revealed a broad spectrum of solution proposals that address these challenges. However, our interview study among software professionals (Chapter 3) also showed that such resources are rarely leveraged by practitioners. Our second interview study in the CPS domain (Chapter 4) confirmed this observation. Moreover, we identified a need for systematic guidance throughout the migration process. To address these challenges, we provide a holistic microservices migration methodology that guides architectural refactoring and focuses on practical applicability.

In our first main contribution, we report on the conceptualization and design of a framework for migrations to microservices. It aims to systematize the transformation process by implementing our in Section 2.4 introduced concept of a quality-driven migration.

This chapter extends the following publication:

> J. Fritzsch, J. Bogner, M. Haug, S. Wagner, and A. Zimmermann. "Towards an Architecture-Centric Methodology for Migrating to Microservices". In: *Agile Processes in Software Engineering and Extreme Programming – Workshops*. Ed. by P. Kruchten and P. Gregory. Cham: Springer Nature Switzerland, 2024, pp. 39–47 [FBH+24]

This concept was first presented at the "Agility with Microservices Programming" (AMP) Workshop, co-located with the XP 2022 Conference. Foundations of the presented quality model have been elaborated in the master's thesis by Koch [Koc22].

## 7.1. Context and Goals

For microservices as a field rooted in practice [DML17], grey literature is dominant for knowledge exchange among practitioners. However, academia has contributed a significant body of knowledge to date, closing the initial gap. Researchers have proposed a variety of approaches and methods to structure and automate the resource-intensive and often long-lasting migration journey. Such studies yielded novel techniques and tools, often backed by industry evaluations. However, our previous studies discerned a distinct gap between industry and academia when it comes to leveraging research results in industrial practice. By overcoming this barrier, we see a significant potential to address the identified shortcomings. Hence, our efforts in providing more systematic guidance on microservices migrations for practitioners build on the existing and often neglected body of research in this field.

During our interview study in the CPS domain, we introduced the concept of a migration framework to nine software professionals, who generally

appreciated such systematic guidance (Section 4.4.4). The framework represented the basic structure of a migration process and intends to guide architects through its three major phases. It would enable them to systematically plan their migrations, make better informed decisions, and use the most appropriate techniques and tools to transition their systems. The draft was well received by the majority of interviewees, who stated that the phases and activities depicted would adequately reflect their understanding.

An essential part of our methodology concerns the incorporation of initially set quality goals into the migration process. They ensure a structured and reasoned migration process and allow for the evaluation of results based on measurable characteristics. These quality goals also allow for monitoring the migration process and predicting the expected outcome early in the process. In particular, the framework assists architects and developers in selecting an appropriate migration approach, as described in Chapter 5. Subsequently, it supports architectural design decisions by recommending proven patterns and best practices, as discussed in Section 2.4.3. Therefore, we frame our research objective by the following two questions:

**RQ1:** How can a process framework provide a holistic view of the key activities in a microservices migration from an architect's perspective?

**RQ2:** How can quality goals be incorporated into such a framework to guide the selection of refactoring approaches, architectural design patterns, and best practices?

In the remainder of this chapter, we first discuss related frameworks that have been proposed by researchers in the context of migrating to microservices. In the same way, we reflect on quality models that pursue similar goals. We then present our methodology by detailing our framework and the incorporated quality model, thereby addressing both of our research questions. Eventually, we discuss the evaluation of our methodology and its implementation as a supporting tool for practitioners.

## 7.2. Method: Design Science Research

We apply design science research [HMPR04; PTRC07; Wie09] as a systematic guide for developing a microservices migration methodology, as described in Section 1.5. We consider the design science methodology appropriate in the context of our overarching research goal of addressing a real-world problem by incorporating state-of-the-art knowledge and seeking to evaluate our solution in practice. According to the process model by Peffers et al. [PTRC07], the following contribution represents the *design and development* in our overarching design science research process. Conboy at al. [CGC15] extended this process model to account for the innovations that agile methodologies have brought to software development. They provide an agile perspective on problem identification that "*balances procedural rigour with the need to consider empirically-driven problem/solution coevolution*" [CGC15]. Our research will follow the guidelines of Peffers et al. [PTRC07], accompanied by Conboy at al. [CGC15], to reflect the characteristics of an agile development process. Specifically, we build on the following foundations throughout our design science research process:

- Our empirical studies in Chapters 3 and 4 reveal the research gap and provide the rationale for our above stated goals. They also guide the design and development of artifacts through the insights gained from industrial practice and the requirements identified.
- We rely on our identified quality attributes and design properties relevant to the microservices architectural style, as discussed in Section 2.3. These qualities form the basis for a quality-driven migration process (Section 2.4) that our framework will implement.
- The refactoring approaches reviewed in Chapter 5 will contribute strategies, techniques, and tools for conducting the architectural refactoring. The quality attributes and design properties associated with each approach (Section 5.3.5) determine their suitability in the context of a quality-driven migration.

- In the same way, we have identified quality attributes associated with patterns and best practices for microservices and cloud-native applications (Section 2.4.3), which will subsequently facilitate a quality-oriented design of the target microservices architecture.

## 7.3. Related Frameworks and Quality Models

In the following, we review related contributions that discuss the migration process to microservices and its entailed activities, as well as quality models in this context. We therefore divide our discussion of related studies into two parts.

**Migration Process Frameworks** In their survey among 18 practitioners, Di Francesco et al. [DLM18] collected the various activities performed in a migration to microservices. They grouped them into three phases, namely reverse engineering, architecture transformation, and forward engineering. The work provides an empirically collected, bottom-up classification of common migration activities. Taibi et al. [TLP17] followed a similar survey-based approach by interviewing 21 practitioners. From their findings, they reconstructed a migration process framework that reflected the procedures and best practices of their interviewees. In addition to Di Francesco et al., they also distinguish between redevelopment and continuous evolution strategies, using the popular strangler pattern.

Wolfart et al. [WAS+21] take a more holistic approach to migration in their proposed migration roadmap. They analyzed six primary studies to come up with a unified process. To do so, the authors conducted a comprehensive systematic mapping of 62 primary studies that addressed legacy modernization to microservices. Their resulting framework represents eight activities grouped into four phases, namely initiation, planning, execution, and monitoring. Similarly, we can consider the roadmap by Bozan et al. [BLR21] on incrementally transitioning to a microservices architecture, which was distilled from interviews with 31 software experts. In contrast to the above-mentioned studies, the authors also reflect on organizational

and business-related impacts. Ayas et al. [ALH23] conducted 19 interviews and analyzed 215 StackOverflow discussions to investigate the structure of migrations to microservices. They emphasize the iterative nature of such an architectural refactoring, and identify 14 distinct activities within three phases: planning, executing, and setting up supporting artifacts. The authors also see their comprehensive analysis of the migration journey as a means for developing "*a framework on which [organizations] can base their migration*" [AA23].

Four of the five empirical studies are based on interviews with practitioners. Thus, the existing frameworks reflect common practices in industry. They structure the activities in a migration process and group them into several phases. Despite their different orientations, we can identify three common groups of activities: analysis and comprehension of the existing system, elaboration of a transformation strategy, and implementation of the transition.

**Quality Models** While all frameworks provide a solid foundation for planning a migration process, software quality and automation aspects are underrepresented in the existing works. Quality assurance aspects are reflected in some of the above discussed frameworks during the initial phase of defining requirements and strategic goals, or in the context of verification and validation activities. A general model that reflects quality characteristics and their validation for microservices was proposed by Yu et al. [YGS21]. Lichtenthäler and Wirtz [LW22] developed a quality model for cloud-native applications which was empirically validated in our follow-up study [LFW23]. The work provides a comprehensive collection of patterns and best practices, and their influence on software quality, as discussed in Section 2.5.3. The authors regard microservice-based systems as a subset or variant of cloud-native applications, justifying its relevance in our context.

However, a quality model that reflects the process of migrating to microservices in a quality-driven migration process has not yet been proposed. While the systematic review by Capuano and Muccini [CM22] discusses the role of quality attributes throughout the migration process, the step of synthesizing

a model has not yet been addressed. Based on these shortcomings and the generally limited research on quality models for microservices migrations, we provided substantial groundwork in Section 2.3. There therein identified quality attributes and design properties for microservices architectures represent the cornerstones for a quality model that integrates with our framework to be designed.

While significant advances have been made in detailing the process of a migration to microservices, a holistic methodology that combines the architectural refactoring and microservices design in a quality-driven migration process is still lacking. We aim to fill this gap by proposing a process framework that reflects all related activities and aligns them with predefined quality goals. Our concept and design also aims at practical applicability and a high degree of automation throughout the process. In this regard, our concept should foresee its potential implementation as a supporting tool for practitioners, which would also allow for its evaluation.

## 7.4. Microservices Migration Framework

In this section, we first outline our view of the migration process by describing the sequence of activities involved within three phases. We then present our concept of a framework that reflects this process, complemented by the resulting artifacts in each phase.

Based on our empirical studies and the works of Wolfart et al. [WAS+21], Taibi et al. [TLP17], Bozan et al. [BLR21], and Ayas et al. [ALH23], we constitute three phases of a migration process, namely *System Comprehension*, *Strategy Definition*, and *Architecture Definition*. Accordingly, we propose an architecture-centric framework for migrating monolithic systems to microservices, as shown in Figure 7.1. It incorporates concepts and foundations of the above-mentioned existing studies [ALH23; BLR21; TLP17; WAS+21], which influenced its structure and the activities involved. The reflected migration process can be followed for a single monolithic system, while a separate application for subsystems is also possible. In the following, we provide

a description of each of the activities involved and the resulting artifacts, which are shown as a flowchart on the right in Figure 7.1.

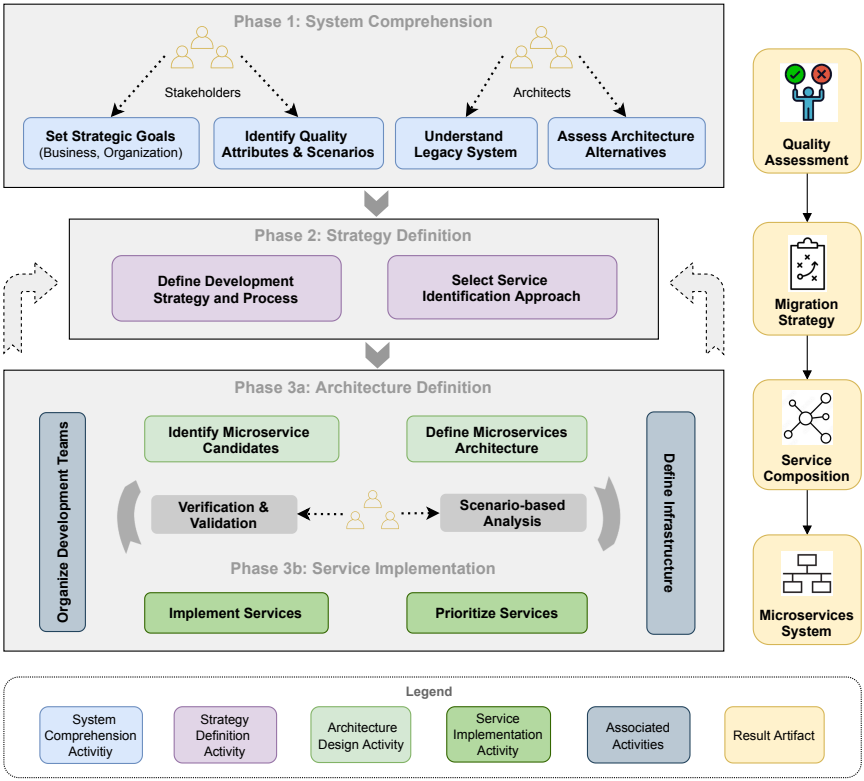



**Figure 7.1.:** Proposed Framework for Microservices Migrations

**Phase 1: System Comprehension** The initiation of a migration typically begins with an understanding of the existing monolithic system and a reflection of its capabilities in terms of goals and requirements. The definition of strategic goals, functional, and non-functional requirements in a mid- to long-term planning process provide the foundation for assessing an existing system. They allow for comparing architectural alternatives based

on strategic goals, quality attributes, and metrics [WAS+21]. Contrasting architectural styles such as monolith and microservices can also be compared using scenarios. We introduced the concept of such scenario-based architecture evaluations in Section 2.4.1.
The major activities in this first phase are typically performed during such an architecture evaluation, using, e.g., ATAM, SAAM, or a more lightweight method [KBWA94; KKC00; SM07]. We propose the use of a scenario-based method to allow for a fine-grained evaluation of design alternatives based on discrete use cases that are associated with certain software product qualities. In contrast to the scenario-based methods, Auer et al. [ALFT21] propose a quality assessment framework specifically tailored to microservices architectures, which can be considered as a time-efficient alternative. In the course of such an evaluation, both opposing architectural styles, monolith and microservices, are assessed for their suitability to fulfill the elicited scenarios and associated qualities.

The elicitation of strategic goals, quality attributes, and scenarios typically involves all system stakeholders, while architectural alternatives should be discussed by architects and developers. The result of an architecture evaluation is a quality assessment of the existing system. It supports the decision for or against an architectural refactoring to microservices, i.e., the continuation of phases two and three. While monolith and microservices represent only two out of several possible architectural styles, we exclusively focus on the contraposition of both in the context of our work.

**Phase 2: Strategy Definition** Since the initial phase of understanding the system led to a decision in favor of a migration, the following phase aims at defining an adequate strategy within two activities. The first activity concerns planning the development process based on different types of software modernization [BBGG21] as discussed in Section 2.1.4. Here, we distinguish between greenfield and brownfield developments, which are further divided into *Re-Factor* and *Re-Build* development types. Other variations, such as applying the *Strangler Application* [Fow04b] pattern or following a *continuous evolution* strategy (Section 3.4.2) involve adjacent constraints like

technological state, parallel operation, or organizational concerns including available resources and envisioned timeframe. The second activity in the strategy definition phase concerns the choice of a refactoring approach, also denoted as *service identification approach* [ASM+21]. It involves the major challenge of finding an appropriate service cut and granularity, i.e., the service boundaries and the number of resulting services.

The refactoring approaches reviewed in Chapter 5 provide a variety of strategies, techniques, and tools that are building blocks for performing both activities in this phase. The quality goals defined in the first phase, along with the specifications of the system to migrate, will guide the selection of appropriate approaches. Our taxonomy proposed in Section 5.4 allows for their systematic characterization and filtering based on various criteria. The completion of phase two will result in a migration strategy, consisting of the process and development type to follow, and the approach for service identification.

**Phase 3: Architecture Definition** After completion of the second phase, the architectural design and implementation follow in the third phase. Candidate microservices are identified according to the chosen strategy and approach, while the target architecture is designed. After prioritizing the identified services, each service implementation represents an iteration of a development cycle. In such a cycle, one or more microservices are implemented, accompanied by a quality assessment of the emerging microservices application. The focus on quality assurance aspects ensures that initially defined measures are applied and met. To this end, the implementation activities can be accompanied by a scenario-based analysis, and a verification and validation activity.

The implementation activities go in line with infrastructure build-up, and establishing DevOps processes. Adjacent constraints outlined in the previous phase set constraints for duration, required resources, as well as implicit organizational changes. The latter are particularly relevant as a consequence of changing the architecture of a system [Con18]. Our interviews with practitioners have shown that such aspects can be even more challenging than

the technical aspects of a migration (Section 3.4.4). Hence, our framework reflects the organization of development teams as a related activity. The design of phases two and three allows for backtracking to phase two to adjust the strategy or service identification approach. In this way, inadequacies in the architecture definition or even an inappropriate decomposition approach can be corrected early on, ensuring an agile migration process.

In the following section, we introduce a quality model that combines all three phases of the presented framework, with regard to a continuous quality assurance throughout the transformation.

## 7.5. A Quality Model for Migrations

We present a quality model that extends the framework presented above and implements a quality-driven migration process, as discussed in Section 2.4. Our previously identified quality attributes and design properties in the context of microservices architectures (Section 2.3), and in particular the migration process (Sections 2.4.2 and 2.4.3), provide the foundation for our proposed quality model.

Figure 7.2 illustrates the role of these quality attributes and design properties in the migration process from a monolith to microservices. The process is reflected by two refactoring activities that change the system's architecture and thereby allow for incorporating specific quality goals: Decomposition and Architecture Design. The refactoring activities correlate to phases two and three in the framework presented above. During the illustrated migration process, quality attributes influence the recommendation for a refactoring approach, as well as for patterns and best practices. For decomposition, we additionally consider the design properties introduced in Section 2.3.2. They act as mediators in addressing specific quality attributes and allow the decomposition to be controlled at a lower level. As such, the decomposition and architecture design incorporate the previously defined quality attributes. Consequently, the resulting microservices architecture satisfies the initially stated quality goals.

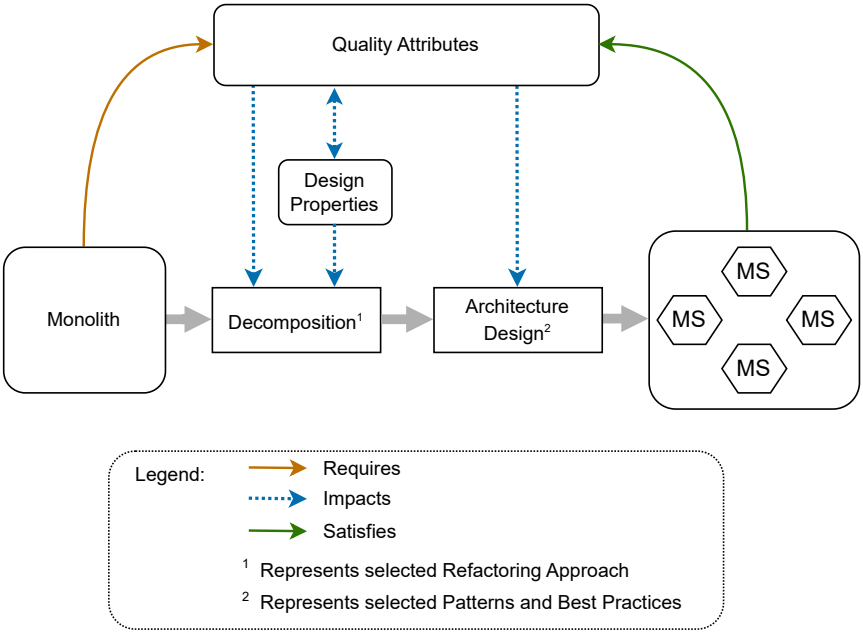



**Figure 7.2.:** Quality Attributes and Design Properties as Drivers in the Migration Process

When integrating quality goals into our above proposed framework, we can combine all three migration phases in a quality-driven migration process, as shown in Figure 7.3. In the remainder of this section, we discuss the two transitions from phase one into phases two (**T1**) and three (**T2**), respectively.

### 7.5.1. From Qualities to Refactoring Approaches

In the following, we detail the transition **T1** from phase one to phase two, as shown in Figure 7.3. We discuss how to quantify the influence of a quality goal on the recommendation of refactoring approaches. During our following considerations, we focus on quality attributes and sub-attributes, while the behavior for design properties will be derived at the end of the section.

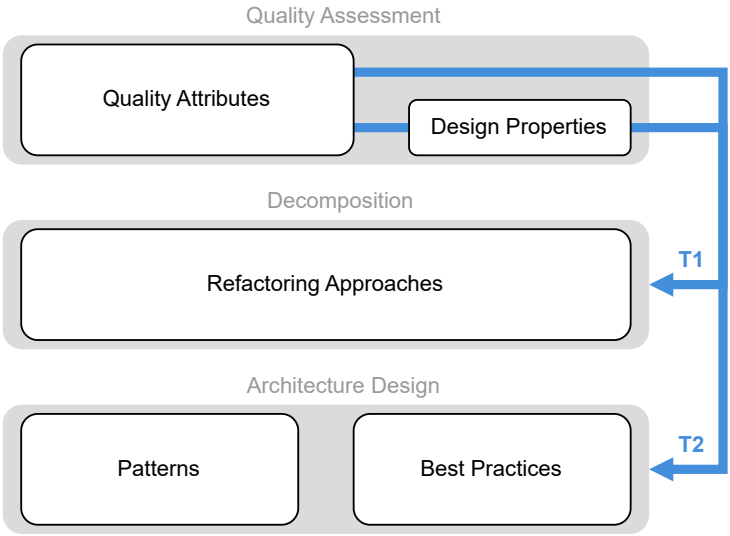



**Figure 7.3.:** Quality-Driven Migration Process

#### 7.5.1.1. Suitability Definition

To allow quantification of influence, we first establish the notion of *'suitability'* within our context. It expresses the degree of congruence with respect to quality goals, including all required quality attributes and their weightings. To quantify the suitability of a refactoring approach, we first define a set of six premises:

**Prem-1**: A desired quality attribute addressed by an approach does have a positive influence.

**Prem-2**: A desired quality attribute that is not addressed by an approach has no influence.

**Prem-3**: A *not* desired quality attribute addressed by an approach has no influence.

**Prem-4**: A *not* desired quality attribute that an approach does *not* address, has no influence.

**Prem-5**: The number of scenarios associated with a desired quality attribute is proportional to the strength of the influence.

**Prem-6**: The importance of a scenario associated with a desired quality attribute is reflected proportionally by the influence strength.

Scenario-based architecture evaluation methods (Section 2.4.1), which can be applied in *Phase 1: System Comprehension*, associate each scenario with a quality attribute. For *Phase 2: Strategy Definition*, we associated refactoring approaches with qualities that they address/optimize in a migration (Section 5.3.5). To identify suitable refactoring approaches based on certain quality goals, we propose to calculate a score that reflects the suitability of an approach for a given set of desired quality criteria. This suitability score ranges from 0% to 100%, with a score of 0% indicating the lowest, and a score of 100% indicating the highest possible suitability.

To calculate the *Suitability* score, we first define *Total Count* as the sum of all desired qualities resulting from a quality assessment. Based on this *Total Count*, we define a *Match Count* for each refactoring approach. It reflects the number of matches between the desired qualities and the qualities that an approach addresses or optimizes. Given a set of desired qualities, we can now define the *Suitability* score for a particular refactoring approach as the quotient of *Match Count* and *Total Count*:

$$Suitability = \frac{Match\ Count}{Total\ Count} \iff \frac{\text{matching qualities}}{\text{desired qualities}} \tag{7.1}$$

If an approach addresses all the desired qualities, its *Suitability* will result in 100%. On the contrary, if an approach does not address any of the desired qualities, its *Suitability* will result in 0%. The *Suitability* score is not defined, if the *Total Count* is zero, i.e., if no qualities are desired. Equation 7.1 thereby satisfies our premises **Prem-1** through **Prem-4**. To satisfy **Prem-5**, we treat all instances of a particular quality attribute separately for each scenario when calculating *Match Count* and *Total Count*. Thereby, a quality that is desired by more than one scenario is given a proportionally higher weight.

Scenario-based architecture evaluation methods also allow for the weighting of scenarios to reflect their importance in the context of a system. In Section 2.4.1, we proposed an ordinal scale based on ATAM [RKH10; Sta08], using the three values A-B-C. Since each scenario can be associated with one or more qualities, we extend a scenario's weighting to all associated qualities. In this regard, we reflect the weight of a quality attribute by assigning a factor to each importance value as follows:

*A* — Factor 3 (High importance)

*B* — Factor 2 (Medium importance)

*C* — Factor 1 (Low importance, default)

Based on these prerequisites, we also define a weighted suitability score that reflects the importance associated with each quality attribute:

$$Suitability_{weighted} = \frac{Match\ Count}{Total\ Count_{weighted}} \tag{7.2}$$

In the following, we detail the calculation of $Total\ Count_{weighted}$. First, we define $S$ as the number of all scenarios in a quality assessment, and $Q_s$ as the number of qualities for a specific scenario $s$. We then define $Total\ Count_{weighted}$ as the sum of all qualities $k$ over all scenarios, where each quality is multiplied by the importance factor $s_{importance}$ of its respective scenario, as follows:

$$Total\ Count_{weighted} = \sum_{s=0}^{S} \sum_{q=0}^{Q_s} k_{sq} \cdot s_{importance} \tag{7.3}$$

Correspondingly, a quality assigned the importance *B* will count double in the calculation of $Total\ Count_{weighted}$. A quality assigned the importance *A* will count three times, while the importance *C* reflects the default importance with a factor of one. Equation 7.2 thereby satisfies premise **Prem-6**.

#### 7.5.1.2. Suitability Score Calculation

To demonstrate the calculation of the *Suitability* and $Suitability_{weighted}$ scores, we construct the example shown in Figure 7.4. On top, it shows the result of a hypothetical quality assessment, consisting of three scenarios. Each scenario is assigned a set of quality attributes and sub-attributes.

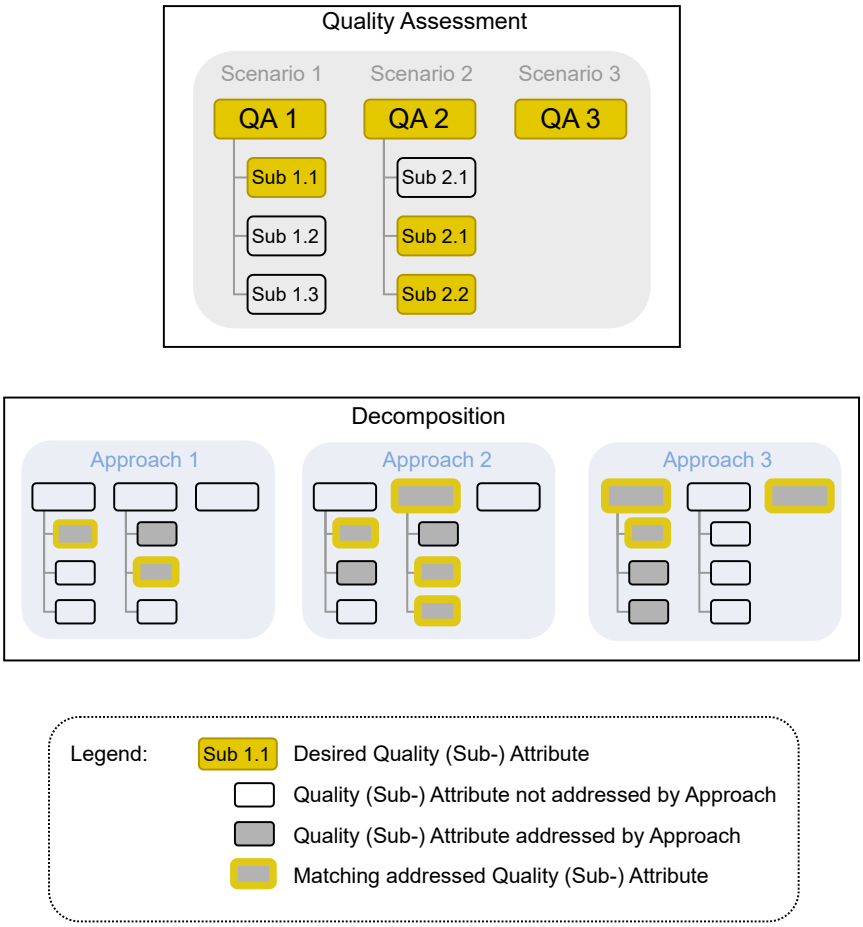



**Figure 7.4.:** Calculation of *Match Count* for Refactoring Approaches

For convenience, we do not use overlapping quality goals between scenarios, i.e., all scenarios target separate qualities (as is also recommended when using ATAM [KKC00]). One or more desired sub-attributes entail the inclusion of their top-level quality attribute, as shown for "QA 1" and "QA 2". Here, the desired quality attribute "QA 3" has no sub-attributes. Thus, our hypothetical quality assessment results in a *Total Count* of 6 qualities, which is the sum of the desired quality attributes and sub-attributes.

Below the quality assessment, three hypothetical refactoring approaches are shown. Each of the approaches addresses a subset of available quality attributes, indicated by the gray filled boxes. To calculate the *Match Count* for each of the three approaches, we sum their addressed quality attributes that match the quality assessment:

**Approach 1:** $\mathit{Match\ Count} = 2$
**Approach 2:** $\mathit{Match\ Count} = 4$
**Approach 3:** $\mathit{Match\ Count} = 3$

The resulting *Suitability* score for each approach can now be calculated according to Equation 7.1:

**Approach 1:** $\mathit{Suitability} = 2/6 \approx \mathbf{33}\%$
**Approach 2:** $\mathit{Suitability} = 4/6 \approx \mathbf{67}\%$
**Approach 3:** $\mathit{Suitability} = 3/6 = \mathbf{50}\%$

This calculated suitability now indicates how well an approach meets the given set of quality criteria. However, weighted scenarios are not yet considered. To demonstrate the calculation of $\mathit{Suitability}_{weighted}$, we assign each of the three scenarios an arbitrary importance from *A* to *C*:

Scenario 1: $\mathit{Importance} = B$ (Factor 2)
Scenario 2: $\mathit{Importance} = C$ (Factor 1)
Scenario 3: $\mathit{Importance} = A$ (Factor 3)

We now calculate the $\mathit{Total\ Count}_{weighted}$ according to Equation 7.3 as follows:

$$\mathit{Total\ Count}_{weighted} = 2 \cdot 2 + 3 \cdot 1 + 1 \cdot 3 = 10 \tag{7.4}$$

To calculate the *Match Count* for each of the three approaches, we again sum the matching qualities with the quality assessment, incorporating a scenario's importance factor:

**Approach 1:** $\mathit{Match\ Count} = 3 \quad (1 \cdot 2 + 1 \cdot 1)$
**Approach 2:** $\mathit{Match\ Count} = 5 \quad (1 \cdot 2 + 1 \cdot 3)$
**Approach 3:** $\mathit{Match\ Count} = 7 \quad (2 \cdot 2 + 1 \cdot 3)$

The resulting $\mathit{Suitability}_{weighted}$ for each of the approaches can now be calculated according to Equation 7.2:

**Approach 1:** $\mathit{Suitability}_{weighted} = 3/10 = \mathbf{30\%}$
**Approach 2:** $\mathit{Suitability}_{weighted} = 5/10 = \mathbf{50\%}$
**Approach 3:** $\mathit{Suitability}_{weighted} = 7/10 = \mathbf{70\%}$

Using a hypothetical example, we have presented the calculation of a score that quantifies the suitability of a refactoring approach for a given set of quality criteria. In addition to quality attributes and sub-attributes, design properties (Section 2.3.2) can also be reflected in the score. Since design properties are generally not part of architecture evaluation methods such as ATAM [KKC00], we suggest that architects decide on their influence separately. In our suitability score calculation, we therefore treat them like individual top-level quality attributes (as in our example "QA 3"). While they are not associated with scenarios, a weighting is not intended. Therefore, the suitability score calculation assumes a default factor of 1 for each desired design property.

### 7.5.2. From Qualities to Patterns and Best Practices

Based on the definitions provided in Section 7.5.1, we now discuss quantifying the influence of a quality goal on the recommendation for patterns and best practices. In this context, we refer to transition **T2** from phase one to phase three in Figure 7.3. While the basic concept of calculating a *Suitability* and $\mathit{Suitability}_{weighted}$ score can be applied to this transition as well, we point out its specifics.

For patterns and best practices, we have not found sufficient empirical evidence regarding their associations with design properties (Section 2.5). Therefore, we base the recommendation of patterns and best practices solely on quality attributes associated with scenarios. Furthermore, our mapping of quality attributes to 51 patterns (Section 2.5.1) and 43 best practices (Section 2.5.3) resulted only in associations with top-level attributes. Hence, in the suitability calculation, sub-attributes are always represented by their top-level quality attribute. Although beyond the scope of this quality model, potential trade-offs should be considered, as pointed out in Section 2.5.2. While a set of quality criteria leads to the recommendation of certain patterns, their application may require a trade-off with other desired qualities. As the suitability score calculation does not take such trad-offs into account, we recommend a separate assessment when selecting individual patterns.

### 7.5.3. Algorithm for Suitability Calculation

To implement the suitability calculation introduced above, we provide Algorithm 7.1. It allows to calculate the $Suitability_{weighted}$ score for a specific refactoring approach, pattern, or best practice. The input parameter $requestedQuality$ contains the desired individual qualities for all scenarios. The second parameter $matchCount$ represents the *Match Count* for the approach/pattern/best practice, which has to be calculated beforehand. Similarly, the leveraged function $getWeightedTotal()$ in line 11 returns the importance factor for an individual quality attribute or sub-attribute, based on the associated scenario and its weight.

**Algorithm 7.1** Quality Suitability Score Calculation

```
1: function CalculateSuitabilityScore(requestedQuality, matchCount)
2:     suitabulityScore ← 0
3:     weightedTotal ← 0
4:     for all qualityAttribute in requestedQuality.qualityInformation do
5:         if qualityAttribute.selection == Include then
6:             weightedTotal ← weightedTotal + getWeightedTotal(quality)
7:         end if
8:     end for
9:     for all qualitySubAttribute in requestedQuality.qualitySublevelInformation do
10:        if qualitySubAttribute.selection == Include then
11:            weightedTotal ← weightedTotal + getWeightedTotal(qualitySub)
12:        end if
13:    end for
14:    suitabilityScore ← round(matchCount / weightedTotal × 100)
15:    return suitabulityScore
16: end function
```

## 7.6. Discussion and Limitations

In our first main contribution, we proposed a holistic architecture-centric methodology for migrating monolithic systems to microservices. We first introduced the concept of a framework based on existing research. It covers the major activities in a migration and the associated architectural refactoring, grouped into three phases. Complemented by a dedicated quality model, our resulting methodology systematizes the transformation process by implementing the concept of a quality-driven migration. Based on an initial quality assessment, we propose the calculation of a suitability score to generate recommendations for refactoring approaches, patterns, and best practices. By providing systematic guidance throughout the migration process, our methodology focuses on practical applicability, addressing the challenges we have identified in industry practice. In doing so, we recognize the wide range of solutions available and improve their visibility and accessibility from a practitioner's perspective.

The resulting recommendations can serve as guidance and decision support for architects, but must be carefully reflected before implementation. In the following, we point out specific limitations and suggest ways to mitigate them. The suitability score calculation does not consider potential trade-offs between two quality attributes. We discussed such trade-offs in Section 2.3.3, where we identified typical quality attributes involved, namely *Maintainability*, *Performance*, and *Security*. While a particular decomposition technique may address or optimize for certain qualities, others may be hindered. Similar trade-offs concern the use of patterns, as discussed in Section 2.5.2. For example, the Gateway Aggregation pattern promotes *Performance* and *Maintainability* (Table A), while *Scalability* and *Fault Tolerance* have been identified as trade-offs (Section 2.5.2) for its implementation. Therefore, the interpretation of the calculated recommendations should include a reflection on such trade-offs.

The presentation of *Suitability* and $Suitability_{weighted}$ scores in the form of percentages suggests an accuracy that the quality model can hardly achieve, for several reasons. First, the decision about scenarios and their associated qualities is usually the result of a consensus among several stakeholders, which limits the accuracy in the first place. Second, the importance of the scenarios is measured on a three-point ordinal scale, which further limits the accuracy. Finally, the various refactoring approaches, patterns, and best practices do not follow a consistent scheme and do not use standardized metrics to evaluate their optimization for specific qualities.

Due to these limitations, we propose a more decent representation of the calculated suitability when implementing the quality model. We suggest the following more coarse-grained representation of the *Suitability* and $Suitability_{weighted}$ scores in the form of a *Quality Tendency*. It visually supports the interpretation of the calculated percentages with five arrow icons. For a balanced reporting, the value ranges defined below can be used to visually promote lower scores. When implementing the quality model as part of a

web-based application in the next chapter, we therefore rely on the following representation:

| | | |
|---|---|---|
| ↑ | Green arrow up | $100\% \geq \textit{Suitability} \geq 75\%$ |
| ↗ | Green arrow inclined up | $75\% > \textit{Suitability} \geq 50\%$ |
| → | Grey arrow horizontal | $50\% > \textit{Suitability} \geq 30\%$ |
| ↘ | Yellow arrow inclined down | $30\% > \textit{Suitability} \geq 15\%$ |
| ↓ | Red arrow down | $15\% > \textit{Suitability}$ |

Chapter 8

# Implementation as Web-based Application

In this second main contribution, we realize our in Chapter 7 conceptualized methodology for migrations to microservices. To guide software architects and developers through a migration scenario, we initiated the development of a web-based application[1]. Our proposed framework and its quality model form the basis for a customizable workflow throughout the transformation process, implementing the concept of a quality-driven migration (Sections 2.4 and 7.5). The application development followed an agile process with early involvement of potential users. Preliminary results of early iterations were subject to evaluations of our framework concept, methodology and usability aspects. The evaluation activities used proven methods of empirical software engineering, including interviews, surveys, and longitudinal case studies in industry. Feedback collected during the evaluations was incorporated into subsequent iterations as appropriate.

[1] https://github.com/jfr609/architecture-refactoring-helper
https://www.refactoringhelper.de

Substantial parts of the work presented in this chapter have been elaborated in the master's theses of Haller [Hal22] and Koch [Koc22]. The design and implementation of the application is attributed to the work of Tobias Haller, Daniel Koch, Qiwen Gu, and supplementary work of student assistants. The contributions are ordered by their extent and importance for this chapter.

## 8.1. Context and Goals

To evaluate the practical applicability of our proposed methodology in an industrial context, the availability of tool support is crucial. In addition, an existing implementation provides a solution for practitioners that can be used immediately. The web-based application to be developed reflects the three phases of our framework described in Section 7.4, while a customizable workflow reflects their processing sequence in a quality-driven migration process. The form of guidance for the activities involved will vary significantly due to their level of abstraction and automation capability. To provide maximum flexibility, we allow certain activities to be handled individually. At its core, the application reflects the transitions described in Section 7.5, visualized in Figures 7.2 and 7.3.

During the initial *Phase 1: System Comprehension*, the application will collect general system specifications for documentation purposes. Afterwards, it systematically collects the artifacts resulting from a conducted scenario-based architecture evaluation. The therein elicited scenarios with their associated quality attributes allow for an assessment of suitable architectural styles in the next step. As a result of this first phase, the user is presented with an architecture assessment that quantitatively compares both alternatives and thereby supports the migration decision.

In the case of a continuation to *Phase 2: Strategy Definition*, the previously collected quality goals will guide the architectural refactoring, as discussed in Section 7.5. An extensible repository of refactoring approaches proposed by academia (Chapter 5) represents the core functionality in phase

two. Therein, the application will help to find an appropriate strategy and approach for performing the architectural refactoring. The suitability of specific approaches will be based on the quality assessment of phase one, complemented by various additional properties that are reflected by our in Section 5.4 proposed taxonomy. Proceeding to *Phase 3: Architecture Definition*, the application will support the design of the emerging microservices architecture by recommending suitable patterns and best practices, also based on the initially collected quality goals. These quality-based recommendations for refactoring approaches, patterns, and best practices implement the concept of a suitability score calculation, as discussed in Sections 7.5.1 and 7.5.2.

To frame our research objective and define the scope for the design and implementation of a web-based application, we formulate the following three research questions:

**RQ1:** How can an extensible repository for migration and refactoring approaches be designed?

**RQ2:** How can a web-based application implement our conceptualized microservices migration framework?

**RQ3:** How can this web-based application be extended with a quality model to implement a quality-driven migration process?

Research question one concerns an extensible repository for hosting existing refactoring approaches based on our proposed taxonomy in Section 5.4. To account for the ongoing emergence of new approaches, the repository should allow for their addition in a user-friendly manner. In this regard, the classification scheme should also allow for adaptations and extensions of the taxonomy by adding values that new approaches may introduce. Regarding the second research question, a web-based application will be designed and implemented that reflects our microservices migration framework conceptualized in Chapter 7. At its core, the framework guides through the migration process while allowing for different workflows. For ease of administration and to prepare for multi-user support, we choose to host the repository

centrally, and thus prefer a web-based solution over a standalone application. The accompanying evaluations via surveys and interviews would equally benefit from this application model. Eventually, with regard to our overall research topic, we want to demonstrate by example the design and implementation of a microservices-based application. Research question three aims to connect all three migration phases through a common quality model. It supports the migration decision based on an initial quality assessment and ensures the adherence to predefined quality goals during the migration process.

The remainder of this chapter starts with an introduction of our agile development process. We then discuss the application structure, which entails three development areas. Subsequently, each development area is detailed, including design decisions, technology considerations, and implementation. We then present our build and deployment concept, and conclude the chapter with the integration tests we created.

## 8.2. Method: Agile Development Process

Our development process for a web-based application follows established software engineering standards, including requirements engineering, model design, implementation, and testing. In this section, we present the requirements engineering process, including prototyping, followed by our testing concept.

### 8.2.1. Requirements Engineering and Prototyping

As a first activity, we elicit and detail the requirements for our application to be developed. These requirements emerge from our analysis studies presented in PART II, refined during several discussions in the research group involving the contributors mentioned at the beginning of this chapter. A requirement is represented by its identifier, title, description, rationale, and priority. Optionally, a requirement may reference dependent requirements that are prerequisites for its implementation. Each requirement is assigned

a priority value from 1 (lowest) to 5 (highest) on an ordinal scale. We define the milestone of a minimal viable product (MVP) as being reached when all requirements with a priority of 4 or 5 have been implemented. Our requirement prioritization scheme distinguishes between the following ratings:

5 – mandatory requirement
4 – important requirement, needed for MVP
3 – important requirement, not needed for MVP
2 – desired requirement for product quality improvement
1 – optional requirement, addressed when resources available

In an agile development process, prototyping is considered important, especially in the early stages [SMMS11]. The concept of prototyping has already been described by Burns and Dennis [BD85], who contrast it with more traditional development approaches. In their systematic review on user-centered design and agile methods in software development, da Silva et al. find that prototypes can particularly support usability evaluations by inspecting and testing functionality. They find that such prototypes can be derived from user stories [SMMS11], while Li et al. derive their prototype from a use case model [LLHL05]. Use cases are a common means of eliciting, capturing, and visualizing requirements. A use case combines a set of scenarios that contribute to a user goal, while a scenario represents a sequence of actions in a well-defined order [Coc00]. Use cases can be defined as text stories or use case diagrams that provide an illustration of the actors, use cases, and the relationships between them. We therefore derive a set of use cases from previously defined requirements to separate the functionality that we implement as a prototype.

### 8.2.2. Test Strategy

To ensure quality standards for the web-based application to be implemented, we elaborate on our testing strategy. Established models distinguish three types of tests that target units of different sizes and cover different subsets of

a software system's functionality. The widely accepted test pyramid model by Cohn [Coh09] distinguishes unit, integration, and end-to-end tests. The pyramid metaphor used reflects the relationship between their number, the cost of creating and maintaining them, and their automation capability. A mixture of all test types is generally recommended to ensure a high degree of test coverage that also includes the user perspective.

However, modern software development and the emergence of web-based applications require reconsidering established testing strategies. Trautsch et al. conducted an empirical study of 38,782 test cases for 17 open-source projects and found that particularly unit and integration tests yielded no significant differences in their defect detection capabilities [THG20]. In the context of microservice-based applications, API testing as a type of integration testing is particularly relevant [Jai22]. Services typically expose their functionality through an API, which allows testing of encapsulated business functionality. Jain notes that the cost of API testing is significantly lower than end-to-end testing at the user interface layer [Jai22]. Since our application design and implementation is largely based on software engineering student assignments, we take into account the common difficulties faced by novice testers [GCOB21]. To maximize the efficiency of our testing strategy by acknowledging the aforementioned limitations, we focus on integration testing by using an established testing framework that allows for automated testing.

Cypress[1] is a modern and open-source JavaScript-based testing tool that uses the Mocha[2] framework. It is considered the industry standard for front-end testing, along with Selenium [MFS20]. The tool is targeted at end-to-end and integration testing, but also allows for unit testing of web user interfaces by mocking API calls. As we develop our web-based application, we employ Cypress for API testing, a type of integration testing. A fine-grained API to be designed should allow for an adequate test coverage of the application's main functionality.

[1] https://cypress.io/

[2] https://mochajs.org/

## 8.3. Application Model and Prototyping

In this section, we describe the application model and its structure, which leads to three main development areas. We then describe the implementation of our vertical prototype.

### 8.3.1. Client-Server Application

The application to be developed adopts the client-server model of a distributed application. Both parts, client and server, will be designed as microservices that communicate via a RESTful[1] HTTP. The design and implementation of our web-based application can be divided into the following three development areas:

1. Extensible repository for approaches, and application back-end
2. User interface and application front-end
3. Quality model and quality-driven migration process

The application back-end includes a database for persisting the extensible repository of refactoring approaches and accompanying tools. Its implementation requires the design of a relational data model for refactoring approaches. The business logic to be implemented combines the persistence layer with functionality exposed through API endpoints. The choice of technologies to be used will be driven by maintainability and performance aspects, with preference given to open-source solutions. The web-based front-end of the application acts as a presentation layer and makes the back-end functionality available to the user. It will be developed using current web technologies and best practices to maximize performance, usability, and maintainability.

### 8.3.2. Vertical Prototype

The prototype aims at providing a proof of concept for our web-based application and refining the requirements for it. We first identified the core

[1] REpresentational State Transfer

functionality that should be subject to an initial evaluation among industry experts. Accordingly, our prototype will present the user with a repository of refactoring approaches proposed by academia. The user would be able to specify a set of filtering criteria, based on the classification of such approaches. It results in a ranked result list of approaches, with further functionality to inspect each result and access the original scientific publication. We opted for a vertical prototype, which implements a specific functionality using real data and a fully functional search capability. A horizontal prototype, on the other hand, would provide a broad view of the application with little implemented functionality, which was not our intent. Based on the described scope of the prototype, we define the following set of five use cases:

1. **Browse Available Approaches** The user can browse the list of all available refactoring approaches in the repository.
2. **Inspect Approach Details** The user can inspect the details of all available refactoring approaches in the repository, based on our proposed taxonomy in Section 5.4.
3. **Refer to Original Source** The user is provided a link to the originating publication describing the refactoring approach.
4. **Search for Approaches** Starting from the home page, the user can filter the repository of available approaches based on a set of criteria according to our proposed taxonomy in Section 5.4.
5. **Inspect Search Results** The presented filtered list of refactoring approaches is ranked by the number of matching criteria, complemented by a score indicating its suitability.

The UML diagram in Figure 8.1 visualizes the use cases defined above. For the implementation, we relied on XAMPP[1], a lightweight web server solution stack that is freely available and open source. For the user interface development, we chose Bootstrap[2], a front-end framework based on HTML and CSS, that communicates with a PHP-based back-end. A detailed discussion

[1] https://www.apachefriends.org, cross-platform **A**pache + **M**ariaDB + **P**HP + **P**erl

[2] https://getbootstrap.com/

of the prototypical implementation can be found in the work by Gu [Gu20], while the results of its evaluation are presented in Section 9.1.

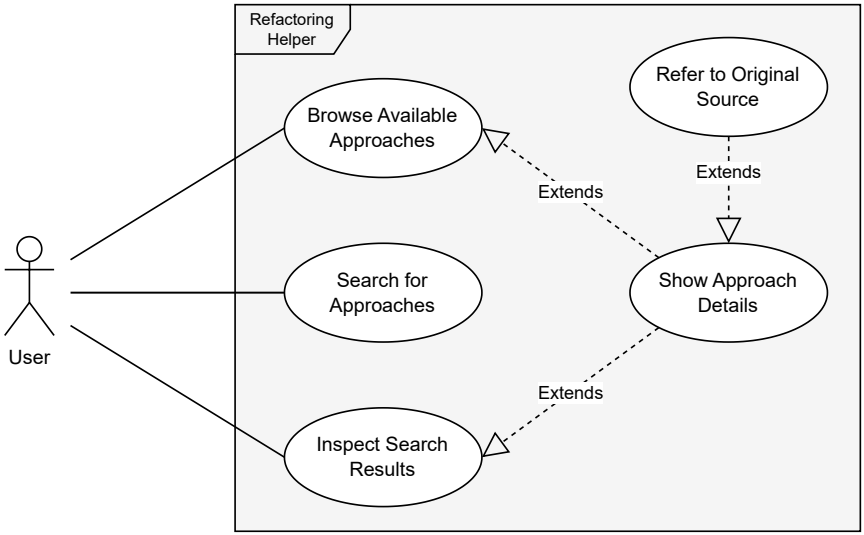



**Figure 8.1.:** Use Case Diagram for Application Prototype

## 8.4. Extensible Repository for Refactoring Approaches

The extensible repository for refactoring approaches represents the foundation of our web-based application. In this section, we present the requirements for the extensible repository and the associated back-end that accommodates the business logic. Furthermore, we discuss constraints on the design of a data model, and reason our choice of a technology stack.

Table 8.1 lists the requirements for the extensible repository and application back-end, along with their priority. While most of them were defined at the beginning of the project, several requirements were added and refined during the agile development process, e.g., as a result of the prototype's evaluation. For a detailed description of the requirements including description, rationale, and dependencies, we refer to Table C.1.

**Table 8.1.:** Requirements for Extensible Repository and Back-End

| ID | Requirement | Priority |
|---|---|---|
| BE-R1 | Extensible data model | 5 |
| BE-R2 | Approach source information | 4 |
| BE-R3 | Readable database file | 2 |
| BE-R4 | API for exposing functionality | 4 |
| BE-R5 | Data availability and exchange | 5 |
| BE-R6 | Data seeding at application start | 3 |
| BE-R7 | Reflect patterns and best practices | 3 |
| BE-R8 | Decomposition tools repository | 3 |

### 8.4.1. Data Model for Refactoring Approaches

The data model for refactoring approaches reflects our taxonomy outlined in Section 5.4. From this taxonomy, we derive entities and data fields for the resulting database tables. The resulting Entity Relationship Model (ERM) is shown in Figure 8.2. Each classification category of a refactoring approach is represented as an entity type, visualized by blue colored tables. An exception is the Quality category, which is subordinate to `Process` in the model. All subcategories are visualized as the green colored tables. The table `ApproachSource` accommodates approach source information and thus fulfills requirement BE-R2.

For the approach category entity types, the primary key is an integer ID, which has been omitted for brevity in Figure 8.2. All approach entity types are characterized by their `Name` and `Description` fields, where the `Name` field acts as primary key. However, for the `Input.Executable` entity type, the primary key is a composite of the `Name` and `Language` fields (omitted in Figure 8.2). The `RefactoringApproach` entity type has an additional unique `Identifier` field that is exposed to the user. This allows the primary key of a `RefactoringApproach` entity to be kept internal.

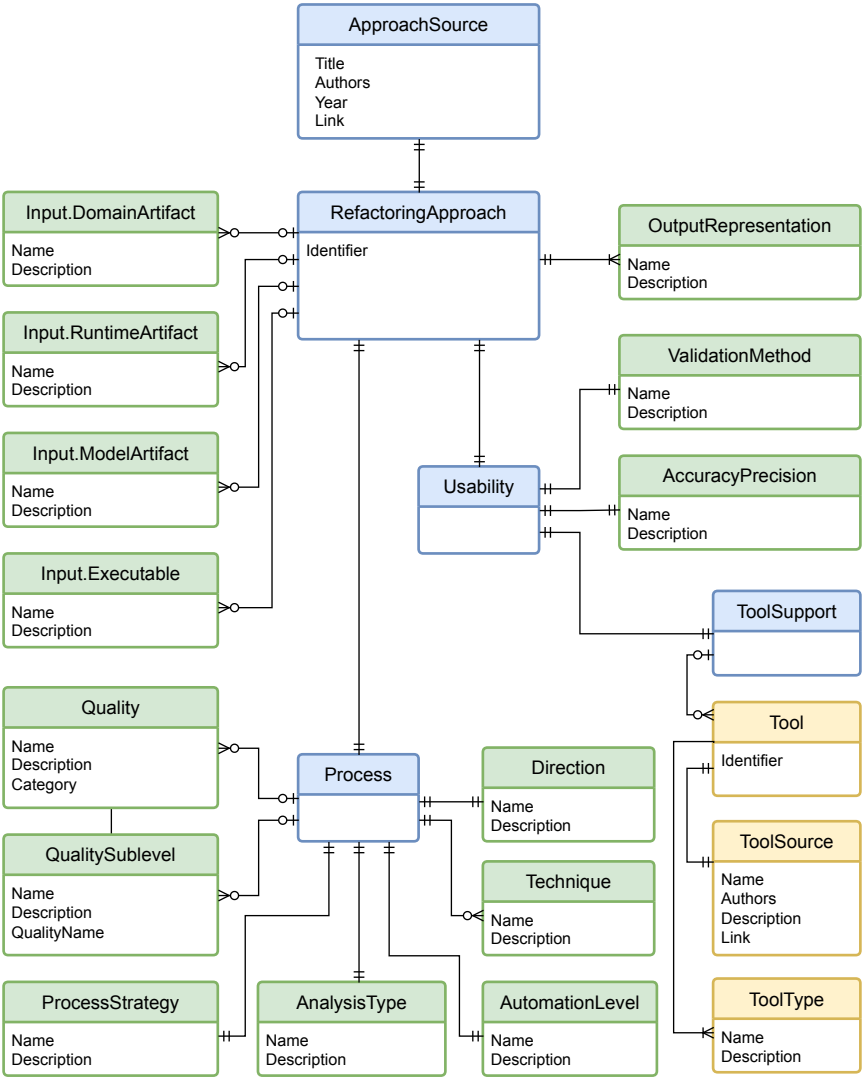



**Figure 8.2.:** Entity-Relationship Model for Refactoring Approach Repository

Subcategories of approaches such as `Technique` or `OutputRepresentation` are represented as entity types at the lowest level in the data model. Thus, the corresponding table contains one entity for each value in our taxonomy 5.4. This design allows new values to be added at runtime. In this respect, our designed data model satisfies requirement BE-R1, i.e., the extension of subcategories with additional values at runtime without changing the database schema. The initial data seeding for refactoring approaches is implemented as part of the application back-end, thereby satisfying requirement BE-R6.

#### 8.4.1.1. Data Model for Patterns and Best Practices

Patterns and best practices are represented by a separate data model, derived from the above detailed model for refactoring approaches. We provide the JSON representation of the exemplary pattern Event Sourcing in Listing 8.1, which reflects the underlying data model. The entity type `ArchitecturalDesignSource` corresponds to `ApproachSource`, and holds source information, while `Category` distinguishes between a pattern and a best practice. Each pattern and best practice entity is associated with a set of quality attributes that refer to the identical set of qualities available for refactoring approaches. The application backend equally allows patterns and best practices to be seeded from a prepared dataset at application startup, which fulfills requirement BE-R7.

### 8.4.2. Technology Stack

In the following, we outline our considerations on the technologies used for the application's back-end, separately for the programming language and framework, and the database management system.

#### 8.4.2.1. Programming Language and Framework

A basic requirement for suitable programming languages and frameworks is their ability to build server-side web applications. This requires native support for implementing **RESTful APIs**. The popular Java programming

```
{
  "architecturalDesignId": 21,
  "identifier": "21",
  "architecturalDesignSource": {
     "architecturalDesignId": 21,
    "name": "Event Sourcing",
    "description": "The pattern suggests recording the sequence of
                    events in a transaction instead of saving the
                    data directly to improve consistency in distributed
                    transactions.",
    "source": "Microservices.io by Chris Richardson",
    "link": "https://microservices.io/patterns/data/event-sourcing.html"
  },
  "category": "Pattern",
  "approachProcess": {
    "qualities": [
      {
        "name": "Security",
        "category": "Attribute"
      },
      {
        "name": "Performance",
        "category": "Attribute"
      }
    ]
  }
}
```


**Listing 8.1:** JSON Representation of the Event Sourcing Pattern

language forms the basis for a number of frameworks such as Spring Boot or Micronaut. While Java represents a more complex ecosystem, more lightweight JavaScript-based frameworks have gained popularity in recent years. While predominantly used for front-end development, its basic syntax and interpreted code allow also for building back-ends in a short time, for instance by using Node.js. Golang (Go) in combination with the Gin Web Framework would be a third alternative, focusing on performance and concurrency. However, the relatively young Golang language still lacks features such as generics, reflection, lambda expressions, or annotations. Another popular combination is C# and the .NET Framework, currently

available in version 6. The feature-rich ecosystem is comparable to Java and enables the creation of scalable and maintainable web applications. The .NET Framework is an open-source development platform that supports cross-platform development for Windows, Linux, and macOS. Compared to Java and Spring Boot, C# with .NET 6 benefits from lower resource consumption and faster HTTP/HTTPS response times [Dha21; Pre21]. Additional libraries would be needed, such as the Entity Framework Core [Mic21] for object-relational mapping (ORM), to simplify data access. Non-technical constraints represent the programmer's skills and prior experience with the technologies mentioned. In this respect, Node.js and Golang are not considered further, in the case of the latter additionally to the limitations mentioned above.

For the stated reasons, **C# and .NET 6** is our preferred back-end technology stack. Maturity, performance, maintainability, and existing expertise all favor this combination. While the .NET platform supports three different programming languages, namely C#, F# and Visual Basic [Mic23d], we adhere to C# exclusively for maintainability reasons. It offers a high degree of maturity and flexibility, as several additional packages and libraries are available exclusively for C#. The .NET Framework includes the functionality needed to build RESTful APIs according to the requirements BE-R4 and BE-R5. For the API design, we ensure their conformance to the OpenAPI specification [Ope23]. To inspect API endpoints and for general testing purposes, we use the Swashbuckle package for ASP.NET Core [Mic23b]. It allows us to generate API endpoint descriptions in accordance with the OpenAPI specification and can be used in conjunction with the included SwaggerUI.

#### 8.4.2.2. Database Management System

Regarding the choice of a database management system, we consider a relational database most appropriate to reflect our entity-relationship data model. Compared with document-oriented databases, they provide a variety of widely used and mature solutions. In addition, they ensure data integrity and consistency of transactions [SAZ+18]. Popular and freely

available relational database systems include PostgreSQL, MariaDB, MySQL, or SQLite. Since our requirements do not impose any particular constraints on performance or data volume, any of these alternatives would suffice in this regard. The .NET Framework's data access layer **Entity Framework Core** [Mic21] acts as an object relational mapper (ORM) between objects in C# and database entities and supports several relational database systems [Mic23a]. Among the mentioned solutions, we prefer **SQLite**[1] because of its wide distribution and portability. Compared to other database systems, SQLite relies on a dynamic type system with a limited set of data types. While dynamic typing offers more flexibility on the one hand, sticking to the provided data types would allow for an unproblematic migration to a more feature-rich DBMS in the future.

## 8.5. User Interface and Application Front-End

The presentation layer of our application is implemented as a web-based user interface and part of the application front-end. In this section, we first present the requirements for the application front-end, covering both the *user* and *administrator* roles. Furthermore, we reason our choice of a technology stack and discuss design and usability aspects. We conclude the section by setting up the integration between the application's front-end and back-end.

To account for differing requirements regarding the application's normal use and its administration, we decided to define two respective user roles, a *user* and an *administrator*. The *user* represents a software architect or developer that uses the application in a microservices migration context. Requirements for this *user* role will be collected in the first subsection. The *administrator* role provides a super-set of this functionality. It allows for performing administrative tasks such as adding and altering existing refactoring approaches in the repository. An *administrator* would also be allowed to extend the data model by specifying new subcategory values for

[1] https://sqlite.org/

approaches. Both user roles will share the same user interface and front-end implementation.
Table 8.2 lists the requirements for the user interface and application front-end, along with their priority. For a detailed description of the requirements including description, rationale, and dependencies, we refer to Tables C.2 and C.3.

**Table 8.2.:** Requirements for User Interface and Application Front-End

| ID | Role | Requirement | Priority |
|---|---|---|---|
| UI-R1 | user | Select framework phase | 3 |
| UI-R2 | user | Show UI Mock-ups for intended functionality | 2 |
| UI-R3 | user | Visual feedback for unimplemented phases | 1 |
| UI-R4 | user | List all existing approaches | 4 |
| UI-R5 | user | Show all information for an approach | 5 |
| UI-R6 | user | List filtered approaches | 5 |
| UI-R7 | user | Show suitability of recommended approaches | 5 |
| UI-R8 | user | Transparency for suitability calculation | 3 |
| UI-R9 | user | Configuration via custom filter | 5 |
| UI-R10 | user | Strategy filter presets | 3 |
| UI-R11 | user | Present patterns and best practices | 3 |
| UI-R12 | user | Architecture assessment representation | 3 |
| UI-R13 | user | Project management functionality | 3 |
| UI-R14 | user | Browse tools for decomposition | 3 |
| UIA-R1 | admin | Add new approaches | 5 |
| UIA-R2 | admin | Edit existing approaches | 4 |
| UIA-R3 | admin | Delete existing approaches | 3 |
| UIA-R4 | admin | Add new approach attributes | 5 |
| UIA-R5 | admin | Delete existing approach attributes | 4 |
| UIA-R6 | admin | Export existing approaches | 2 |
| UIA-R7 | admin | Import approaches from file | 1 |
| UIA-R8 | admin | Toggle administrator and user roles | 3 |
| UIA-R9 | admin | Tool management functionality | 3 |

### 8.5.1. Technology Stack

In the following, we outline our considerations on the technologies used for the application front-end. A basic requirement for front-end programming languages and frameworks is their ability to build client-side web applications. Besides, they should employ current best practices of user interface design and dynamic web pages, supported by guidelines, components, and tools. A variety of languages and frameworks are available for web-based front-end development. JavaScript is the basis for a number of popular frameworks and can be considered the de facto standard for client-side web applications. Popular examples include React.js, Vue.js, Angular, or Razor – a markup language for web pages based on .NET. While the latter integrates seamlessly with our .NET 6 and C# back-end technology, we consider this advantage less important. While looking for the most appropriate and maintainable technology, we also want to avoid potential tight coupling between client and server code.

The Angular framework is one of the most popular technologies for modern web-based user interface development, offering a wide range of features such as routing and dependency injection. Angular is primarily based on the statically typed scripting language TypeScript. In addition to class-based object-oriented programming, it offers additional language features such as generics and lambda expressions. Compared to the other JavaScript-based frameworks React.js and Vue.js, **Angular**[1] and **TypeScript** allow us to leverage existing developer expertise. For the stated reasons, we opted for the Angular framework in its most recent version 13 at project start.

For the user interface design, we adhere to Google's established design standard *Material Design* [Goo23]. It ensures a consistent layout and behavior of UI elements, while allowing the developer to leverage numerous predefined components. In this respect, the Angular framework offers **Angular Material**[2], which consists of *Material Design*-conform components for use in mobile and desktop web applications, to ensure a consistent look

[1] https://angular.io/

[2] https://material.angular.io/

and feel [Ang23]. The structure of browser views is modeled using **HyperText Markup Language (HTML)**, the standard markup language for web content. Cascading Style Sheets (CSS) are a common way to define styles for UI elements, while its extension **Sassy CSS (SCSS)**[1] allows for creating custom color themes based on Angular Material. We therefore use SCSS for describing the presentation of HTML documents. Complementary, we use **Angular Flex-Layout**[2], a third-party module that *"provides a sophisticated layout API"* [Per20] and simplifies the styling of browser views. For generating Angular model interfaces and web service clients based on the back-end provided OpenAPI specification (Section 8.4.2.1), we leverage the **ng-openapi-gen**[3] NPM module.

### 8.5.2. Design and Navigation

This section introduces the design and navigation of the user interface by describing its structure, routing, view design, and complementary functionality.

Entering the application's URL in a web browser displays the *Application Start Page* view (Figure 8.3), which represents the application's entry point. It presents the user with an image of the migration framework, with each of its phases being a clickable link to the corresponding view. This functionality to navigate to a framework phase fulfills requirement UI-R1. An "Info" icon next to the title provides a brief textual introduction for novice users. In early iterations of the development process, UI-R2 and UI-R3 were addressed to adequately reflect yet unimplemented functionality. This was applied to phases one and three, as implementation activities started with the prototyped functionality in phase two. The routing structure of the application front-end components is illustrated in Figures 8.4 and 8.5.

[1] https://sass-lang.com/

[2] https://github.com/angular/flex-layout

[3] https://www.npmjs.com/package/ng-openapi-gen

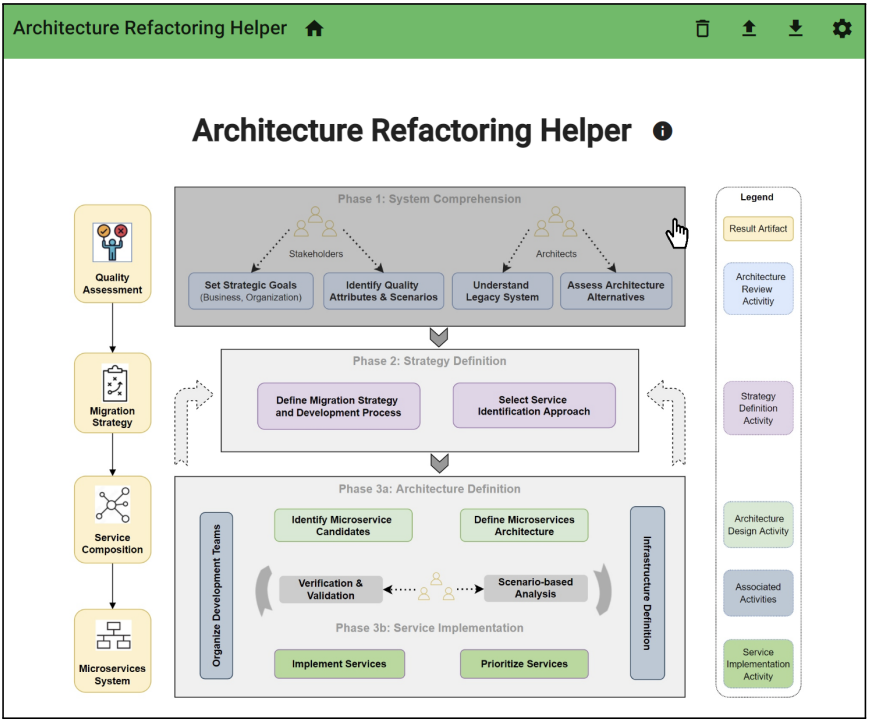



**Figure 8.3.:** Application Start Page With “Phase 1” Selected

In the following, we will proceed with the design and implementation of user interface requirements according to Table 8.2, and Appendix Tables C.2 and C.3, respectively.

**Header Toolbar** The application’s header toolbar contains the name and a “Home” button for navigating to the *Application start page* from any other view. In addition, the toolbar accommodates a “Settings” button, which allows for further configuration within a dedicated dialog. It contains functionality exclusively related to the *administrator* role and is protected by password-based authentication. Accessing the settings therefore requires switching from the default *user* role to the *administrator* role, thereby implementing requirement UIA-R8. This functionality concerns the import

and export of refactoring approaches and tools, as well their addition and modification (shown as red views in Figure 8.4). It thereby meets the requirements UIA-R6, UIA-R7, and UIA-R9. The header toolbar also includes project management functionality to persist all user-specified data for a migration project. Three icons allow to import, export, and delete a project (shown as yellow views in Figure 8.4) and thereby satisfies requirement UI-R13.

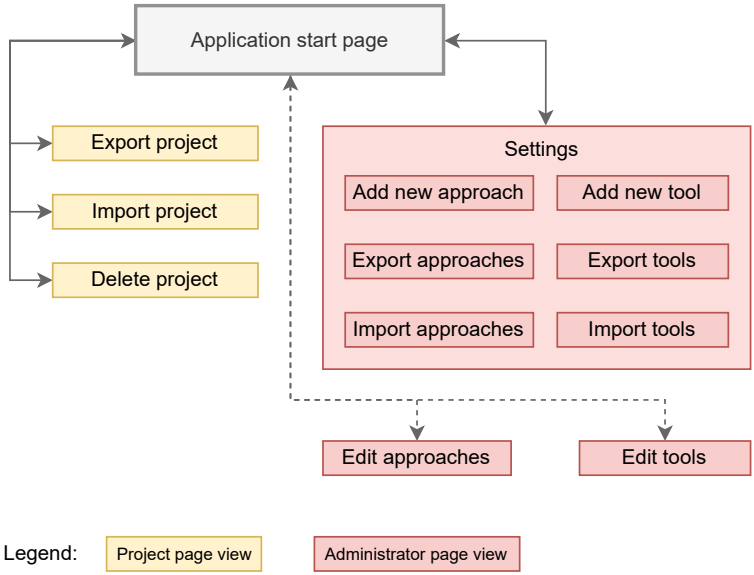



**Figure 8.4.:** Routing and Navigation for Project and Administrator Views

All views that are available to both the *user* and *administrator* roles are shown in Figure 8.5. The two administrator views "Edit approaches" and "Edit tools" in Figure 8.4 are offered to administrative users exclusively and enhance the respective "View approach details" and "View tool details" views in Figure 8.5 by an "Edit" and "Delete" button.

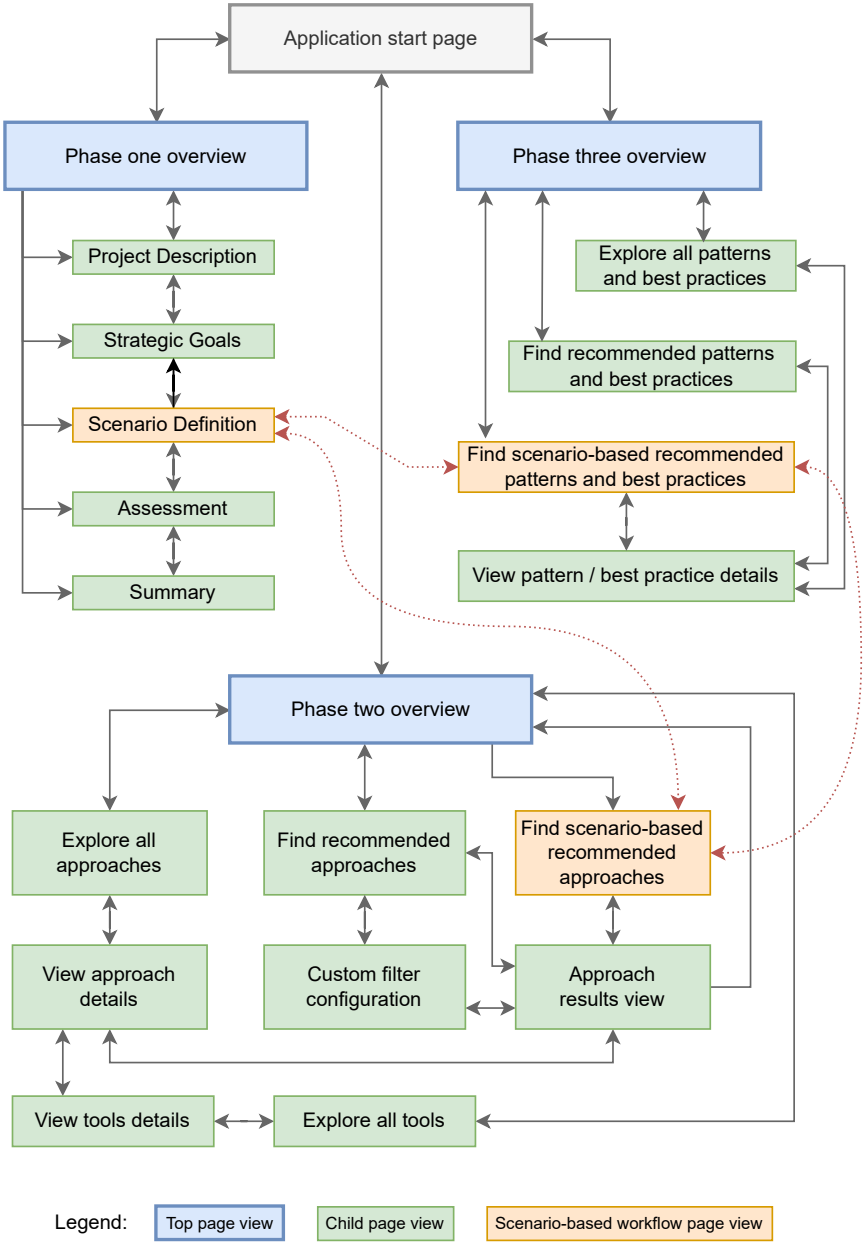



**Figure 8.5.:** Routing and Navigation for User Views

**Adding and Editing a Refactoring Approach** The functionality to add a new refactoring approach is available for administrative users, as shown in Figure 8.4. The respective dialog view based on *Material Design* guidelines is shown in Figure 8.6 and thereby realizes requirement UIA-R1. Additionally, the view implements requirements UIA-R4 and UIA-R5 for adding and removing new values for the various subcategories of an approach. The successful addition of a new refactoring approach will lead to the *View approach details* view, while canceling the action will route back to the *Application start page*. The view to edit an existing approach offers the same functionality as for adding an approach, and thereby realizes requirement UIA-R2. Both a successful or canceled editing will route back to the *View approach details* view.

**List Existing Refactoring Approaches** Starting from *Phase two overview* in Figure 8.5, one of the three routes leads to *Explore all approaches*. This view implements the functionality for requirements UI-R4 and UIA-R3 to list all approaches and delete a specific approach. For each of the listed approaches, its details can be inspected, while administrative users can also edit an approach. In addition, potential tools associated with a specific approach can be viewed.

**Detail View of Refactoring Approach** The detail view shows all available information for a refactoring approach, thereby satisfying requirement UI-R5. The view presents categories, subcategories, and values according to our taxonomy in Section 5.4. It can be accessed from *Explore all approaches*, from the *Recommended Approaches* results page, and from *View tool details* of an associated tool. Within the detail view, a "Back" button allows for navigating back to the originating view.

**Recommendation Overview** The second route originating from *Phase two overview* in Figure 8.5 leads to the *Find recommended approaches* view. It allows for filtering all cataloged refactoring approaches based on their underlying taxonomy. In addition, one of three predefined strategy filters can be applied, as shown in Figure 8.7. The predefined filters *Re-Factor*,

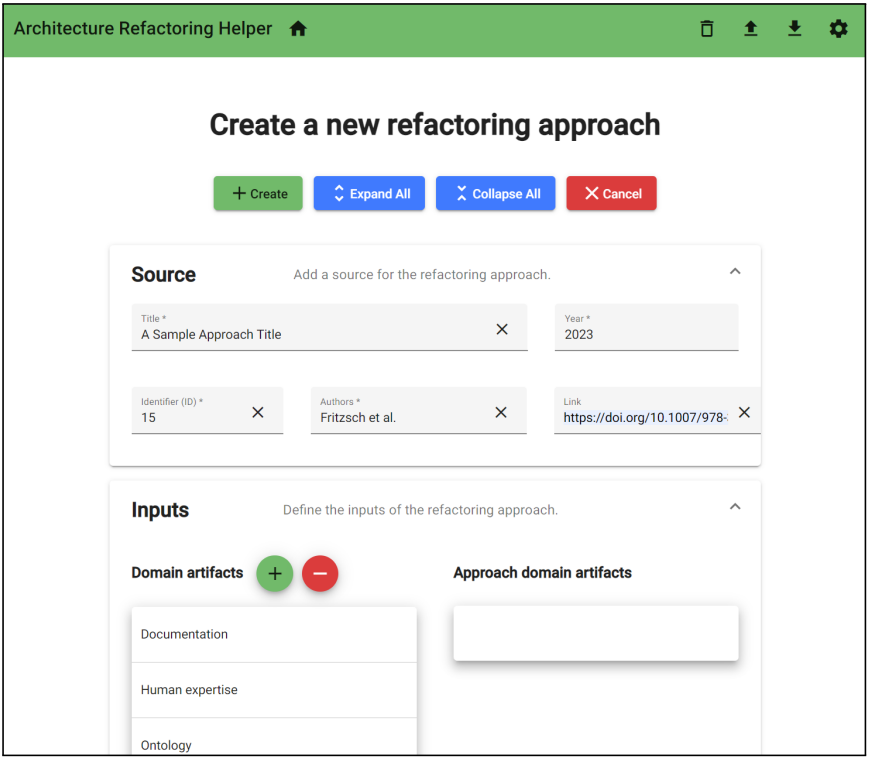



**Figure 8.6.:** Dialog View for Requirement UIA-R1 Using Material Design

*Re-Write/Re-Build*, and *New Application* development have been introduced in Section 2.1.4, and reflect the *Process Strategy* in our taxonomy (Table 5.5). Selecting a predefined filter leads directly to the "Recommended approaches" result view, while the "Configure" button leads to the intermediate "Custom filter configuration", shown in Figure 8.8. Here, the user can specifically "promote" or "neglect" an attribute. A match (presence of a promoted attribute) increases the number of matches by one, while a mismatch (presence of a neglected attribute) decreases this number. The results view presents each refactoring approach with its sum of matching and mismatching attributes. After specifying custom filter settings, the "Show Results" button routes to

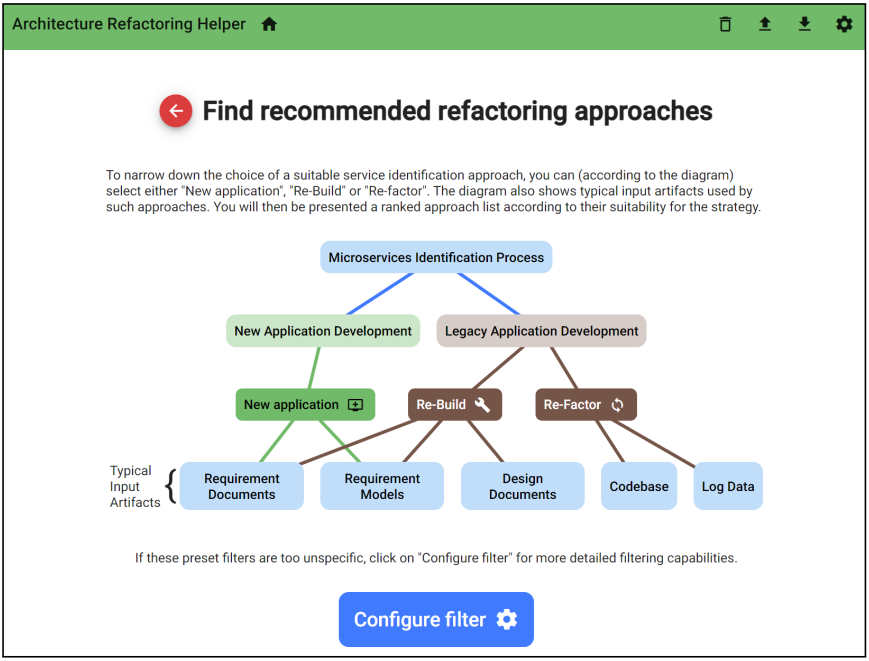



**Figure 8.7.:** View for Predefined Strategy Filters (Illustration adapted from [BBGG21])

the "Recommended approaches" result view and presents a ranked list of recommended refactoring approaches. A "Back" button allows the user to navigate back to the originating view. The outlined implementation fulfills requirements UI-R9 and UI-R10.

**Recommended Refactoring Approaches** The *Recommended Approaches* view can be accessed from the two filter views described in the previous paragraph, as well as from the *Scenario-based filter* view. It presents the user with the ranked list of recommended refactoring approaches, as shown in Figure 8.11. The ranking is determined by the matching and mismatching filter criteria, reflected as "Matches". Every item is expandable and shows the matching and mismatching attributes previously specified during

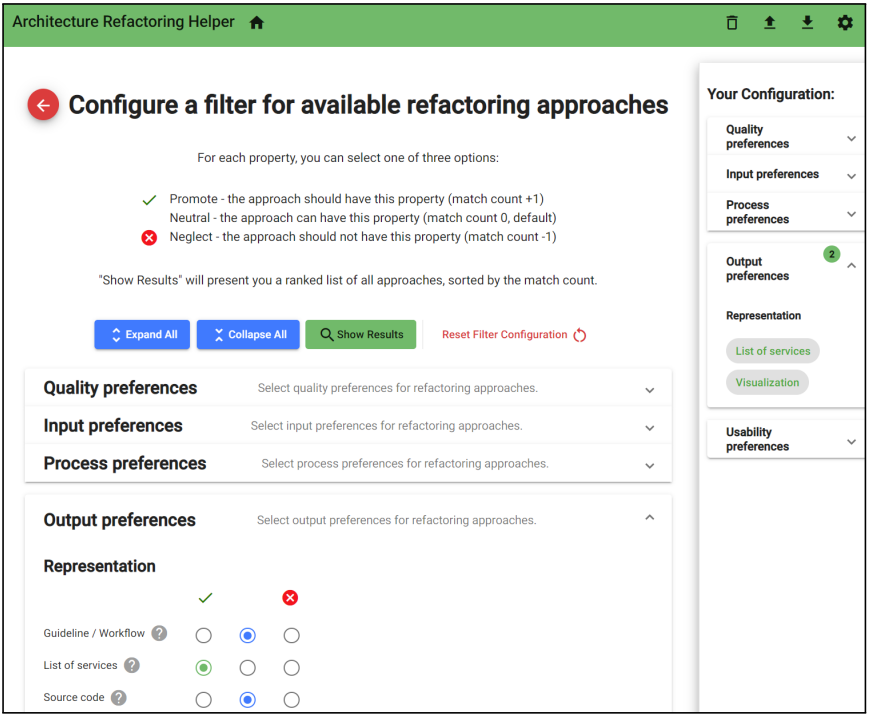



**Figure 8.8.:** Custom Filter Configuration for Refactoring Approaches

the custom filter configuration (Figure 8.8). The outlined implementation thereby meets requirements UI-R6 and UI-R7. In addition to the total number of matching filter criteria, a separate suitability score is provided for qualities. We will discuss the presentation of this quality tendency below in Section 8.6.2. A “Back” button allows navigating back to the originating view, which is especially important if the user wants to refine the current filter configuration.

**Patterns and Best Practices** The user interface design for patterns and best practices in phase three is derived from phase two. Available patterns and best practices are presented in the same view within two different tabs.

The detail view for patterns and best practices reflects their data model, as shown in Listing 8.1. The *Find recommended patterns and best practices* view uses the identical filter configuration as for refactoring approaches in phase two, but only evaluates quality criteria to generate a ranked results list. Accordingly, requirement UI-R11 for presenting patterns and best practices has been met.

### 8.5.3. Integration with Back-End

In Section 8.4.2.1, we introduced our communication concept based on a RESTful API that the application back-end uses to provide its functionality. To establish communication between client and server, we encode the server's URL in the client's environment configuration file `environment.prod.ts`. Accordingly, the file needs to be customized for a specific environment at application startup. To avoid implementing and maintaining back-end model classes and front-end API functionality twice, we use both Swashbuckle and `ng-openapi-gen` for better integration. When starting the back-end in development mode, we create a `swagger.json` file that contains a description of the API endpoints according to the OpenAPI specification. This file is then consumed by `ng-openapi-gen` to automatically generate model classes and Angular services for the application front-end. The generated services then point to the provided API endpoints and can be used throughout the front-end implementation.

## 8.6. Quality-Model Implementation

In this section, we address our third research question by implementing the quality model described in Section 7.5. Based on the above detailed back-end and front-end implementation, the quality model will extend both parts to implement a quality-driven migration process. Table 8.3 lists the requirements for the quality model, along with their priority. For a detailed description of the requirements including description, rationale, and dependencies, we refer to Table C.4.

**Table 8.3.:** Requirements for Quality Model Implementation

| ID | Requirement | Priority |
|---|---|---|
| QM-R1 | Quality model representation | 3 |
| QM-R2 | Quality assessment model representation | 3 |
| QM-R3 | Recommendations incorporating quality assessment | 3 |
| QM-R4 | Creation of scenarios | 3 |
| QM-R5 | Scenario-based workflow for approaches | 3 |
| QM-R6 | Recommended approaches reflecting suitability | 3 |
| QM-R7 | Scenario-based workflow for patterns and best practices | 3 |
| QM-R8 | Configuration overview | 2 |

### 8.6.1. Quality Assessment (Phase 1)

The quality-driven migration process starts with a quality assessment in phase one of the framework (Section 7.4). Here, we implement functionality to capture the results of a scenario-based architecture evaluation based on ATAM, as introduced in Section 2.4.1. The underlying data model reflects the elicited scenarios along with their qualities and weightings, as discussed in Section 7.5.1. To this end, we propose the entity-relationship model shown in Figure 8.9 to represent a quality assessment. The two quality entity types `Quality` and `QualitySublevel` are identical to those for refactoring approaches in Figure 8.2, and represent the underlying quality model. Its implementation in the application back-end fulfills requirement QM-R1, while the extending quality assessment model fulfills requirement QM-R2.

Figure 8.10 shows the corresponding view *Scenario Definition*. It allows entering a number of scenarios that are characterized by their `Name`, `Description` , `Importance`, and `Difficulty`. The view allows for adding new scenarios, editing and removing existing ones, and rearranging them. Each scenario can be assigned several quality attributes and sub-attributes, as shown on the right side in Figure 8.10. Sub-attributes become visible

after expanding their top-level attributes. The described implementation to create scenarios fulfills requirement QM-R4.

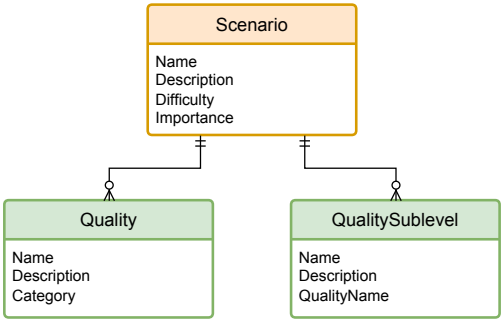



**Figure 8.9.:** Entity-Relationship Model for Scenario Representation

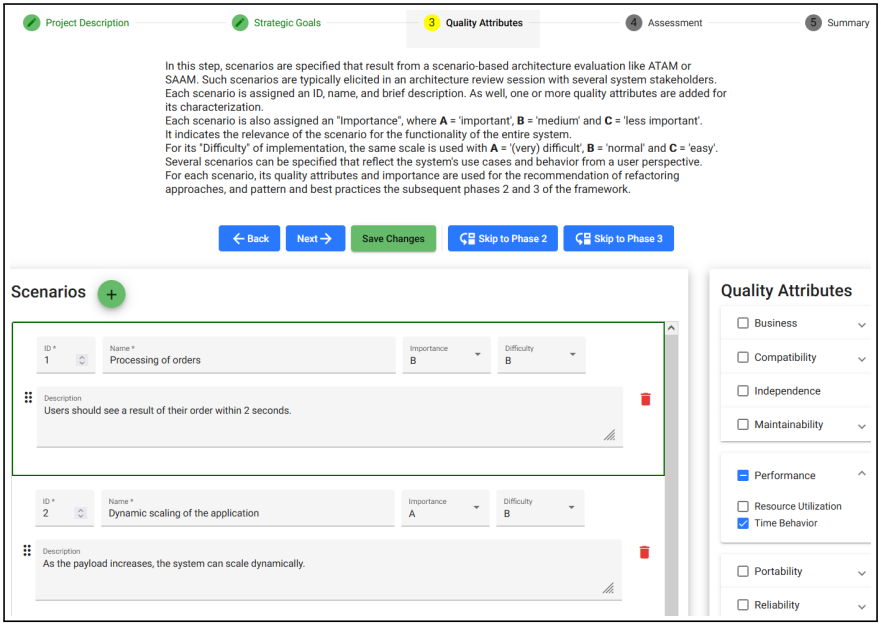



**Figure 8.10.:** User Interface View for Scenario Definition

The inclusion of scenarios and their associated qualities allows a scenario-based workflow to continue through phases two and three. While the already existing custom filtering for approaches allows for the individual inclusion/exclusion of quality attributes and design properties, the now available scenario-based workflow considers the specified scenarios and their associated qualities from phase one. To this end, additional views have been added to phases two and three to reflect this scenario-based workflow. The routing and navigation overview in Figure 8.5 highlights this workflow by the connected orange views.

### 8.6.2. Quality-Based Recommendations (Phases 2 and 3)

In phase two, a separate view *Find scenario-based recommended approaches* has been implemented, which can be accessed directly from the *Scenario Definition* view in phase one. Following the concept discussed in Section 7.5.1, the *Approach results view* has been enriched with quality-related information such as *Match Count* for quality attributes and design properties. It thereby fulfills requirement QM-R6. Figure 8.11 shows the *Approach results view* of phase two in the scenario-based workflow.

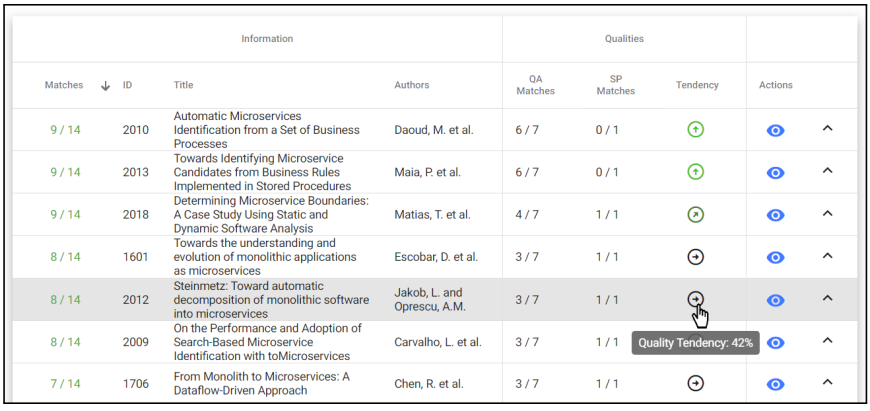



**Figure 8.11.:** Approach Results View in Scenario-Based Workflow

For each refactoring approach in the results list, the first column “Matches” shows the total number of matches for all applied filters (including qualities). It follows “ID”, “Title”, and “Authors” of the approach’s original source. The “Qualities” section contains three columns for quality-related matches: The first column “QA Matches” displays the *Match Count/Total Count* ratio for quality attributes, including sub-attributes. The second column, “DP Matches”, shows the same ratio for design properties. Eventually, the column *Tendency*, shows the calculated suitability score based on Algorithm 7.1. To ease its interpretation, we opted for a visual representation of the score in the form of tendency arrows, as elaborated in Section 7.6. However, we implemented a hover effect to also allow for inspecting the precise percentage of each approach’s suitability score.

The user interface design for patterns and best practices in phase three was derived from the implementation for refactoring approaches in phase two. Accordingly, the *Find scenario-based recommended patterns and best practices* view can be directly accessed from the *Scenario Definition* view in phase one, as well as from the corresponding view in phase two (Figure 8.5). The view generates a ranked result list of patterns and best practices, based on the quality attributes associated with scenarios. Additional filters are not evaluated, as only (top-level) quality attributes are relevant in phase three (Section 7.5.2). We thereby fulfill requirement QM-R7 for integrating patterns and best practices in the scenario-based workflow.

### 8.6.3. Configuration Overview

Providing users with a means to track the current filter configuration economizes navigation and improves the overall usability of the application. To this end, a view component has been implemented that displays scenarios and their qualities, along with other applied filters. This configuration overview complements the main views in phases two and three. To not disturb the existing layout, a side panel was used. This panel can be expanded and collapsed for each filter category. Figure 8.12 shows the configuration overview in the scenario-based workflow. During the custom filter configuration, the

configuration overview slightly differs: Scenarios are not shown, and quality attributes are combined with design properties in the “Quality Preferences” category, as shown in Figure 8.8. The implementation of the configuration overview satisfies requirement QM-R8.

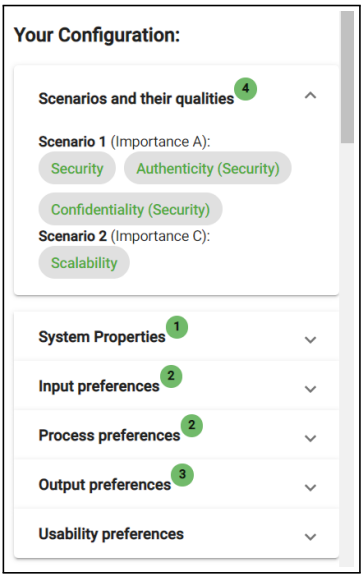



**Figure 8.12.:** Configuration Overview in Side Panel

## 8.7. Build, Test, and Deployment

In this section, we discuss the build, deployment, and testing concept of our web-based application. The developed application comprises two microservices, one containing the application back-end and one for the front-end. It can be built and deployed using either a local system or a containerized Docker environment, with the local deployment intended for development purposes. In the following, we detail the Docker configuration aimed for production use. Further information on setting up a local development

environment can be found in our application's GitHub repository[1]. We conclude the section with describing our 16 created integration test cases using Cypress.

### 8.7.1. Deployment Architecture

To handle the traffic between the application back-end and front-end, we used a reverse proxy solution. Reverse proxies help to increase scalability, performance, resilience, and security [Chr19; New15]. For our application, its main purpose is to route the incoming HTTP/HTTPS traffic to the appropriate Docker container. We use the industry-standard solution NGINX[2] which is currently used by 34% of 10,000 websites dealing with the highest traffic[3]. Figure 8.13 shows the resulting deployment architecture, which includes three containers.

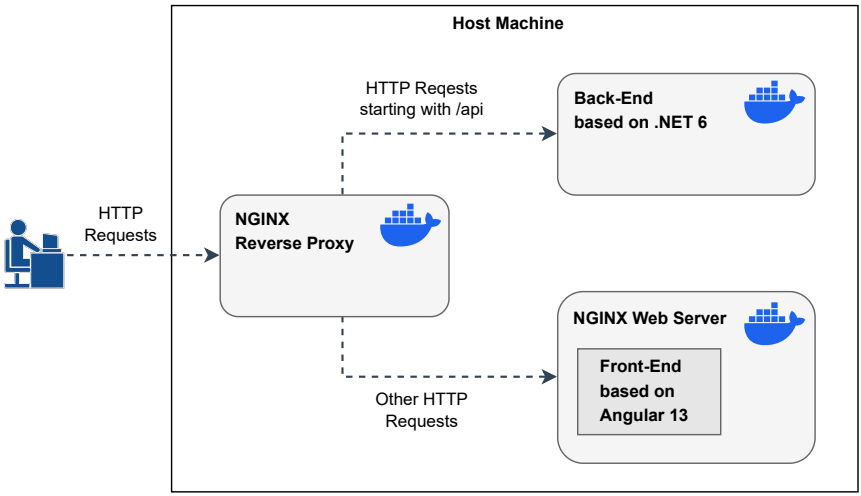



**Figure 8.13.:** Deployment Architecture (Adapted from Haller [Hal22])

[1] https://github.com/jfr609/architecture-refactoring-helper

[2] https://www.nginx.com/

[3] https://w3techs.com/technologies/cross/web_server/ranking

```
upstream frontend {
    server client;
}

upstream backend {
    server api;
}

server {
    listen 80;

    location / {
        proxy_pass http://frontend;
    }

    location /api {
        proxy_pass http://backend;
    }
}
```

**Listing 8.2:** Configuration File for the NGINX Reverse Proxy

Listing 8.2 shows the reverse proxy's adapted configuration file. The two `upstream` sections refer to the application services for the front-end (`client`) and back-end (`api`). The `server` section concerns the NGINX proxy itself and defines two routing rules: Incoming HTTP/HTTPS requests starting with `/api` are routed to back-end container, while all other HTTP/HTTPS request are routed to the front-end container.

### 8.7.2. Containerization with Docker

To build and deploy the application in a Docker environment, a `Dockerfile` for both the back-end and front-end of the application has been created. Listing 8.3 shows the `Dockerfile` for the front-end. It installs the necessary dependencies using `'npm install'`. Then, the build is started by running `'npm run build --configuration=production'`. The generated `index.html` and additional JavaScript files are copied to a separate NGINX web server instance for hosting. We further adapted the NGINX proxy's configuration to

```
FROM node:18.0.0 AS build
WORKDIR /app

RUN npm install -g @angular/cli

COPY ./package.json .
RUN npm install
COPY . .
RUN npm run build --configuration=production

FROM nginx:alpine AS final
COPY "./nginx/default.conf" "/etc/nginx/conf.d/default.conf"
COPY --from=build /app/dist/ui /usr/share/nginx/html
```

**Listing 8.3:** `Dockerfile` for the Application Front-End

ensure that all URI requests are forwarded to the web server's `index.html` file.

Containerizing the application's back-end requires building and publishing the project inside a Docker container that provides the .NET 6 SDK. Listing 8.4 shows the `Dockerfile` created for this purpose. After setting up the C# project file, the build and publish tasks are started (`'RUN dotnet build'` and `'RUN dotnet publish'`). The generated binaries are then replicated into another Docker container that provides the .NET 6 runtime environment. The listing concludes by defining the entry point for the application back-end.

To orchestrate the three resulting microservices, we use the orchestration tool Docker Compose[1]. It allows us to manage, scale, and maintain applications consisting of multiple containers. The corresponding YAML file for defining the microservices is shown in Listing 8.5. It defines the three microservices `proxy`, `client`, and `api`. The `Dockerfile` for each service is referenced in its respective `build` section. Additionally, the exposed `port` for

[1] https://docs.docker.com/compose/

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY "Repository/Repository.csproj" "Repository/"
RUN dotnet restore "Repository/Repository.csproj"
COPY . .
WORKDIR "/src/Repository"
RUN dotnet build "Repository.csproj" -c Release -o /app/build -p:
   GenerateSwagger=false

FROM build AS publish
RUN dotnet publish "Repository.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Repository.dll"]
```

**Listing 8.4:** `Dockerfile` for the Application Back-End

each service is specified. The last setting `restart` ensures that the services are always restarted automatically, e.g., after a crash of the container.

### 8.7.3. Integration Testing

To ensure the correctness of our implemented artifacts, we outline our testing concept in the following. We employed Cypress for API testing, a variant of integration testing that targets the functionality provided by the application back-end. Our total of 16 test cases cover the main use cases of our web-based application, thereby ensuring a high test coverage. The tests examine the Create, Read, Update and Delete (CRUD) operations for scenarios, refactoring approaches, and tools, complemented by a more comprehensive test case for the filtering functionality. Table 8.4 provides an

```yaml
version: "3"
services:
    proxy:
        build:
            context: ./Proxy
            dockerfile: Dockerfile
        ports:
            - "127.0.0.1:8080:80"
        restart: always
    client:
        build:
            context: ./UI
            dockerfile: Dockerfile
        ports:
            - "9000:80"
        restart: always
    api:
        build:
            context: ./ArchitectureRefactoringHelper
            dockerfile: Dockerfile
        ports:
            - "5000:80"
        restart: always
```

**Listing 8.5:** Orchestrated Services in the `docker-compose.yml` File

overview of all test cases, which can be found in our application's GitHub repository[1].

In this chapter, we have demonstrated the creation of a containerized version of our web-based applications. It allows for flexible build and deployment on any underlying infrastructure and lays the groundwork for subsequent evaluation activities.

[1] https://github.com/jfr609/architecture-refactoring-helper/tree/main/Tests

**Table 8.4.:** Integration Test Cases for the Web-Based Application

| ID | Test Case | Description |
|---|---|---|
| A1 | EmptyApproaches | Queries all existing approaches and deletes them |
| A2 | PopulateApproaches | Populates the database with 5 random approaches |
| A3 | DulicateApproach | Creates a new and duplicate approach |
| A4 | QueryApproaches | Queries all existing approaches |
| A5 | QueryApproachByID | Queries an existing and non-existing approach by ID |
| F1 | FilterApproaches | Apply a custom filter configuration for approaches |
| S1 | EmptyScenarios | Queries all existing scenarios and deletes them |
| S2 | CRUDScenario | Performs CRUD operations for a scenario |
| S3 | DulicateScenario | Creates a new and duplicate scenario |
| S4 | QueryAllScenarios | Queries all existing scenarios |
| S5 | QueryScenarioByID | Queries an existing and non-existing scenario by ID |
| T1 | EmptyTools | Queries all existing tools and deletes them |
| T2 | PopulateTools | Populates the database with 5 random tools |
| T3 | DulicateTools | Creates a new and duplicate tool |
| T4 | QueryTools | Queries all existing tools |
| T5 | QueryToolByID | Queries an existing and non-existing tool by ID |

# Part IV.

# EVALUATION AND CONCLUSION

Chapter 9

# Application Concept and Usability Evaluation

In this chapter, we present the evaluation activities conducted throughout the agile development process of our migration framework and its associated web-based application, as presented in PART III. The evaluations are an integral part of our design science process outlined in Section 7.2, and seek to provide empirical evidence on the practicality, soundness, and usability of our migration methodology and its implementation. We begin with a concept evaluation survey based on our vertical prototype among nine practitioners, researchers, and students. We then present a more comprehensive survey targeting the concept and usability of our web-based application. Nine practitioners, one professor, and one researcher evaluated its first iteration. The concept and usability of our quality-driven migration were subject to evaluation in the second and third iterations of our implemented application. We first surveyed three experienced practitioners on the quality model implementation and subsequently interviewed three experts more holistically on the quality-driven migration process.

## 9.1. Survey on Prototype (9 Participants)

To gather initial feedback on the effectiveness and usefulness of our migration concept, we conducted a web-based survey involving our implemented vertical prototype. It reflects the main use case of querying a repository of refactoring approaches based on 39 cataloged publications. A detail view would allow for inspecting all approach attributes, while a customizable filter for the various criteria would allow identifying suitable approaches for a specific system. We wanted to gather expert opinions on this core functionality of our planned web-based application, and formulated our research question accordingly:

**RQ1:** How effective and usable do potential users perceive the prototypical implementation of our microservices migration concept?

Our created questionnaire provided an introduction to the prototype's concept and purpose. It then asked for demographic information about the participant. Next, participants were asked to complete a task that was accessible on a linked GitHub page. It instructed on filtering the repository and inspecting the results and details for a specific refactoring approach. The participant could then customize the filtering and review the generated output. After that, four questions about concept, usefulness, and usability were asked, along with an input field for the time needed to complete the task. The answers to the four questions could be given on a five-point Likert scale. An additional free-text input field allowed for additional comments and feedback.

We preferred potential participants to have a technical role in software development and prior professional experience. In addition, we sought expertise with microservices and their migration from monolithic systems. Hence, we also invited the participants of our previous interview study (Chapter 3), as well as other professional contacts of the author. The survey was hosted together with a running instance of the vertical prototype for four weeks in August and September 2020. In the end, we received 9 valid responses that we subsequently analyzed using descriptive statistics.

**Table 9.1.:** Prototype Survey: Participant Demographics (N=9)

| Participant | Job Role | Professional Experience | Microservices Experience |
|---|---|---|---|
| P1 | Researcher | 5 years | 0.5 years |
| P2 | Researcher | 8 years | 3 years |
| P3 | Student and SW Developer | 1 year | 0.5 years |
| P4 | IT Consultant | 6 years | 3 years |
| P5 | SW Developer | 8 years | 5 years |
| P6 | Student | n/a | n/a |
| P7 | Student | 1 year | n/a |
| P8 | Student | 1 year | n/a |
| P9 | SW Developer | 2 years | 2 years |
| Median: | | 2 years | 0.5 years |
| Mean: | | ~3.5 years | ~1.5 years |

Table 9.1 shows the participants' demographics. Of our nine participants, four were active in industry (P3, P4, P5, P9), two were researchers (P1, P2), and three were computer science students (P6, P7, P8). P3 stated to be active in two roles, software engineering student and software developer. The average professional experience of all participants was 3.5 years, and their average experience with microservices was 1.5 years. Participant P6 had no experience in either area, while P7 and P8 had no prior experience with microservices. The demographics reveal two clusters of participants, with four of them having significant professional and microservices-related experience, while the participating students had less experience.

### 9.1.1. Results

Table 9.2 shows the participants' summarized evaluation of the prototype, based on the four questions asked. On average, our participants spent about 13 minutes completing the task. We collected their subsequent ratings on a 5-point Likert scale for the following aspects: understandability, effectiveness,

usability, and their expectations. Accordingly, each response could be given on a scale of one to five, with one indicating disagreement or the lowest rating and five indicating full agreement or the highest rating.

**Table 9.2.:** Prototype Assessment on a 5-Point Likert Scale (N=9)

| Participant | Understandability | Effectiveness | Usability | Met Expectations | Time (min) |
|---|---|---|---|---|---|
| P1 | 3 | 2 | 2 | 2 | 10 |
| P2 | 3 | 3 | 2 | 2 | 20 |
| P3 | 2 | 4 | 3 | 4 | 5 |
| P4 | 4 | 2 | 4 | 3 | 15 |
| P5 | 3 | 4 | 5 | 3 | 5 |
| P6 | 5 | 3 | 4 | 3 | 30 |
| P7 | 4 | 2 | 4 | 5 | 15 |
| P8 | 4 | 4 | 4 | 3 | 10 |
| P9 | 3 | 4 | 2 | 4 | 5 |
| Median: | 3 | 3 | 4 | 3 | 10 |
| Mean: | ~3.4 | ~3.1 | ~3.3 | ~3.2 | ~12.8 |

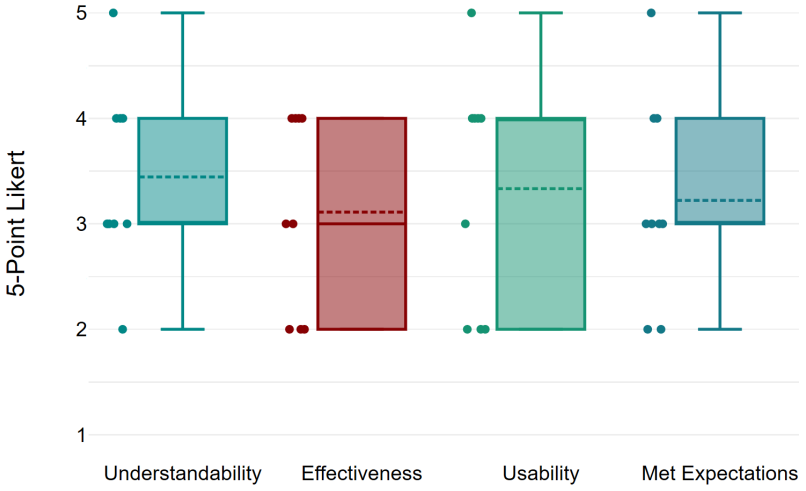



**Figure 9.1.:** Assessment of Vertical Prototype by Participants (N=9)

The "Understandability" aspect indicates whether users understood our concept, the terminology used, and the functionality provided. The "Effectiveness" aspect indicates the overall relevance and helpfulness of the information provided in the context of an architectural refactoring process, while the "Usability" aspect aimed to validate a convenient usage of the tool. Eventually, we asked participants about the extent to which the presented results "Met [their] Expectations" in a potential decision-making process.

Figure 9.1 reflects the statistical analysis via box plots, including mean, median, maximum, minimum, and quartiles of the results from Table 9.2. In summary, the ratings for all aspects were slightly positive, with a mean above 3.0. Understandability received the highest mean of 3.4. This indicates that the prototypical implementation adequately conveyed our concept, supported by the ratings for effectiveness and degree of met expectations. The usability ratings show the highest spread, which we partly attribute to the limited time and resources available to develop the prototype.

The free-text input field yielded more in-depth qualitative data, such as reasons and additional suggestions. Table 9.3 shows the participants' feedback grouped into general, filtering, and result presentation. Comments No. 1 to 4 concern the filtering view of refactoring approaches. The number of 35 possible filters was considered too many by one participant, while two others asked for a proper explanation of each filter criterion. The design and effectiveness of the radio buttons in this context was not intuitive for two participants. In this regard, comment No. 4 suggested redesigning the workflow and guiding the user through the filtering process by narrowing down the filters along the way.

Comments No. 5 to 7 address the search functionality. Our prototype had a limited set of 39 refactoring approaches available in the repository. Hence, when several filters were applied, the result list may have been empty, as two commenters noted. Comment No. 6 suggested still showing the results that the filter does not include, but for example graying them out. To improve usability, a "Reset" button should allow to clear all applied filters at once.

Comments No. 8 to 12 regard the result view, which the first comment said should be explained on the tool's introductory page. The result view provided

**Table 9.3.:** Prototype Survey: Feedback From Participants

| No. | Feedback Entry | # of Mentions |
|---|---|---|
| 1 | Too many filtering parameters | 1 |
| 2 | An explanation on each point/option would be helpful | 2 |
| 3 | The "Include/Neutral/Exclude" radio buttons are not intuitive | 2 |
| 4 | UI itself could be simplified by starting with generic selection and then getting more and more into detail, instead of having to select everything in the first place | 1 |
| 5 | The tool sometimes returns no results after search | 2 |
| 6 | Either place a "remove filters" button or always show a complete list of results when clicking "Search", but gray out those that are not relevant according to the selected search string | 1 |
| 7 | Reset button is recommended | 1 |
| 8 | Clarify the result view in the introduction | 1 |
| 9 | The algorithm behind this filtering is unclear to the user | 1 |
| 10 | Is it necessary to differ between a "Score" and "Recommendation"? | 1 |
| 11 | The "Score" and its definition is unclear | 1 |
| 12 | "Recommendation" in percent is not very helpful. A percentage range mapping to a term would be helpful (e.g. 0-20% = not worth, 20-40% = xyz etc.) | 1 |
| 13 | Quality Metrics/Intention part feels a little obsolete | 1 |
| 14 | Explain tool's function and purpose: How to help making decisions about the migration | 1 |
| 15 | Possibility of using NLP to automatically label new papers | 1 |
| 16 | Use Bootstrap to help to generate the front-end pages quickly | 1 |

a "Recommendation" percentage that was calculated based on the number of matching and mismatching filter criteria. This percentage determined the order of the results, while the underlying algorithm was not clear to one participant. To this end, comment No. 12 suggested providing a textual indication of the approach's suitability instead of a percentage. Another "Score" assigned to each approach characterized the quality, comprehensiveness, and feasibility of the underlying publication. Due to a lack of explanation,

this additional "Score" caused confusion for one participant, while another one saw a conflict with the "Recommendation" percentage. The remaining comments No. 13 to 16 address several other issues. Comment No. 13 expresses discomfort with the limited top-level quality attributes according to ISO25010 [ISO11a]. The following commenter expected more documentation on the tool's purpose in guiding a migration. Finally, comment No. 15 suggests an alternative way to catalog new approaches, while a technical comment addressed our prototype implementation.

### 9.1.2. Discussion

To evaluate our vertical prototype, we surveyed nine practitioners, researchers, and students in the field of IT/software engineering, most of whom had previous experience with microservices. We aimed at collecting first indications on the concept and potential usefulness of our intended web-based application. The survey revealed an overall positive sentiment about the prototype's understandability, effectiveness, usability, and fulfillment of expectations, but also pointed to potential issues. From the 16 additionally collected comments, we discern partly lacking documentation of our concept and the prototype's functionality. We attribute this fact to the limited resources spent on the implementation.

Nevertheless, we consider the results to be a solid basis for the prototype's evolution towards a web-based application. The usability improvements mentioned in comments No. 3, 6, and 7, should be prioritized in this regard. The filtering should be refined and restructured to guide the user confidently through the available options. Comments 2, 8, and 14 should be addressed by improving the documentation throughout the application, e.g., by providing additional explanations or tooltips where appropriate. Assigning a custom "Score" rating to an approach during its classification has not proven useful. As a consequence, we will remove this rating for refactoring approaches (comments No. 10 and 11). Moreover, we consider a different representation of "Recommendation" score (comment No. 12) and aim to make its calculation more transparent (comment No. 9).

Our efforts to develop a quality model for microservices migrations (Section 7.5) are not yet reflected in this prototype, but we see potential and need for an application that combines all phases of a quality-driven migration process, from architecture assessment, over decomposition, to microservices design. In this regard, we consider comment No. 13 about the available qualities for a follow-up evaluation. The vertical prototype evaluation helped us create and refine requirements for the extensible repository and application back-end (Table C.1), as well as for the user interface and application front-end (Tables C.2 and C.3). For example, to address the suggested usability improvements and ensure a consistent and intuitive user interface design, we decided to base our web-based application on the established Material Design standard, as discussed in Section 8.5.1. With respect to our initially stated research question (RQ1), we discern a generally positive sentiment for the effectiveness and usability of our microservices migration concept and the intended web-based application.

## 9.2. Survey on Web-based Application (11 Participants)

To evaluate the first iteration of our web-based application, we conducted a survey among 11 practitioners. We were looking for in-depth qualitative evidence on the application concept, its comprehensibility, perceived usefulness, and usability. The results should also reveal rationales, potential ambiguities, and suggestions for improvement. Our objective is expressed by the following second research question:

**RQ2:** How comprehensible, useful, and usable do practitioners perceive our migration methodology and web-based application?

We required potential participants to have a technical role in software development and significant professional experience. In addition, we sought for substantial knowledge and prior experience in microservices architectures. In this section, we discuss the questionnaire design, the survey process, and its results. At the time of the survey, the requirements listed below had been implemented (colored green) and were subject to our evaluation.

Extensible Repository and Back-End:
BE-R1, BE-R2, BE-R3, BE-R4, BE-R5, BE-R6, BE-R7, BE-R8

User Interface and Front-End (User):
UI-R1, UI-R2, UI-R3, UI-R4, UI-R5, UI-R6, UI-R7, UI-R8, UI-R9, UI-R10, UI-R11, UI-R12, UI-R13

User Interface and Front-End (Administrator):
UIA-R1, UIA-R2, UIA-R3, UIA-R4, UIA-R5, UIA-R6, UIA-R7, UIA-R8, UIA-R9

The full list of requirements can be found in the Appendix C. While functionality for the administrator role had been implemented as well, this survey exclusively concerns the user perspective. The evaluated state of development is documented by the referenced commit[1] of the project's GitHub repository.

[1] https://github.com/jfr609/architecture-refactoring-helper/releases/tag/v0.1.0-alpha

The design and implementation of the survey were guided by the seven-step process proposed by Kasunic [Kas05]. Accordingly, research objectives and target audience were identified in a first step, followed by creating a sampling plan. Next, the questionnaire was designed, and a pilot survey was conducted within the research group. Eventually, the refined questionnaire was distributed, and the results were collected and analyzed.

### 9.2.1. Questionnaire Design

We divided our questionnaire into five sections, which we discuss below.

The **start section** introduced the questionnaire and briefly described the context and purpose of the application. The questionnaire subsequently informed about the use of the collected information in future scientific publications, the guaranteed anonymity, and asked participants' consent hereto. We then collected basic demographic information such as job role, work experience, and experience with microservices.

In the **exploration section**, we introduced our application to the participants. To this end, we hosted a publicly available instance of the application that we referenced in the questionnaire. To familiarize the participants with the concept, we asked them to complete two tasks. Solving the tasks would guide the participants through the relevant views of the user interface and enable them to answer subsequent questions. The first task engaged the participant with the recommendation presets. We asked, how many refactoring approaches qualify for a "Re-Factor" activity with a suitability of 100%. Solving the first task would require to enter the second phase of the depicted framework, click on "Find recommended approaches", followed by the choosing the "Re-Factor" preset. The participant would then be presented a list of suitable refactoring approaches.

The second task addressed the **custom filter configuration** for refactoring approaches. We asked participants to find the approach that is 100% suitable for a given custom filter configuration. Solving the task would require locating the filter functionality in phase two and configuring it accordingly. The custom filter configuration can be accessed by clicking on "Configure"

instead of a preset. Eventually, clicking the "Search" button would result in the same results view, but with differently ranked approaches based on their suitability. The results of both tasks were queried in the questionnaire.

The following **application structure section** contained seven questions about the overall structure and usefulness of the implemented functionality. We asked about the way the underlying migration framework is presented, in terms of its understandability and summarizing the activities in a migration to microservices. In this context, we asked about the planned and already implemented functionality for phases one and two of the framework. We were also interested in the usefulness of both the filter presets and the custom filter configuration. As pointed out during our prototype evaluation in Section 9.1, we sought for feedback on the transparency of the recommendation score calculation. Specifically, we wanted to know if participants could understand the calculated score that leads to the ranking of results.

The following **usability section** aimed at evaluating the perceived ease and intuitiveness in solving the two tasks. This included the design of user interface elements, views, and navigation. We also asked participants about the perceived difficulty of the two tasks themselves. In this context, the five-point Likert scale rating would range from "very easy" (one) to "very difficult" (five). If a participant did not complete the task, we would allow that question to be skipped. Since the variety of usability aspects can hardly be covered by predefined questions, we again provided a free text input field for additional comments and feedback.

The final **additional functionality section** aimed at collecting feedback on potential enhancements, derived in part from our prototype evaluation in Section 9.1. We presented four preselected enhancements targeting a quality-driven migration, project and user management, and crowdsourced data collection. This section also included a free-text input field to collect additional ideas from participants.

### 9.2.2. Results

In the following, we present the survey results by starting with the participant demographics, followed by an analysis of each questionnaire section.

#### 9.2.2.1. Participant Demographics

The survey was hosted together with a running instance of the application for four weeks in April 2022. As for our prototype evaluation, we again invited participants of our earlier interview study (Chapter 3) as well as other professional contacts of the author. To extend the reach, we also sought contacts in our Empirical Software Engineering research group at the University of Stuttgart. In the end, we counted eleven valid responses among the submitted forms. Table 9.4 shows the resulting demographics of the participants.

**Table 9.4.:** Application Survey: Participant Demographics (N=11)

| Participant | Job Role | Professional Experience | Microservices Experience |
|---|---|---|---|
| P1 | Software Architect | 20 years | 2 years |
| P2 | CIO | 15 years | 10 years |
| P3 | Software Developer | 1 year | 1 year |
| P4 | Software Architect | 15 years | 8 years |
| P5 | Software Architect | 16 years | 1 years |
| P6 | Software Architect | 25 years | 3 years |
| P7 | Professor | 20 years | 8 years |
| P8 | Software Architect | 26 years | 10 years |
| P9 | Project Manager | 13 years | 2 years |
| P10 | Software Architect | 19 years | 2 years |
| P11 | Researcher | 15 years | 3 years |
| Median: | | 16 years | 3 years |
| Mean: | | ~16.8 years | ~4.5 years |

Of our 11 participants, the majority of six were in a software architect role with at least 15 years of professional experience. The roles of CIO, project

manager, and software developer round out the nine industry participants. The list of participants also includes two experienced participants from academia — a professor with 20 years and a researcher with 15 years of professional experience. The average professional experience of 16.8 years reflects a remarkable level of expertise among all participants. There is a similarly high level of expertise in the area of microservices. Four participants (P2, P4, P7, P8) stand out with 8 and 10 years of experience. While the term microservices became popular only around 2015, we relate this experience to service-oriented architectures and the concept of this architectural style behind the term microservices. The fact that each participant had at least one year of experience with microservices can be considered a solid foundation for our evaluation.

#### 9.2.2.2. Application Structure

To evaluate the structure of the application, we asked our participants the following seven questions:

**Application Structure**

1) Is the depicted framework an understandable way of summarizing the common migration activities? [1 = not understandable; 5 = very understandable]
2) How helpful do you rate the intended functionality in phase 1 (System Comprehension)? [1 = not helpful; 5 = very helpful]
3) How helpful do you rate the functionality in phase 2 (Strategy definition)?
4) How helpful do you rate predefined configurations such as "New Application", "Re-Build" and "Re-factor"?
5) How helpful do you rate the "Configure" functionality to filter for custom requirements?
6) Could you comprehend how the selection of recommended approaches was done by the tool? [1 = could not; 5 = fully could]
7) How helpful do you find the calculated percentage on the results page?

All questions were answered on a five-point Likert scale, with the lowest and highest scores indicated in square brackets for each question. The results

are visualized in Figure 9.2, where the labels on the x-axis correspond to the questions.

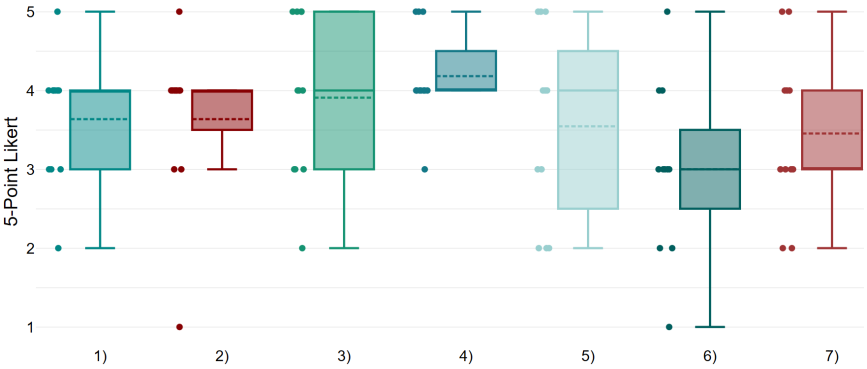



**Figure 9.2.:** Application Structure on a 5-Point Likert Scale (N=11)

The result for question one shows that the framework presentation was slightly well understood by our participants (mean: 3.6). A similar result was obtained for the usefulness of the yet unimplemented functionality in phase one (mean: 3.6). Significantly better perceived was the already implemented functionality in phase two, as the mean of 3.9 indicates. Only one participant tended to a slightly negative rating for this third question. The predefined filters for the “New Application”, “Re-Build”, and “Re-Factor” presets received the highest acceptance with a mean of 4.2. The custom filter functionality still scored an average mean of 3.5, but was also rated slightly negatively by three respondents, as shown in the graph for question five. This question was the most divisive, as reflected by the highest standard deviation of the seven questions in this section. The perceived transparency of the calculation for the approach-result view was equally wide distributed and resulted in a mean of 3.0. We detect a high standard deviation for this question too, indicating conflicting views on the aspect. The corresponding percentage shown for each approach on the results page was perceived slightly better by our participants, with a mean of 3.5.

#### 9.2.2.3. Usability

As a precursor to evaluating the usability of the application, we asked our participants to complete two tasks and then rate their difficulty. Seven participants provided a rating for the first task, while the subsequent question after completing the task revealed that six of these seven participants solved the task correctly. Six participants entered a difficulty rating for the second task, which was solved correctly four times. After completing these tasks and questions, we asked our participants two usability-related questions:

**Usability**

1) How difficult was it to solve the first task? (leave empty if you did not solve) [1 = very easy; 5 = very difficult]
2) How difficult was it to solve the second task? (leave empty if you did not solve)
3) How intuitive and easy is the navigation in the tool? [1 = not intuitive; 5 = very intuitive]
4) How would you rate the overall usability of the tool? [1 = not usable; 5 = very usable]

The results are visualized in Figure 9.3, where the labels on the x-axis correspond to the questions above.

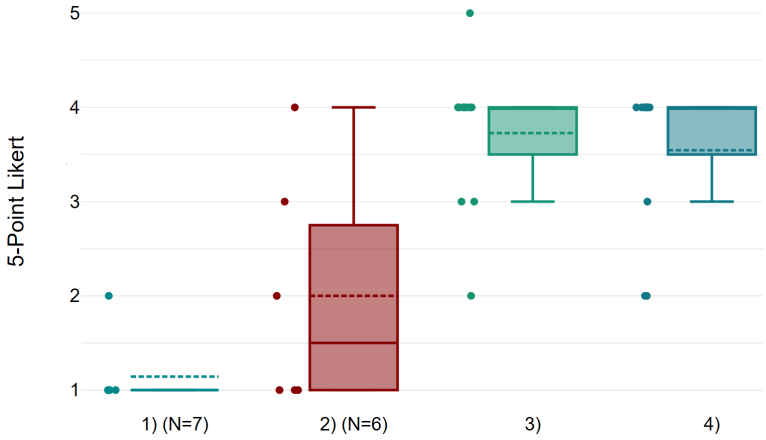



**Figure 9.3.:** Application Usability on a 5-Point Likert Scale (N=11)

As Figure 9.3 shows, six out of seven participants rated the difficulty of the first task as very low, with a mean of 1.1. The rating of the second task was significantly higher with a mean of 2.0, but only one participant tended to a higher difficulty than neutral. It is interesting to note that the four participants who solved the second task correctly rated its difficulty higher, with a mean of 2.5. For the analysis of this question, we excluded one answer because the participant entered an approach ID of "42" as the solution to the second task. However, we included fewer approaches for this evaluation, so this ID did not exist in the repository.

Questions three and four dealt with the overall usability of the application, and the ease and intuitiveness of navigation. Both questions were answered very similarly, with a slightly better rating for the navigation aspect (mean: 3.7 vs. mean: 3.5). We received several comments and suggestions in the free-text field regarding usability aspects. Three participants mentioned a certain difficulty in understanding the whole set of attributes that characterize a refactoring approach. A more detailed description of these attributes was requested. To facilitate the understanding of the user interface, one participant suggested showing explanatory videos that could, for example, detail the functionality provided for each phase of the framework. A short explanation of each approach in the form of an abstract was also considered a useful enhancement.

#### 9.2.2.4. Additional Features and General Feedback

Based on the evaluation result of our vertical prototype in Section 9.1), and subsequent brainstorming in the research group, we considered four additional features for the application. Participants could rate each of the four features from "not helpful" (1) to "very helpful" (5) on a five-point Likert scale. We collected responses from nine participants, which are shown in Figure 9.4. The labels on the x-axis correspond to the four features above. We asked our participants for their opinions on:

Additional Functionality

1) Functionality to use system specifications/results of phase one as an input for the filter in phase two [1 = not helpful; 5 = very helpful]
2) Functionality to save entered information/progress in form of project
3) Multi-user management (Login/Logout/Session Management)
4) Functionality to gather user feedback or star-ratings for certain approaches to see other users' opinions

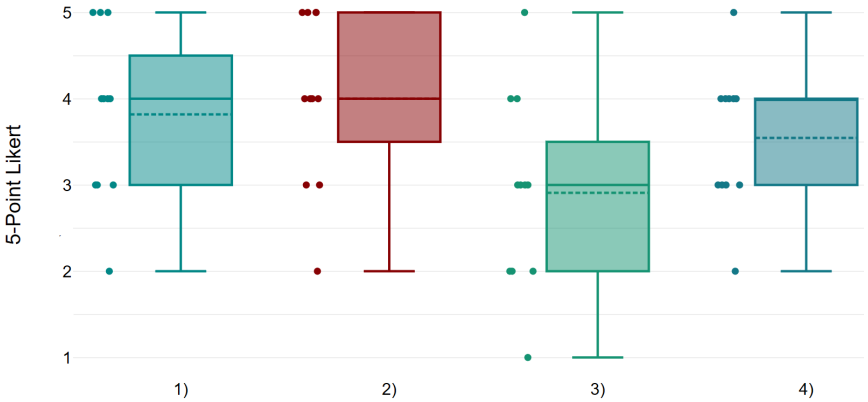



**Figure 9.4.:** Potential Additional Features of the Application (N=9)

Allowing users to persist their progress by saving and reloading the entered information (feature 2) was considered most helpful, with a mean of 4.0. Using the system specifications and architecture assessment results from phase one in phase two of the application (feature 1) was similarly desired with a mean of 3.8. This feature suggestion anticipated our concept of a quality-driven migration. Somewhat less desired, but still perceived as helpful, was the ability to share usage experiences from other users via feedback or "star-ratings" of refactoring approaches (feature 4). The least acceptance with a mean of 2.9 received a multi-user management of the

application (feature 3), likely due to the low added value for the application's core functionality.

Besides, we received additional feedback through the free-text input field. Almost all the comments suggested providing more detailed information about specific refactoring approaches. Among the aspects mentioned were the availability of (commercial) tool support, the size of supported code bases, or the size of the authoring team. Information about potential domain specifics of the approaches was also considered helpful. Two participants suggested expanding the list of approaches to include non-academic publications such as books, articles, blog posts, or other industry practices.

The final free-text field for general comments provided additional feedback beyond the topics discussed above. One participant questioned the overall usefulness of the application and had doubts that a migration decision process could be modeled in such a tool. Another participant criticized the fact that the linked publications for the refactoring approaches are often behind a "paywall", which would significantly reduce the value of the tool. One additional participant expressed dissatisfaction with navigating and answering the questionnaire. According to the comment, jumping back and forth between the questionnaire and the application made solving the tasks cumbersome.

### 9.2.3. Discussion

In this section, we discuss the survey results and their implications for guiding the future development activities of our web-based application. This includes the usefulness of the underlying migration framework and methodology, its implementation, usability, and potential future enhancements and improvements.

Our presented web application received an overall positive response from the survey participants. In particular, the functionality implemented in the second phase was found to be very helpful. Especially the predefined filters were well received compared to the more spread out opinions on the custom filter functionality. We interpret this result as a need for closer guidance

in the selection of appropriate refactoring approaches, which would allow the application to be used with less prior knowledge of the specifics of such approaches. The potentially overwhelming number of options for custom filter configuration could be restructured and adjusted in this regard to improve their comprehensibility. However, we still see a need for such fine-grained filtering, which may appeal to more experienced users. Reducing the number of options could be perceived as an unnecessary restriction for advanced users. Therefore, we are considering to keep the functionality, but providing more detailed explanations for the various filtering criteria.

While there was a slightly positive perception of the helpfulness of the suitability percentage, participants could not always understand how it was calculated. Therefore, we believe that this feature is generally useful, but should be presented in a more understandable way, e.g., as a graphical representation. As well, ratios of matches and mismatches in relation to the number of desired attributes could be shown. For example, if two out of four desired input artifacts match for an approach, its result entry could display "2/4 (50%)".

Based on the analysis of the tasks in our questionnaire, we constitute an acceptable degree of usability. The tasks were easy to solve for the majority of participants, and the subsequent questions about the intuitiveness of the navigation and the overall usability showed a clearly positive tendency. The free text responses revealed current shortcomings, which, we believe, could greatly benefit the overall usability in future iterations. In this regard, a written documentation, context-sensitive help, or even embedded short videos should be considered. A common way to familiarize users with new applications and workflows are tutorial overlays that guide users through their first use.

The feedback on our suggested four additional features shows a clear tendency towards usability and holistic guidance. We intend to address the two highest rated features in the next iteration. Implementing a quality model will allow all phases to be combined in a quality-driven migration process. A feature to save the entered data in the form of a project management feature will also be addressed. In addition, participants requested more

detailed information about refactoring approaches. We will address this as we refine our underlying taxonomy. In this context, a diversification of publication sources beyond academia was also mentioned. While we see this as an enhancement that could add significant value, we do not see it as being within the scope of our research project.

With respect to our second research question (RQ2), our survey revealed an overall positive sentiment on the comprehensibility, usefulness, and usability of the web-based application, thereby underscoring its relevance to industry. Only a few participants questioned its usefulness in part, while one participant questioned it in general. This survey provided us with valuable insights and suggestions for improving our web-based application. Combined with the observations from our prototype evaluation, it will guide future development activities.

## 9.3. Evidence on Quality-Driven Migration (6 Participants)

In evaluating the second iteration of our web-based application, we focused on the implemented quality model, while the subsequent third iteration focused more holistically on the quality-driven migration across all three phases. In both evaluations, we looked for in-depth qualitative evidence on understandability, perceived usefulness, and usability of our methodology by focusing on the newly implemented requirements and improvements. The results should also reveal possible ambiguities, workflow improvements, and other suggestions and ideas for directing our future development efforts beyond the scope of this thesis. According to this objective, we formulate our third research question as follows:

**RQ3:** How comprehensible, useful, and usable do practitioners perceive our implemented quality model and quality-driven migration process?

Again, we required potential participants to have a technical role in software development and significant professional experience. In addition, we were looking for substantial knowledge and prior experience with microservices architectures and their migration from monolithic systems. According to Virzi [Vir92], even few participants can provide thorough evidence for usability evaluations, with 80% of usability problems detected by four or five subjects. While these findings have been relativized in more recent studies [AM14; HS10], we believe a small sample is still appropriate given our solid foundation of previous evaluations. Hence, we conducted a survey among three industry experts in a second development iteration, followed by three expert interviews in a third iteration.

### 9.3.1. Survey

Our research objective for the survey was to validate the comprehensibility, usefulness, and usability of our application with respect to its integrated quality model. At the time of survey preparation, the following application

requirements (in addition to the requirements mentioned in Section 9.2) had been implemented and were additionally subject to our evaluation:

Extensible Repository and Back-End:
BE-R7

User Interface and Front-End (User):
UI-R11

Quality Model:
QM-R1, QM-R2, QM-R3, QM-R4, QM-R5, QM-R6, QM-R7, QM-R8

The evaluated state of development is documented by the referenced commit[1] of the project's GitHub repository.

#### 9.3.1.1. Questionnaire Design

We followed the same survey design and procedure as described in Section 9.2. We also built on the proven tactic of including tasks for participants that would help familiarize them with the interface while introducing them to the functionality. To do this, we hosted a publicly accessible instance of our web-based application dedicated to the survey participants. We designed a basic questionnaire structure based on our previous experiences. An introduction would explain the application's background and the quality model concept. The questionnaire then informed the participants about the use of the collected information, the guaranteed anonymity, and asked for their consent hereto. We collected basic demographic information such as current job role, professional experience, and experience with microservices. Next, participants were asked to complete two tasks that introduced the target functionality in the form of user stories. The first task involved creating a scenario in phase one of the framework, including assigning quality attributes. The newly implemented scenario-based workflow would yield recommended refactoring approaches in phase two, and subsequently patterns and best practices in phase three. The second task would engage the participant with

[1] https://github.com/jfr609/architecture-refactoring-helper/releases/tag/v0.2.0-alpha

the quality model in conjunction with the custom filter configuration. Participants were then presented with statements to agree or disagree with. To obtain more in-depth qualitative data, we dropped the Likert scale rating in favor of a complementary free-text input field for each statement or question. While there is a risk that participants would not fill in the field, the benefit of additional rationales would outweigh the risk of a missing response. Such a free-text field was presented again at the end of the questionnaire to collect general comments and feedback.

**Table 9.5.:** Quality Evaluation Survey: Participant Demographics (N=3)

| Participant | Job Role | Professional Experience | Microservices Experience |
|---|---|---|---|
| P1 | Lead Consultant | 14 years | 5 years |
| P2 | Software Architect | 11 years | 2.5 years |
| P3 | Software Architect | 25 years | 5 years |
| Median: | | 14 years | 5 years |
| Mean: | | ~16.7 years | ~4.2 years |

The survey was hosted along with a running instance of the application for three weeks in November to December 2022. Four software professionals were recruited through the network of our research group and participants in our previous industry case study (Chapter 4). We included three participants in our analysis: two software architects and one lead consultant, as shown in Table 9.5. Our participants had significant professional experience with 14, 11, and 25 years respectively. Two of them stated to have five years of experience with microservices and migration. One participant reported two and a half years of experience with microservices and one year of experience with migrating to them. The remaining fourth participant stated to have no experience with microservices, while we excluded this dataset from our analysis.

#### 9.3.1.2. Results

After our participants completed the two tasks, we first asked about the ease and intuitiveness of creating/editing scenarios and assigning quality attributes. For each aspect, two out of three participants confirmed ease and intuitiveness. One response indicated that there was ambiguity about how detailed the scenarios should be described, while another suggested a change to the user interface for more clarity. Two participants stated that the instructions provided in the user interface were helpful in performing these operations. We then asked if the structure of the user interface was useful and easy to understand. Again, two participants agreed, and one participant expressed doubts. The option to navigate to other phases from within phase one was perceived questionable. In this regard, it was suggested to use breadcrumbs for better orientation within phases and their nested pages.

Entering phase two of the framework, we first asked about the understandability and usefulness of the newly introduced configuration overview on the right-hand side. All three participants found it helpful, with two agreeing that it was easy to understand. One respondent stated that its purpose would only become clear during the process of creating a filter configuration. When asked if the three different workflows (preset filters, custom filter configuration, and scenario-based workflow) in phase two were easy to understand, there was one agreement and one expression of misunderstanding. The usefulness of specifying additional filters in the scenario-based workflow was agreed by two participants, with the remaining participant leaving no comment.

All participants agreed on the appropriateness of the result presentation. Its helpfulness in finding appropriate approaches based on quality aspects was agreed to by two participants. The third participant elaborated his impressions as follows: "*The criteria for classification of approaches are quite interesting. However, I was not sure how specific scenarios that I defined could be addressed by these approaches. More explanation and context could help.*" All three participants confirmed the comprehensibility of the calculated

ranking, i.e., the presented order of the resulting refactoring approaches. One participant stated that it took a moment to understand the abbreviations "QA" and "SP" for quality attributes and system properties (the notion of *system properties* was later replaced by *design properties*). When asked about the tendency arrow icons, which visualize the quality-related match score for a particular approach, one participant confirmed their usefulness. A second participant stated that he did not need this information, while the third participant was unsure.

On the question of whether the quality model extension helps to support a migration in terms of quality goals, two participants agreed, while one did not give an answer. One explanation for the agreement was that it specifically provides the appropriate approaches, patterns and best practices, which would make the application very useful in this regard.

In the final section of the questionnaire, we received comments and general feedback through the optional free-text field. One response suggested a redesign of the navigation concept: "*Consider to drop the nested wizards, I would prefer a search engine that allows to modify the filters next to the result set*". To make the custom filter configuration more intuitive, one comment suggested replacing the radio buttons with tri-state switches. Another suggestion was to remove the ID column for listing refactoring approaches. One of the participants concluded the questionnaire with "*All feedback is really meant to be constructive, discard at your peril. There are lots of good things about that tool that I did not take the time to mention. Rock on!*" A second participant left the following closing remark: "*It is a nice tool and can be useful in practice. Definition of scenarios was not clear to me and how they can result in more accurate results compared to just configuring the attributes*".

### 9.3.2. Expert Interviews

Complementary to the survey, we conducted a series of semi-structured interviews. Our research objective for the interviews was to evaluate our application in the context of a quality-driven migration. In this context, we validated the comprehensibility and usability of the three workflows, with a

focus on the scenario-based workflow. We also wanted to ask participants about the use of the presented results and the application's potential for further automation using artificial intelligence techniques. At the time of the interview preparation, the following application requirements (in addition to the requirements for 9.2 and 9.3.1) had been implemented and were subject to our evaluation:

Extensible Repository and Back-End:
BE-R8

User Interface and Front-End (User):
UI-R12, UI-R13, UI-R14

User Interface and Front-End (Administrator):
UIA-R9

The evaluated state of development is documented by the referenced commit[1] of the project's GitHub repository.

#### 9.3.2.1. Interview Preparation

The interviews, like our previous survey, consisted of two parts: The first part was a demonstration of the application, followed by a semi-structured interview. The demonstration part made the participants familiar with three use cases: The first use case demonstrated browsing the repository for available refactoring approaches and tools, while the second use case employed the custom filter configuration in phase two. Eventually, the third use case introduced the scenario-based workflow that covers all three phases of the framework. For the subsequent interview, we prepared a set of open-ended questions that would encourage interviewees to engage in discussion and provide rationales for their opinions. We prepared several follow-up questions in case interviewees would give short answers. Four groups of questions covered the three phases of the framework separately, followed by questions about the overall user experience.

[1] https://github.com/jfr609/architecture-refactoring-helper/releases/tag/v0.2.5-alpha

**Table 9.6.:** Quality Evaluation Interviews: Participant Demographics (N=3)

| Participant | Job Role | Professional Experience | Microservices Experience |
|---|---|---|---|
| P1 | Technology Expert | 10 years | 1 year |
| P2 | Software Architect & Technical Lead | 15 years | 6 years |
| P3 | Project Engineer & Team Lead | 5 years | 3 years |
| Median: | | 10 years | 3 years |
| Mean: | | 10 years | ~3.3 years |

Over the course of two weeks in October and November 2023, we interviewed three software professionals with experience in microservices and architectural refactoring towards them. Table 9.6 shows the demographics of the participants. The interviews were moderated by the author and a group of two to three software engineering students. We recorded the interviews and coded the responses according to our predefined structure. We then synthesized the responses thematically using tabulation.

#### 9.3.2.2. Results

The newly implemented phase one of the framework reflects the process of an architecture evaluation, as shown in Figure 9.5. We asked our interviewees about the appropriateness of the five steps involved, including the definition of scenarios, the assignment of quality attributes, and the accuracy of their weighting. Interviewees P2 and P3 expressed their perceived usefulness for this phase in general. That some of the entered information serves for documentation purposes only, was not obvious to either of them. P3 stated that it should be made more transparent which of the information entered influences the subsequent filtering. He suggested including the company name besides the project name, which would be relevant for, e.g., consulting companies. P1 considered the collected information to be at too high a

level of abstraction, which would make it difficult to relate it to the concrete component refactoring. For example, he expressed uncertainty about the information summary in step five of phase one and how the recommendations in subsequent phases would be generated from it.

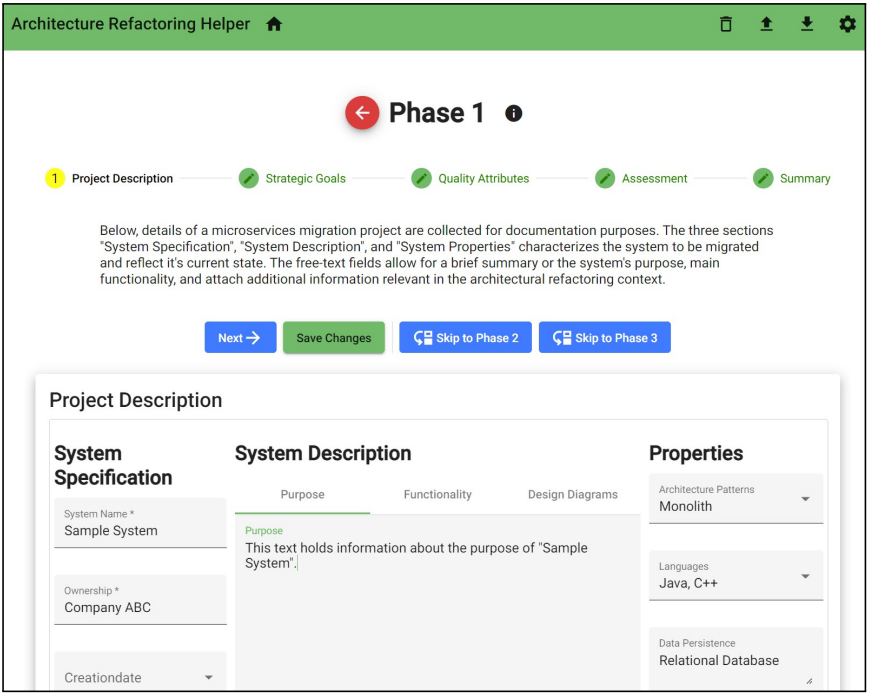



**Figure 9.5.:** User Interface of Framework Phase 1

We further asked if the proposed workflow would allow the identification of scenarios and associated quality attributes. In this context, we asked about the perceived accuracy of this assessment approach. Interviewees P2 and P3 confirmed the appropriateness of the workflow and its accuracy. P2 added that it would be helpful to show, in an additional view, which scenario(s) a particular quality attribute is derived from. In this regard, P3 suggested adding tooltips that provide more information about a quality attribute.

Interviewee P1 expressed uncertainty about how to define the scenarios, and the available quality attributes. He stated that qualities beyond those covered by ISO25010 [ISO11a] may be relevant for certain products, which would also require the involvement of a domain expert. Eventually, all interviewees confirmed the application's capability to accurately reflect and weight qualities in the context of a migration to microservices.

Our questions for phase two of the framework targeted the appropriate presentation of refactoring approaches and tools, the scenario-based workflow, and the intuitive use of the available filters. For refactoring approaches, we asked about the understandability of the results (matches, mismatches, tendency) and the interviewees' preferred tactic for selecting an approach. Listing all approaches and tools for reference was found useful by all respondents. P1 and P3 would have liked additional descriptive documentation, for example in the form of frequently asked questions at the top of the page or in the navigation bar.

The scenario-based workflow, which involves propagating qualities from phase one to phases two and three, was also found useful by all participants. However, the manual filter configuration was described as too extensive and not easy to understand. P1 and P2 suggested more guidance when specifying filter criteria. The current way of using tooltips for each filter option was considered necessary and appropriate, but could be extended, e.g. by automatic selection/suggestion of related filter criteria. P2 and P3 found the results view expressive and comprehensive. Both interviewees found the coloring helpful in structuring the results, but suggested that it could better reflect the filter mismatches. They also expressed discomfort with the interpretation of the quality-related tendency score. Here, P3 suggested showing the exact percentage when hovering over the tendency icon.

Regarding the selection of an approach from the result view, all three interviewees agreed on considering a maximum of three approaches from the top of the list for further consultation. P3 added that he would still like to see the full list of approaches in the results view, even if certain filter criteria do not match, i.e., the corresponding properties of an approach are undesirable. When asked about other aspects that influence the choice of an

approach, P2 was satisfied with the amount of information presented by the application, while P1 preferred approaches that explain algorithms or tools at a higher level of abstraction for easier understanding. In this context, he noted that the results view should show a small abstract of each refactoring approach, as the title is sometimes not expressive enough.

The overall presentation of patterns and best practices, which conforms to the structuring in phase two, was perceived as comprehensible and appropriate by interviewees P2 and P3. Interviewee P1 did not participate in the evaluation of phase three. Since the recommendation of patterns and best practices is based solely on the specified quality attributes, additional filter settings are not evaluated. Here, interviewee P3 stated that dedicated filters for the architecture design phase could be considered for future development of the application.

When asking for other suggestions and feedback not covered by our questions, P1 suggested automating the classification of refactoring approaches, for example by using a mining tool. In general, he confirmed the usefulness of the application and remarked that it moves academia in the right direction. As a final question, we asked all participants for their opinion on using artificial intelligence techniques to improve the automation, accuracy, and usability of the application. All three interviewees responded affirmatively: P2 and P3 would like to see a chatbot for user interaction, while P1 saw potential for using artificial intelligence during the above-mentioned cataloging of approaches.

### 9.3.3. Discussion

In this section, we discuss the combined results of both the survey and interviews with three participants each.

The quality model extension was found useful by two of our three survey participants. The concept of selecting appropriate refactoring approaches based on a prior architecture evaluation was found useful by all three interviewees, as was the subsequent selection of patterns and best practices.

These results indicate the general usefulness and appropriateness of the implemented quality model.

Our survey and interview results attest phase one a solid and intuitive design, the basic structure and presentation received majorly positive feedback. Participants were able to use the application in creating scenarios, assigning quality attributes, and comprehending their relevance in a quality-driven migration process. The instructions provided were considered helpful, but could be expanded to provide more detailed guidance. One suggestion from the interviews was to add a "final report" view for documentation, which could keep stakeholders up to date on the outcome of the quality assessment. It could also clarify what information is for documentation purposes only versus data that will be considered in subsequent phases of the framework.

Phase two of the framework provides three workflows for finding appropriate refactoring approaches: preset filters, custom filter configuration, and the scenario-based workflow. They were found to be understandable and useful by two interviewees, while the difference between the latter two was not clear to one interviewee. More context-sensitive documentation or a tutorial could help to distinguish their purpose and use. For the scenario-based workflow, more transparency was requested on how each scenario targets the recommended approaches. Scenario matches for each approach in the results could address this, but would also increase the complexity of the view. The current results view, with its tabular presentation and expandable elements, was found to be appropriate and useful by all respondents. The presentation of the quality tendency, however, was not obvious to all participants. Tooltips could provide additional information, including the calculated percentage, as one interviewee suggested. Surprisingly, all three interviewees would consider a maximum of three results from the top of the list before changing the filter configuration. This fact underscores the importance of an accurately calculated ranking of results.

The most contrary opinions received the custom filter configuration in phase two. While detailed filtering would provide a high degree of flexibility for experienced users, the variety of options could be overwhelming for others. One way to serve both user profiles would be to divide the filter view

into several smaller sections, ideally processed step-by-step, similar to the navigation in phase one. One interviewee suggested providing abstracts for each refactoring approach, containing its high-level characterization. This would be an alternative way to find suitable approaches and could reduce the importance of too detailed filtering.

The presentation of patterns and best practices, which follows the structure of phase two, was found to be understandable and appropriate in our interviews. Phase three does not offer a separate custom filter configuration and, moreover, relies solely on selected quality attributes. Hence, this less complex results view was considered comprehensible and would even offer potential for expansion, as one interviewee remarked. The overall user interface design and the cross-phase elements, especially the configuration overview sidebar, were perceived well and useful by our participants. A minor helpful enhancement would be to also reflect the current phase in this side panel, e.g., by showing cross-phase breadcrumbs. With respect to our third research question (RQ3), the combined survey and interview results yielded positive results for the comprehensibility, usefulness, and usability of our implemented quality-driven migration process and its underlying quality model.

## 9.4. Limitations

In this section, we discuss threats to the validity of our survey and interview results in Sections 9.1, 9.2, and 9.3.

**Internal validity** concerns the questionnaire design and whether our formulated questions precisely target the concepts to be evaluated. In this regard, statements and concepts should be formulated in an understandable and unambiguous manner. As we have carefully designed our questionnaires and piloted the surveys, we consider this risk to be sufficiently mitigated. To make the surveys as accessible as possible, we partly made the answers optional or allowed to skip tasks. As well, we also hosted a prepared and freely accessible instance of the application to minimize the barrier to participation. However, one participant of the survey in Section 9.2 expressed an inconvenience about jumping back and forth between the questionnaire and the application.

**External validity** concerns the degree to which the results are applicable in other scenarios and different contexts. The recruitment of suitable participants for our specific target group was challenging and only possible through convenience sampling, mainly through the research group's professional network. Here, a potential threat to external validity is the volunteer/selection bias, which can threaten the representativeness of the sample and thus may affect the generalizability of our results. We tried to mitigate this threat by opting for questionnaires over interviews for the prototype and the first two application evaluations. We believe that the resulting anonymity of the participants would reduce volunteer bias. While our sample sizes are too small for deriving general trends, we consider the qualitative feedback collected to be substantial for evaluating the application concept and implementation method. This interpretation is supported by the exceptional professional expertise of our participants, which suggests a high reliability of their judgments.

For our interviews, we obtained a limited sample size of only three participants. This raises the risk of selection bias, as the sample interviewed may

not accurately represent the broader target population. Another potential threat is response bias, which may have influenced the results. It occurs when respondents provide untruthful answers in an effort to meet the researchers' expectations. While inviting unaffiliated interviewees who matched our target population was not feasible due to time and resource constraints, we could have conducted for anonymous surveys instead. However, after weighing both options, we opted for interviews in the final iteration because of the opportunity to collect more in-depth responses and feedback.

## 9.5. Conclusion

In the course of evaluating our web-based application during an agile development process, we conducted three surveys and one interview study. While the first survey addressed the vertical prototype, the subsequent evaluations focused on three iterations of the developed application over a period of 1.5 years. The collected results influenced and guided the development process by generating new requirements and refining existing ones. In this regard, they substantiated the initially defined goals, derived from our analysis studies presented in PART II.

To evaluate our vertical prototype, we surveyed nine practitioners, researchers, and students in the field of the IT and software engineering. Our goal was to get first indications about the concept and the potential usefulness of our intended web-based application. The prototype survey revealed an overall positive sentiment regarding understandability, effectiveness, usability, and meeting of expectations. The additional comments also pointed out potential issues and suggested improvements, such as context-sensitive documentation and guidance when specifying filter criteria. We considered the results to be a solid foundation for evolving the concept into a web-based application. Other valuable insights included the choice of technology and design standards, like utilizing the established Material Design.

In our survey targeting the first iteration of our web-based application, we validated the usefulness of the underlying migration framework, usability,

and potential future enhancements. The eleven participating practitioners, with an exceptional average of professional experience, expressed an overall positive perception, particularly for the core functionality in phase two. The predefined filters were well received, while we saw more spread out opinions on the custom filter functionality. We observed this pattern throughout all evaluations and attribute it to different user preferences. While less experienced users may benefit from more guidance and fewer choices, more advanced users may prefer maximum flexibility, reflected in more fine-grained filtering capabilities. Therefore, we consider keeping both functionalities, but look for other ways to improve comprehension. The slightly positive perceived helpfulness of the suitability score would also benefit from a better documentation and a twofold visual representation.

Based on the little difficulty the participants had in solving the two tasks we provided, we constitute an acceptable level of usability for the application's main use cases. The feedback on our four proposed feature enhancements confirmed the usefulness of our quality-driven migration concept. By additionally implementing functionality to saving the progress in form of a project management, we addressed the two most desired features by the 11 participants who evaluated this first iteration (9.2.2.4). In summary, we see the survey results for this first iteration as a testament to the usefulness of the application and its relevance to industry. Only one participant questioned this aspect in general. As for the earlier prototype, we received valuable feedback that focused on more comprehensive documentation, such as embedded short videos or tutorial overlays.

Our third round of surveys and interviews targeted the iterations two and three of our application. They addressed the implemented quality model and the reflected quality-driven migration to microservices. For this purpose, we surveyed and interviewed a total of six practitioners with substantial professional experience, also covering migrations to microservices. The quality model itself was found useful by two of three survey participants, while the underlying concept of selecting refactoring approaches based on a prior architecture assessment was found useful by all interviewees. The newly implemented framework phase one was attested a solid and intuitive

design that received majorly positive feedback. Our survey participants were able to use the application in creating scenarios, assigning quality attributes, and understanding their relevance in a quality-driven migration process. For the scenario-based workflow, more transparency was requested on how each scenario influences the recommended refactoring approaches. The result view for approaches was found to be appropriate and useful by all survey participants. Potential for improvement was identified in the presentation of the quality tendency, where tooltips could be beneficial.

The custom filter configuration was also discussed controversially among the survey and interview participants. Solution ideas centered around alternative ways, such as a search-engine-like behavior or providing expressive approach abstracts. The presentation of patterns and best practices, that corresponds to the structure in phase two, was perceived as comprehensible and appropriate in our interviews and would offer room for future enhancements. The configuration overview was also well received by our participants and could be improved, for example, with cross-phase breadcrumbs. All evaluations indicate that the application's industry readiness would benefit greatly from more context-sensitive documentation, supplemented by introductory tutorials or videos.

In conclusion, our conducted evaluation studies have proven the relevance, feasibility, and usefulness of our quality-driven migration concept. We were able to provide and significantly improve guidance throughout a migration process by refining the developed web-based application over three iterations. Our participants preferred different ways of using the application, which was particularly evident in the design of the filtering functionality in phase two. Our design decisions aimed to best serve these different user needs by providing multiple workflows with varying degrees of guidance. The evaluation activities yielded additional requirements for future improvements and adaptations that would further improve the application's industry readiness.

Chapter 10

# Case Studies in Industry

In this chapter, we contribute two case studies for evaluating our migration methodology and its accompanying web-based application *Architecture Refactoring Helper*[1] (*ARH*), as presented in PART III.

The first case study of a monolithic application in the *Services* domain was conducted between February and August 2023. It aimed at developing a concept for its migration to a microservices-based architecture. To this end, the need for a migration was assessed first, based on software product quality aspects. The developed migration concept was then applied in a proof-of-concept, which involved the extraction of a single microservice from the monolithic application. The case study was accompanied by the creation of structured field notes that collected in-depth qualitative data on the applicability of our methodology and its implementation as a web-based application. Our presentation of the first case study summarizes and condenses its detailed documentation, which was elaborated in the software engineering master's thesis of Knodel:

[1] https://github.com/jfr609/architecture-refactoring-helper

M. Knodel, “Migration monolithischer Anwendungen in Microservices-basierte Architekturen: Fallstudie einer Service/Sales-Applikation.” University of Stuttgart, 2023 [Kno23]

While our first case study addresses the primary use case of migrating a monolithic application to microservices, the second case study aimed to evaluate the suitability of our methodology for improving an existing microservices-based architecture. It involves a document management solution and was conducted between September 2023 and March 2024. The system under study was previously migrated from a monolithic system to a microservices architecture. Despite the improvements achieved through the architectural refactoring, the company sought further improvement opportunities to address remaining issues. While the vast majority of our in Chapter 5 reviewed refactoring and migration approaches require the existence of a monolithic application as a starting point, most of the decomposition approaches can still be used to verify and refine an existing decomposition. The same applies to patterns and best practices used for the architecture design. Therefore, in this second case study, we present an evaluation of our methodology with respect to the refactoring of an existing microservices architecture in an industrial environment. Our presentation summarizes its detailed elaboration in the software engineering bachelor’s thesis of Herrmann:

A. Herrmann, “Toolunterstütztes Refactoring von Microservices-Architekturen: Eine industrielle Fallstudie” University of Stuttgart, 2024 [Her24]

In the following, we introduce our applied case study research method, before presenting the two case studies. The discussion of limitations combines both studies and leads to a joint conclusion.

## 10.1. Case Study Research

We use the qualitative method of case study research and adopt the subjectivist and interpretivist perspectives, as the researcher is closely involved

with the study participants. We thereby acknowledge the importance of human behavior, beliefs, and interpretation in the process of knowledge creation. Case studies are appropriate when the conducted research aims to answer the "why" and "how" questions, along with seeking in-depth and detailed descriptions. In this regard, we consider them the most appropriate to address our research objective.

Case study research aims to establish a detailed relationship between the phenomenon under investigation and its surroundings [Zai07]. Phenomena are explored in their natural setup to gain a deep and accurate understanding, which is largely based on observations and statements of participants, and their interpretations [Cro20]. Runeson and Höst regard this methodology as "*well suited for many kinds of software engineering research*" [RH08]. They note that the field often investigates contemporary phenomena that would be difficult to study in isolation. Yin similarly notes that the case study method is appropriate when "*boundaries between phenomenon and context are not clearly evident*" [Yin03]. Zainal further distinguishes between interpretive and evaluative case studies. In the latter one, which we consider most relevant for our objective, researchers go beyond interpreting the data, by "*adding their judgment to the phenomena found in the data*" [Zai07].

Based on these considerations, we conducted two case studies to holistically evaluate our developed methodology in real industrial environments.

## 10.2. A Microservices Migration in the Services Domain

The company L-mobile[1] decided to migrate their established web-based service/sales application from a monolithic system to a microservices-based architecture. The application is based on a customer relationship management System (CRM). Its core functionality centers around organizing order types for service objects and planning of service operations, while customizing allows for the implementation of specific extensions. The application is offered to customers via a Software-as-a-Service (SaaS) cloud model, but can

[1] https://l-mobile.com/

also be used on-premise. In the latter case, the application is operated as a standalone application that allows for integration with established enterprise resource planning (ERP) systems such as proALPHA[1], SAP[2], Infor COM[3] or Microsoft Dynamics 365 Business Central[4].

During this study, we evaluate our migration framework and methodology in an industrial context, by planning a migration of the L-mobile service/sales application. A migration concept is elaborated first, and then demonstrated by an exemplary implementation as a proof-of-concept. This study takes a software architect's perspective and thus focuses on architectural refactoring. Organizational and infrastructure-related topics are not subject to our investigation. As such, The following research questions formulate our objectives and frame the scope for this case study:

**RQ1:** How appropriate is the microservices migration framework for refactoring the L-mobile monolith into microservices?

**RQ2:** What migration processes and refactoring approaches are best suited for the monolith's decomposition?

**RQ3:** How can the framework facilitate the refactoring and design of the emerging microservices architecture?

### 10.2.1. Methods

This first case study entails the development of a migration concept for the monolithic system, followed by prototypical implementation, and evaluation of the results. Our efforts aim to evaluate the developed framework and critically examine its practical applicability. In case the framework's guidance reveals deficiencies or inadequacies, further research is conducted to successfully continue the concept creation and implementation. The thereby yielded

[1] https://www.proalpha.com/en/

[2] https://www.sap.com/products/erp.html

[3] https://www.infor.com

[4] https://www.microsoft.com/dynamics-365/products/business-central

insights can help to extend and improve the methodology. In the following, we outline our process and the associated methods for data collection.

**Migration Framework** Our migration framework will guide through the three phases of a migration, starting with an architecture evaluation based on Svahnberg and Mårtensson [SM07]. The elicited scenarios and associated quality goals will provide the basis for a quality-based decision on the need of a migration to microservices. In the framework's second phase, a strategy for the migration process is elaborated in a first step. It is followed by the identification of suitable refactoring approaches for the monolith's decomposition into microservices, in which the quality goals of phase one are incorporated. In the third phase, microservices are identified and the resulting microservices architecture is designed. The framework supports this architecture design by recommending patterns and best practices, which are again based on the quality goals of phase one.

**Prototypical Implementation** Prioritizing individual microservices leads to a suggested order for their subsequent implementation. In the context of this case study, the prioritization will also determine the service selected for a prototypical implementation. To ensure that code quality is preserved after the refactoring, code reviews are conducted by senior software developers familiar with the application. To further ensure that the application's functionality is preserved, a series of automated tests are performed on the involved functionality before and after the refactoring.

**Architecture and Process Review** The resulting migration concept will be subject to an architecture review. It aims to assess its appropriateness and soundness from the perspective of a software architect familiar with the application. This review involves the generated decomposition, architecture description, created diagrams, recommended patterns and best practices, as well as decisions made along the process. An additional process review aims at critically reflecting the migration concept and implementation from a product management perspective. It seeks to assess the elaborated strategy's suitability for an envisioned migration of the entire application, also

involving related aspects like DevOps practices. To this end, we conducted two interviews with the "Head of Development" and "Product Manager" of the involved product.

**Field Notes** To allow for a holistic assessment of the migration concept and process with regard to its applicability and usefulness, we employ the qualitative method of structured field notes [Sea08]. In this regard, we seek for more in-depth qualitative data from observations of the involved participants. Field notes are commonly used in various domains besides empirical software engineering, such as medicine or psychology [Don09; PL18]. Seaman describes field notes as a detailed qualitative data collection method that is based on participant observations and interviews. Field notes record participant's actions, discussions, goals, motivations, including distractions or other sentiments. Seaman also suggests adding extra comments for documenting impressions that arise from unspoken aspects or occurrences. Field notes should be enriched with as many details and taken as soon as possible when the observer still remembers all the details. The researcher's goals would eventually determine their degree of detail. To analyze the collected field notes, Seaman suggests the constant comparison and cross-case analysis for the latter generation of a theory. For the collection of field notes in our work, we created the following template based on the suggestions by Seaman [Sea08]:

- Date, time, location
- Phase of framework and step
- Conducted activity
- Affected component of application
- Things that went well
- Things that did no go well
- Sentiments
- Other comments

### 10.2.2. Results

In this section, we first present the elaborated migration concept, divided into three subsections that correspond to the three phases of our framework. Afterwards, we present results for the validation activities, namely architecture review, code review including automated tests, and process review. The concluding analysis of the collected field notes provides further insights and reflections of the involved participants along the process.

#### 10.2.2.1. Phase One: Architecture Evaluation

Phase one of the framework focuses on the system comprehension and aims to systematically determine the feasibility and need for a migration to microservices. To this end, the conduct of an established architecture evaluation method is recommended. Our framework suggests using a scenario-based architecture evaluation method based on ATAM [KKC00] or SAAM [KBWA94]. Both are proven methods that employ a scenario-based assessment based on quality goals. The architecture assessment framework by Auer et al. [ALFT21] was specifically developed to guide the decision-making on whether an existing system should be migrated to a microservices architecture. However, the framework by Auer et al. only takes into account quality attributes and does not consider scenarios. Svahnberg and Mårtensson present a more lightweight architecture evaluation method that is based on ATAM and SAAM, but can be conducted within 4 to 5 hours [SM07]. Because of various factors such as size of the application, number of stakeholders, and resource limitations, the decision was made to employ the method by Svahnberg and Mårtensson [SM07]. It passes through the following six steps:

1. Introduction to project
2. Identification of quality requirements
3. Scenario elicitation
4. Architecture presentation
5. Scenario and architecture analysis
6. Conclusion

The architecture evaluation for the service/sales application was held within three weeks of March 2023, while the workshop lasted three hours and 41 minutes in total. The participating stakeholders were active in the roles "Head of Development", "Product Manager", "Software Architect" and "Senior Developer". A software engineering master student [Kno23] assumed the role of the moderator and protocol writer. The results of this scenario-based architecture evaluation were aligned with our web-based application *ARH*, which allows for reflecting them within five steps. Accordingly, we pass through these steps, namely (1) project description, (2) strategic goals, (3) quality goals, (4) quality assessment, and (5) results.

**Step 1: Project Description** The system under investigation is a service/sales web application that is developed and deployed as a modular monolith. The code base contains about 300,000 lines of code in 4810 classes, mainly implemented in C#, JavaScript, TypeScript, and HTML. Its client-server architecture is shown in Figure 10.1. The communication between the frontend and backend is based on HTTPS. The backend is implemented in ASP.NET and operated on a Microsoft Windows server using a relational SQL database. To guarantee high availability, several servers can be operated by hosting them in a web farm. In this case, the SQL database is operated on several instances in failover clusters.

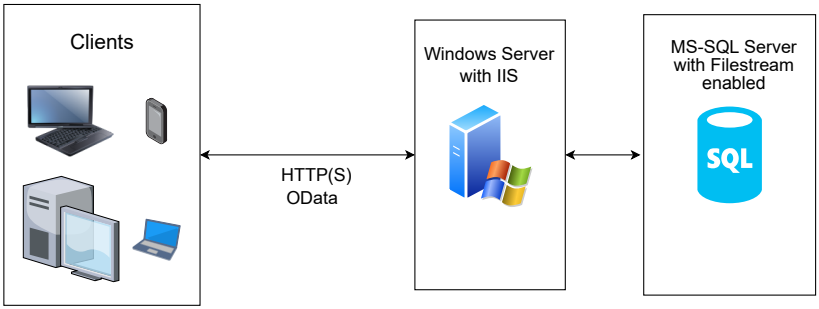



**Figure 10.1.:** Architecture of Service/Sales Application (Adapted from L-mobile [L-M23])

**Step 2: Strategic Goals** The strategic goals that the company L-mobile pursues in the context of this migration project were defined by the participants as follows:

- Drive further modularization of the software
- Cost reduction by using cloud-based resources
- Prepare for microservices-based operation and DevOps automation

The modularization goal aims for better maintainability and evolvability of the code, as individual services can be developed in isolation [MCL17]. Depending on the chosen cloud model, cost reductions can be achieved by dynamic scaling of small units [VGO+16]. The last strategic goal reflects the company's strategy of moving towards microservices and DevOps.

**Steps 3-5: Quality Goals, Assessment, and Summary** The identification of quality attributes was conducted based on ISO 25010 [ISO11a] and Auer et al. [ALFT21]. The stakeholders then prioritized the identified quality goals as suggested by Svahnberg and Mårtensson [SM07]. As a result, nine quality attributes were identified, with altogether 17 quality sub-attributes. For the three highest prioritized quality attributes, the authors suggest a restriction to two sub-attributes per quality attribute [SM07]. The resulting quality goals for the L-mobile application are shown in Figure 10.2.

*Security* was prioritized highest, with its sub-attributes *Data Integrity* and *Authentication*. It was followed by *Reliability* with its sub-attributes *Recoverability* and *Data Consistency*. *Functional Suitability* was ranked third, with *Functional Correctness* and *Functional Completeness*. For each of the six quality sub-attributes, two scenarios were elicited, resulting in 12 scenarios altogether. Each scenario was then assigned an importance and difficulty rating from A to C, as described in Chapter 2.4.1. The resulting quality attributes with their assigned scenarios are shown in Table 10.1.

Eventually, each scenario was examined regarding architectural styles that allow for its implementation, and the preferred style for its implementation. The results revealed no immediate need for a migration to microservices, since the existing monolithic architecture fulfills all scenarios with regard to the quality goals. Nevertheless, the participants agreed to further research

on microservices and migration strategies according to the defined strategic goals. In this regard, the participants were seeking for profound knowledge

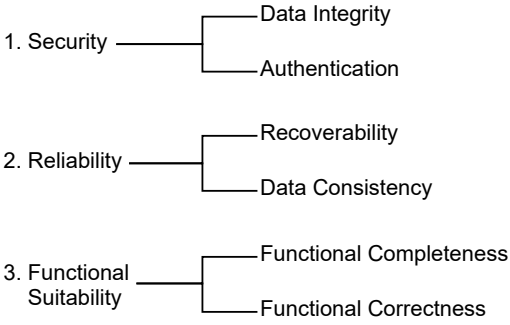



**Figure 10.2.:** Prioritized Quality Goals for the L-mobile Application (Adapted From Knodel [Kno23])

**Table 10.1.:** Scenarios for the Highest Prioritized Quality Attributes

| Attribute | Sub-Attribute | Scenario | Importance | Difficulty |
|---|---|---|---|---|
| Security | Data Integrity | S1 | C | B |
| | | S2 | B | A |
| | Authentication | S3 | A | C |
| | | S4 | C | C |
| Reliability | Recoverability | S5 | A | B |
| | | S6 | A | A |
| | Data Consistency | S7 | B | C |
| | | S8 | A | C |
| Functional Suitability | Functional Completeness | S9 | C | A |
| | | S10 | C | C |
| | Functional Correctness | S11 | A | C |
| | | S12 | A | C |

on a process for a migration to microservices, and its feasibility. A migration concept along with a prototypical implementation was thus regarded eminently important. Due to the presented rationales, the elaboration of a migration concept for the L-mobile monolith will continue through phases two and three of our framework as a proof of concept.

#### 10.2.2.2. Phase 2: Strategy Definition

Our framework entails two steps in its second phase: (1) Definition of migration strategy and development process, and (2) Selection of a service identification approach. At the time of conducting the case study, the repository of refactoring approaches populated 90 publications. Several of them may contribute to the elaboration of a specific migration strategy and service identification approach. In a first filtering, approaches were considered that fulfill the quality goals elicited in phase one. For an exhaustive quality-based filtering, all nine quality attributes and 17 sub-attributes were included. To narrow down the number of results, the process strategy "Re-Factor" was specified as an additional criterion. The yielded approaches of this first filtering round are shown in Table 10.4 with their number of matching quality attributes and sub-attributes.

**Table 10.2.:** Approaches Filtered for Quality Goals

| Publications | # Quality Goals |
|---|---|
| [SK17] | 7 |
| [DKTT22], [GCL+20], [AIE19] | 4 |
| [ACC+22b], [DBFP21], [ZSS+21], [CGC+20a], [CGC+20b], [PKY20], [Lau19], [HLT18] | 3 |
| 16 additional publications | 2 |
| 15 additional publications | 1 |

In a second round of filtering, strategies and approaches were retained that promote one or more of the design properties *Coupling*, *Cohesion*, *Granularity*,

and *Complexity*. As well, approaches that solely rely on model artifacts as input were excluded due to the rare availability of such artifacts for the L-mobile application. An overview of the remaining publications after the second filtering is shown in Table 10.3.

**Table 10.3.:** Approaches Filtered for Quality Goals and Input Artifacts

| Publication | Design Properties | # Quality Goals |
|---|---|---|
| [SK17] | - | 7 |
| [CGC+20a] | cohesion, coupling | 3 |
| [HLT18] | coupling | 3 |
| [DBFP18] | cohesion, coupling | 2 |
| [LML20] | - | 2 |
| [DBT21] | cohesion, coupling | 1 |
| [AM21] | cohesion | 1 |

Of the filtered publications, [DBT21], [CGC+20a], [DBFP18], and [SK17] focus on service identification, while [AM21], [LML20], and [HLT18] describe a service identification approach along with a migration strategy. The approach by Shashwat and Kumar [SK17] addresses seven of the quality attributes and sub-attributes identified in phase one. Among them are the most relevant ones, like *Security* and *Reliability*, which are not addressed by the other filtered approaches. The approach by Li et al. [LML20] employs the Strangler Fig pattern [Mic23c], which describes a gradual migration process to microservices. As the approach incorporates a model-driven analysis for service identification, it is considered exclusively for the migration strategy definition, but omitted for the service identification.

A complementary filtering was performed on tools for service identification. At that time, the tool repository contained 22 approaches associated with tool support. For the filtering, all tools were excluded that operate solely on Java-based applications. Besides, we excluded tools that require source code, API specifications, or version control system data as input. Furthermore, publications were excluded that could not be successfully executed

in the course of our preliminary usability report [FCBW23]. The remaining five tools and their associated publications [ZSS+21], [ZZ21], [KZH+20], [NUEH18], and [MCF+20] were analyzed in detail on whether and how they can be beneficial in migrating the L-mobile monolith.

To ensure that no potentially relevant publications were mistakenly excluded or not yet covered by our repository, a manual literature review was conducted in addition. It aimed specifically at identifying approaches offering tool support. Ten additional publications had been identified and considered relevant after title and abstract reading. It included five new publications, yielded by the review of Akkaya and Ovatman [AO22]. However, the additionally identified approaches and tools have not been further considered for the following reasons: The described tool was not available for download; the implementation was specifically for Java-based applications; the documentation was insufficient for an effective use.

After the rounds of quality-driven and tool-oriented filtering, the following suitable approaches describing a migration strategy or service identification approach for the decomposition of the L-mobile monolith have been identified:

| | |
|---|---|
| Migration Strategy: | [AM21], [LML20], [HLT18] |
| Decomposition: | [DBT21], [CGC+20a], [DBFP18], [SK17] |
| Decomposition Tool Support: | [ZSS+21], [ZZ21], [KZH+20], [NUEH18] |

After close inspection of all eleven publications, two approaches were selected. The approach by Li et al. [LML20] was selected for guiding the migration strategy, since it is based on the Strangler Fig pattern [Mic23c]. According to the strategic goals identified in phase one, the company L-mobile seeks for a strategy that allows a gradual migration towards microservices. The ability to operate the not yet entirely restructured monolith in parallel with extracted microservices was also considered important. The approach by Krause et al. [KZH+20] was chosen for service identification. Due to the size of the service/sales application, such a tool-assisted approach would be less dependent on human expertise and allow for adaptions and repetitions

of the process with higher accuracy. In this regard, a prototypical extraction of one microservice should prove the concept and serve as a reference for future migration activities.

#### 10.2.2.3. Phase 3: Architecture Definition

Our framework entails the following four essential steps in its third phase: (1) Identify Microservice Candidates, (2) Define Microservices Architecture, (3) Prioritize Services, and (4) Implement Services. The framework supports the architecture design in step two by recommending suitable patterns and best practices that facilitate quality goals elicited in phase one. At the time of conducting the case study, the repository cataloged 51 patterns and 43 best practices (Appendix A). In the following, we present the design decisions along the four steps while focusing on the incorporated patterns and best practices.

**Step 1: Identify Microservice Candidates** The decomposition of the monolith according to Krause et al. [KZH+20] started with the identification of bounded contexts. To this end, the tool Structure101[1] was used for static code analysis. The resulting graphical representation served for a manual identification of contexts with class-level granularity. This procedure partially required domain expertise to decide on merging items whose context was not obvious. For instance, the three contexts of "PDF Generation", "Time Intervals", and "Login" were created after consultation with the product managers. Altogether, 39 bounded contexts could be identified. According to Krause et al. [KZH+20], the dynamic analysis tool ExploreViz[2] would complement the static analysis and refine its results. Due to incompatibilities with the L-mobile application, this step could not be carried out. However, as the static analysis yielded promising results, it was decided to omit further research on alternative dynamic analysis tools. The database was eventually

[1] https://structure101.com/

[2] https://explorviz.dev/

**Table 10.4.:** Filtering Patterns and Best Practices

| Filter Step | # Patterns | # Best Practices |
|---|---|---|
| Improves one or more of three highest prioritized quality attributes, including sub-attributes | 30 | 16 |
| Exclude, if irrelevant for system context | -9 | -1 |
| Include from all, if still relevant for system context | +9 | +12 |
| Postpone, if related to messaging or containerization | -8 | -2 |
| Add, if not yet covered by framework | +4 | 0 |
| **Total included** | **26** | **25** |

analyzed via the tool DBeaver[1]. Krause et al.[KZH+20] state that if a database table encompasses several bounded contexts, it should be split for the various contexts.

**Step 2: Define Microservices Architecture** As a preparatory step for the architecture definition, the framework supports architects in this phase by recommending patterns and best practices based on predefined quality goals. In the following, we describe the selection of patterns and best practices before outlining the design decisions for the resulting microservices architecture.

To filter the cataloged 51 patterns and 43 best practices, a consecutive filtering was conducted as shown in Table 10.4. The filtering started based on the initially defined quality goals, while subsequent rounds ensured a relevance for the system context. As well, other potentially relevant patterns and best practices were considered that do not specifically address the quality goals. Eventually, 26 patterns (Table 10.5) were considered for designing the microservices architecture, complemented by a similar number of 25 best practices (Table 10.6). In addition to the already cataloged patterns, four additionally identified patterns were considered for the design.

[1] https://dbeaver.io/

**Table 10.5.:** Selected Patterns[1] After Filtering

| | |
|---|---|
| Aggregator (New) | Health Check |
| API Gateway | **Internal Load Balancer** |
| Auth-Service | **Load Balancer/Load-Balancing** |
| Bulkhead (New) | Mulitple Service per Host |
| **Circuit Breaker** | **REST Integration** |
| Correlation ID | **Service Discovery** |
| CQRS | **Service Locator** |
| Edge Server | **Service Registry** |
| **Enable CI** | **Service Registry Client/Server** |
| **Externalized Configuration** | **Shared DB Server** |
| Gatekeeper | **Strangler** |
| Gateway Offloading | **Microservice Chassis** (New) |
| Gateway Routing | **Microservice Template** (New) |

[1] Patterns printed in **bold** are considered for PoC implementation

**Table 10.6.:** Selected Best Practices[2] After Filtering

| | |
|---|---|
| **Access Restriction** | **Guarded Ingress** |
| **Account Separation** | **Loose Coupling** |
| **API-Based Communication** | **Manageable Connections** |
| Authentication Delegation | **Manageable Standards** |
| Automated Monitoring | Seamless Upgrades |
| Automated Restarts | Secrets Management |
| Autonomous Fault Handling | **Separate Libraries** |
| Built-in Autoscaling | Separation by Gateways |
| Cloud Vendor Abstraction | **Service Independence** |
| **Coarse-Grained Microservices** | **Standardization** |
| **Configuration Management** | **Standardized Deployment Unit** |
| Data Encryption in Transit | **Versioned APIs** |
| Dynamic Scheduling | |

[2] Best Practices printed in **bold** are considered for PoC implementation

The targeted gradual migration of the monolith to microservices relies on existing technologies and follows the Strangler pattern [Fow04b; LML20; Mic23c; YM22]. The pattern suggests to extract existing or add new functionality as microservices, which is then disabled in the monolith. While the remaining monolith is being operated in parallel, it communicates with the newly added microservices [YM22]. On the monolith's complete "strangulation", it would be replaced by an API Gateway [Ric23; TLP18]. The API Gateway acts as a single entry point for requests and forwards them to the corresponding services. To allow the API Gateway addressing the respective services, a Service Registry [Ric23; TLP18] is added. The microservices and monolith would interact exclusively via synchronous REST-based communication in early stages of the migration, complemented by Asynchronous Messaging [Ric23] in later stages.

Due to highly cross-referenced database tables, the Shared Database pattern is applied [Ric23]. Following the propagated Database per Service pattern as suggested by Krause et al. [KZH+20] was rejected by stakeholders due to the necessary complex migration scenarios for existing customer installations. A consensus was found that prescribes to either duplicate database tables or use a shared database. Similarly, the Log Aggregation pattern was desired for consolidating the logging of all microservices. To ensure that only authorized users can send a request to a microservice, using Access Tokens [Ric23] was prescribed. In addition, the Health Check pattern and "Automated Monitoring" practice were considered [Ric23; VLL+20]. To prevent failures from cascading, the patterns Circuit Breaker [Ric23] in conjunction with Load Balancer [VLL+20] were incorporated. An Externalized Configuration [Ric23] file is used to control settings that affect several microservices. Eventually, the Microservice Chassis [Ric23] and Microservice Template [Ric23] patterns have applied to ensure that all microservices are based on a unified template with basic functionality.

Based on these considerations, the microservices-based architecture in Figure 10.3 was designed.

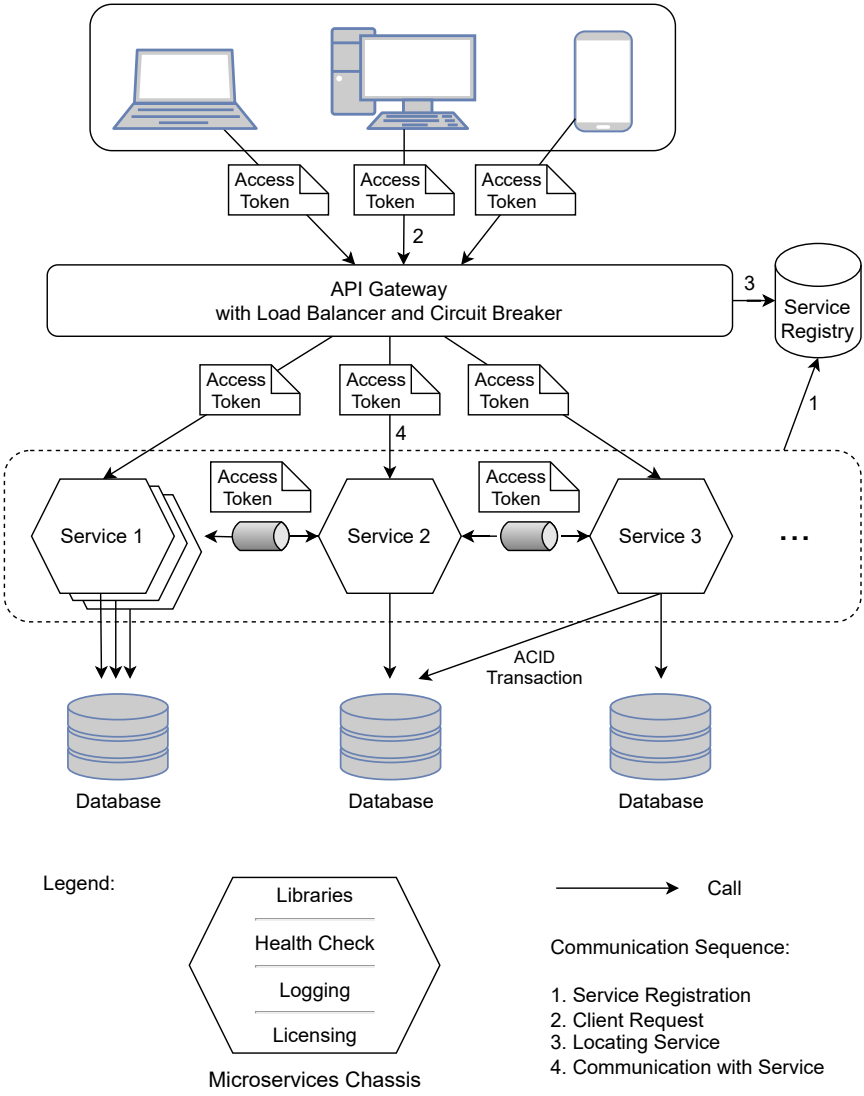



**Figure 10.3.:** Microservices Architecture for the Service/Sales Application (Adapted From [Kno23])

**Step 3 and 4: Prioritize and Implement Services** Li et al. [LML20] suggest several criteria for prioritizing services. For instance, performance problems or a loose coupling with the monolith are facilitators. Yoder et al. recommend to start with a small service [YM22], while Taibi et al. add a high number of bugs as a criterion [TLP17]. Except for the last criterion, all others apply to the "PDF generation service" of the L-mobile application. After consulting the "Head of Development" and "Product Manager", its extraction was chosen for the PoC in this case study. Further details on the refactoring and implementation are provided by Knodel [Kno23].

#### 10.2.2.4. Validation

In the following, we summarize the validation activities for the before created artifacts, including the implementation. It entails an architecture review, code review, automated tests, and concludes with a process review.

**Architecture Review** The review of the designed architecture was conducted by the software architect who already participated in the initial architecture evaluation. He completed the review within approximately two hours, in which he inspected all created documents, diagrams, and decisions made. As a result, the architect suggested smaller refinements of the generated microservice decomposition. For instance, the license management should not be handled by each microservice individually, but once for entire application. Moreover, he recommended using the Log Aggregator pattern. The findings and subsequent adaptions of the architecture are already reflected in our above discussed architecture design.

**Code Review and Automated Tests** The aforementioned software architect and an information security engineer reviewed the generated code of the extracted microservice, along with the adaptions made for the remaining monolith. Both the software architect and the information security engineer needed around two hours for reviewing the code repository. The review revealed a total of 27 findings, of which five could be implemented

within this PoC. The remaining 22 findings required additional research and implementation efforts and could thus not be addressed.

Seven automated tests existed for the functionality of PDF generation in the monolith. After extraction of the service, the automated tests would require to run the monolith and microservice in combination. All seven existing automated tests for the generation of a PDF file were successful. Three additional tests were implemented which specifically test methods of the PDF generation microservice. The successful execution of the automated tests, both in the monolithic and in the microservices implementation, imply that the application's functionality was preserved during the refactoring.

**Process Review** To review the migration concept and process strategy, two interviews of 30 minutes in length were conducted with the "Head of Development" and the "Product Manager". The participants were presented details of the architectural changes, communication mechanisms, and the migration process. The "Head of Development" considered the elaborated migration strategy suitable, it should be followed for further migration activities. He anticipated changes for the created code, once asynchronous messaging and container technologies are incorporated. Besides, he expressed uncertainty about the necessary development process adaptions in alignment with the company strategy. The "Product Manager" similarly stated that the PoC has reached its goal from his perspective. While he regarded the selected migration strategy suitable for the PoC and most likely also or the further migration process, he still considered it beneficial and necessary to evaluate alternative strategies. Moreover, he expressed the wish for a guide the describes the generation of ideal microservices for the service/sales application, incorporating the code review results. Such an internally published guide could then be used in a follow-up PoC to extract five to six additional microservices of differing complexities.

In summary, both the "Head of Development" and the "Product Manager" considered the PoC a success. However, before rolling out a migration of the service/sales application to microservices, alternative migration strategies should be evaluated and additional exemplary services should be extracted.

The use of containers and messaging should be investigated further, and adaptions to the development process with regard to DevOps practices should be elaborated. While not in the scope of this case study, an expense assessment was also considered advantageous.

#### 10.2.2.5. Evaluation of Field Notes

Throughout the entire PoC, structured field notes have been created to document observations of the involved participants. In the timeframe of March to July 2023, a total of 32 structured field notes were created by Knodel [Kno23]. Nine field notes refer to the first phase, seven to the second phase, and 16 to the third phase of the framework.

In phase one, four field notes emphasize constructive discussions during the architecture evaluation, while it was noted twice that participants had to be encouraged to engage in the discussions. The elicitation of scenarios and identification of measurable criteria was perceived difficult by some participants. According to few participants, all quality attributes would equally important, which would make a prioritization difficult. Linking the suitability of architectural styles to scenarios likewise revealed some unfamiliarity, but embarked into a fruitful discussion. One participant stated that it would have been interesting to see results yielded by alternative architecture assessment methods. Suggestions for improvements were given three times, two times related to the *ARH* application and its usability.

In phase two, it was noted five times that the available migration strategies and refactoring approaches were well presented. One note emphasized the framework's capability for allowing several migration strategies from which a suitable one can be selected. Two comments expressed uncertainties during the filtering of the approaches, while it was noted once that not all linked publications a freely accessible. As well, the availability of tools for other languages than Java was perceived unfavorable.

In the field notes for phase three, it was mentioned that the already well-structured monolith allowed for an unproblematic identification of bounded contexts and its subsequent decomposition. As well, confidence

about the accepted microservices architecture and the PoC implementation were recorded. Seven comments centered around source code and implementation aspects, e.g., on consulting other software developers for support. Three comments mention positive review results in this regard. Problems targeted testing activities one time, and the selection of a tool for service identification two times. Here it was noted the that the documentation of tools varies significantly and that not all tools could be executed successfully. Filtering patterns and best practices was perceived generally positive, but uncertainties were expressed about the accuracy. A better documented guideline for pattern selection was regarded favorable, as well as a completion by not yet cataloged patterns.

### 10.2.3. Discussion

The initial architectural evaluation yielded strategic goals and quality goals, that served as a solid foundation for the migration of the L-mobile monolith. Our migration methodology and its associated *ARH* application supported the identification and efficient filtering of 90 refactoring approaches. Eleven publications were considered for defining a migration strategy and to subsequently identifying services. Ultimately, the approach of Li et al. [LML20] was selected to guide the strategy definition. For service identification and decomposition, the tool-based approach by Krause et al. [KZH+20] was used as the primary source.

The developed migration concept and the identified microservices were accepted by the “Software Architect”, “Head of Development”, and “Product Manager” with only minor adjustments. Accordingly, the framework’s guidance led to an appropriate process strategy and service identification approach. An obstacle to automation was that many of the tools associated with the approaches were only available for Java-based applications and often insufficiently documented. Refactoring and splitting the database turned out to be a particular challenge. The catalog of patterns and best practices was found to be helpful and contributed to a solid architecture design. However,

their filtering in the *ARH* was considered not yet sufficiently accurate and may have led to a disregard of relevant patterns and best practices.

Challenges arouse from limited experience that participants had with methods for architecture evaluation and the microservices concept in general. Here, it was suggested that our methodology could better support inexperienced users, which confirms our earlier observations in Chapter 9. For service implementation, examples, tutorials, or videos could be referenced, such as the one[1] mentioned by Knodel [Kno23]. As a summarizing result of his work, the author concludes that the framework and its accompanying tool are well suited for migrating the service/sales application of L-mobile towards microservices.

## 10.3. Refactoring of a Document Management Solution

Our second case study investigates the document management solution jadice flow[2]. The application is developed by the company levigo solutions[3], a small enterprise with over 30 employees, of which about six were involved in the development of jadice flow. During its lifetime, the product was migrated from the monolithic solution jadice server[4] to a microservices-based architecture. Both products offer workflow-based document processing that can manage complex data streams. For example, the email converter of jadice flow allows to split emails into body and attachments, to convert them into specific target formats, and to process them individually. jadice flow offers additional features such as comprehensive monitoring, deep analysis and processing of archives in appendices, which go beyond the functionality of standard email solutions. From an architectural point of view, the transition to microservices allows for efficient scaling of individual

[1] .NET Microservices (YouTube) https://youtu.be/CqCDOosvZIk?si=AFP8hIsqvYh5cVPg

[2] https://www.levigo.de/jadice-flow/

[3] https://www.levigo.de/dokumentenmanagement/

[4] https://www.levigo.de/jadice-server/

processing steps. While most of the services are implemented in Java, the use of container technologies promotes technological heterogeneity.

To optimize and further improve jadice flow, the development team focuses on three design aspects of the application: optimal service granularity, communication model and paradigm to control the individual units, and reduction of the current I/O bottleneck. Accordingly, the following research questions formulate our research objective and frame the scope of this case study:

**RQ1:** How can our methodology be used to improve specific qualities of an existing microservices architecture?

**RQ2:** Which techniques are suitable for determining the optimal service granularity of a microservices architecture?

**RQ3:** Which approaches, patterns or best practices allow for optimizing the inter-service communication?

### 10.3.1. Methods

This case study aims to develop a refactoring concept to improve an existing microservices architecture. We want to further evaluate our developed methodology and examine its applicability specifically for this application scenario. Again, we take a software architect's perspective and neglect organizational and infrastructure-related aspects. We apply our in Chapter 7 introduced framework, including all three phases. In the second phase, we aim to identify suitable refactoring approaches that address the specific optimization areas of the jadice flow application. The preferred approaches are then critically reflected in three expert interviews with members of the development team. In the following, we describe our research process and the associated data collection methods in detail.

**Migration Framework** Our framework will guide the creation of a refactoring concept as described in Section 10.2.1. Phase one starts with an architecture evaluation based on Svahnberg and Mårtensson [SM07] (Section 10.2.2.1). The elicited scenarios and associated quality goals guide

the selection of refactoring approaches in the framework's second phase. Therein, we perform several rounds of filtering, including additional filters that focus on our targeted improvements to jadice flow, i.e., the optimization of service granularity and inter-service communication. Based on the generated recommendations, candidate refactoring approaches are selected and evaluated for their suitability. In the framework's third phase, suitable patterns and best practices for the architecture design are selected that address the quality goals defined in phase one.

**Expert Interviews** After the selection and analysis of suitable refactoring approaches, their appropriateness is evaluated in three interviews with experts from the development team. Next to the approaches themselves, the interviewees critically reflect on our migration methodology as a whole. To follow a basic structure, but also to capture related aspects and rationales, we opted for semi-structured interviews based on the guidelines by Seaman [Sea08]. Due to the qualitative nature of case studies, Runeson and Höst recommend involving participants with different backgrounds and perspectives [RH08] to allow for the collection of in-depth qualitative data that covers different viewpoints. Therefore, in addition to the software architect of the system, we invited the product owner (PO) of the application, who brings a holistic view of the product and its functionality. Finally, a software developer familiar with the implementation completes our three participants.

In preparation, all interviewees were provided with an interview preamble, a summary of our migration methodology, and the candidate refactoring approaches. To ensure accurate analysis of all statements made, the interviews were audio recorded. Consent hereto was obtained when the of interviewees were invited. The recordings were then transcribed by summarizing and explicating, as suggested by Mayring and Fenzl [MF19]. To provide a quick overview of the results per participant, the answers to each question were first classified and then complemented by a more in-depth discussion that aggregates all answers [Her24].

**Field Notes** As with our first case study, we created field notes as we applied our methodology during phases two and three of the framework. To this end, we used the following slightly revised version of the template used in Section 10.2.1:

- Date, time, location
- Involved persona
- Phase of framework and step
- Conducted activity and decisions
- Things that went well
- Things that did no go well
- Sentiments
- Other comments

All field notes were documented by the software engineering bachelor student who conducted the case study [Her24]. His assessment is reflected in the documentation of "Things that went / did not go well", thereby aiming for a more consistent weighting and reporting of activities involving multiple personas.

### 10.3.2. Results

In this section, we present the framework-guided process towards a refactoring concept for the jadice flow application. We then present its evaluation through expert interviews with three stakeholders of the system, and conclude the section with an analysis of the collected field notes.

#### 10.3.2.1. Phase One: Architecture Evaluation

Following our discussion in Section 10.2.2.1, an architecture evaluation based on Svahnberg and Mårtensson [SM07] was conducted during two days in November 2023. Participants included four software developers and architects, as well as the product owner (PO) and the software engineering bachelor student who took the moderator role. The user's perspective was adequately represented by the close interaction between the development

team and customers. The structure described in Section 10.2.2.1 was followed with the exception of the fourth step "architecture presentation", since all participants were familiar with the system. To align the results with the workflow of the *ARH* in phase one, we iterate through the five steps of 1) project description, 2) strategic goals, 3) quality goals, 4) quality assessment, and 5) results.

**Step 1: Project Description** The system under study is a document management solution and was previously migrated to a microservices architecture. Its client-server architecture is shown in Figure 10.4. The jadice flow controller serves as a communication endpoint for clients. It manages the execution of jobs by scalable worker services and provides the current job status to clients [lev24]. Job metadata is persisted in an attached relational database, such as MariaDB. The Data Store is implemented as a Eureka service (according to the source, S3 and WebDAV are also supported) and persists documents and related data. Three to four base services provide the application's core functionality, which can be equipped with up to 50 worker services on demand. Kubernetes acts as the orchestrator for the containers. The application's code base contains approximately 200,000 lines of code in about 200 classes, implemented in Java and Spring Boot.

**Step 2: Strategic Goals** The strategic goals of the company are motivated by the evolution of the successful product jadice server into a new solution that can meet current and future customer requirements. Overcoming the limitations of the existing architecture was seen as a prerequisite for this. A transformation to a microservices architecture should enable the efficient handling of varying workloads and the implementation of an agile, cloud-capable deployment concept.

**Step 3: Quality Goals, Assessment, and Summary** For the identification of quality goals, each participant was asked to document the three most important quality sub-attributes for the system. While the identification aimed at sub-characteristics based on ISO 25010 [ISO11a], several participants felt more comfortable using top-level quality attributes.

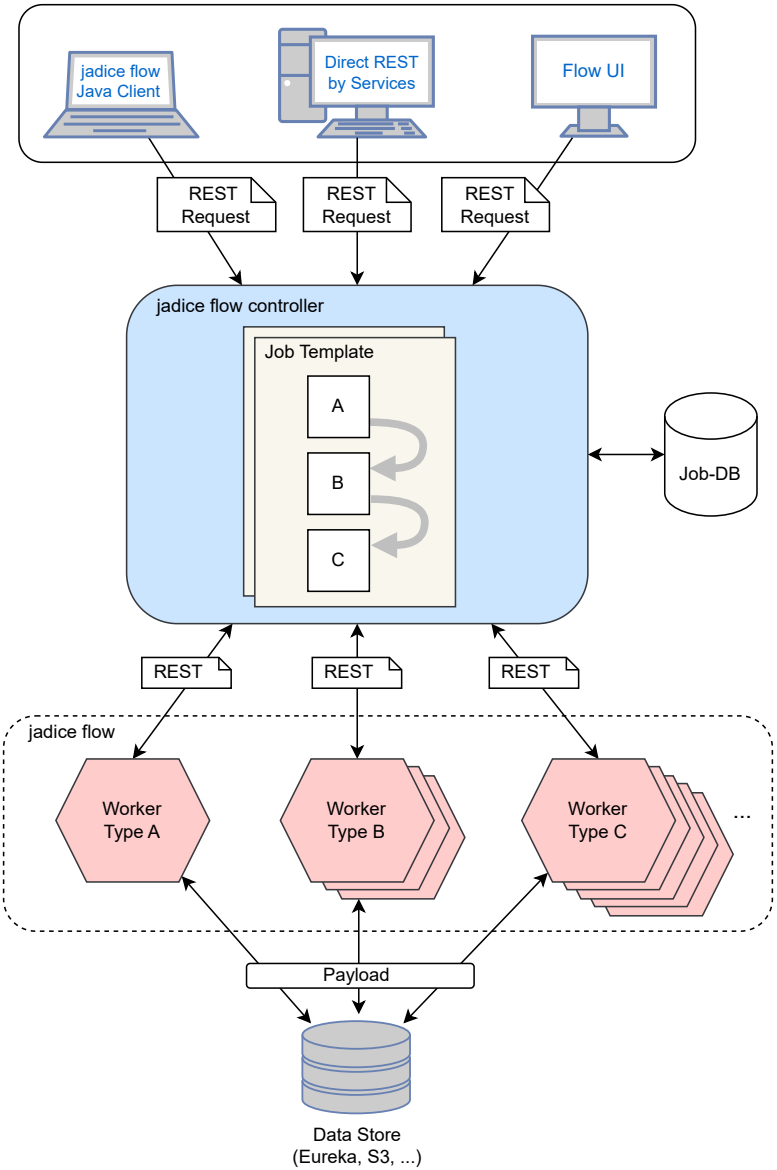



**Figure 10.4.:** Architecture of Document Management Solution (Adapted from levigo solutions [lev24])

A subsequent free discussion led to a set of six quality (sub-)attributes and their prioritization: 1) *Scalability*, 2) *Modularity*, 3) *Time Behavior*, 4) *Deployability*, 5) *Fault Tolerance*, and 6) *Recoverability*. In the following open discussion, the team agreed to elicit a maximum of two scenarios per quality (sub-)attribute. The scenarios were then ranked in terms of importance and difficulty of implementation, resulting in the utility tree shown in Figure 10.5. In a second session, the elicited scenarios were discussed again by all stakeholders and enriched with additional characterizing quality attributes. Finally, each scenario was assigned a preferred architectural style – monolith, microservices, or both. While consensus was quickly reached on, for example, *Scalability*, weighing the pros and cons for other attributes turned out to be more difficult. An open discussion led to an unanimous classification for each scenario, indicated by the green (microservices) and red (monolith) arrows per scenario in Figure 10.5. A microservices architecture was seen to be better suited for five scenarios, while three would preferably be realized in a monolith. The fact that jadice flow is already implemented using microservices was not seen as an obstacle to performing this task. Instead, the participants interpreted these results as a confirmation of their earlier decision to migrate to microservices.

#### 10.3.2.2. Phase 2

In the second phase of the framework, we focus on selecting service identification approaches that address the quality goals elicited in the phase one and also contribute to the aspects expressed by our second and third research questions. To combine both goals, we performed several rounds of filtering on the repository of refactoring approaches. At the time of conducting this case study, the *ARH* repository contained 110 approaches. We collected the results for all filter configurations and subsequently aggregated them.

**Filter Selection** The filter configuration was created by the software engineering bachelor student and the product owner of jadice flow. Table 10.7 shows the two created filter configurations “Filter 1” and “Filter 2”. All the

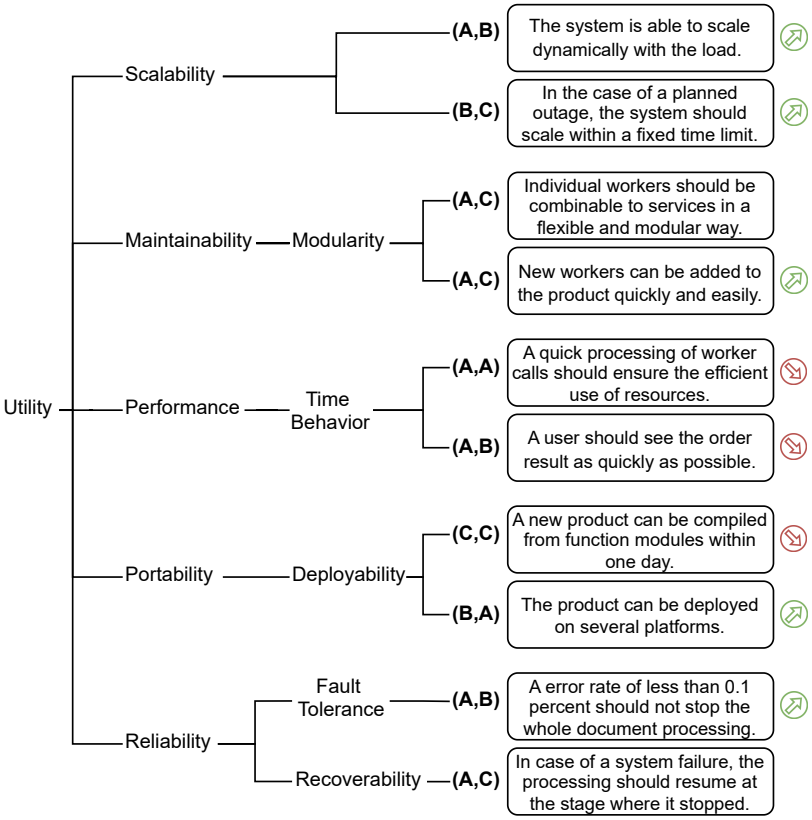



**Figure 10.5.:** Utility Tree for the jadice flow Application with Elicited Scenarios, Importance and Difficulty Rating, and Preferred Architecture: Red Arrow = Monolith, Green Arrow = Microservices (Adapted From Hermann [Her24])

properties listed have been set to “include”, while the “exclude” functionality has not been used. The nine prioritized properties for “Filter 1” reflect the configuration that would have been used for the primary query. The six properties for “Filter 2” were specified in addition to reflect lower-priority aspects. When creating the filter configurations, the following aspects were

taken into account: 1) The goal was to select only the necessary filters to ensure a high correlation with the generated results. 2) Design properties were considered particularly important, leading to the inclusion of five out of seven properties. 3) High importance was given to the *Input* and *Output* categories. 4) The *Process* and *Usability* categories were allowed the greatest variability, as the research objective required fewer restrictions in this regard. Table 10.8 shows the filtering rounds performed in combination with the quality goals.

**Table 10.7.:** Filter Configurations for Refactoring Approaches in Phase Two

| Category | Sub-Category | Filter 1 | Filter 2 |
|---|---|---|---|
| Quality | Design Properties | Automomy, Granularity, Technology Heterogeneity | Complexity, Isolation |
| Input | Domain Artifacts | Human Expertise | VCS History |
| | Runtime Artifacts | | Log Traces |
| | Executables | API/Interface, Source Code (No Specification) | Source Code (Java) |
| Process | Process Strategy | Refactor | |
| Output | Representation | List of Services, Splitting Recommendations | Guideline/Workflow |

**Table 10.8.:** Rounds of Filtering and Their Configuration in Phase Two

| Round | Quality Attributes | Filter 1 | Filter 2 |
|---|---|---|---|
| 1 | × | | |
| 2 | × | × | |
| 3 | × | × | × |
| 4 | | × | |

The results of all four filtering rounds were aggregated by considering the five highest-ranked approaches per round. If multiple approaches had the

same number of matches as the fifth-ranked approach, they were all included. An exception was made for round four, where 12 equally ranked approaches followed the first two. Due to the low priority of this configuration without quality attributes, it was decided to include only the first two results. The aggregated results are shown in Table 10.9.

**Table 10.9.:** Resulting 8 Refactoring Approaches With Their Ranking in Four Rounds of Filtering

| Publication | Rank 1 | Rank 2 | Rank 3 | Rank 4 |
|---|---|---|---|---|
| Habibullah et al. [HLT18] | 1 | 1 | 1 | – |
| Daoud et al. [DEF+21] | 3 | 2 | 2 | 2 |
| Vera-Rivera et al. [VPAG21] | 2 | 3 | 3 | – |
| Freitas et al. [FFC21] | 4 | 5 | 4 | – |
| Matias et al. [MCF+20] | 5 | 6 | 5 | – |
| De Alwis et al. [DBFP21] | – | 4 | – | – |
| Assunção et al. [ACC+22b] | – | 7 | – | – |
| Selmadji et al. [SSB+20] | – | – | – | 1 |

**Resulting Recommended Approaches** In the following, the eight resulting refactoring approaches were examined in detail to assess their suitability for the jadice flow application. Two publications were excluded after a first inspection: The approach by Freitas et al. [FFC21] was considered unsuitable for verifying the granularity of an existing microservices application. Similarly, the approach by Habibullah et al. [HLT18] was considered inappropriate due to its high level of abstraction that lacks a concrete method for service identification.

The required input artifacts for the remaining six approaches differ significantly. While three approaches process existing source code ([ACC+22b; DBFP21; MCF+20]), class diagrams are leveraged in addition in [ACC+22b]. Extensive human expertise in addition to source code is needed to apply the approach by Selmadji et al. [SSB+20], while the De Alwis at al. [DBFP21]

require an existing database schema in addition. Vera-Rivera et al. [VPAG21] leverage user stories, while the approach by Daoud et al. [DEF+21] works on business process level. The reviewed approaches create a graph representation of the processed input based on the corresponding atomic unit such as class or function. For the refactoring of jadice flow, the input artifact types of user stories and business processes were considered most appropriate, as they would allow for manual intervention in their generation. Approaches operating on source code, however, commonly assume an originating monolithic application.

Four approaches use clustering techniques to generate candidate microservices [DBFP21; DEF+21; MCF+20; SSB+20], with coupling and cohesion as the primary target metrics. Vera-Rivera et al. [VPAG21] employ a genetic algorithm, and Assunção et al. [ACC+22b] rely on an evolutionary strategy. Regarding the techniques employed by an approach, no preference was given to either cluster-based approaches or evolutionary/genetic algorithms. Due to lack of experience and supportive comparisons, participants felt uncertain about how different algorithms would affect the decomposition.

The automation capability of an approach was generally desired. Therefore, all six remaining approaches were examined for their level of automation and tool support. For two approaches, a tool is mentioned but not accessible or publicly available ([DEF+21; VPAG21]). De Alwis et al. [DBFP21] employ a tool for static code analysis, while the tool used by Matias et al. [MCF+20] was incompatible with the application's technologies. The two remaining approaches do not provide tool support ([ACC+22b; SSB+20]).

**Candidate Approaches** After reviewing the six refactoring approaches, two approaches were identified as candidates and considered for detailed investigation. The approach by Vera-Rivera et al. [VPAG21] proved to be the most promising for existing microservices architectures. A large number of metrics and visualizations offer comprehensive guidance, but also increases its complexity. However, due to the unavailable tool, the case study author and product owner regarded an alternative manual implementation as too time-consuming within the scope of this case study.

The second preferred approach by Daoud et al. [DEF+21] was also considered applicable due to its operation at the business process level. The authors propose a procedure based on their pseudocode provided "cHAC" algorithm. An attempt has been made to implement this algorithm in Java. However, an insufficiently detailed description of the formulas and functions used prevented its timely implementation. Due to the expected manual effort for both candidate approaches, a refactoring concept could not be elaborated in the course of this case study. However, the collected results serve as a basis for the following expert interviews, which aim to evaluate their potential from different perspectives.

#### 10.3.2.3. Phase 3: Architecture Definition

To improve and optimize an existing microservices architecture, design recommendations in the third phase of our framework can yield appropriate patterns and best practices that address the predefined quality goals, and facilitate the specific optimizations expressed by research questions two and three. At the time of the case study, the repository cataloged 51 patterns and 43 best practices (Appendix A), that were filtered based on the quality goals specified in phase one. The first 20 search results for both patterns and best practices are shown in Table 10.10. Each item is listed with the number of matching quality (sub-)attributes from a total of attributes 17 specified in phase one. A preliminary review of the results by the case study author and product owner revealed promising recommendations among the individual patterns and best practices. Their detailed review was scheduled for the following round of expert interviews.

#### 10.3.2.4. Expert Interviews

Three expert interviews were conducted with the product owner, the architect, and a software developer of the jadice flow application. The demographics are shown in Figure 10.11. All participants had extensive experience with microservices of at least six years. The product owner participated in all

**Table 10.10.:** Filter Results for Patterns and Best Practices

| Pattern | Matches | Best Practice | Matches |
|---|---|---|---|
| Service Registry | 11/17 | Isolated State | 11/17 |
| CQRS | 10/17 | Dynamic Scheduling | 9/17 |
| Pipes and Filters | 10/17 | Infrastructure Abstraction | 8/17 |
| Asynchronous Messaging | 9/17 | Use Infrastructure as Code | 8/17 |
| Gateway Offloading | 9/17 | Acyclic Calls | 7/17 |
| Load Balancer | 9/17 | Automated Infrastructure | 6/17 |
| Ambassador | 7/17 | Automated Monitoring | 6/17 |
| Backend for Frontend | 7/17 | Operation Outsourcing | 6/17 |
| Container | 7/17 | Cloud Vendor Abstraction | 5/17 |
| Database is the Service | 7/17 | Immutable Artifacts | 5/17 |
| Scalable Store | 7/17 | Standardized Deployment Unit | 5/17 |
| API Gateway | 6/17 | Built-in Autoscaling | 4/17 |
| Circuit Breaker | 6/17 | Persistent Communication | 4/17 |
| Deploy Cluster and Orchestrate Containers | 6/17 | API-based Communication | 3/17 |
| Edge Server | 6/17 | Automated Restarts | 3/17 |
| External Load Balancer | 6/17 | Autonomous Fault Handling | 3/17 |
| Gateway Aggregation | 6/17 | Coarse-Grained Microservices | 3/17 |
| Internal Load Balancer | 6/17 | Communication Indirection | 3/17 |
| Local Database Proxy | 6/17 | Guarded Ingress | 3/17 |
| Local Sharding-based Router | 6/17 | Loose Coupling | 3/17 |

activities of the case study, while the other two participants were informed about the previous activities via an interview preamble. It included information about our framework, the activities performed, and their results. The interviews were scheduled to last 45 minutes, with an additional 15 minutes buffer to avoid time pressure and interruptions for longer answers. In the following, we summarize the four groups of questions by aggregating all responses and highlighting important statements and findings. A comprehensive analysis including the participants' responses to each question, encoded in a 4-point ordinal scale, can be found in the work of Herrmann [Her24].

**Table 10.11.:** Interview Demographics (Adapted From [Her24])

| Participant | Job Role | Experience | | |
|---|---|---|---|---|
| | | Professional | Microservices | Product |
| P1 | Product Owner | 20 years | 6 years | 13 years |
| P2 | Software Architect | 13 years | 7 years | 2 years |
| P3 | Software Developer | 8 years | 8 years | 6 years |
| Median: | | 13 years | 7 years | 6 years |
| Mean: | | ~13.7 years | 7 years | 7 years |

**Question Group 1: Refactoring Approaches for jadice flow** The first group of questions asked about the suitability of the recommended refactoring approaches for the jadice flow application. Both candidate approaches [DEF+21; VPAG21] (Section 10.3.2.2) were rated as potentially suitable by all participants. This assessment was strongly influenced by the required input artifacts and the expected implementation effort of the approaches. One expert favored the approach by Vera-Rivera et al. [VPAG21], while the other two experts would prefer the approach by Daoud et al. [DEF+21]. The decision for the former was based on its suitability for an existing microservices architecture and an optimization of its service cut, supported by a visualization of the metrics involved. The other respondents rated these aspects as less important. Instead, both preferred BPMN diagrams over user stories as input artifacts, as such diagrams would provide a more accurate representation of the application's workflows and thus the communication between services. Nevertheless, both experts considered the first approach as potentially useful too, but only in addition to the second.

The interviewees were then asked about potential technical challenges or hurdles that could arise from using these two approaches. One expert did not see any significant problems, while the other two expressed minor challenges: relying on BPMN diagrams as input would limit the level of detail to the expressiveness of this modeling technique. Also, this type of input artifact would lack the ability to handle unforeseen problems. For example,

edge cases might not be considered by the approach, but could have a major impact on the success of an architectural refactoring. When asked about the objective expressed in our second research question, optimizing the service granularity of the jadice flow application, one interviewee was very confident about the capabilities of the favored approach in this regard. The two other respondents also expressed a positive tendency, but were still unsure due to the lack of a concrete concept and therefore the hypothetical nature of this question.

**Question Group 2: Patterns and Best Practices for jadice flow** The second group of questions targeted the recommended patterns and best practices for the jadice flow application (Section 10.3.2.3). The questions aimed at evaluating the *ARH*'s ability to recommend appropriate patterns and best practices based on the quality goals specified in phase one. All interviewees were initially provided with the complete list of all 51 cataloged patterns and 43 best practices (Appendix A). A short explanation for each item helped the participants to become familiar with less common ones.

In a first question, the experts were asked to rate the ranking of patterns and best practices based on the scenarios and quality attributes. The ranking was generally perceived as meaningful by all participants. In some places, the experts expected a different prioritization, but most of these objections could be put into perspective after the experts double-checked the specified quality goals. However, one expert noted that unlike the recommended refactoring approaches in phase two, a restriction to a small subset of results is not necessary for patterns and best practices, as several of them are usually combined. According to him, this fact would reduce the importance of a precise order of the results.

In the following, the results provided by the *ARH* web application were contrasted with the already implemented patterns and best practices for jadice flow. An aggregation of the interviewees' answers resulted in an existing (but partially incomplete) implementation of 13 patterns and 15 best practices. Of the remaining recommended items, five patterns and four best practices were considered potentially useful for a future implementation.

The last question in this group addressed our third research question, which aims to optimize the inter-service communication. The experts identified between one and two patterns and best practices to this end. Accordingly, they noted that only a few results would be suitable and that they did not rank high in the list of results. One expert aptly related this fact to the specified quality attributes, which were less focused on the aspect of inter-service communication.

**Question Group 3: Framework and Methodology** The third group of questions addressed the migration framework and methodology, including our *ARH* web application. First, it was explicitly mentioned to assess its practical applicability in general, independent of the results obtained in this case study. While two participants needed clarification to distinguish our meta-approach from specific refactoring approaches, one interviewee immediately referred to our methodology. The responses showed a general acceptance of the reflected process covering three migration phases, their included activities, and their quality-oriented integration with each other. The ability to compare and select from a large number of methods and approaches in phase two was seen as a desired and helpful functionality. It would help to consider a variety of refactoring approaches that would otherwise be difficult to identify.

When asked about potential problems with our meta-approach, one participant pointed to its documentation, e.g., for manual filter configuration. It was also suggested to provide more guidance on the migration process and to enrich the collected information with visualizations to provide a better basis for discussion within the team. Another interviewee stated that the *ARH* application's usefulness is closely related to the scope and timeliness of the cataloged refactoring approaches. Although our implemented quality model is specifically tailored to microservices migrations, it was mentioned that some desired product qualities are not reflected, which would require manual effort, e.g., to map them to available quality attributes.

Finally, the overall usefulness of the migration framework and methodology was queried, now including the results of this case study. The intervie-

wees were asked to distinguish between optimizing the current microservices architecture and refactoring the legacy monolith. Two participants found the methodology useful for refactoring the current architecture, while the third participant expressed a tendency towards its usefulness. He was unsure whether using the migration methodology would reveal new, yet unknown optimization potential. However, all experts agreed that the *ARH* web application provides a valuable overview for architects and promotes new solutions proposed by academia. In assessing the overall usefulness, the interviewer found a correlation with the professional experience of the expert – the more experienced interviewee tended to a better overall rating.

#### 10.3.2.5. Field Notes

Throughout the case study, structured field notes were collected to document the activities and observations of the personas involved. From November 2023 to February 2024, a total of 18 structured field notes were created by Herrmann [Her24]. While the architecture evaluation in the first phase was excluded, 14 notes refer to the second phase and four to the third phase of our applied framework. Three field notes concern the filter configuration for refactoring approaches in phase two and were performed by the bachelor student and the product owner of jadice flow. They indicate positive results of all performed activities, two times complemented by a positive sentiment. In one of the field notes, the author recorded uncertainty in applying the various available filters. This uncertainty was attributed, in part, to the author's lack of prior experience and the limited documentation of filters.

In the following, eight field notes were created while inspecting the recommended refactoring approaches, one for each approach. Three more field notes refer to meetings in which the resulting approaches were discussed and prioritized between the two personas. The field notes show mixed results, with a slight tendency towards positive outcomes. The author attributes this number of negative results to the refactoring approaches that were deemed inappropriate for the jadice flow application – ultimately, only two candidates out of the eight approaches were considered. The notes

also show that the meetings with the product owner in this process led to quick and tangible results and were therefore rated mostly positive. Four field notes were created when applying the third phase of the migration framework. They focus on the attempts to apply the two favored refactoring approaches and to create a migration concept in phase 3a, while the filtering for patterns and best practices was not covered. The overall negative results and sentiments for phase 3a were attributed to the complexity and limited experience of the bachelor student in working on a migration concept.

In summary, we see mostly positive results and sentiments for the activities related to the *ARH* web application in phase two. The eventual examination of specific refactoring approaches and their incorporation into a migration concept raised new challenges that were not directly related to our framework and methodology. While no field notes were created for phase one, the author separately emphasizes the positive results of the initial architecture evaluation [Her24].

### 10.3.3. Discussion

In this case study, we evaluated the appropriateness of our methodology for improving an existing microservices architecture with respect to specific product quality aspects. In addition, the evaluation aimed to determine the optimal service granularity of an existing decomposition and to optimize the inter-service communication. The initial architecture evaluation resulted in the elicitation of 10 scenarios, each with one or more associated quality attributes. The resulting quality assessment confirmed microservices as the preferred architectural style for the jadice flow application.

To identify suitable refactoring approaches in the second phase, four rounds of manual filtering with different filter configurations were performed on the repository cataloging 110 approaches. Three rounds produced consistent results that were strongly influenced by the quality goals initially defined in phase one. The fourth round was performed without these quality goals and yielded an additional refactoring approach. The selection of two candidate approaches out of eight recommended approaches was mainly

determined by the types of input artifacts required: user stories and BPMN diagrams. Most of the remaining approaches were identified as inappropriate due to their required input, automation capability, or requirement of an existing monolithic system. As such, the filtering functionality in phase two was considered reliable and largely satisfactory. Both the student conducting the study and the product owner of jadice flow were convinced that a manual search without the methodology used would not have yielded comparable results in terms of breadth and adequacy. The available timeframe for the case study did not allow the successful elaboration of a migration concept based on one of the two approaches. There are several reasons to mention: Neither approach provided a usable tool, requiring manual implementation. An attempt to implement the provided algorithm for one of the approaches was not possible due to its insufficient documentation. Lastly, the concept was elaborated by a software engineering bachelor student with limited experience in software architecture and refactoring. However, the three interviewed experts considered the identified refactoring approaches as basically appropriate for the jadice flow application. This assessment included both their practical applicability and their suitability for optimizing the application's service granularity. Given the necessary resources, one expert was convinced of a successful implementation, while the other two experts cautiously signaled a positive tendency. We have thereby answered our second research question about suitable refactoring approaches for determining the optimal service granularity of an existing microservices architecture.

In phase three, a filtering for patterns and best practices was performed based on the quality goals specified in phase one. The three experts were queried regarding the recommendations generated by the *ARH* web application. They assessed the ranking of recommendations as appropriate and, moreover, found out that 13 patterns and 15 best practices were already implemented in the current version of jadice flow. Another five patterns and four best practices would be considered in the future. The results indicate that the generated recommendations were appropriate for the jadice flow application. Next to the generated results, the experts identified additional patterns and best practices from the repository that could potentially im-

prove the communication model and the associated network overhead of the jadice flow application, thereby addressing research question three. Since the quality goals specified in phase one do not specifically target this aspect, two experts suggested extending the filtering in phase three to include such aspects. However, the experts also noted that holistically addressing the issue of network overhead would first require the generation of an optimized service cut, which should be complemented by suitable patterns and best practices. For example, identifying frequently communicating services through performance modeling could lead to the design of larger microservices that require less inter-service communication.

## 10.4. Limitations

The majority of both case studies were conducted by software engineering students with little or no work experience. Support was provided by senior developers, architects, product owner, and supervisors in the form of feedback and reviews. The case study participants had a vested interest in the positive outcome of this work and may have been tempted to omit negative comments or provide more positive feedback. This threat was countered by making it clear to all participants that negative results would not affect the grading of the case study results. In particular, the collected field notes confirm that also negative feelings were expressed in various ways. Furthermore, the semi-structured interviews conducted need to be critically reflected upon regarding their limitations, as discussed in Section 9.4.

The implementation phase in our first case study targeted only a small component that was loosely coupled by design. The results may be different for larger services, which would require more extensive database refactoring. However, the scheme applied should be similarly applicable to larger components due to the already well modularized service/sales application. During the service identification, only static analysis tools were applied. The identified services could be different if dynamic analysis were added.

However, we are confident that the inclusion of human expertise from senior developers sufficiently compensated for this drawback.

In both case studies, the chosen scenario-based architecture evaluation method was more compact than, for example, ATAM or SAAM. Nevertheless, we are confident that the results obtained are reliable in the microservices migration context. Since the first case study aimed to create a migration concept accompanied by a PoC, a more comprehensive method could be chosen later to justify the actual migration to microservices.

Finally, the results obtained from the field notes cannot be generalized and would likely differ for other organizations and applications. The limited reliability due to the influencing subjectivity of the field note writers must be considered when interpreting the results [RH08]. We attempted to mitigate this risk by closely following established guidelines [Sea08] and by involving the students' supervisor in the analysis of the field notes.

Considering the outlined limitations and our countermeasures, we still consider both case studies a successful exemplary demonstration of our methodology. We expect promising results for its application to similarly object-oriented, well-structured, and modularized systems like the L-mobile monolith. The generalizability regarding optimizations of existing microservices-based applications like jadice flow is more limited due to our specific objective, but the applied scenario-based workflow still proved effective for this application area.

## 10.5. Conclusion

With both case studies, we aimed at a holistic evaluation of our developed methodology and *ARH* web application in an industrial environment. While our first case study dealt with the application scenario of migrating a monolithic system to microservices, our second case study aimed at improving an existing microservices architecture. We successfully applied our methodology in both environments in the course of two software engineering theses.

Both case studies began with a scenario-based architecture evaluation of the existing system, reflecting the first phase of our framework. Although most of the participants had little experience with architecture evaluation methods, the method proposed by Svahnberg and Mårtensson [SM07] yielded promising results.

In the first case study, filtering the repository of 90 refactoring approaches yielded two publications that were selected as primary sources for elaborating a migration concept. The approaches by Li et al. [LML20] and Krause et al. [KZH+20] guided the definition of a migration strategy and the tool-assisted decomposition. The developed concept was applied in a proof-of-concept where a single microservice was extracted from the monolith. Internal reviews by senior developers confirmed the appropriateness of the service cut, resulting in 41 microservices. The subsequent design of the microservices architecture was facilitated by a catalog of 51 patterns and 43 best practices for architectural designs, from which 14 patterns and the same number of best practices were applied. Automated testing proved the preservation of existing functionality, while code reviews ensured compliance with corporate standards. While the L-mobile stakeholders considered the concept a success, the company's journey to microservices raised additional process-related questions that were not entirely within the scope of this work. The variety of available migration strategies and approaches encouraged the product manager to conduct further evaluations involving alternative concepts. He also suggested fostering microservices expertise through a microservices migration guide that could serve as an internal knowledge base. The case study was accompanied by the creation of structured field notes that yielded in-depth qualitative data throughout the process, such as suggestions for improving the *ARH* web application by providing more guidance for inexperienced users.

The second case study aimed at improving the jadice flow application with respect to specific quality aspects, in addition to finding the optimal service cut and optimizing the inter-service communication. In four rounds of filtering with different configurations, two candidate refactoring approaches were identified in the second phase: the approaches of Vera-Rivera et al. [VPAG21]

and Daoud et al. [DEF+21]. Several obstacles, such as the lack of tool support and the limited timeframe, did not allow a comprehensive refactoring concept to be elaborated. However, in the course of three expert interviews with a software developer, the architect, and the product owner, both candidate approaches were discussed in detail and found their proponents among the experts. Furthermore, it was found that out of the recommendations in phase three, 13 patterns and 15 best practices had already been implemented, while another five patterns and four best practices will be considered in the future. These results support the appropriateness of the generated recommendations for the jadice flow application. Finally, all interviewees agreed that the *ARH* web application is a valuable resource for architects and promotes new solutions proposed by academia, which would be difficult to identify by other means. With regard to the overall evolution of jadice flow, the case study initiated important future steps. Applying our methodology revealed existing problems and new approaches to solving them.

The provided suggestions for improvements in both case studies largely confirm the aspects already identified in Section 9, underpinning their relevance in practice. We conclude that both case studies demonstrated a successful application of our migration methodology and *ARH* web application in an industrial setting.

## 10.6. Outlook

Future case studies, as the one presented in Chapter 4, could focus on identifying requirements that emerge from recent technological advancements, such as the "No-Code/Low-Code" [AEA23; Rym17] movement for rapid application development. The use of such platforms by enterprises is growing steadily and has already reached a significant level [Sch23a], with one prominent example being SAP Build[1]. It could be investigated, how an architectural refactoring process can generate pre-built and reusable microservices that allow for developing complex application by means of graphical user

[1] https://www.sap.com/products/technology-platform/low-code.html

interfaces exclusively. The gained flexibility would promote code reuse, allow for quicker adaptions, and speed up application development. Besides, relieving developers from the need to write code, would open this field for a variety of other creators beyond the classical software developer [AEA23]. Combining the no-code/low-code paradigm with a transformation to microservices would thereby not just break up complex monolithic systems, but also revolutionize the application development approach.

Another technological advancement marks the emergence of generative artificial intelligence, driven by the emergence of large language models (LLMs) [AVM+24] and popularized by OpenAI's generative pre-trained transformer ChatGPT[1]. While ChatGPT itself has proven to be a solid tool for generating new source code, its refactoring capabilities still reveal potential for improvement [AVM+24], as "*it exhibits limitations in comprehending the broader context*" [DMH+24]. GitHub Copilot[2], a joint venture by Microsoft and OpenAI, has become another popular AI-based assistant specifically for software developers. Hence, we asked ChatGPT, how GitHub Copilot can support the refactoring of legacy applications to microservices. Referring to its knowledge base as of January 2022, the chatbot responded with a list of possible areas where Copilot can provide insight and assistance, including code discovery and navigation, dependency analysis, and code understanding. While not being specifically tailored to refactoring legacy applications, it would actively support the development of new artifacts through code reviews, and recommending patterns and best practices.

Combining no-code/low-code development with the use of artificial intelligence could further increase productivity, and even alleviate the IT skills shortage [Sch23b]. In the context of improving and enhancing our methodology for migrations to microservices, leveraging both advancements could mark and evolutionary leap forward.

[1] https://chat.openai.com

[2] https://github.com/features/copilot

Chapter 11

# Discussion and Conclusion

In this final chapter, we summarize the contributions of this thesis with respect to the initially formulated research objective. We further discuss limitations of the presented studies and sketch out possible future research directions that can extend our work. By placing our results in the context of recent technological advances, we conclude with a holistic picture of the relevance, implications, and future prospects of our work.

## 11.1. Summary of Contributions

**RQ1: What are intentions, strategies, and challenges in the context of migrating monolithic systems to microservices?** Based on two primary empirical interview studies, we answer research question one. In our first analysis contribution (Chapter 3), we interviewed 16 software professionals from 10 German-based companies, and talked about 14 migration scenarios towards a microservices architecture (in the following denoted as *enterprise interviews*). In our second analysis contribution (Chapter 4), we conducted a mixed-method empirical study in the domain of cyber-physical systems (CPS). During an industry-academia collaboration, we reviewed 146 scientific

papers on microservices and DevOps in the CPS domain, followed by an interview-based case study with 9 CPS experts in different business units at Siemens AG (in the following denoted as *CPS interviews*). While we found that CPS and enterprise application literature differ in the importance of several microservices-related challenges, our interviewees' perception of challenges and their applied practices at Siemens AG were generally much closer to the enterprise application domain than to the CPS literature. We thus regard the interview results valid and relevant in the context of our overarching research objective, which abstracts from a specific application domain.

- **Intentions** In our *enterprise interviews*, we identified the quality aspects of maintainability and scalability as main drivers for a migration to microservices. This tendency was also confirmed by related studies published between 2017 and 2023 [HOAM23; KH19; LZJ+21; TLP17], and reflects the qualities that our interviewees commonly associate with microservices-based systems [BFWZ19b]. While many systems were in place for several years or even decades, current functional requirements could not be met anymore — which constitutes the third modernization driver. The system's general operability follows, while time to market for new features was not a main driver, and comparable to the mentions of a prescribed company-wide strategy, or the scalability of development teams. The often propagated reliability and high availability of microservices-based systems were less decisive for a migration. In our *CPS interviews*, maintainability was a likewise prominent migration intention, but not so for scalability. Here, the main driver was overall development productivity, while the remaining aspects show a similar distribution compared to enterprise literature and interviews.
- **Strategies** The majority of systems in our *enterprise interviews* were migrated in a *Re-Write* strategy using current technologies. Solely refactoring the existing code base was performed in two cases, while four systems were built in a *greenfield* manner, mostly as replacement

of a COTS solution. This tendency towards a rewrite of the system was also confirmed in another industry survey by Di Francesco et al. [DLM18]. During the migration phase, old and new systems were either operated in parallel or, more often, architects used the popular Strangler pattern to gradually replace the monolith. Particularly for long-serving legacy systems, comprehensive functionality enhancements accompanied the architectural refactoring. A similar distribution show the results of our *CPS interviews*, where the *Re-Write/Re-Build* and *Re-Factor* strategies sometimes differed for subsystems of a bigger system. Here, we observed a tendency towards slow-paced migration processes in a continuous evolution manner, based on the Strangler pattern. Strong requirements for stability and compliance were stated as reasons against a swift *big bang* migration.

- **Challenges** In our *enterprise interviews*, the decomposition task turned out to be the prevalent technical challenge. For decomposing the monolith into services, most interviewees relied on a non-systematic and manual approach. Often, the existing system's structure served as the basis for identifying functional units. The explicit use of Domain-Driven Design was less favored, likely related to its associated difficulties that we pointed out in Section 2.2.1. Our *CPS interviews* revealed similar challenges related to the decomposition, such as coupling and hidden dependencies, or general insufficiencies of legacy systems like accumulated technical debt. Interestingly, in both interview studies, participants did rarely consider or were not aware of specially tailored refactoring approaches proposed by academia. Organizational challenges were prevalent in both interviews and regarded sometimes even more challenging than technical ones, especially for larger systems built by many teams. While prevalent in the *enterprise interviews*, also three CPS experts attested their teams lacking personnel experienced with microservices-related technologies and concepts — a challenge also confirmed by Knoche and Hasselbring [KH19]. Other organizational challenges centered around initiating a mindset change when

simultaneously establishing agile processes, facilitating communication between teams, or the degree of central governance.

**RQ2: What architectural refactoring techniques are applicable for the decomposition into a set of microservices?** We answer this second research question through our extensive literature review in Chapter 5, along with the tool usability evaluations reported in Chapter 6. Over four iterations, we reviewed the scientific literature on approaches for architectural refactoring and migrations to microservices, covering 110 publications between 2016 and 2023. We developed a classification scheme for such approaches that focuses on practical applicability and the targeted quality attributes. We found that the reviewed approaches vary significantly regarding level of abstraction, required input data, employed techniques, and automation capability. A universally applicable approach could not be identified, which justifies the variety of solutions for different contexts and systems. Approaches based on static analysis of source code dominate the applied techniques, followed by dynamic and lexical analysis, while historic data was rarely leveraged. Often applied is a combination of two or more analysis types, which promises more precise service cuts.

Approaches that suggest a broad applicability are often described at a higher level of abstraction and rarely offer practically usable tool support. Still, many approaches claim to provide an automated or semi-automated solution. We identified a total of 31 accompanying tools, 20 of which address the decomposition into microservices. For five of them, we generated service cuts using reference applications and reported our experiences on installation, documentation, usability, and support. Java-based source code is the dominant input artifact, with *Service Cutter* [GKGZ16] being the only language-agnostic tool we tested. While it has potential to represent a general purpose tool for automating the decomposition, it demonstrates by example the most prevalent issue: automatically transforming existing system artifacts into the tool's custom input format. Besides, another obstacle for the straightforward adoption of academia-proposed tools becomes apparent: the limited maintenance and support by the tool creators. In

comparison, the tool *Mono2Micro* [KXL+20] offers a mature and technologically advanced, AI-assisted solution. However, its drawback is the limited applicability to WebSphere application servers.

Moreover, we identified the quality goals associated with each reviewed approach, thereby laying the ground for implementing a quality-driven migration methodology. We anticipate that new approaches for migrating monolithic systems to microservices will continue to emerge, where our proposed taxonomy can support a rapid classification of new solution proposals and serve as a communication tool between researchers and practitioners.

**RQ3: How can predefined quality goals steer the migration process?** By means of a literature review in Section 2.3, we identified quality attributes and design properties relevant for microservices architectures, along with their trade-offs. We further analyzed their role during a migration (Section 2.4), and found evidence for their involvement in the major migration phases: architecture assessment, decomposition, and microservices design. The concept of a quality-driven migration describes a modernization process, in which a subset of quality attributes is addressed and potentially improved [CM22].

To implement this concept, we proposed a quality model in Section 7.5 that integrates with the three phases of our microservices migration framework. An initial architecture evaluation (Section 2.4.1) justifies a migration decision based on predefined quality goals and tangible measures. These predefined quality goals guide the decomposition during the second phase, since quality expectations strongly influence the partitioning [LZJ+21]. By introducing the role of quality attributes in a decomposition (Section 2.4.2), we motivated the classification of 110 refactoring approaches (Chapter 5) according to their associated quality goals. In Table 5.4, we have shown that 74 approaches state to address one or more quality attributes, while 51 approaches state to address one or more sub-attributes. For design properties, 71 approaches stated to address at least one of them.

Subsequently, the predefined quality goals guide architects in designing a microservices architecture by recommending proven patterns and best prac-

tices. By reviewing secondary studies and a public repository (Section 2.5), we sought evidence on the qualities, that patterns and best practices for microservices address. Our review of seven sources yielded 51 patterns (Table A.1), for which we discussed their impact on quality attributes and sub-attributes, while also pointing out potential trade-offs. We similarly collected evidence on 43 best practices (Table A.2). While commonly originating from industry practice, they capture knowledge similarly to patterns, but often in a less structured manner.

The designed quality model has been implemented in our web-based application (Chapter 8), and was in this context subject to several evaluations among software professionals (Chapter 9). Eventually, its practical relevance was examined during two longitudinal case studies in industry (Chapter 10).

**RQ4: How can a practically applicable migration methodology guide architects?** The answer to our fourth research question incorporates the foundations that contributed to the questions RQ1 to RQ3, and as such addresses our overarching research objective. Our state of practice analysis in Chapters 3 and 4 yielded a multi-faceted picture of the situation practitioners are confronted with. The decomposition into services was identified as a major technical challenge during a migration to microservices. Furthermore, we discovered a need for systematic guidance and ways to partly automate this process. Eventually, we found that practitioners rarely leverage existing academia-proposed solutions or tools to this end. This manifestation of a research gap encouraged us to collect requirements for a migration methodology concept (Section 4.4.4) that aims to address these shortcomings. By incorporating our extensive state of research analysis in Chapters 5 and 6, we developed a framework for migrations to microservices in Chapter 7. Complemented by a dedicated quality model, we proposed a holistic migration methodology that implements the concept of a quality-driven migration to microservices.

To provide actionable guidance for architects and developers, we built a web-based application in an agile development process with early involvement of potential users (Chapter 8). At its core, it accommodates an extensi-

ble repository of refactoring and migration approaches, complemented by microservices design patterns and best practices. A scenario-based workflow realizes our concept of a quality-driven migration: In a first phase, the application captures results of an architecture evaluation activity. The therein elicited scenarios and quality goals then guide the decomposition and architecture design during the subsequent phases. Starting with a prototypical implementation, we refined the initial prototype into a microservices-based web application that uses latest technologies and modern user interface concepts. Our iterative development process adhered to established software engineering standards, including requirements engineering, model design, testing, and continuous evaluation.

During multifaceted evaluations, we collected empirical evidence on the practicability, soundness, and usability of our methodology and its implementation. In a survey with nine participants, we evaluated the vertical prototype, followed a by a survey with 11 participants that targeted the web-based application in its first iteration. The implemented quality-driven migration process was examined by six practitioners in the second and third iteration by a mix of surveys and interviews. Eventually, we conducted two longitudinal case studies to evaluate our methodology in an industrial context. In the first case study on the migration of a monolithic application in the *Services* domain, our methodology guided the creation of a migration concept that was subsequently implemented in a proof-of-concept. In the second case study on a document management solution, we evaluated our methodology with respect to improving an existing microservices-based architecture for specific quality aspects.

By answering the above research questions, we proposed

*A practically applicable methodology*
*For transforming monolithic applications into microservices*

and thereby regard our initially stated research objective as achieved.

## 11.2. Limitations and Future Research

In addition to the limitations already mentioned for the individual contributions in this thesis, we want to critically reflect on our research process and the developed methodology as a whole. As many of these limitations could be addressed by future research, we discuss such possibilities alongside.

With respect to our **state of practice analysis** contributions, the sample size of the interviews compromises generalizability. In particular, the industry case study trades external validity for richness and depth of results. Nevertheless, we are confident that we have identified the critical points in a microservices migration, supported by ample evidence from other empirical studies that confirm our identified research gap.

The refactoring approaches reviewed in our **state of research analysis** have been examined only theoretically, based on the authors' assertions and provided supplementary material. Thus, our classification is based on documented properties and evaluations, which may be biased by our interpretation of the concepts presented. This is particularly true for the quality and design attributes that such approaches seek to address or optimize. While the extent of such optimization can only be reliably assessed through exemplary application, the use of our classification must take this into account. Here, we see potential for future research that could comparatively evaluate such approaches based on defined measures, ideally involving real-world applications. Such an objective examination may also reveal drawbacks and trade-offs that the authors may have been less explicit about.

Complementarily, we see the evaluation of accompanying decomposition tools as an important next step, i.e., assessing the plausibility and soundness of the service cuts they generate. This would require the selection of appropriate reference applications, the preparation of input data, and defined measures for their comparison — which our usability-focused tool evaluation did not cover. Our review of scientific literature has revealed a number of such automation solutions that are in high demand by practitioners. In general, we believe that future research aimed at making such tools more accessible and usable would be beneficial. For microservices as a field rooted

in practice [DML17], grey literature may provide valuable additional solutions that could be incorporated. Besides, future research could empirically validate our proposed taxonomy for refactoring approaches. Ensuring its consistency and comprehensiveness would facilitate its adoption potential in academia and practice.

Furthermore, our in Section 7.5 proposed **quality model** has not been evaluated separately. Although indirectly evaluated as part of our migration methodology, the soundness and accuracy of our computed suitability measures are not supported by empirical evidence. In addition, the model does not yet incorporate potential quality trade-offs. Future research could continue here, ideally in conjunction with a validation and unification of the qualities targeted by refactoring approaches, as mentioned above. Besides, empirical evidence on how specific patterns and best practices influence software quality is limited to only a few secondary studies. Further research could provide more insights into the quality-driven design of microservices architectures by expanding the knowledge of existing patterns and their trade-offs, as we also detailed in our related work [VCG+22].

Eventually, our web-based application realizing the conceptualized **framework and methodology**, does not represent a final product. Further industry case studies across different domains could substantially improve its maturity. Our evaluations throughout the Chapters 9 and 10 already resulted in great consensus, but have also pointed out the limitations of such an expert system. While practitioners seek guidance and automation capabilities to reduce complexity, new tools always require an initial learning curve to their adoption. Setting this barrier as low as possible should be a main research direction to improve our developed application. This could be achieved, for example, by providing short introductory videos or tutorial overlays for new users, as suggested in our evaluations. The complexity of specifying filter criteria based on our taxonomy (Table 5.5) has already been reduced by additional preset filters, but could be further simplified using artificial intelligence techniques. We are currently investigating how a large language model for refactoring approaches can be built based on the knowledge in scientific publications. Our long-term goal is to enable interactions via natural

language input, that generate targeted and more precise recommendations in a human-like conversational dialog. In combination with the current trends discussed in Section 10.6, entirely new possibilities may arise from the use of artificial intelligence techniques in a refactoring context.

We further see potential for publicly hosting the developed application and extending it with functionality to incorporate user feedback, e.g., in the form of a rating and commenting system. In this way, the results of our work could be directly applied in practice and address its core motivation of facilitating knowledge transfer between academia and practice (Section 1.3). Researchers developing industry-focused methods can also benefit from our insights and experiences in a more general sense, beyond a more efficient transformation of monolithic legacy systems towards microservices.

## 11.3. Final Remarks

In this section, we conclude with final remarks, expressing perceptions on the relevance and impact of our work.

The publications contributed to the scientific community throughout this research project (Section 1.7) marked a significant gain to the body of knowledge in the field. A meta-analysis of 837 publications on secure microservices saw our publication on classifying refactoring approaches [FBZW19] leading the third most representative cluster [FPA+21]. In addition, our empirical studies of industry practice [BFWZ19b; FBWZ19] have been well received by the research community. In the sense of academic research that benefits the public, we promote transparency, traceability, and replicability of our research. As such, we share important study artifacts and supplemental material when appropriate, as is the case with the work presented herein. Beyond contributing to the field of software engineering, we believe that our design science approach to leveraging academic research by practitioners is transferable to other scientific disciplines too.

Still, our initial goal of bridging the gap between academia and practice is an ambitious one, and we can only advance it in small steps through our

work. While large enterprises often have the necessary resources to develop their own individual modernization strategies, it is small and midsize businesses that would benefit the most from our research. They often lack such possibilities and experienced personnel to systematically plan and conduct the transformation of their legacy systems to a new architecture and platform. As a result, many of them outsource such activities as much as possible or rely on the offerings of cloud service providers. Both alternatives have drawbacks that usually manifest only in the long term: While outsourcing reduces flexibility and agility due to a lack of in-house expertise, leveraging a variety of third-party offerings also carries the risk of a vendor lock-in. Compliance and regulatory constraints may require additional precautionary measures on security and privacy that could render both alternatives infeasible.

While grey literature is dominated by success stories of microservices adoption, it can be assumed that many attempts also reached an impasse, as such experiences are rarely published [Lit18; MBMR21]. Su et al. investigated the phenomenon of returning to a monolithic architecture, which can have several reasons [SLT23]. It could be a sign that microservices as an emerging technological trend has reached its *trough of disillusionment*. In this broader sense, we hope that our contributions support practitioners in making informed decisions on microservices adoption, and furthermore guide them through a systematic transformation process.

# Bibliography


[AA23] A.-S. Ayisha, Z. Al-Khanjari. “Achieving Loose Coupling and High Quality through Migrating the Monolithic Applications to Microservices Applications”. In: *International Journal of Advanced Natural Sciences and Engineering Researches* 7.2 (2023), pp. 5–10 (cit. on pp. 62, 198).

[AAC+17] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, P. Avgeriou. “A mapping study on design-time quality attributes and metrics”. In: *Journal of Systems and Software* 127 (2017), pp. 52–77 (cit. on pp. 57, 58, 62).

[AAE16] N. Alshuqayran, N. Ali, R. Evans. “A systematic mapping study in microservice architecture”. In: *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA*. 2016, pp. 44–51 (cit. on pp. 63, 172).

[ACC+22a] W. K. G. Assunção, T. E. Colanzi, L. Carvalho, A. Garcia, J. A. Pereira, M. J. de Lima, C. Lucena. “Analysis of a many-objective optimization approach for identifying microservices from legacy systems”. In: *Empirical Software Engineering* 27.2 (Feb. 2022) (cit. on p. 70).

[ACC+22b] W. K. G. Assunção, T. E. Colanzi, L. Carvalho, A. Garcia, J. A. Pereira, M. J. de Lima, C. Lucena. “Analysis of a many-objective optimization approach for identifying microservices from legacy systems”. In: *Empirical Software Engineering* 27.2 (Feb. 2022) (cit. on pp. 301, 322, 323).

[ADY19] A. Andriyanto, R. Doss, P. Yustianto. “Adopting SOA and Microservices for Inter-enterprise Architecture in SME Communities”. In: *2019*

*International Conference on Electrical, Electronics and Information Engineering (ICEEIE)*. IEEE, Oct. 2019, pp. 282–287 (cit. on p. 25).

[AEA23] M. AIT Said, A. Ezzati, S. Arezki. "Microservices, a Step from the Low-Code to the No-Code". In: *Proceedings of the 8th International Conference on Advanced Intelligent Systems and Informatics 2022*. Ed. by A. E. Hassanien, V. Snášel, M. Tang, T.-W. Sung, K.-C. Chang. Cham: Springer International Publishing, 2023, pp. 779–788 (cit. on pp. 335, 336).

[AFS12] A. Ampatzoglou, G. Frantzeskou, I. Stamelos. "A methodology to assess the impact of design patterns on software quality". In: *Information and Software Technology* 54.4 (2012), pp. 331–346 (cit. on p. 71).

[AGCC21] I. Alfonso, K. Garcés, H. Castro, J. Cabot. "Self-adaptive Architectures in IoT Systems: A Systematic Literature Review". In: (2021). arXiv: 2109.03312 (cit. on p. 109).

[AI16] M. Ahmadvand, A. Ibrahim. "Requirements reconciliation for scalable and secure microservice (de)composition". In: *Proceedings - 2016 IEEE 24th International Requirements Engineering Conference Workshops, REW*. 2016, pp. 68–73 (cit. on pp. 64, 70).

[AIE19] M. Abdullah, W. Iqbal, A. Erradi. "Unsupervised learning approach for web application auto-decomposition into microservices". In: *Journal of Systems and Software* 151 (May 2019), pp. 243–257 (cit. on pp. 172, 301).

[ALFT21] F. Auer, V. Lenarduzzi, M. Felderer, D. Taibi. "From monolithic systems to Microservices: An assessment framework". In: *Information and Software Technology* 137 (Sept. 2021), p. 106600 (cit. on pp. 44, 64, 67, 201, 297, 299).

[ALH23] H. M. Ayas, P. Leitner, R. Hebig. "An empirical study of the systemic and technical migration towards microservices". In: *Empirical Software Engineering* 28.4 (May 2023) (cit. on pp. 198, 199).

[AM14] R. Alroobaea, P. J. Mayhew. "How many participants are really enough for usability studies?" In: *2014 Science and Information Conference*. 2014, pp. 48–56 (cit. on p. 275).

[AM21] O. Al-Debagy, P. Martinek. “A Microservice Decomposition Method Through Using Distributed Representation of Source Code”. In: *Scalable Computing: Practice and Experience* 22.1 (Feb. 2021), pp. 39–52 (cit. on pp. 302, 303).

[AME+23] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, P. Clarke. “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review”. In: *IEEE Transactions on Software Engineering* (2023), pp. 1–32 (cit. on pp. 155, 158, 165, 166).

[Ami18] M. J. Amiri. “Object-Aware Identification of Microservices”. In: *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, July 2018, pp. 253–256 (cit. on p. 159).

[AMNO21] E. A. AlOmar, M. W. Mkaouer, C. Newman, A. Ouni. “On Preserving the Behavior in Software Refactoring: A Systematic Mapping Study”. In: *CoRR* (2021). eprint: 2106.13900 (cit. on p. 25).

[Ang23] Angular. *Angular Material*. online. URL: https://material.angular.io/. 2023 (cit. on p. 232).

[AO22] K. Akkaya, T. Ovatman. “A Comparative Study of Meta-Data-Based Microservice Extraction Tools”. In: *International Journal of Service Science, Management, Engineering, and Technology* 13.1 (June 2022), pp. 1–26 (cit. on p. 303).

[ASM+21] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, Y.-G. Guéhéneuc. “A taxonomy of service identification approaches for legacy software systems modernization”. In: *Journal of Systems and Software* 173 (Mar. 2021) (cit. on pp. 25, 52–55, 154, 155, 158–160, 163–166, 168, 169, 175, 178, 180, 202).

[ASS22] B. Andrade, S. Santos, A. R. Silva. “From Monolith to Microservices: Static and Dynamic Analysis Comparison”. pp. 2022. arXiv: 2204.11844 (cit. on p. 51).

[AVM+24] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. D. Newman, A. Ouni. *How to Refactor this Code? An Exploratory Study on Developer-ChatGPT Refactoring Conversations*. 2024 (cit. on p. 336).

[AWS18] R. C. Amantha Hutapea, A. P. Wahyudi, Suhardi. “Design Quality Measurement for Service Oriented Software on Service Computing System: a Systematic Literature Review”. In: *2018 International Conference on Information Technology Systems and Innovation (ICITSI)*. 2018, pp. 375–380 (cit. on pp. 57, 62).

[BBB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham. *Manifesto for Agile Software Development*. 2001 (cit. on p. 102).

[BBD+22] P. Billawa, A. Bambhore Tukaram, N. E. Díaz Ferreyra, J.-P. Steghöfer, R. Scandariato, G. Simhandl. “SoK: Security of Microservice Applications: A Practitioners’ Perspective on Challenges and Best Practices”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ARES ’22. Vienna, Austria: Association for Computing Machinery, 2022 (cit. on p. 71).

[BBG+17] R. Barcia, K. Brown, C. Gang, S. Daya, M. Martins, R. Osowski, J. Verwaayen. “Microservices Decision Guides”. 2017 (cit. on p. 44).

[BBGG21] D. Bajaj, U. Bharti, A. Goel, S. C. Gupta. “A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts”. In: *Journal of Web Engineering* (June 2021) (cit. on pp. 25, 47, 48, 52, 180, 201, 238).

[BCK21] L. Bass, P. Clements, R. Kazman. *Software architecture in practice*. 4th ed. SEI Series in Software Engineering. Boston, MA: Addison Wesley, 2021 (cit. on pp. 37, 38, 58, 70, 71).

[BCS21] M. Brito, J. Cunha, J. Saraiva. “Identification of Microservices from Monolithic Applications through Topic Modelling”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1409–1418 (cit. on p. 187).

[BCW17] M. Brambilla, J. Cabot, M. Wimmer. *Model-Driven Software Engineering in Practice*. Springer International Publishing, 2017 (cit. on pp. 50, 51).

[BD85] R. N. Burns, A. R. Dennis. “Selecting the appropriate application development methodology”. In: *SIGMIS Database* 17.1 (Sept. 1985), pp. 19–23 (cit. on p. 219).

[BDD+18] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, M. Mazzara. "From Monolithic to Microservices: An Experience Report from the Banking Domain". In: *IEEE Software*. Vol. 35. 3. IEEE, 2018, pp. 50–55 (cit. on p. 83).

[Bev95] N. Bevan. "Measuring usability as quality of use". In: *Software Quality Journal* 4.2 (1995), pp. 115–130 (cit. on pp. 56, 57).

[BFWZ18] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann. "Limiting technical debt with maintainability assurance". In: *Proceedings of the 2018 International Conference on Technical Debt - TechDebt '18*. New York, New York, USA: ACM Press, 2018, pp. 125–133 (cit. on p. 34).

[BFWZ19a] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann. "Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 546–556 (cit. on pp. 34, 82).

[BFWZ19b] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann. "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2019, pp. 187–195 (cit. on pp. 34, 58, 82, 135, 139, 142, 338, 346).

[BFWZ21] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann. "Industry practices and challenges for the evolvability assurance of microservices". In: *Empirical Software Engineering* 26.5 (Sept. 2021), p. 104 (cit. on pp. 34, 135).

[BGLL08] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (2008), P10008 (cit. on p. 187).

[BGM87] I. Benbasat, D. K. Goldstein, M. Mead. "The Case Research Strategy in Studies of Information Systems". In: *MIS Quarterly* 11.3 (1987), pp. 369–386 (cit. on p. 84).

[BGR+17] L. Baresi, M. Garriga, A. D. Renzis, D. Elettronica, I. Bioingegneria. “Microservices Identification Through Interface Analysis”. In: *ESOCC 2017: Service-Oriented and Cloud Computing*. 2017, pp. 19–33 (cit. on pp. 160, 185).

[BHJ+18] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, T. Lynn. “Microservices migration patterns”. In: *Software: Practice and Experience* 48.11 (July 2018), pp. 2019–2042 (cit. on pp. 49, 83).

[BKB+07] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, M. Khalil. “Lessons from applying the systematic literature review process within the software engineering domain”. In: *Journal of Systems and Software* 80.4 (2007), pp. 571–583 (cit. on pp. 29, 150).

[BLBV04] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet. “Architecture-level modifiability analysis (ALMA)”. In: *Journal of Systems and Software* 69.1 (2004), pp. 129–147 (cit. on p. 68).

[BLR21] K. Bozan, K. Lyytinen, G. M. Rose. “How to transition incrementally to microservice architecture”. In: *Communications of the ACM* 64.1 (Jan. 2021), pp. 79–85 (cit. on pp. 197, 199).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-oriented software architecture*. en. Wiley Software Patterns Series. Nashville, TN: John Wiley & Sons, July 1996 (cit. on p. 71).

[BNJ03] D. M. Blei, A. Y. Ng, M. I. Jordan. “Latent dirichlet allocation”. In: *The Journal of Machine Learning Research* 3.null (Mar. 2003), pp. 993–1022 (cit. on p. 187).

[BNK18] S. Baškarada, V. Nguyen, A. Koronios. “Architecting Microservices: Practical Opportunities and Challenges”. In: *Journal of Computer Information Systems* (2018) (cit. on pp. 24, 25, 139).

[BNZ+15] M. Bozorgi, R. Nayak, A. Zaffar, M. I. Hoque, S. A. Ghouri, H. Singh, P. S. Kalshan. “A Case Study on Quality Attribute Measurement using MARF and GIPSY”. In: *CoRR* abs/1505.0 (2015) (cit. on pp. 57, 58).

[Bog20] J. Bogner. *On the evolvability assurance of microservices: metrics, scenarios, and patterns*. online. URL: http://elib.uni-stuttgart.de/handle/11682/10967. 2020 (cit. on pp. 22, 39, 48, 166).

[BOP22] G. Blinowski, A. Ojdowska, A. Przybylek. “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation”. In: *IEEE Access* 10 (2022), pp. 20357–20374 (cit. on pp. 40, 44, 45).

[Bos99] J. Bosch. “Design and Use of Industrial Software Architectures”. In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*. TOOLS29-99. IEEE Comput. Soc, 1999 (cit. on p. 67).

[BPRS18] L. Banica, P. Polychronidou, M. Radulescu, C. Stefan. “When IoT Meets DevOps: Fostering Business Opportunities”. In: *Economies of the Balkan and the Eastern European Countries in the Changing World (Ebeec 2018)* 2018 (2018), pp. 250–264 (cit. on p. 109).

[BS14] M. Broy, A. Schmidt. “Challenges in Engineering Cyber-Physical Systems”. In: *IEEE Computer* 47.2 (2014), pp. 70–72 (cit. on pp. 108, 109).

[BVS13] R. Buyya, C. Vecchiola, S. T. Selvi. “Mastering Cloud Computing”. In: *Mastering Cloud Computing*. Ed. by R. Buyya, C. Vecchiola, S. T. Selvi. Boston: Morgan Kaufmann, 2013, pp. 3–27 (cit. on p. 22).

[BWZ17a] J. Bogner, S. Wagner, A. Zimmermann. “Automatically Measuring the Maintainability of Service- and Microservice-based Systems - a Literature Review”. In: *2017 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, IWSM-MENSURA* (2017) (cit. on pp. 62, 63).

[BWZ17b] J. Bogner, S. Wagner, A. Zimmermann. “Towards a practical maintainability quality model for service-and microservice-based systems”. In: *Proceedings of the 11th European Conference on Software Architecture Companion Proceedings - ECSA ’17* 3 (2017), pp. 195–198 (cit. on pp. 61, 62).

[Cam18] G. Campeanu. “A mapping study on microservice architectures of Internet of Things and cloud computing solutions”. In: 2018 7th Mediterranean Conference on Embedded Computing (MECO) June. IEEE, June 2018, pp. 1–4 (cit. on p. 109).

[Cap21] R. Capuano. “Enhancing System Quality Attributes via Microservices Adoption”. In: *15th European Conference on Software Architecture ECSA'21: Doctoral Symposium*. 2021, pp. 0–4 (cit. on pp. 48, 65).

[CCRZ23] M. Camilli, C. Colarusso, B. Russo, E. Zimeo. “Actor-Driven Decomposition of Microservices through Multi-Level Scalability Assessment”. In: *ACM Transactions on Software Engineering and Methodology* 32.5 (July 2023) (cit. on pp. 70, 166).

[CD11] D. S. Cruzes, T. Dyba. “Reded Steps for Thematic Synthesis in Software Engineering”. In: 2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, Sept. 2011, pp. 275–284 (cit. on p. 114).

[CGC+20a] L. Carvalho, A. Garcia, T. E. Colanzi, W. K. G. Assuncao, J. A. Pereira, B. Fonseca, M. Ribeiro, M. J. de Lima, C. Lucena. “On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Vol. 1. IEEE, Sept. 2020, pp. 569–580 (cit. on pp. 301–303).

[CGC+20b] L. Carvalho, A. Garcia, T. E. Colanzi, W. K. G. Assunção, M. J. Lima, B. Fonseca, M. Ribeiro, C. Lucena. “Search-Based Many-Criteria Identification of Microservices from Legacy Systems”. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 305–306 (cit. on p. 301).

[CGC15] K. Conboy, R. Gleasure, E. Cullina. “Agile Design Science Research”. In: *New Horizons in Design Science: Broadening the Research Agenda*. Ed. by B. Donnellan, M. Helfert, J. Kenneally, D. VanderMeer, M. Rothenberger, R. Winter. Cham: Springer International Publishing, 2015, pp. 168–180 (cit. on p. 196).

[Che18] L. Chen. “Microservices: Architecting for Continuous Delivery and DevOps”. In: *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*. IEEE, 2018, pp. 39–46 (cit. on p. 44).

[Chr19] B. Christudas. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress, 2019 (cit. on p. 246).

[CIO22] CIO, CSO, COMPUTERWOCHE. *Studie Legacy-Modernisierung 2022*. online. URL: https://web.archive.org/web/20220714101629/https://shop.computerwoche.de/portal/studie-legacy-modernisierung-2022-pdf-download-direkt-im-shop-10440. 2022 (cit. on p. 23).

[CLL17] R. Chen, S. Li, Z. Li. "From Monolith to Microservices: A Dataflow-Driven Approach". In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 2017-Decem. IEEE, Dec. 2017, pp. 466–475 (cit. on p. 185).

[CM22] R. Capuano, H. Muccini. "A Systematic Literature Review on Migration to Microservices: a Quality Attributes perspective". In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2022, pp. 120–123 (cit. on pp. 57, 58, 65, 66, 70, 198, 341).

[CMR+20] M. Ciavotta, G. D. Maso, D. Rovere, R. Tsvetanov, S. Menato. "Towards the Digital Factory: A Microservices-Based Middleware for Real-to-Digital Synchronization". In: *Microservices*. Cham: Springer International Publishing, 2020 (cit. on p. 108).

[Coc00] A. Cockburn. *Writing Effective Use Cases*. en. The Crystal Collection for Software Professionals. Boston, MA: Addison Wesley, Oct. 2000 (cit. on p. 219).

[Coh04] M. Cohn. *User stories applied*. en. The Addison-Wesley signature series. Boston, MA: Addison-Wesley Educational, Mar. 2004 (cit. on p. 67).

[Coh09] M. Cohn. *Succeeding with agile*. en. The Addison-Wesley signature series. Boston, MA: Addison-Wesley Educational, Oct. 2009 (cit. on p. 220).

[Con18] M. Conway. *Conway's Law*. online. URL: http://melconway.com/Home/Conways_Law.html. 2018 (cit. on pp. 43, 202).

[COU19] M. D. Cojocaru, A. Oprescu, A. Uta. “Attributes assessing the quality of microservices automatically decomposed from monolithic applications”. In: *Proceedings - 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC 2019* 1 (2019), pp. 84–93 (cit. on pp. 25, 57, 61–63, 67).

[CPS20] B. Cartaxo, G. Pinto, S. Soares. “Rapid Reviews in Software Engineering”. In: *Contemporary Empirical Methods in Software Engineering*. Springer, 2020 (cit. on pp. 29, 111, 181).

[CR22] J. Correia, A. Rito Silva. “Identification of monolith functionality refactorings for microservices migration”. In: *Software: Practice and Experience* July (Aug. 2022), pp. 1–20 (cit. on p. 183).

[Cro20] M. Crotty. *The foundations of social research: Meaning and perspective in the research process*. Routledge, Aug. 2020 (cit. on p. 293).

[CW20] B. Combemale, M. Wimmer. “Towards a Model-Based DevOps for Cyber-Physical Systems”. In: *Lecture Notes in Computer Science*. Vol. 12055. Springer International Publishing, 2020 (cit. on p. 108).

[DB20] M. Daghaghzadeh, S. M. Babamir. “A model driven and clustering method for service identification directed by metrics”. In: *Software: Practice and Experience* September (Oct. 2020), spe.2913 (cit. on pp. 62, 159).

[DBFP18] A. A. C. De Alwis, A. Barros, C. Fidge, A. Polyvyanyy. “Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11230 LNCS. Springer International Publishing, 2018, pp. 60–79 (cit. on pp. 302, 303).

[DBFP19] A. A. C. De Alwis, A. Barros, C. Fidge, A. Polyvyanyy. “Availability and Scalability Optimized Microservice Discovery from Enterprise Systems”. In: *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*. Ed. by H. Panetto, C. Debruyne, M. Hepp, D. Lewis, C. A. Ardagna, R. Meersman. Cham: Springer International Publishing, 2019, pp. 496–514 (cit. on pp. 42, 160).

[DBFP21] A. A. C. De Alwis, A. Barros, C. Fidge, A. Polyvyanyy. *Microservice Remodularisation of Monolithic Enterprise Systems for Embedding in Industrial IoT Networks*. Vol. 12751 LNCS. Springer International Publishing, 2021, pp. 432–448 (cit. on pp. 301, 322, 323).

[DBT21] U. Desai, S. Bandyopadhyay, S. Tamilselvam. "Graph Neural Network to Dilute Outliers for Refactoring Monolith Application". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.1 (2021), pp. 72–80 (cit. on pp. 302, 303).

[DEF+21] M. Daoud, A. El Mezouari, N. Faci, D. Benslimane, Z. Maamar, A. El Fazziki. "A multi-model based microservices identification approach". In: *Journal of Systems Architecture* 118 (Sept. 2021), p. 102200 (cit. on pp. 322–324, 326, 335).

[Dha21] H. K. Dhalla. "A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core". In: *Journal of Physics: Conference Series* 1933.1 (2021), p. 12041 (cit. on p. 228).

[DKJ05] T. Dyba, B. A. Kitchenham, M. Jorgensen. "Evidence-based software engineering for practitioners". In: *IEEE Software* 22.1 (2005), pp. 58–65 (cit. on p. 27).

[DKTT22] M. Dehghani, S. Kolahdouz-Rahimi, M. Tisi, D. Tamzalit. "Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning". In: *Software and Systems Modeling* (Feb. 2022) (cit. on pp. 51, 170, 301).

[DLM18] P. Di Francesco, P. Lago, I. Malavolta. "Migrating Towards Microservice Architectures: An Industrial Survey". In: *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*. 2018, pp. 29–38 (cit. on pp. 83, 104, 197, 339).

[DLM19] P. Di Francesco, P. Lago, I. Malavolta. "Architecting with microservices: A systematic mapping study". In: *Journal of Systems and Software* 150.April (2019), pp. 77–97 (cit. on pp. 26, 82).

[DMH+24] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, E. A. AlOmar. "Exploring ChatGPT's code refactoring capabilities: An empirical study". In: *Expert Systems with Applications* 249 (2024), p. 123602 (cit. on p. 336).

[DML17] P. Di Francesco, I. Malavolta, P. Lago. “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption”. In: *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*. 2017, pp. 21–30 (cit. on pp. 60, 64, 190, 194, 345).

[Don09] M. G. Donoff. “Field notes: assisting achievement and documenting competence”. In: *Canadian Family Physician* 55.12 (2009), pp. 1260–1262 (cit. on p. 296).

[ECA+16] D. Escobar, D. Cardenas, R. Amarillo, E. Castro, K. Garces, C. Parra, R. Casallas. “Towards the understanding and evolution of monolithic applications as microservices”. In: *2016 XLII Latin American Computing Conference (CLEI)*. IEEE, Oct. 2016, pp. 1–11 (cit. on p. 61).

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. USA: Prentice Hall PTR, 2005 (cit. on p. 39).

[ES22] E. Ersoy, H. Sozer. “Effort Estimation for Architectural Refactoring of Data Tier Software”. In: *Proceedings - IEEE 19th International Conference on Software Architecture, ICSA 2022* (2022), pp. 80–89 (cit. on p. 49).

[ESSD08] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian. “Selecting Empirical Methods for Software Engineering Research”. In: *Guide to Advanced Empirical Software Engineering*. Ed. by F. Shull, J. Singer, D. I. K. Sjøberg. London: Springer London, 2008, pp. 285–311 (cit. on pp. 27–29).

[Eva04] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004 (cit. on pp. 43, 49, 166).

[FAB+21] M. Fahmideh, A. A. Abbasi, A. Behnaz, J. Grundy, W. Susilo. “Software Engineering for Internet of Things”. In: *IEEE Transactions on Software Engineering* (2021) (cit. on p. 109).

[FBB+14] M. Falkenthal, J. Barzen, U. Breitenb"ucher, C. Fehling, F. Leymann. “From Pattern Languages to Solution Implementations”. In: *Proceedings of the 6\textsuperscript{th} International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2014, pp. 12–21 (cit. on p. 71).

[FBBO99] M. Fowler, K. Beck, J. Brant, W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999 (cit. on pp. 48, 58).

[FBH+22] J. Fritzsch, J. Bogner, M. Haug, A. C. Franco da Silva, C. Rubner, M. Saft, H. Sauer, S. Wagner. "Adopting microservices and DevOps in the cyber-physical systems domain: A rapid review and case study". In: *Software: Practice and Experience* (Nov. 2022), pp. 1–20 (cit. on pp. 26, 33, 108, 112, 115, 130).

[FBH+24] J. Fritzsch, J. Bogner, M. Haug, S. Wagner, A. Zimmermann. "Towards an Architecture-Centric Methodology for Migrating to Microservices". In: *Agile Processes in Software Engineering and Extreme Programming – Workshops*. Ed. by P. Kruchten, P. Gregory. Cham: Springer Nature Switzerland, 2024, pp. 39–47 (cit. on pp. 33, 116, 194).

[FBWZ19] J. Fritzsch, J. Bogner, S. Wagner, A. Zimmermann. "Microservices Migration in Industry: Intentions, Strategies, and Challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2019, pp. 481–490 (cit. on pp. 24–26, 33, 82, 100, 128, 135, 139, 163, 346).

[FBZW19] J. Fritzsch, J. Bogner, A. Zimmermann, S. Wagner. "From Monolith to Microservices: A Classification of Refactoring Approaches". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. DEVOPS 2018*. Ed. by J.-M. Bruel, M. Mazzara, B. Meyer. Cham: Springer International Publishing, 2019, pp. 128–141 (cit. on pp. 25, 33, 148, 153, 180, 346).

[FCBW23] J. Fritzsch, F. Correia, J. Bogner, S. Wagner. "Tools for Refactoring to Microservices: A Preliminary Usability Report". In: *5th International Conference on Microservices (Microservices 2023), Pisa, Italy*. 2023 (cit. on pp. 33, 180, 303).

[FFC21] F. Freitas, A. Ferreira, J. Cunha. "Refactoring Java Monoliths into Executable Microservice-Based Applications". In: *Proceedings of the 25th Brazilian Symposium on Programming Languages*. SBLP '21. Joinville, Brazil: Association for Computing Machinery, 2021, pp. 100–107 (cit. on p. 322).

[FFZ+18] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, A. Barros. “Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency”. In: *IEEE Software* 35.3 (2018), pp. 63–72 (cit. on p. 96).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Vienna, 2014 (cit. on p. 43).

[Fow04a] M. Fowler. *Definition Of Refactoring*. online. URL: https://martinfowler.com/bliki/DefinitionOfRefactoring.html. 2004 (cit. on p. 48).

[Fow04b] M. Fowler. *StranglerApplication*. online. URL: https://www.martinfowler.com/bliki/StranglerApplication.html. 2004 (cit. on pp. 48, 95, 99, 128, 163, 201, 307).

[Fow15] M. Fowler. *Microservice Trade-Offs*. online. URL: https://martinfowler.com/articles/microservice-trade-offs.html. 2015 (cit. on p. 43).

[FPA+21] P. Ferreira, W. D. Prá, D. N. Araújo, G. d. O. Alves, L. M. C. e. Martins, W. F. Giozza, R. T. de Sousa Jr. “Secure Microservices - A Review Applying the Theory of the Consolidated Meta-analytic Approach”. In: *IADIS International Journal on WWW/Internet* 18.2 (2021), pp. 57–75 (cit. on p. 346).

[FSC+21] A. F. A. A. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros, N. C. Mendonça. “Migrating production monolithic systems to microservices using aspect oriented programming”. In: *Software: Practice and Experience* 51.6 (Mar. 2021), pp. 1280–1307 (cit. on p. 70).

[FST23a] R. Frank, G. Schumacher, A. Tamm. “Mastering Software as a Core Competence”. In: *Cloud Transformation : The Public Cloud Is Changing Businesses*. Wiesbaden: Springer Fachmedien Wiesbaden, 2023, pp. 133–168 (cit. on p. 45).

[FST23b] R. Frank, G. Schumacher, A. Tamm. “The Cloud Transformation”. In: *Cloud Transformation : The Public Cloud Is Changing Businesses*. Wiesbaden: Springer Fachmedien Wiesbaden, 2023, pp. 203–245 (cit. on pp. 45, 47).

[FWBW21] J. Fritzsch, M. Wyrich, J. Bogner, S. Wagner. “Résumé-Driven Development: A Definition and Empirical Characterization”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. IEEE, 2021 (cit. on p. 40).

[Gar20] Gartner. *Gartner 2020 Hype Cycle for Supply Chain*. online. URL: https://www.gartner.com/en/newsroom/press-releases/2020-09-09-gartner-2020-hype-cycle-for-supply-chain-strategy-shows-internet-of-things-is-two-to-five-years-away-from-transformational-impact. 2020 (cit. on p. 108).

[Gar84] D. A. Garvin. “What Does ’Product Quality’ Really Mean?” In: *MIT Sloan Management Review* 26:1 (1984), pp. 25–43 (cit. on p. 57).

[GB19] J. Ghofrani, A. Bozorgmehr. “Migration to Microservices: Barriers and Solutions”. In: *Applied Informatics*. Ed. by H. Florez, M. Leon, J. M. Diaz-Nafria, S. Belli. Vol. 1051. Communications in Computer and Information Science. Cham: Springer International Publishing, 2019, pp. 269–281 (cit. on p. 166).

[GCL+20] M. Gao, M. Chen, A. Liu, W. H. Ip, K. L. Yung. “Optimization of Microservice Composition Based on Artificial Immune Algorithm Considering Fuzziness and User Preference”. In: *IEEE Access* 8 (2020), pp. 26385–26404 (cit. on p. 301).

[GCOB21] B. Gopal, S. Cooper, J. Olmanson, R. Bockmon. “Student difficulties in Unit Testing, Integration Testing and Continuous Integration: An exploratory pilot qualitative study”. In: *Annual Workshop of the Psychology of Programming Interest Group*. 2021 (cit. on p. 220).

[GKGZ16] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. “Service Cutter: A Systematic Approach to Service Decomposition”. In: *Lecture Notes in Computer Science*. Vol. 9846. 2016, pp. 185–200 (cit. on pp. 158, 161, 184, 185, 340).

[GL18] J. Ghofrani, D. Lübke. “Challenges of Microservices Architecture: A Survey on the State of the Practice”. In: *10th Central European Workshop on Services and their Composition (ZEUS)* 10th.May (2018) (cit. on p. 83).

[Gle18] A. Glen. *[DZone Research] Microservices Priorities and Trends*. online. URL: https://dzone.com/articles/dzone-research-microservices-priorities-and-trends. 2018 (cit. on p. 65).

[GM20] M. H. Gomes Barbosa, P. H. M. Maia. "Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures". In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2020, pp. 41–48 (cit. on p. 70).

[Goo23] Google. *Material Design*. online. URL: https://m3.material.io/. 2023 (cit. on p. 231).

[GP09] A. C. Gemino, D. Parker. "Use Case Diagrams in Support of Use Case Modeling: Deriving Understanding from the Picture". In: *Journal of Database Management (JDM)* 20 (2009), pp. 1–24 (cit. on p. 159).

[Gre06] S. Gregor. "The Nature of Theory in Information Systems". In: *MIS Quarterly* 30.3 (2006), p. 611 (cit. on p. 28).

[GS94] D. Garlan, M. Shaw. *An Introduction to Software Architecture*. Tech. rep. CMU-CS-94-166. Carnegie Mellon University, Jan. 1994 (cit. on p. 39).

[Gu20] Q. Gu. *A meta-approach to guide architectural refactoring from monolithic applications to microservices*. en. online. URL: http://elib.uni-stuttgart.de/handle/11682/11518. 2020 (cit. on pp. 153, 223).

[HA05] S. E. Hove, B. Anda. "Experiences from conducting semi-structured interviews in empirical software engineering research". In: *Proceedings - International Software Metrics Symposium* 2005.October 2005 (2005), pp. 203–212 (cit. on pp. 29, 83).

[Hab16] O. Habib. *A Quick Primer on Microservices*. online. URL: https://dzone.com/articles/a-quick-primer-on-microservices. 2016 (cit. on p. 41).

[Hal22] T. Haller. *Design, implementation and evaluation of an application for guiding architectural refactoring to microservices*. en. online. URL: http://elib.uni-stuttgart.de/handle/11682/12289. 2022 (cit. on pp. 216, 246).

[Her24] A. Herrmann. *Toolunterstütztes Refactoring von Microservices-Architekturen: Eine industrielle Fallstudie*. de. 2024 (cit. on pp. 292, 315, 316, 320, 325, 326, 329, 330).

[Hev07] A. R. Hevner. "A Three Cycle View of Design Science Research". In: *Scandinavian Journal of Information Systems* 19.2 (2007), pp. 87–92 (cit. on p. 29).

[HHL+19] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, M. Wojcieszak. "Industrial DevOps". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 123–126 (cit. on p. 110).

[HLT18] S. Habibullah, X. Liu, Z. Tan. "An Approach to Evolving Legacy Enterprise System to Microservice-Based Architecture through Feature-Driven Evolution Rules". In: *International Journal of Computer Theory and Engineering* 10.5 (2018), pp. 164–169 (cit. on pp. 301–303, 322).

[HMPR04] A. R. Hevner, S. T. March, J. Park, S. Ram. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), pp. 75–105 (cit. on pp. 28, 30, 196).

[HNSC21] S. G. Haugeland, P. H. Nguyen, H. Song, F. Chauvel. "Migrating Monoliths to Microservices-based Customizable Multi-tenant Cloud-native Apps". In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2021, pp. 170–177 (cit. on p. 166).

[HOAM23] M. H. Hasan, M. H. Osman, N. I. Admodisastro, M. S. Muhammad. "Legacy systems to cloud migration: A review from the architectural perspective". In: *Journal of Systems and Software* 202 (2023), p. 111702 (cit. on pp. 45, 105, 338).

[HR19] A. Henry, Y. Ridene. "Migrating to Microservices". In: *Microservices*. Springer International Publishing, Dec. 2019, pp. 45–72 (cit. on p. 46).

[HS10] W. Hwang, G. Salvendy. "Number of people required for usability evaluation: the 10±2 rule". In: *Communications of the ACM* 53.5 (2010), pp. 130–133 (cit. on p. 275).

[HSA19] M. S. Hamzehloui, S. Sahibuddin, A. Ashabi. “A study on the most prominent areas of research in microservices”. In: *International Journal of Machine Learning and Computing* 9.2 (2019), pp. 242–247 (cit. on p. 45).

[HWB17] S. Haselböck, R. Weinreich, G. Buchgeher. “Decision Models for Microservices: Design Areas, Stakeholders, Use Cases, and Requirements”. In: *Lecture Notes in Computer Science*. Ed. by A. Lopes, R. de Lemos. Vol. 10475. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 155–170 (cit. on p. 25).

[IEE90] IEEE. “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (1990), pp. 1–84 (cit. on p. 37).

[IGP+21] V. Ibarra-Junquera, A. Gonzalez-Potes, C. Paredes, D. Martinez-Castro, R. A. Nunez-Vizcaino. “Component-Based Microservices for Flexible and Scalable Automation of Industrial Bioprocesses”. In: *IEEE Access* 9 (2021), pp. 1–1 (cit. on p. 108).

[ISO11a] ISO. *ISO/IEC 25010:2011 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* Vol. 34:2910. International Organization for Standardization, Geneva, CH, 2011 (cit. on pp. 36, 57, 59, 60, 62, 68, 171, 261, 283, 299, 317).

[ISO11b] ISO. *ISO/IEC/IEEE 24765 — Systems and software engineering — Vocabulary — Second edition*. Vol. 34:2910. International Organization for Standardization, Geneva, CH, 2011 (cit. on p. 63).

[Jai22] J. Jain. *Introduction to API Testing*. Berkeley, CA: Apress, 2022, pp. 1–9 (cit. on p. 220).

[JC19] C. T. Joseph, K. Chandrasekaran. “Straddling the crevasse: A review of microservice software architecture foundations and recent advancements”. In: *Software - Practice and Experience* 49.10 (2019), pp. 1448–1484 (cit. on pp. 109, 110).

[JHT18] Y. Jiang, Z. Huang, D. H. K. Tsang. “Challenges and Solutions in Fog Computing Orchestration”. In: *IEEE Network* 32.3 (May 2018), pp. 122–129 (cit. on p. 109).

[JLC+19] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng. “Service Candidate Identification from Monolithic Systems based on Execution Traces”. In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1 (cit. on p. 104).

[JLZ+18] W. Jin, T. Liu, Q. Zheng, D. Cui, Y. Cai. “Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering”. In: *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, July 2018, pp. 211–218 (cit. on p. 160).

[JPM+18] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, S. Tilkov. “Microservices: The journey so far and challenges ahead”. In: *IEEE Software* 35.3 (2018), pp. 24–35 (cit. on pp. 24, 26, 82).

[JR19] A. Janes, B. Russo. “Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices”. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Oct. 2019, pp. 163–168 (cit. on p. 45).

[Kas05] M. Kasunic. *Designing an effective survey*. Tech. rep. September. Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, 2005, p. 143 (cit. on p. 264).

[Kay03] D. Kaye. *Loosely coupled*. Kentfield, CA: RDS Press, Aug. 2003 (cit. on p. 62).

[KBAR21] L. J. Kirby, E. Boerstra, Z. J. Anderson, J. Rubin. “Weighing the Evidence: On Relationship Types in Microservice Extraction”. In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 2021, pp. 358–368 (cit. on pp. 25, 52, 53, 180).

[KBWA94] R. Kazman, L. Bass, M. Webb, G. Abowd. “SAAM: A Method for Analyzing the Properties of Software Architectures”. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE ’94. Washington, DC, USA: IEEE Computer Society Press, 1994, pp. 81–90 (cit. on pp. 67, 201, 297).

[KC07] B. Kitchenham, S. Charters. *Guidelines for performing Systematic Literature reviews in Software Engineering*. Tech. rep. EBSE-2007-01. Keele, UK: School of Computer Science and Mathematics, Keele University, 2007, p. 65 (cit. on pp. 111, 150, 152, 177).

[KCAT21] I. Karabey Aksakalli, T. Celik, B. Ahmet, B. Tekinerdogan. “Deployment and communication patterns in microservice architectures: A systematic literature review”. In: *Journal of Systems and Software* 180 (Oct. 2021), p. 111014 (cit. on p. 60).

[KDJ04] B. A. Kitchenham, T. Dyba, M. Jorgensen. “Evidence-Based Software Engineering”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. USA: IEEE Computer Society, 2004, pp. 273–281 (cit. on pp. 27, 29).

[KH18] H. Knoche, W. Hasselbring. “Using Microservices for Legacy Software Modernization”. In: *IEEE Software* 35.3 (May 2018), pp. 44–49 (cit. on p. 48).

[KH19] H. Knoche, W. Hasselbring. “Drivers and Barriers for Microservice Adoption-A Survey among Professionals in Germany”. In: *Enterprise Modelling and Information Systems Architectures (EMISA)* 14.1 (2019), pp. 1–35 (cit. on pp. 24, 83, 104, 338, 339).

[KI12] H. Kniberg, A. Ivarsson. “Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds”. 2012 (cit. on p. 95).

[Kit04] B. Kitchenham. *Procedures for Performing Systematic Reviews*. Tech. rep. TR/SE-0401. Keele University, 2004 (cit. on p. 29).

[KKC00] R. Kazman, M. Klein, P. Clements. *ATAM: Method for Architecture Evaluation*. Tech. rep. CMU/SEI-2000-TR-004. Carnegie Mellon University, Software Engineering Institute’s Digital Library, Aug. 2000 (cit. on pp. 39, 67–69, 201, 209, 210, 297).

[KM19] J. Kazanavicius, D. Mazeika. “Migrating Legacy Software to Microservices Architecture”. In: *2019 Open Conference of Electrical, Electronic and Information Sciences, eStream 2019 - Proceedings* (2019) (cit. on pp. 40–43, 45, 47).

[KMM18] M. Kalske, N. Mäkitalo, T. Mikkonen. “Challenges When Moving from Monolith to Microservice Architecture”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10544 LNCS. 2018, pp. 32–47 (cit. on pp. 24, 40, 41, 43–45).

[Kno23] M. Knodel. *Migration monolithischer Anwendungen in Microservices-basierte Architekturen : Fallstudie einer Service/Sales-Applikation*. de. 2023 (cit. on pp. 292, 298, 300, 308, 309, 311, 313).

[Koc22] D. Koch. *Migrating monolithic architectures to microservices: a study on software quality attributes*. en. online. URL: http://elib.uni-stuttgart.de/handle/11682/12695. 2022 (cit. on pp. 59–61, 64, 73–75, 77, 78, 153, 194, 216).

[KP08] B. A. Kitchenham, S. L. Pfleeger. "Personal opinion surveys". In: *Guide to Advanced Empirical Software Engineering* (2008), pp. 63–92 (cit. on p. 29).

[KSS21] R. Kiran Mallidi, M. Sharma, J. Singh. "Legacy Digital Transformation: TCO and ROI Analysis". In: *International journal of electrical and computer engineering systems* 12.3 (Aug. 2021), pp. 163–170 (cit. on pp. 45, 47).

[KVGJ17] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, S. Jansen. "Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures". In: *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA* (2017), pp. 11–20 (cit. on p. 64).

[KXK+21] A. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, D. Banerjee. "Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices". In: *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23-28, 2021, Athens, Greece* 1.1 (July 2021) (cit. on pp. 161, 170, 185, 186).

[KXL+20] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. Rofrano, M. Vukovic, D. Banerjee. "Mono2Micro: an AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1606–1610 (cit. on pp. 185, 187, 341).

[KZH+20] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, D. Kroger. “Microservice Decomposition via Static and Dynamic Analysis of the Monolith”. In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 9–16 (cit. on pp. 170, 303–305, 307, 312, 334).

[L-M23] L-Mobile. *Internal Product Documentation*. online. URL: https://l-mobile.com/. 2023 (cit. on p. 298).

[LAO+18] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, R. Bonifácio. “An experience report on the adoption of microservices in three Brazilian government institutions”. In: *Proceedings of the XXXII Brazilian Symposium on Software Engineering - SBES ’18*. Vol. 10. New York, New York, USA: ACM Press, 2018, pp. 32–41 (cit. on pp. 82, 83).

[Lau19] L. D. Lauretis. “From Monolithic Architecture to Microservices Architecture”. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Oct. 2019, pp. 93–96 (cit. on p. 301).

[LCA21] N. Lapuz, P. Clarke, Y. Abgaz. “Digital Transformation and the Role of Dynamic Tooling in Extracting Microservices from Existing Software Systems”. In: *Communications in Computer and Information Science*. Springer, 2021, pp. 301–315 (cit. on pp. 52, 180).

[lev24] levigo solutions GmbH. *Jadice Flow Documentation: System Architecture*. online. URL: https://levigo.github.io/jadice-flow-documentation/docs/main-concepts/architecture/. 2024 (cit. on pp. 317, 318).

[LF14] J. Lewis, M. Fowler. *Microservices - a definition of this new architectural term*. online. URL: http://martinfowler.com/articles/microservices.html. 2014 (cit. on pp. 40–42, 44, 100, 123).

[LFW23] R. Lichtenthäler, J. Fritzsch, G. Wirtz. “Cloud-Native Architectural Characteristics and their Impacts on Software Quality: A Validation Survey”. In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2023, pp. 9–18 (cit. on pp. 34, 71, 77, 198).

[LG18] F. Li, L. Gelbke. “Microservice architecture in industrial software delivery on edge devices”. In: *Proceedings of the 19th International Conference on Agile Software Development: Companion*. New York, NY, USA: ACM, 2018 (cit. on p. 109).

[Lin16] D. S. Linthicum. “Practical Use of Microservices in Moving Workloads to the Cloud”. In: *IEEE Cloud Computing* 3.5 (2016), pp. 6–9 (cit. on p. 23).

[Lit18] M. Little. *Why Segment Returned to a Monolith from Microservices*. 2018 (cit. on p. 347).

[LJ17] Y. Lu, F. Ju. “Smart Manufacturing Systems based on Cyber-physical Manufacturing Services (CPMS)”. In: *IFAC-PapersOnLine* 50.1 (2017), pp. 15883–15889 (cit. on p. 107).

[LLHL05] X. Li, Z. Liu, J. He, Q. Long. “Generating a Prototype from a UML Model of System Requirements”. In: *Distributed Computing and Internet Technology*. Ed. by R. K. Ghosh, H. Mohanty. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 255–265 (cit. on p. 219).

[LLST20] V. Lenarduzzi, F. Lomio, N. Saarimäki, D. Taibi. “Does migrating a monolithic system to microservices decrease the technical debt?” In: *Journal of Systems and Software* 169.July (Nov. 2020), p. 110710 (cit. on pp. 40, 46, 162).

[LML20] C.-Y. Li, S.-P. Ma, T.-W. Lu. “Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System”. In: *2020 International Computer Symposium (ICS)*. IEEE, Dec. 2020, pp. 519–524 (cit. on pp. 302, 303, 307, 309, 312, 334).

[LS18] V. Lenarduzzi, O. Sievi-Korte. “On the negative impact of team independence in microservices software development”. In: *Second International Workshop on Microservices: Agile and DevOps Experience (MADE2018)*. June. 2018 (cit. on p. 60).

[LW22] R. Lichtenthäler, G. Wirtz. “Towards a Quality Model for Cloud-native Applications”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2022, pp. 109–117 (cit. on pp. 71, 77, 198, 400–402).

[LZJ+19] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan. “A dataflow-driven approach to identifying microservices from monolithic applications”. In: *Journal of Systems and Software* 157 (Nov. 2019), p. 110380 (cit. on p. 185).

[LZJ+21] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, M. A. Babar. “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review”. In: *Information and Software Technology* 131 (Mar. 2021), p. 106449 (cit. on pp. 57, 58, 60, 62, 64, 65, 74, 338, 341, 398, 399).

[MA19] G. Marquez, H. Astudillo. “Actual Use of Architectural Patterns in Microservices-Based Open Source Projects”. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* 2018-Decem.January 2019 (Dec. 2019), pp. 31–40 (cit. on pp. 74, 397–399).

[Man+95] J. C. Mankins et al. “Technology readiness levels”. In: *White Paper, April* 6 (1995) (cit. on p. 190).

[Mar03] R. C. Martin. “SRP: The Single Responsibility Principle”. In: *Agile Software Development: Principles, Patterns, and Practices* (2003) (cit. on p. 42).

[Mau22] J. Mauerer. *IT-Steinzeit in deutschen Unternehmen*. online. URL: https://www.cio.de/a/it-steinzeit-in-deutschen-firmen. 2022 (cit. on p. 23).

[MBMR21] N. C. Mendonça, C. Box, C. Manolache, L. Ryan. “The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture”. In: *IEEE Software* 38.5 (2021), pp. 17–22 (cit. on pp. 24, 347).

[MCF+20] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, A. Restivo. “Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis”. In: *Lecture Notes in Computer Science*. Vol. 12292 LNCS. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 315–332 (cit. on pp. 34, 135, 188, 303, 322, 323).

[MCL17] G. Mazlami, J. Cito, P. Leitner. “Extraction of Microservices from Monolithic Software Architectures”. In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 524–531 (cit. on pp. 104, 299).

[MDN20] S. A. Maisto, B. Di Martino, S. Nacchia. *From Monolith to Cloud Architecture Using Semi-automated Microservices Modernization*. Vol. 96. Springer International Publishing, 2020, pp. 638–647 (cit. on p. 41).

[MF19] P. Mayring, T. Fenzl. "Qualitative Inhaltsanalyse". In: *Handbuch Methoden der empirischen Sozialforschung*. Ed. by N. Baur, J. Blasius. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, pp. 633–648 (cit. on p. 315).

[MFC+18] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, N.-L. Hsueh. "Using Service Dependency Graph to Analyze and Test Microservices". In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE, July 2018, pp. 81–86 (cit. on p. 63).

[MFS20] J. Moon, B. Farnsworth, R. Smith. "The effectiveness of client-side JavaScript Testing". In: *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. AST '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 101–102 (cit. on p. 220).

[MGM06] M. Mattsson, H. Grahn, F. Mårtensson. "Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability". In: *Second International Conference on the Quality of Software Architectures (QoSA 2006)*. 2006 (cit. on p. 67).

[MHAK08] S. Moaven, J. Habibi, H. Ahmadi, A. Kamandi. "A Decision Support System for Software Architecture-Style Selection". In: *2008 Sixth International Conference on Software Engineering Research, Management and Applications*. IEEE, 2008, pp. 213–220 (cit. on pp. 37, 39).

[Mic21] Microsoft Corporation. *Entity Framework Core*. online. URL: `https://learn.microsoft.com/en-us/ef/core/`. 2021 (cit. on pp. 228, 229).

[Mic23a] Microsoft Corporation. *.NET - Entity Framework Core - Database Providers*. online. URL: `https://learn.microsoft.com/en-us/ef/core/providers/`. 2023 (cit. on p. 229).

[Mic23b] Microsoft Corporation. *Get started with Swashbuckle and ASP.NET Core*. online. URL: `https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle`. 2023 (cit. on p. 228).

[Mic23c] Microsoft Corporation. *Strangler Fig pattern*. online. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig. 2023 (cit. on pp. 163, 302, 303, 307).

[Mic23d] Microsoft Corporation. *What is .NET? Introduction and overview*. online. URL: https://learn.microsoft.com/en-us/dotnet/core/introduction. 2023 (cit. on p. 228).

[MM20] K. Munonye, P. Martinek. "Enhancing Performance of Distributed Transactions in Microservices via Buffered Serialization". In: *Journal of Web Engineering* 19.5-6 (2020), pp. 647–684 (cit. on p. 45).

[MM22] M. Milić, D. Makajić-Nikolić. "Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures". In: *Symmetry* 14.9 (Sept. 2022), p. 1824 (cit. on p. 64).

[MPB09] E. Murphy-Hill, C. Parnin, A. P. Black. "How we refactor, and how we know it". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 287–297 (cit. on p. 25).

[MRM21] I. Mizutani, G. Ramanathan, S. Mayer. "Integrating Multi-Disciplinary Offline and Online Engineering in Industrial Cyber-Physical Systems through DevOps". In: 11th International Conference on the Internet of Things (IoT '21). ACM, 2021 (cit. on p. 108).

[MSV21] R. K. Mallidi, M. Sharma, S. R. Vangala. "Semantic Review on Software Architectures for Web-based Applications". In: *COMPUSOFT, An international journal of advanced computer technology* 10.3 (2021) (cit. on p. 39).

[MT04] T. Mens, T. Tourwe. "A survey of software refactoring". In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139 (cit. on p. 48).

[MY20] P. Merson, J. Yoder. "Modeling Microservices with DDD". In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2020, pp. 7–8 (cit. on p. 166).

[NB06] B. Niehaves, J. Becker. "Epistemological Perspectives on Design Science in IS Research". In: *Americas Conference on Information Systems*. 2006 (cit. on p. 28).

[New15] S. Newman. *Building Microservices*. O'Reilly, 2015 (cit. on pp. 24, 42, 43, 246).

[New21] S. Newman. *Building microservices*. "O'Reilly", 2021 (cit. on pp. 24, 180).

[NG04] M. E. J. Newman, M. Girvan. "Finding and evaluating community structure in networks". In: *Physical Review E* 69.2 (Feb. 2004) (cit. on p. 188).

[NOLP21] V. M. Nino-Martinez, J. Octavio Ocharan-Hernandez, X. Limon, J. C. Perez-Arriaga. "Microservices Deployment: A Systematic Mapping Study". In: *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, Oct. 2021, pp. 24–33 (cit. on p. 44).

[NSR19] L. Nunes, N. Santos, A. Rito Silva. "From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 37–52 (cit. on pp. 183, 184).

[NUEH18] R. Nakazawa, T. Ueda, M. Enoki, H. Horii. "Visualization Tool for Designing Microservices with the Monolith-First Approach". In: *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018* (2018), pp. 32–42 (cit. on p. 303).

[NW14] A. Nuraini, Y. Widyani. "Software with service oriented architecture quality assessment". In: *Proceedings of 2014 International Conference on Data and Software Engineering, ICODSE 2014* (2014) (cit. on p. 57).

[NZP+20] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, S. Geiger. "Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 3–20 (cit. on p. 71).

[ÖBB23a] O. Özkan, Ö. Babur, M. Brand. "Refactoring with domain-driven design in an industrial context: An action research report". In: *Empirical Software Engineering* 28 (2023) (cit. on p. 50).

[ÖBB23b] O. Özkan, Ö. Babur, M. van den Brand. *Domain-Driven Design in Software Development: A Systematic Literature Review on Implementation, Challenges, and Effectiveness*. 2023. arXiv: 2310.01905 [cs.SE] (cit. on p. 50).

[ODGG20] T. d. Oliveira Rosa, J. F. L. Daniel, E. M. Guerra, A. Goldman. “A Method for Architectural Trade-off Analysis Based on Patterns: Evaluating Microservices Structural Attributes”. In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. EuroPLoP ’20. New York, NY, USA: ACM, 2020 (cit. on pp. 71, 72).

[OMA19] F. Osses, G. Márquez, H. Astudillo. “An Exploratory Study of Academic Architectural Tactics and Patterns in Microservices: A systematic literature review”. In: *Avances en Ingenieria de Software a Nivel Iberoamericano, CIbSE 2018* (Feb. 2019) (cit. on p. 72).

[Ope23] OpenAPI Initiative. *OpenAPI Specification*. online. URL: https://swagger.io/specification/. 2023 (cit. on p. 228).

[Pap03] M. Papazoglou. “Service-oriented computing: concepts, characteristics and directions”. In: *Proceedings of the 7th International Conference on Properties and Applications of Dielectric Materials (Cat. No.03CH37417)*. IEEE Comput. Soc, 2003, pp. 3–12 (cit. on p. 39).

[PCF21] I. M. Pereira, T. Carneiro, E. Figueiredo. “A systematic review on the use of DevOps in internet of things software systems”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, Mar. 2021 (cit. on p. 110).

[PD20] R. Pradhan, A. K. Dash. “An Overview of Microservices”. In: *ICDSMLA 2019*. Springer Singapore, 2020, pp. 620–625 (cit. on p. 45).

[Per20] J. Perrott. *Angular Flex-Layout*. online. URL: https://github.com/angular/flex-layout. 2020 (cit. on p. 232).

[PFMM08] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson. “Systematic mapping studies in software engineering”. In: *EASE’08 Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering* (2008), pp. 68–77 (cit. on p. 111).

[PJ16] C. Pahl, P. Jamshidi. “Microservices: A Systematic Mapping Study”. In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. January. 2016, pp. 137–146 (cit. on pp. 82, 166).

[PKY20] J. Park, D. Kim, K. Yeom. “An Approach for Reconstructing Applications to Develop Container-Based Microservices”. In: *Mobile Information Systems* 2020 (Jan. 2020), pp. 1–23 (cit. on p. 301).

[PL18] J. Phillippi, J. Lauderdale. “A guide to field notes for qualitative research: Context and conversation”. In: *Qualitative health research* 28.3 (2018), pp. 381–388 (cit. on p. 296).

[PMA19] F. Ponce, G. Márquez, H. Astudillo. “Migrating from Monolithic architecture to microservices: A Rapid Review”. In: *Proceedings of 38th International Conference of the Chilean Computer Science Society (SCCC 2019), Chile*. 2019 (cit. on pp. 25, 180).

[Pre21] P. Prema. *Spring Boot vs ASP.NET Core: A Showdown*. 2021 (cit. on p. 228).

[PRF07] M. Perepletchikov, C. Ryan, K. Frampton. “Cohesion metrics for predicting maintainability of service-oriented software”. In: *Proceedings - International Conference on Quality Software* Qsic (2007), pp. 328–335 (cit. on pp. 62, 63).

[PS22] S. Pulnil, T. Senivongse. “A Microservices Quality Model Based on Microservices Anti-patterns”. In: *2022 19th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2022, pp. 1–6 (cit. on pp. 77, 400–402).

[PT19] T. Poklepović Peričić, S. Tanveer. *Why systematic reviews matter*. online. URL: `https://www.elsevier.com/connect/why-systematic-reviews-matter`. 2019 (cit. on p. 150).

[PTRC07] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee. “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77 (cit. on pp. 28, 30, 196).

[PVK15] K. Petersen, S. Vakkalanka, L. Kuzniarz. “Guidelines for conducting systematic mapping studies in software engineering: An update”. In: *Information and software technology* 64 (2015), pp. 1–18 (cit. on p. 150).

[PW92] D. E. Perry, A. L. Wolf. “Foundations for the Study of Software Architecture”. In: *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), pp. 40–52 (cit. on p. 37).

[RH08] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (Dec. 2008), pp. 131–164 (cit. on pp. 29, 84, 115, 293, 315, 333).

[Ric15] C. Richardson. *Introduction to Microservices*. online. URL: https://web.archive.org/web/20240214235828/https://www.nginx.com/blog/introduction-to-microservices/. 2015 (cit. on p. 60).

[Ric23] C. Richardson. *A pattern language for microservices*. online. URL: https://microservices.io/patterns/index.html. 2023 (cit. on pp. 74, 307, 397–400).

[RKH10] V. Reijonen, J. Koskinen, I. Haikala. “Experiences from Scenario-Based Architecture Evaluations with ATAM”. In: *Software Architecture*. Ed. by M. A. Babar, I. Gorton. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 214–229 (cit. on pp. 68, 69, 207).

[RR22] V. Raj, S. Ravichandra. “Enhanced Service Point Approach for Microservices Based Applications Using Machine Learning Techniques”. In: *Advanced Informatics for Computing Research*. Springer International Publishing, 2022, pp. 78–90 (cit. on p. 49).

[RSHR15] K. Rostami, J. Stammel, R. Heinrich, R. Reussner. “Architecture-based assessment and planning of change requests”. In: *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. 2015, pp. 21–30 (cit. on p. 168).

[RSS18] F. Rademacher, J. Sorgalla, S. Sachweh. “Challenges of domain-driven microservice design: A model-driven perspective”. In: *IEEE Software* 35.3 (May 2018), pp. 36–43 (cit. on pp. 50, 82, 100).

[RSZ17] F. Rademacher, S. Sachweh, A. Zundorf. “Differences between Model-Driven Development of Service-Oriented and Microservice Architecture (SOA vs. MSA)”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 38–45 (cit. on p. 50).

[RWW+18] Z. Ren, W. Wang, G. Wu, C. Gao, W. Chen, J. Wei, T. Huang. “Migrating Web Applications from Monolithic Structure to Microservices Architecture”. In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware - Internetware ’18*. New York, New York, USA: ACM Press, 2018, pp. 1–10 (cit. on pp. 40, 43, 44).

[Rym17] J. R. Rymer. *Vendor Landscape: A Fork In The Road For Low-Code Development Platforms*. 2017 (cit. on p. 335).

[SAZ+18] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi, F. Ismaili. "Comparison between relational and NOSQL databases". In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2018, pp. 216–221 (cit. on p. 228).

[SB21] T. Schirgi, E. Brenner. "Quality Assurance for Microservice Architectures". In: *2021 IEEE 12th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, Aug. 2021, pp. 76–80 (cit. on pp. 72, 77, 400–402).

[SBH+22] X. Sun, S. Boranbaev, S. Han, H. Wang, D. Yu. "Expert system for automatic microservices identification using API similarity graph". In: *Expert Systems* 41.5 (Oct. 2022) (cit. on p. 185).

[Sch06] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (2006), pp. 25–31 (cit. on p. 50).

[Sch23a] A. Schaffry. *Licht und Schatten*. 2023 (cit. on p. 335).

[Sch23b] A. Schaffry. *Wie KI die No-Code/Low-Code-Entwicklung verändert*. 2023 (cit. on p. 336).

[Sch91] D. A. Schön. *The reflective practitioner*. London, England: Ashgate Publishing, Aug. 1991 (cit. on p. 28).

[SCKP08] B. Shim, S. Choue, S. Kim, S. Park. "A Design Quality Model for Service-Oriented Architecture". In: *2008 15th Asia-Pacific Software Engineering Conference*. 2008, pp. 403–410 (cit. on pp. 57, 60–63).

[SDJ07] D. I. K. Sjøberg, T. Dyba, M. Jorgensen. "The Future of Empirical Methods in Software Engineering Research". In: *Future of Software Engineering (FOSE '07)*. 2007, pp. 358–378 (cit. on pp. 27, 29).

[Sea08] C. B. Seaman. "Qualitative Methods". In: *Guide to Advanced Empirical Software Engineering*. Guide to Advanced Empirical Software Engineering. London: Springer London, 2008, pp. 35–62 (cit. on pp. 29, 83, 84, 115, 296, 315, 333).

[SF08] F. Shull, R. L. Feldmann. “Building Theories from Multiple Evidence Sources”. In: *Guide to Advanced Empirical Software Engineering*. Ed. by F. Shull, J. Singer, D. I. K. Sjøberg. London: Springer London, 2008, pp. 337–364 (cit. on p. 29).

[SFP+19] N. Santos, N. Ferreira, M. Pereira, C. E. Salgado, F. Morais, M. Melo, S. Silva, R. Martins, M. Pereira, H. Rodrigues, R. J. Machado. “A logical architecture design method for microservices architectures”. In: *Proceedings of the 13th European Conference on Software Architecture - ECSA ’19 - volume 2*. Vol. 2. New York, New York, USA: ACM Press, 2019, pp. 145–151 (cit. on pp. 47, 82).

[Sil15] A. da Silva. “Model-driven engineering: A survey supported by the unified conceptual model”. In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155 (cit. on p. 50).

[SK17] A. Shashwat, D. Kumar. “A service identification model for service oriented architecture”. In: *3rd IEEE International Conference on "Computational Intelligence and Communication Technology" (IEEE-CICT 2017)*. 2017, pp. 1–5 (cit. on pp. 301–303).

[SKM20] C. Schröer, F. Kruse, J. Marx Gómez. “A Qualitative Literature Review on Microservices Identification Approaches”. In: *Communications in Computer and Information Science*. Vol. 1310. Springer International Publishing, 2020, pp. 151–168 (cit. on pp. 25, 51, 52, 166, 168).

[SLT23] R. Su, X. Li, D. Taibi. “Back to the Future: From Microservice to Monolith”. In: *5th International Conference on Microservices (Microservices 2023), Pisa, Italy*. 2023, pp. 1–6. arXiv: 2308.15281v1 (cit. on p. 347).

[SM07] M. Svahnberg, F. Mårtensson. “Six years of evaluating software architectures in student projects”. In: *Journal of Systems and Software* 80.11 (Nov. 2007), pp. 1893–1901 (cit. on pp. 69, 201, 295, 297, 299, 314, 316, 334).

[SMMS11] T. da Silva, A. Martin, F. Maurer, M. Silveira. “User-Centered Design and Agile Methods: A Systematic Review”. In: *2011 Agile Conference*. 2011, pp. 77–86 (cit. on p. 219).

[Sne12] H. M. Sneed. “Nachhaltigkeit durch gesteuerte Software-Evolution”. In: *Nachhaltiges Software Management*. Bonn: Gesellschaft für Informatik e.V., 2012, pp. 50–68 (cit. on pp. 21, 22).

[SS13] H. M. Sneed, R. Seidl. *Softwareevolution: Erhaltung und Fortschreibung bestehender Softwaresysteme*. dpunkt.verlag, Germany, 2013 (cit. on pp. 22, 23, 46, 162).

[SS17] Sarita, S. Sebastian. “Transform Monolith into Microservices using Docker”. In: *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*. 2017, pp. 1–5 (cit. on pp. 40–44).

[SS21] A. Stojkov, Z. Stojanov. “Review of methods for migrating software systems to microservices architecture”. In: *Journal of Engineering Management and Competitiveness* 11.2 (2021), pp. 152–162 (cit. on p. 51).

[SS22] S. Santos, A. R. Silva. “Microservices Identification in Monolith Systems: Functionality Redesign Complexity and Evaluation of Similarity Measures”. In: *Journal of Web Engineering* (Aug. 2022) (cit. on pp. 70, 183, 184).

[SSB+20] A. Selmadji, A.-d. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, C. Dony. “From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach”. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. Section III. IEEE, Mar. 2020, pp. 157–168 (cit. on pp. 322, 323).

[SSS16] G. Samarthyam, G. Suryanarayana, T. Sharma. “Refactoring for software architecture smells”. In: *Proceedings of the 1st International Workshop on Software Refactoring* (2016), pp. 1–4 (cit. on p. 49).

[ST20] R. A. Schmidt, M. Thiry. “Microservices identification strategies : A review focused on Model-Driven Engineering and Domain Driven Design approaches”. In: *Iberian Conference on Information Systems and Technologies, CISTI* 2020-June.June (2020), pp. 24–27 (cit. on pp. 50, 51).

[Sta08] G. Starke. *Effektive Software-Architekturen*. de. 3rd ed. Munich, Germany: Hanser, Jan. 2008 (cit. on pp. 66, 68, 69, 207).

[STH17] A. Sayara, M. S. Towhid, M. S. Hossain. “A probabilistic approach for obtaining an optimized number of services using weighted matrix and multidimensional scaling”. In: *2017 20th International Conference of Computer and Information Technology (ICCIT)*. Vol. 2018-Janua. IEEE, Dec. 2017, pp. 1–6 (cit. on pp. 42, 47).

[STV18] J. Soldani, D. A. Tamburri, W. J. Van Den Heuvel. “The pains and gains of microservices: A Systematic grey literature review”. In: *Journal of Systems and Software* 146.September (2018), pp. 215–232 (cit. on p. 82).

[SWG21] C. Schröer, S. Wittfoth, J. M. Gomez. “A Process Model for Microservices Design and Identification”. In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2021, pp. 1–8 (cit. on pp. 65, 66).

[Tay18] R. N. Taylor. “Only the Architecture You Need”. In: *The Essence of Software Engineering*. Ed. by V. Gruhn, R. Striemer. Cham: Springer International Publishing, 2018, pp. 77–89 (cit. on p. 37).

[TEPN20] D. Taibi, N. El Ioini, C. Pahl, J. Niederkofler. “Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020 (cit. on p. 46).

[TG18] M. Törngren, P. Grogan. “How to Deal with the Complexity of Future Cyber-Physical Systems?” In: *Designs* 2.4 (Oct. 2018), p. 40 (cit. on p. 109).

[THG20] F. Trautsch, S. Herbold, J. Grabowski. “Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects”. In: *Journal of Systems and Software* 159 (2020), p. 110421 (cit. on p. 220).

[THLL18] S. Tyszberowicz, R. Heinrich, B. Liu, Z. Liu. “Identifying Microservices Using Functional Decomposition”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10998 LNCS. 2018, pp. 50–65 (cit. on pp. 44, 47, 168).

[TL18] D. Taibi, V. Lenarduzzi. “On the Definition of Microservice Bad Smells”. In: *IEEE Software* 35.3 (May 2018), pp. 56–62 (cit. on p. 72).

[TLP17] D. Taibi, V. Lenarduzzi, C. Pahl. “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation”. In: *IEEE Cloud Computing* 4.5 (Sept. 2017), pp. 22–32 (cit. on pp. 46, 49, 83, 104, 197, 199, 309, 338).

[TLP18] D. Taibi, V. Lenarduzzi, C. Pahl. “Architectural Patterns for Microservices: A Systematic Mapping Study”. In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science* March (2018), pp. 221–232 (cit. on pp. 74, 76, 77, 82, 100, 307, 397–400).

[TLP19] D. Taibi, V. Lenarduzzi, C. Pahl. “Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study”. In: Cloud Computing and Services Science. CLOSER 2018. Communications in Computer and Information Science. Springer, 2019, pp. 126–151 (cit. on p. 110).

[TMD09] R. N. Taylor, N. Medvidovic, E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Chichester, England: John Wiley & Sons, 2009 (cit. on pp. 37, 39).

[TS18] M. Törngren, U. Sellgren. “Complexity Challenges in Development of Cyber-Physical Systems”. In: vol. 10760 LNCS. Lecture Notes in Computer Science. Springer, 2018 (cit. on p. 109).

[TS19] D. Taibi, K. Systä. “From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining”. In: *9th International Conference on Cloud Computing and Services Science, CLOSER , 2019*. Heraklion, Greece, 2019 (cit. on p. 104).

[TVS18] K. Thramboulidis, D. C. Vachtsevanou, A. Solanos. “Cyber-physical microservices: An IoT-based framework for manufacturing systems”. In: *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*. IEEE, May 2018, pp. 232–239 (cit. on p. 107).

[VCG+22] G. Vale, F. F. Correia, E. M. Guerra, T. de Oliveira Rosa, J. Fritzsch, J. Bogner. “Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs”. In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, Mar. 2022, pp. 69–79 (cit. on pp. 24, 34, 71, 74, 76, 77, 135, 180, 345, 397–400).

[VF23] V. Velepucha, P. Flores. “A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges”. In: *IEEE Access* 11 (2023), pp. 88339–88358 (cit. on pp. 40, 41).

[VFD+16] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan. “Osmotic Computing: A New Paradigm for Edge/Cloud Integration”. In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83 (cit. on p. 141).

[VGC+15] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Colombian Computing Conference, 10CCC 2015* (2015), pp. 583–590 (cit. on p. 44).

[VGO+16] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang. “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures”. In: *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid* (2016), pp. 179–182 (cit. on p. 299).

[Vir92] R. A. Virzi. “Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough?” In: *Human Factors* 34.4 (1992), pp. 457–468 (cit. on p. 275).

[VK21] H. Vural, M. Koyuncu. “Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?” In: *IEEE Access* (2021), pp. 1–1 (cit. on p. 50).

[VKG17] H. Vural, M. Koyuncu, S. Guney. “A Systematic Literature Review on Microservices”. In: *Computational Science and Its Applications – ICCSA 2017*. Cham: Springer International Publishing, 2017, pp. 203–217 (cit. on p. 82).

[VLC19] J. A. Valdivia, X. Limon, K. Cortes-Verdin. “Quality attributes in patterns related to microservice architecture: a Systematic Literature Review”. In: *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, Oct. 2019, pp. 181–190 (cit. on pp. 71, 72, 74, 82, 397–400).

[VLL+20] J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, J. Ocharán-Hernández. “Patterns Related to Microservice Architecture: a Multivocal Literature Review”. In: *Programming and Computer Software* 46.8 (Dec. 2020), pp. 594–608 (cit. on pp. 71, 74, 307, 397–400).

[Vog09] W. Vogels. “Eventually consistent”. In: *Communications of the ACM* 52.1 (2009), pp. 40–44 (cit. on p. 43).

[VPAG20] F. H. Vera-Rivera, E. G. Puerto-Cuadros, H. Astudillo, C. M. Gaona-Cuevas. “Microservices Backlog - A Model of Granularity Specification and Microservice Identification”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12409 LNCS. 2020, pp. 85–102 (cit. on p. 185).

[VPAG21] F. H. Vera-Rivera, E. Puerto, H. Astudillo, C. Gaona. “Microservices backlog – a genetic programming technique for identification and evaluation of microservices from user stories.” In: *IEEE Access* (2021), pp. 1–1 (cit. on pp. 185, 322, 323, 326, 334).

[VTR+18] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, E. Figueiredo. “Microservices in Practice: A Survey Study”. In: *VI Workshop on Software Visualization, Evolution and Maintenance, São Carlos, Brazil* abs/1808.0.September (2018). arXiv: 1808.04836 (cit. on p. 82).

[WA11] G. Wilson, J. Aranda. “Empirical Software Engineering”. In: *American Scientist* 99.6 (2011), p. 466 (cit. on p. 27).

[WA15] C. Wohlin, A. Aurum. “Towards a decision-making structure for selecting a research design in empirical software engineering”. In: *Empirical Software Engineering* 20.6 (2015), pp. 1427–1455 (cit. on pp. 27–29).

[Wag13] S. Wagner. *Software Product Quality Control*. Springer Berlin Heidelberg, 2013 (cit. on p. 57).

[WAS+21] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, D. d. N. Paza. “Modernizing Legacy Systems with Microservices: A Roadmap”. In: *Evaluation and Assessment in Software Engineering*. New York, NY, USA: ACM, June 2021, pp. 149–159 (cit. on pp. 197, 199, 201).

[Wei76] K. E. Weick. “Educational Organizations as Loosely Coupled Systems”. In: *Administrative Science Quarterly* 21.1 (1976), pp. 1–19 (cit. on p. 62).

[Wie09] R. Wieringa. “Design science as nested problem solving”. In: *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*. DESRIST ’09. New York, NY, USA: Association for Computing Machinery, 2009 (cit. on pp. 28, 30, 196).

[Wie14] R. J. Wieringa. “Research Goals and Research Questions”. In: *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 13–23 (cit. on p. 26).

[Wig17] A. Wiggins. *The twelve-factor App*. online. URL: `https://12factor.net/`. 2017 (cit. on p. 93).

[WLS20] M. Waseem, P. Liang, M. Shahin. “A Systematic Mapping Study on Microservices Architecture in DevOps”. In: *Journal of Systems and Software* 170 (Dec. 2020), p. 110798 (cit. on p. 109).

[Woh14] C. Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE ’14* (2014), pp. 1–10 (cit. on pp. 113, 152).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. Vol. 9783642290. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–236 (cit. on p. 144).

[XW05] R. Xu, D. Wunsch. “Survey of clustering algorithms”. In: *IEEE Transactions on Neural Networks* 16.3 (2005), pp. 645–678 (cit. on p. 165).

[YGS21] J. Yu, J. Ge, J. Sun. “Research on Quality Model and Measurement for Microservices”. In: *Proceedings of the 3rd International Workshop on Experience with SQuaRE Series and Its Future Direction (IWESQ 2021) co-located with 28th Asia-Pacific Software Engineering Conference (APSEC 2021), Virtual Event, Taipei, Taiwan, Dec 8, 2021*. Ed. by T. Nakajima, T. Komiyama. Vol. 3114. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 5–10 (cit. on pp. 62, 63, 198).

[Yin03] R. K. Yin. *Case Study Research: Design and Methods*. 3rd ed. Applied Social Research Methods. Thousand Oaks, CA: SAGE Publications, Feb. 2003 (cit. on p. 293).

[YM22] J. W. Yoder, P. Merson. "Strangler Patterns". In: *Proceedings of the 27th Conference on Pattern Languages of Programs*. PLoP '20. USA: The Hillside Group, 2022 (cit. on pp. 48, 307, 309).

[YPS17] P. Yugopuspito, F. Panduwinata, S. Sutrisno. "Microservices architecture: Case on the migration of reservation-based parking system". In: *2017 IEEE 17th International Conference on Communication Technology (ICCT)*. Vol. 2017-Octob. IEEE, Oct. 2017, pp. 1827–1831 (cit. on p. 42).

[Zai07] Z. Zainal. "Case study as a research method". In: *Jurnal kemanusiaan* 5.1 (2007) (cit. on p. 293).

[Zim09] O. Zimmermann. *An architectural decision modeling framework for service oriented architecture design*. online. URL: http://elib.uni-stuttgart.de/handle/11682/2682. 2009 (cit. on pp. 39, 71, 72).

[Zim15] O. Zimmermann. "Architectural refactoring: A task-centric view on software evolution". In: *IEEE Software* 32.2 (2015), pp. 26–29 (cit. on pp. 49, 58).

[Zim16a] O. Zimmermann. "Microservices tenets: Agile approach to service development and deployment". In: *Computer Science - Research and Development* (2016), pp. 1–10 (cit. on pp. 39, 42, 43).

[Zim16b] T. Zimmermann. "Card-sorting: From text to themes". In: Perspectives on Data Science for Software Engineering. Elsevier, 2016, pp. 137–141 (cit. on p. 116).

[Zim17] O. Zimmermann. "Architectural refactoring for the cloud: a decision-centric view on cloud migration". In: *Computing* 99.2 (2017), pp. 129–145 (cit. on pp. 48, 49, 67).

[ZLC+23] X. Zhou, S. Li, L. Cao, H. Zhang, Z. Jia, C. Zhong, Z. Shan, M. A. Babar. "Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry". In: *Journal of Systems and Software* 195 (2023), p. 111521 (cit. on p. 26).

[ZLJ+19] H. Zhang, S. Li, Z. Jia, C. Zhong, C. Zhang. “Microservice Architecture in Reality: An Industrial Inquiry”. In: 2019 IEEE International Conference on Software Architecture (ICSA). IEEE, Mar. 2019, pp. 51–60 (cit. on pp. 139, 166).

[ZSS+18] A. Zimmermann, R. Schmidt, K. Sandkuhl, D. Jugel, J. Bogner, M. Mohring. “Evolution of Enterprise Architecture for Digital Transformation”. In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)* November (2018), pp. 87–96 (cit. on p. 38).

[ZSS+21] P. Zaragoza, A.-D. Seriai, A. Seriai, H.-L. Bouziane, A. Shatnawi, M. Derras. “Refactoring Monolithic Object-Oriented Source Code to Materialize Microservice-oriented Architecture”. In: *Proceedings of the 16th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, 2021, pp. 78–89 (cit. on pp. 301, 303).

[ZZ21] J. Zhao, K. Zhao. “Applying Microservice Refactoring to Object-Oriented Legacy System”. In: *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, Aug. 2021, pp. 467–473 (cit. on p. 303).

All URLs were last checked on 21/05/2024.

# List of Figures


1.1. Thesis Structure — Contributions and Methods . . . . . 30

2.1. Domain-Driven Design (DDD) and Model-Driven Engineering (MDE) in System Development . . . . . 51
2.2. Refactoring Techniques Found in Existing Reviews . . . . . 53
2.3. Taxonomy of Service Identification Approaches . . . . . 55
2.4. Microservices Qualities . . . . . 59
2.5. Quality Attributes and Sub-Attributes in Microservices . . . . 61
2.6. ATAM Utility Tree . . . . . 69

4.1. General Research Process . . . . . 112
4.2. Rapid Review Process: Number of Publications per Stage . . 117
4.3. Rapid Review: Challenge and Practice Categories . . . . . 119
4.4. CPS Interviews: Distribution of Challenge Categories . . . . . 124
4.5. CPS Interviews: Distribution of Practice Categories . . . . . 127
4.6. CPS Interviews: Distribution of Migration Rationales . . . . . 129
4.7. Interviewees' Sentiment on Framework Adoption . . . . . 132
4.8. Automation Potential and Aspects for Tools in Migrations . . 133
4.9. Challenges: Comparing CPS and Enterprise Applications . . 137

4.10. Practices: Comparing CPS and Enterprise Applications . . . . 137

5.1. Search Strategy for Literature Review . . . . . 152
5.2. Refactoring Approaches by Year of Publication (N=110) . . 154
5.3. Characteristics of Tools (N=31) . . . . . 170
5.4. Refactoring Approaches by Quality Attributes (N=110) . . . 174

6.1. Review Process: Filtering Tools for Decomposition . . . . . 182

7.1. Proposed Framework for Microservices Migrations . . . . . 200
7.2. Quality Attributes and Design Properties as Drivers in the Migration Process . . . . . 204
7.3. Quality-Driven Migration Process . . . . . 205
7.4. Calculation of Match Count for Refactoring Approaches . . . 208

8.1. Use Case Diagram for Application Prototype . . . . . 223
8.2. Entity-Relationship Model for Refactoring Approaches . . . . 225
8.3. Application Start Page With “Phase 1” Selected . . . . . 233
8.4. Routing and Navigation for Project and Admin. Views . . . . 234
8.5. Routing and Navigation for User Views . . . . . 235
8.6. Dialog View Using Material Design . . . . . 237
8.7. View for Predefined Strategy Filters . . . . . 238
8.8. Custom Filter Configuration for Refactoring Approaches . . . 239
8.9. Entity-Relationship Model for Scenario Representation . . . 242
8.10. User Interface View for Scenario Definition . . . . . 242
8.11. Approach Results View in Scenario-Based Workflow . . . . . 243
8.12. Configuration Overview in Side Panel . . . . . 245
8.13. Deployment Architecture . . . . . 246

9.1. Assessment of Vertical Prototype by Participants (N=9) . . . 258
9.2. Application Structure on a 5-Point Likert Scale (N=11) . . . 268
9.3. Application Usability on a 5-Point Likert Scale (N=11) . . . 269
9.4. Potential Additional Features of the Application (N=9) . . . 271

9.5. User Interface of Framework Phase 1 . . . . . 282

10.1. Architecture of Service/Sales Application . . . . . 298
10.2. Prioritized Quality Goals for the L-mobile Application . . . . 300
10.3. Microservices Architecture for the Service/Sales Appl. . . . . 308
10.4. Architecture of Document Management Solution . . . . . 318
10.5. Utility Tree for the jadice flow Application . . . . . 320

# List of Tables


2.1. Analysis Technique Types in Existing Reviews . . . . . 52
2.2. Interplay Between Design Properties and Quality Attributes . . . . . 64
2.3. 51 Microservices Patterns From 7 Publications . . . . . 73
2.4. Qualities Promoted by 51 Patterns . . . . . 75
2.5. Quality Trade-Offs for 13 Patterns . . . . . 77
2.6. Qualities Promoted by 43 Best Practices . . . . . 78

3.1. Company and Participant Demographics . . . . . 86
3.2. System and Organization Overview . . . . . 87
3.3. Intentions, Strategies, and Challenges per System . . . . . 98

4.1. Most Mentioned Challenges and Practices . . . . . 121
4.2. Interview Participant Demographics . . . . . 122
4.3. System and Migration Properties . . . . . 123
4.4. Challenges Affecting Two or More Systems . . . . . 126
4.5. Empirical Studies on Enterprise Application Microservices . 135

5.1. Iterations of This Review Between 2018 and 2023 . . . . . 153

5.2. Classification of 110 Approaches by Input, Output, and Process . . . 156
5.3. Classification of 110 approaches by Usability and Quality . 157
5.4. Quality Goals of Refactoring Approaches (N=110) . . . . . 173
5.5. Taxonomy for Refactoring Approaches . . . . . 176

6.1. Characterization of Executable Tools for Decomposition . . 186

8.1. Requirements for Extensible Repository and Back-End . . . 224
8.2. Requirements for User Interface and Front-End . . . . . 230
8.3. Requirements for Quality Model Implementation . . . . . 241
8.4. Integration Test Cases for the Web-Based Application . . . . 251

9.1. Prototype Survey: Participant Demographics (N=9) . . . . 257
9.2. Prototype Assessment on a 5-Point Likert Scale (N=9) . . . 258
9.3. Prototype Survey: Feedback From Participants . . . . . 260
9.4. Application Survey: Participant Demographics (N=11) . . . 266
9.5. Quality Evaluation Survey: Demographics (N=3) . . . . . 277
9.6. Quality Evaluation Interviews: Demographics (N=3) . . . . 281

10.1. Scenarios for the Highest Prioritized Quality Attributes . . 300
10.2. Approaches Filtered for Quality Goals . . . . . 301
10.3. Approaches Filtered for Quality Goals and Input Artifacts . 302
10.4. Filtering Patterns and Best Practices . . . . . 305
10.5. Selected Patterns After Filtering . . . . . 306
10.6. Selected Best Practices After Filtering . . . . . 306
10.7. Filter Configurations for Refactoring Approaches . . . . . 321
10.8. Rounds of Filtering and Their Configuration in Phase Two . 321
10.9. Resulting 8 Refactoring Approaches With Their Ranking . . 322
10.10. Filter Results for Patterns and Best Practices . . . . . 325
10.11. Interview Demographics . . . . . 326

A.1. Quality Attributes Associated With Patterns . . . . . 397

A.2. Quality Attributes Associated With Best Practices . . . . . 400

B.1. Reviewed Publications From 2016 to 2023 (N=110) . . . . 403

C.1. Requirements for Extensible Repository and Back-End . . . 409
C.2. Requirements for User Interface and Front-End . . . . . 412
C.3. Requirements for Administrator User Interface . . . . . 416
C.4. Requirements for Quality Model . . . . . 419

Appendix A

# Patterns and Best Practices

**Table A.1.:** Quality Attributes Associated With Patterns

| ID | Pattern | Quality Attributes | Reference |
|---|---|---|---|
| 1 | Ambassador | Reliability, Maintainability, Scalability, Security, Availability, Monitorability | [VCG+22; VLL+20] |
| 2 | Anti-Corruption Layer | Performance, Compatibility | [VCG+22; VLL+20] |
| 3 | API Gateway | Maintainability, Reliability, Security, Compatibility, Availability | [MA19; TLP18; VLC19; VLL+20] |
| 4 | Asynchronous Completion Token | Reliability | [VLC19; VLL+20] |
| 5 | Asynchronous Messaging | Reliability, Performance, Maintainability | [Ric23; VLL+20] |
| 6 | Asynchronous Query | Reliability | [VLC19; VLL+20] |

**Continued on next page**

**Table A.1 – Continued from previous page**

| ID | Pattern | Quality Attributes | Reference |
|---|---|---|---|
| 7 | Auth-Service | Reliability, Security | [LZJ+21; VLC19; VLL+20] |
| 8 | Backend for Frontend | Compatibility, Performance, Maintainability, Scalability | [MA19; VCG+22; VLC19; VLL+20] |
| 9 | Change Code Dependency to Service Call | Compatibility | [VLC19; VLL+20] |
| 10 | Circuit Breaker | Maintainability, Reliability, Availability | [LZJ+21; VLC19; VLL+20] |
| 11 | Competing Consumers | Maintainability | [VLC19; VLL+20] |
| 12 | Container | Maintainability, Reliability, Scalability | [LZJ+21; MA19; VLC19; VLL+20] |
| 13 | Correlation ID | Maintainability | [VLC19; VLL+20] |
| 14 | CQRS | Scalability, Performance, Maintainability, Reliability, Security, Availability | [Ric23; VCG+22; VLL+20] |
| 15 | Database is the Service | Maintainability, Reliability, Scalability, Availability | [MA19; VLL+20] |
| 16 | Database per Service | Scalability, Independence, Security | [Ric23; TLP18] |
| 17 | Deploy Cluster and Orchestrate Containers | Maintainability, Reliability | [VLC19; VLL+20] |
| 18 | Edge Server | Maintainability, Reliability | [VLC19; VLL+20] |
| 19 | Enable Continuous Integration | Maintainability | [MA19; VLL+20] |
| 20 | Event Notification | Reliability | [VLC19; VLL+20] |
| 21 | Event Sourcing | Security, Performance | [Ric23; VLL+20] |
| 22 | External Load Balancer | Maintainability, Reliability | [VLC19; VLL+20] |
| 23 | Externalized Configuration | Security, Maintainability, Scalability, Availability | [Ric23; VCG+22; VLC19; VLL+20] |
| 24 | Gatekeeper | Security | [VLC19; VLL+20] |
| 25 | Gateway Aggregation | Performance, Maintainability | [VCG+22; VLL+20] |

**Continued on next page**

**Table A.1 – Continued from previous page**

| ID | Pattern | Quality Attributes | Reference |
|---|---|---|---|
| 26 | Gateway Offloading | Security, Maintainability, Performance, Reliability, Monitorability | [VCG+22; VLL+20] |
| 27 | Health Check | Maintainability | [Ric23; VLL+20] |
| 28 | Internal Load Balancer | Maintainability, Reliability | [VLC19; VLL+20] |
| 29 | Key-Value Store | Reliability, Security | [VLC19; VLL+20] |
| 30 | Load Balancer/Load-balancing | Performance, Reliability, Maintainability | [LZJ+21; VLC19; VLL+20] |
| 31 | Local Database Proxy | Maintainability, Reliability | [VLC19; VLL+20] |
| 32 | Local Sharding-based Router | Maintainability, Performance | [VLC19; VLL+20] |
| 33 | Log Aggregator | Maintainability, Performance | [MA19; VLC19; VLL+20] |
| 34 | Microservice DevOps | Maintainability | [VLC19; VLL+20] |
| 35 | Multiple Service per Host | Scalability, Performance | [Ric23; TLP18] |
| 36 | Page Cache | Performance | [MA19; VLC19; VLL+20] |
| 37 | Pipes and Filters | Maintainability, Reliability, Performance, Scalability, Reusability | [VCG+22; VLC19; VLL+20] |
| 38 | Priority Queue | Maintainability | [VLC19; VLL+20] |
| 39 | REST Integration | Maintainability, Compatibility | [VLC19; VLL+20] |
| 40 | Result Cache | Performance | [MA19; VLC19; VLL+20] |
| 41 | Scalable Store | Maintainability, Reliability, Scalability, Availability | [MA19; VLL+20] |
| 42 | Secure Channel | Security | [VLC19; VLL+20] |
| 43 | Self-Containment of Services | Maintainability | [VLC19; VLL+20] |
| 44 | Service Discovery | Security, Maintainability | [VLC19; VLL+20] |
| 45 | Service Locator | Security, Maintainability | [VLC19; VLL+20] |
| 46 | Service Registry | Portability, Maintainability, Reliability, Understandability, Availability | [LZJ+21; TLP18; VLC19; VLL+20] |

**Continued on next page**

**Table A.1 – Continued from previous page**

| ID | Pattern | Quality Attributes | Reference |
|---|---|---|---|
| 47 | Service Registry Client | Maintainability, Reliability | [VLC19; VLL+20] |
| 48 | Shared Database Server | | [Ric23; TLP18] |
| 49 | Sidecar | Maintainability, Scalability, Security, Monitorability, Testability | [VCG+22; VLL+20] |
| 50 | Single Service per Host | | [Ric23; TLP18] |
| 51 | Strangler | Scalability, Maintainability, Compatibility | [VCG+22; VLL+20] |

**Table A.2.:** Quality Attributes Associated With Best Practices

| ID | Best Practice | Quality Attributes | Reference |
|---|---|---|---|
| 1 | Access Restriction | Security, Integrity | [LW22] |
| 2 | Account Separation | Security, Accountability | [LW22] |
| 3 | Acyclic Calls | Independence, Scalability, Reliability, Understandability, Reusability, Modifiability, Modularity, Deployability | [PS22], [SB21] |
| 4 | API-based Communication | Maintainability, Compatibility, Testability, Interoperability | [LW22] |
| 5 | Appropriate Service Relationship | Modularity, Modifiability, Confidentiality, Fault Tolerance | [PS22] |
| 6 | Authentication Delegation | Security, Authenticity | [LW22] |
| 7 | Automated Infrastructure | Maintainability, Reliability, Modifiability, Recoverability | [LW22] |
| 8 | Automated Monitoring | Reliability, Maintainability, Recoverability, Analyzability | [LW22] |
| 9 | Automated Restarts | Reliability, Recoverability | [LW22] |

**Continued on next page**

**Table A.2 – Continued from previous page**

| ID | Best Practice | Quality Attributes | Reference |
|---|---|---|---|
| 10 | Autonomous Fault Handling | Reliability, Fault Tolerance | [LW22] |
| 11 | Built-in Autoscaling | Performance, Scalability Elasticity, Resource Utilization | [LW22] |
| 12 | Cloud Vendor Abstraction | Portability, Adaptability | [LW22] |
| 13 | Coarse-Grained Microservices | Scalability, Modularity, Modifiability, Understandability, Reusability | [PS22], [SB21] |
| 14 | Communication Indirection | Compatibility, Maintainability, Modularity, Interoperability | [LW22] |
| 15 | Configuration Management | Reliability, Portability, Availability, Adaptability | [LW22] |
| 16 | Cost Variability | Performance, Resource Utilization | [LW22] |
| 17 | Data Encryption in Transit | Security, Confidentiality | [LW22] |
| 18 | Dynamic Scheduling | Maintainability, Reliability, Performance, Recoverability, Modifiability, Resource Utilization | [LW22] |
| 19 | Guarded Ingress | Reliability, Availability | [LW22] |
| 20 | Immutable Artifacts | Portability, Replaceability | [LW22] |
| 21 | Infrastructure Abstraction | Portability, Maintainability, Adaptability, Modifiability | [LW22] |
| 22 | Isolated State | Portability, Performance, Maintainability, Replaceability, Resource Utilization, Modularity | [LW22] |
| 23 | Loose Coupling | Maintainability, Modularity | [LW22] |
| 24 | Manageable Connections | Understandability, Modifiability | [PS22], [SB21] |
| 25 | Manageable Standards | Understandability, Modifiability | [PS22], [SB21] |
| 26 | Non-ESB Microservices | Scalability, Understandability, Modifiability, Reusability, Fault Tolerance | [PS22] |

**Continued on next page**

**Table A.2 – Continued from previous page**

| ID | Best Practice | Quality Attributes | Reference |
|---|---|---|---|
| 27 | Operation Outsourcing | Maintainability, Performance, Resource Utilization | [LW22] |
| 28 | Persistent Communication | Reliability, Recoverability, Modularity | [LW22] |
| 29 | Replication | Performance, Time Behavior | [LW22] |
| 30 | Resolved Endpoints | Scalability, Modifiability | [PS22], [SB21] |
| 31 | Right Cuts | Modularity, Modifiability | [PS22], [SB21] |
| 32 | Seamless Upgrades | Reliability, Availability | [LW22] |
| 33 | Secrets Management | Security, Confidentiality | [LW22] |
| 34 | Separate Persistency | Modularity, Modifiability, Confidentiality, Fault Tolerance | [PS22], [SB21] |
| 35 | Separation by Gateways | Maintainability, Reliability, Security, Reusability, Availability, Integrity | [LW22] |
| 36 | Separate Libraries | Independence, Modularity, Modifiability, Fault Tolerance | [PS22], [SB21] |
| 37 | Service Independence | Maintainability, Compatibility, Modifiability, Co-Existence | [LW22] |
| 38 | Service-orientation | Maintainability, Modularity | [LW22] |
| 39 | Sparsity | Maintainability | [LW22] |
| 40 | Standardization | Maintainability, Reusability | [LW22] |
| 41 | Standardized Deployment Unit | Portability, Installability | [LW22] |
| 42 | Use Infrastructure as Code | Portability, Maintainability Installability, Modifiability | [LW22] |
| 43 | Versioned APIs | Understandability, Modifiability | [PS22] |

Appendix B

# Refactoring Approaches

**Table B.1.:** Reviewed Publications From 2016 to 2023 (N=110)

| ID | Authors | Title | Year |
|---|---|---|---|
| 1 | Abdellatif, M. et al. | A Type-Sensitive Service Identification Approach for Legacy-to-SOA Migration | 2020 |
| 2 | Abdullah, M. et al. | Unsupervised learning approach for web application auto-decomposition into microservices. | 2019 |
| 3 | Ahmadvand, M. and Ibrahim, A. | Requirements reconciliation for scalable and secure microservice (de)composition | 2016 |
| 4 | Al-Debagy, O. and Martinek, P. | Extracting Microservices? Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach | 2020 |
| 5 | Al-Debagy, O. and Martinek, P. | A Microservice Decomposition Method Through Using Distributed Representation of Source Code | 2021 |
| 6 | Al-Debagy, O. and Martinek, P. | Dependencies-based microservices decomposition method | 2021 |
| 7 | Alaasam, A.B.A. et al. | Refactoring the Monolith Workflow into Independent Micro-Workflows to Support Stream Processing. | 2021 |
| 8 | Ali, S. T. et al. | An Approach to Break Down a Monolithic App into Microservices | 2022 |
| 9 | Amiri, M.J. | Object-Aware Identification of Microservices | 2018 |
| 10 | Assunção, W.K.G. et al. | Analysis of a Many-Objective Optimization Approach for Identifying Microservices from Legacy Systems | 2022 |

**Continued on next page**

**Table B.1 – Continued from previous page**

| ID | Authors | Title | Year |
|---|---|---|---|
| 11 | Assunção, W.K.G. et al. | A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices: An Industrial Case Study | 2021 |
| 12 | Ayisha, A. and Al-Khanjari, Z. | Achieving Loose Coupling and High Quality through Migrating the Monolithic Applications to Microservices Applications | 2023 |
| 13 | Bajaj, D. et al. | GreenMicro: Identifying Microservices from Use Cases in Greenfield Development | 2022 |
| 14 | Bajaj, D. et al. | Partial Migration for Re-architecting a Cloud Native Monolithic Application into Microservices and FaaS | 2020 |
| 15 | Baresi, L. et al. | Microservices Identification Through Interface Analysis | 2017 |
| 16 | Borges, M. et al. | Cloud restriction solver: A refactoring-based approach to migrate applications to the cloud | 2018 |
| 17 | Brito, M. | Identification of Microservices from Monolithic Applications through Topic Modelling | 2020 |
| 18 | Bucchiarone, A. et al. | A Model-Driven Approach Towards Automatic Migration to Microservices | 2020 |
| 19 | Camilli, M. et al. | Actor-driven Decomposition of Microservices through Multi-level Scalability Assessment | 2023 |
| 20 | Camilli, M. et al. | Domain Metric Driven Decomposition of Data-Intensive Applications | 2020 |
| 21 | Carvalho, L. et al. | On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices | 2020 |
| 22 | Chaieb, M. et al. | Automate migration to microservices architecture using Machine Learning techniques | 2023 |
| 23 | Chen, L. et al. | Dynamic and Static Feature-Aware Microservices Decomposition via Graph Neural Networks | 2023 |
| 24 | Chen, R. et al. | From Monolith to Microservices: A Dataflow-Driven Approach | 2017 |
| 25 | Correia, J., and Rito Silva, A. | Identification of monolith functionality refactorings for microservices migration | 2022 |
| 26 | Daghaghzadeh, M. and Babamir, S.M. | A model driven and clustering method for service identification directed by metrics | 2020 |
| 27 | Daoud, M. et al. | Automatic Microservices Identification from a Set of Business Processes | 2020 |
| 28 | Daoud, M. et al. | A multi-model based microservices identification approach | 2021 |
| 29 | De Alwis, A.A.C. et al. | Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic | 2018 |
| 30 | De Alwis, A.A.C. et al. | Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems | 2018 |
| 31 | De Alwis, A.A.C. et al. | Availability and Scalability Optimized Microservice Discovery from Enterprise Systems | 2019 |

**Continued on next page**

**Table B.1 – Continued from previous page**

| ID | Authors | Title | Year |
|---|---|---|---|
| 32 | De Alwis, A.A.C. et al. | Microservice Remodularisation of Monolithic Enterprise Systems for Embedding in Industrial IoT Networks | 2021 |
| 33 | De Lauretis, L. | From Monolithic Architecture to Microservices Architecture | 2019 |
| 34 | Dehghani, M. et al. | Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. Software and Systems Modeling | 2022 |
| 35 | Desai, U. et al. | Graph Neural Network to Dilute Outliers for Refactoring Monolith Application | 2021 |
| 36 | Dinh-Tuan, H. and Beierle, F. | MS2M: A message-based approach for live stateful microservices migration | 2022 |
| 37 | Escobar, D. et al. | Towards the understanding and evolution of monolithic applications as microservices | 2016 |
| 38 | Eski, S. and Buzluca, F. | An automatic extraction approach: Transition to Microservices Architecture from Monolithic Application | 2018 |
| 39 | Freire, A.F.A.A. et al. | Migrating production monolithic systems to microservices using aspect oriented programming | 2021 |
| 40 | Freitas, F. | Refactoring Java Monoliths into Executable Microservice-Based Applications | 2021 |
| 41 | Freitas, F. et al. | A methodology for refactoring ORM-based monolithic web applications into microservices | 2023 |
| 42 | Gao, M. et al. | Optimization of Microservice Composition Based on Artificial Immune Algorithm Considering Fuzziness and User Preference | 2020 |
| 43 | Gysel, M. et al. | Service Cutter: A Systematic Approach to Service Decomposition | 2016 |
| 44 | Habibullah, S. et al. | An Approach to Evolving Legacy Enterprise System to Microservice-Based Architecture through Feature-Driven Evolution Rules | 2018 |
| 45 | Hassan, S. et al. | Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity | 2017 |
| 46 | Haugeland, S.G. et al. | Migrating Monoliths to Microservices-based Customizable Multi-tenant Cloud-native Apps | 2021 |
| 47 | Ivanov, N | A Hot Decomposition Procedure : Operational Monolith System to Microservices | 2021 |
| 48 | Jakob, L. and Oprescu, A.M. | Steinmetz: Toward automatic decomposition of monolithic software into microservices | 2020 |
| 49 | Jamshidi, P. et al. | Pattern-based multi-cloud architecture migration | 2016 |
| 50 | Jin, W. et al. | Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering | 2018 |
| 51 | Jin, W. et al. | Service Candidate Identification from Monolithic Systems based on Execution Traces | 2019 |

**Continued on next page**

**Table B.1 – Continued from previous page**

| ID | Authors | Title | Year |
|---|---|---|---|
| 52 | Josélyne, M.I. et al. | Partitioning microservices | 2018 |
| 53 | Kalia, A. et al. | Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. | 2021 |
| 54 | Kamimura, M. et al. | Extracting Candidates of Microservices from Monolithic Application Code | 2018 |
| 55 | Kazanavičius, J. et al. | An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database | 2022 |
| 56 | Kecskemeti, G. et al. | The ENTICE approach to decompose monolithic services into microservice | 2016 |
| 57 | Khoshnevis, S. | A Search-Based Identification of Variable Microservices for Enterprise SaaS | 2022 |
| 58 | Klock, S. et al. | Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures | 2017 |
| 59 | Knoche, H. | Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices | 2016 |
| 60 | Krause, A. et al. | Microservice Decomposition via Static and Dynamic Analysis of the Monolith | 2020 |
| 61 | Levcovitz, A. et al. | Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems | 2016 |
| 62 | Li, C.-Y. et al. | Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System | 2020 |
| 63 | Li, J. et al. | A Novel Method for Identifying Microservices by Considering Quality Expectations and Deployment Constraints | 2022 |
| 64 | Li, S. et al. | A dataflow-driven approach to identifying microservices from monolithic applications | 2019 |
| 65 | Lin, J. et al. | Migrating web applications to clouds with microservice architectures | 2016 |
| 66 | Maia, P. et al. | Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures | 2020 |
| 67 | Maisto, S.A. et al | From Monolith to Cloud Architecture Using Semi-automated Microservices Modernization | 2020 |
| 68 | Mathai, A. et al. | Monolith to Microservices: Representing Application Software through Heterogeneous GNN | 2021 |
| 69 | Matias, T. et al. | Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis | 2020 |
| 70 | Mazlami, G. et al. | Extraction of Microservices from Monolithic Software Architectures | 2017 |
| 71 | Mustafa, et al. | GranMicro: a black-box based approach for optimizing microservices based app?s | 2017 |

**Continued on next page**

**Table B.1 – Continued from previous page**

| ID | Authors | Title | Year |
|---|---|---|---|
| 72 | Nakazawa, R. et al. | Visualization Tool for Designing Microservices with the Monolith-First Approach | 2018 |
| 73 | Nitin, V. et al. | CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture | 2022 |
| 74 | Nordli, E. T. et al. | Migrating monoliths to cloud-native microservices for customizable SaaS | 2023 |
| 75 | Nunes, L. et al. | From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts | 2019 |
| 76 | Park, J. et al. | An Approach for Reconstructing Applications to Develop Container-Based Microservices | 2020 |
| 77 | Paula, A. and Paula, H. De | Microservice decomposition and evaluation using dependency graph and silhouette coefficient | 2021 |
| 78 | Pigazzini, I. et al. | Tool Support for the Migration to Microservice Architecture: An Industrial Case Study | 2019 |
| 79 | Procaccianti, G. et al. | Towards a MicroServices Architecture for Clouds | 2016 |
| 80 | Raj, V. and Ravichandra, S. | A service graph based extraction of microservices from monolith services of service?oriented architecture | 2022 |
| 81 | Rathod, T. et al. | Improving Industry 4.0 Readiness: Monolith Application Refactoring using Graph Attention Networks | 2023 |
| 82 | Ren, Z. et al. | Migrating Web Applications from Monolithic Structure to Microservices Architecture | 2018 |
| 83 | Rochimah, S. and Nuralamsyah, B. | Decomposing Monolithic to Microservices: Keyword Extraction and BFS Combination Method to Cluster Monolithic's Classes | 2023 |
| 84 | Romani, Y. et al. | Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification | 2022 |
| 85 | Saidani, I. et al. | Towards Automated Microservices Extraction Using Muti-objective Evolutionary Search | 2019 |
| 86 | Saidi, M. et al. | From a Monolith to a Microservices Architecture Based Dependencies | 2023 |
| 87 | Santos, N. et al. | A logical architecture design method for microservices architectures | 2019 |
| 88 | Santos, S. and Silva, A. R. | Microservices Identification in Monolith Systems: Functionality Redesign Complexity and Evaluation of Similarity Measures | 2022 |
| 89 | Sayara, A. et al. | A probabilistic approach for obtaining an optimized number of services using weighted matrix and multidimensional scaling | 2017 |
| 90 | Schröer, C. | Towards Microservice Identification Approaches for Architecting Data Science Workflows | 2021 |
| 91 | Schröer, C. et al. | Multi-metric Approach for Decomposition of Microservice-Based Data Science Workflows | 2023 |

**Continued on next page**

**Table B.1 – Continued from previous page**

| ID | Authors | Title | Year |
|---|---|---|---|
| 92 | Sellami, K. and Ouni, A. | A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications | 2022 |
| 93 | Sellami, K. et al. | Combining Static and Dynamic Analysis to Decompose Monolithic Application into Microservices | 2022 |
| 94 | Selmadji, A. et al. | From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach | 2020 |
| 95 | Selmadji, A. et al. | Re-architecting OO Software into Microservices | 2018 |
| 96 | Shimoda, A. and Sunada, T. | Priority Order Determination Method for Extracting Services Stepwise from Monolithic System | 2018 |
| 97 | Silva, C. E. da et al. | BPM2DDD: A Systematic Process for Identifying Domains from Business Processes Models | 2022 |
| 98 | Singhal, N. et al. | Efficient hybrid research for QoS-aware microservice composition | 2019 |
| 99 | Sun, X. et al. | Expert system for automatic microservices identification using API similarity graph | 2022 |
| 100 | Taibi, D. and Systä, K. | From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining | 2019 |
| 101 | Trabelsi, I. et al. | From legacy to microservices: A type?based approach for microservices identification using machine learning and semantic analysis | 2022 |
| 102 | Tusjunt, M. and Vatanawood, W. | Refactoring Orchestrated Web Services into Microservices Using Decomposition Pattern | 2018 |
| 103 | Tyszberowicz, S. et al. | Identifying Microservices Using Functional Decomposition | 2018 |
| 104 | Vera-Rivera, F. H. et al. | Microservices Backlog - A Model of Granularity Specification and Microservice Identification | 2020 |
| 105 | Vera-Rivera, F.H. et al. | Microservices backlog ? a genetic programming technique for identification and evaluation of microservices from user stories | 2021 |
| 106 | Wei, Y. et al. | A Feature Table approach to decomposing monolithic applications into microservices | 2021 |
| 107 | Zaragoza, P. et al. | Materializing Microservice-oriented Architecture from Monolithic Object-oriented Source Code | 2022 |
| 108 | Zaragoza, P. et al. | Refactoring Monolithic Object-Oriented Source Code to Materialize Microservice-oriented Architecture | 2021 |
| 109 | Zhang, Y. et al. | Automated microservice identification in legacy systems with functional and non-functional metrics. | 2020 |
| 110 | Zhao, J. and Zhao, K. | Applying Microservice Refactoring to Object-Oriented Legacy System | 2021 |

Appendix C

# Web Application Requirements

**Table C.1.:** Requirements for Extensible Repository and Back-End (detailed)

| ID | Requirement |
|---|---|
| BE-R1 | Extensible data model<br>**Description:** An administrative user should be able to extend the predefined taxonomy with additional values. The data model needs to reflect this functionality by allowing to add new values at runtime for subcategories.<br>**Rationale:** A static data model reflecting a fixed taxonomy would be inflexible and obsolete too quickly. New migration approaches that are emerging continuously introduce new aspects that need to be reflected by the model.<br>**Priority:** 5 |

**Continued on next page**

**Table C.1 – Continued from previous page**

| ID | Requirement |
|---|---|
| BE-R2 | Approach source information<br>**Description:** The user should be able to view descriptive information about a refactoring approach. This includes its title, authors, publication year, and a link to the originating publication.<br>**Rationale:** To identify a refactoring approach and its associated publication, basic information should be made accessible.<br>**Priority:** 4 |
| BE-R3 | Readable database file<br>**Description:** An application developer should be able to read the database file by means of a suitable tool. This requires standardized identifiers for tables and additional one-to-one relations for related data.<br>**Rationale:** The database file should be readable and allow understanding its content easily. This supports maintainability and portability when dealing with several application instances.<br>**Priority:** 2 |
| BE-R4 | API for exposing functionality<br>**Description:** The application back-end should expose an API that allows to access the functionality implemented by in back-end.<br>**Rationale:** The user will utilize the web-based front-end which interacts with the back-end using its API.<br>**Priority:** 4 |
| BE-R5 | Data availability and exchange<br>**Description:** The application back-end should be able to utilize all functionality provided by the DBMS such as inserting, updating, querying, and deleting data.<br>**Rationale:** To provide a seamless operation between application front-end and repository, the application back-end should offer all functionalities provided by the DBMS. Model classes represent database entities for exchange with the application front-end.<br>**Priority:** 5 |
| BE-R6 | Data seeding<br>**Description:** The application back-end should allow to seed data from an external source into the database on application startup.<br>**Rationale:** This functionality makes the application usable after startup without additional steps to populate the database.<br>**Priority:** 3<br>**Dependencies:** BE-R5 |

**Continued on next page**

**Table C.1 – Continued from previous page**

| ID | Requirement |
|---|---|
| BE-R7 | Reflect patterns and best practices<br>**Description:** Patterns and best practices should be accommodated by the application based of a data model to create. A data set containing patterns and best practices are seeded at application start.<br>**Rationale:** Patterns and best practices represent the architecture design phase of the migration framework.<br>**Priority:** 3<br>**Dependencies:** BE-R4, BE-R6 |
| BE-R8 | Decomposition tools repository<br>**Description:** Tools that are used by refactoring approaches should be cataloged in a separate repository. Each tool should be represented by its name, author, link, and type (e.g. static analysis, decomposition, etc.). A user should be able to browse tools independently of refactoring approaches. A tool should be linked to approaches that uses the tool and vice versa.<br>**Rationale:** The automation in the course of migration to microservices is important for practitioners. For the decomposition into services, several approaches offer tool support. To make them easier accessible and allow a user to filter approaches specifically regarding their tool support, the application should maintain them separately.<br>**Priority:** 3<br>**Dependencies:** BE-R1, BE-R4, BE-R5, BE-R6 |

**Table C.2.:** Requirements for User Interface and Front-End (detailed)

| ID | Requirement |
|---|---|
| UI-R1 | Select framework phase<br>**Description:** The UI should provide a visualization of the migration framework. The user should be able to select one of the three framework phases, resulting in a navigation to the provided functionalities in that phase.<br>**Rationale:** Since the tool's functionality is structured according to the underlying migration framework, the visualization sets the context for the functionalities to expect. Moreover, the framework provides a high-level view on the process, covering also other activities that are not implemented by the web-based application.<br>**Priority:** 3 |
| UI-R2 | Show UI Mock-ups for intended, but unavailable functionality<br>**Description:** When selecting intended, but unavailable functionality of the application, the UI should provide a UI mock-up of the intended functionality.<br>**Rationale:** To conduct evaluations of the application early in the agile development process, we will show intended functionality by providing sketches of UI sequences. In particular, we want to use this method for a demonstration of the workflow in phase one of the framework. That way, we can incorporate early user feedback while keeping the implementation effort low.<br>**Priority:** 2<br>**Dependencies:** UI-R1 |
| UI-R3 | Visual feedback for unimplemented phases<br>**Description:** When selecting a not yet implemented phase of the framework, the UI should provide visual feedback similar to the mock-ups in UI-R2.<br>**Rationale:** Since we allow the user to select all three phases of the framework, a special handling for unimplemented phases should guide the user.<br>**Priority:** 1<br>**Dependencies:** UI-R1 |
| UI-R4 | List all existing approaches<br>**Description:** The user interface should allow displaying a list of all existing refactoring approaches in repository.<br>**Rationale:** Retrieving a list of all approaches allows a user to explore the entire repository and find approaches, e.g., through sorting them by tile, author, or year.<br>**Priority:** 4<br>**Dependencies:** Repo-R4 |

**Continued on next page**

**Table C.2 – Continued from previous page**

| ID | Requirement |
|---|---|
| UI-R5 | Show all information for an approach<br>**Description:** The UI should allow a user to view all information related to a specific refactoring approach, i.e. all values for the categories and subcategories according to the taxonomy.<br>**Rationale:** To assess the suitability of a migration approach for further consideration, all contained information related to an approach needs to be visible.<br>**Priority:** 5<br>**Dependencies:** Repo-R4 |
| UI-R6 | List filtered approaches<br>**Description:** The user interface should allow displaying a list approaches after filter have been applied. The filter allows three settings for each subcategory value: include, neutral, exclude.<br>**Rationale:** The main functionality in phase two of the application is to identify an migration approach based on an individual filtering. This individual filtering can be applied separately, or in addition to the quality-driven recommendation of approaches.<br>**Priority:** 5<br>**Dependencies:** Repo-R4, UI-R9, UI-R10 |
| UI-R7 | Show suitability of recommended approaches<br>**Description:** The UI should display a suitability indicator for recommended approaches, based on the applied filter settings.<br>**Rationale:** The calculation of a suitability score allows for displaying a ranking with listing the most suitable approach first. This score can be represented by a suitability indicator which will facilitate the understandability of the resulting approaches and their ranking in the result view.<br>**Priority:** 5 |
| UI-R8 | Transparency for suitability calculation<br>**Description:** The UI should make the calculation of the suitability score and indicator transparent. To do so, matching and mismatching attributes of each approach is the result list should be highlighted.<br>**Rationale:** Similar to UI-R7, the suitability score and indicator should be traceable to specific attributes of an approach based on the underlying taxonomy. Transparency and traceability of calculated results aims to improve the user acceptance.<br>**Priority:** 3 |

**Continued on next page**

**Table C.2 – Continued from previous page**

| ID | Requirement |
|---|---|
| UI-R9 | Configuration via custom filter<br>**Description:** The user interface should provide a way for users to define a custom filter on all attributes of an approach according to the underlying taxonomy.<br>**Rationale:** A filter functionality is required to allow the user to filter for approaches that fir to their system specifications.<br>**Priority:** 5 |
| UI-R10 | Strategy filter presets<br>**Description:** The user interface should provide predefined filters based on a certain migration strategy.<br>**Rationale:** Subsets of migration approaches are suitable for one of the three migration strategies *Refactor*, *Rewrite*, and *New Application*. By providing these three predefined filters, users can explore the repository based on their suitability for the overreaching migration strategy. This filtering offers a more structured overview compared to a plain list of the approaches. Likewise, user who do not have complete system specifications at hand, can explore the available approaches in a structured manner.<br>**Priority:** 3 |
| UI-R11 | Present patterns and best practices<br>**Description:** Patterns and best practices should be presented in phase three of the framework. The basic UI functionality implemented for approaches in phase two can be leveraged.<br>**Rationale:** Patterns and best practices represent the architecture design phase of the migration framework. The user should be able to browse available patterns and best practices independently and view their attributes and associated qualities.<br>**Priority:** 3<br>**Dependencies:** BE-R7, UI-R4, UI-R5, UI-R10 |

**Continued on next page**

**Table C.2 – Continued from previous page**

| ID | Requirement |
|---|---|
| UI-R12 | Architecture assessment representation<br>**Description:** The user interface should reflect an architecture assessment in phase one of the framework. In this regard, the five essential steps project description, strategic goals, quality attributes, assessment, and summary should be allow to track the results accordingly and eventually advise or advise against a migration towards microservices.<br>**Rationale:** Phase one of the framework guides through an architecture assessment (e.g. ATAM) by collecting the results during each step. The information is used for defining the scenarios that are the foundation of a quality-driven migration to microservices. Based on the suitable architecture alternatives for each scenario, the summary either advises or advises against a migration to microservices.<br>**Priority:** 3<br>**Dependencies:** QM-R1, QM-R2, QM-R4 |
| UI-R13 | Project management functionality<br>**Description:** The user should be able to save the progress, i.e., persist all information that has been entered during framework phase one. This includes project description, strategic goals, scenarios, and scenario assessment. The user interface should provide functionality to save and load a project to/from a file.<br>**Rationale:** To document an architecture assessment in phase one, specifications of the system are entered for documentation purposes and to further enable a quality-driven migration. This information needs to be persisted to allow using the application over consecutive sessions and across several instances.<br>**Priority:** 3<br>**Dependencies:** QM-R1, QM-R2, QM-R4 |
| UI-R14 | Browse tools for decomposition<br>**Description:** As for refactoring approaches, the user should be able to list all available tools, view details for each tool, and identify associated refactoring approaches.<br>**Rationale:** Similarly to refactoring approaches, we want to offer basic functionality for browsing tools offered by such refactoring approaches. Hence, we require functionality to list them, show their details, and find approaches that use a specific tool.<br>**Priority:** 3<br>**Dependencies:** BE-R8, UIA-R9 |

**Table C.3.:** Requirements for Administrator User Interface (detailed)

| ID | Requirement |
|---|---|
| UIA-R1 | Add new approaches<br>**Description:** The application should allow to add new refactoring approaches to the extensible repository.<br>**Rationale:** Adding new migration approaches is essential for keeping the repository up-to-date and is therefor crucial for the application. Alternatives would be a HTTP/HTTPS request to the API or through a direct connection to the database. To provide a reasonable degree of user-friendliness, we want to offer the functionality through the UI.<br>**Priority:** 5<br>**Dependencies:** BE-R4 |
| UIA-R2 | Edit existing approaches<br>**Description:** The application should allow to edit and update approaches that exist in the repository.<br>**Rationale:** Similar to UI-R11, the altering of existing refactoring approaches should be possible via the administrator UI.<br>**Priority:** 4<br>**Dependencies:** BE-R4, BE-R6, UIA-R1, UIA-R7 |
| UIA-R3 | Delete existing approaches<br>**Description:** The application should allow to delete approaches that exist in the repository.<br>**Rationale:** Similar to UI-R11 and UI-R12, deletion of existing refactoring approaches should be possible via the administrator UI.<br>**Priority:** 3<br>**Dependencies:** BE-R4, BE-R6, UIA-R1, UIA-R7 |
| UIA-R4 | Add new approach attributes<br>**Description:** The application should allow extending the existing taxonomy for migration approaches by adding new values to approach subcategories.<br>**Rationale:** Repo-R1 describes an extensible data model to be implemented in the application back-end. To make the functionality available to administrative users, the UI should allow specifying new subcategory values when adding a new approach to the repository.<br>**Priority:** 5<br>**Dependencies:** JF:BE-R1, BE-R4, JF:UIA-R1 |

**Continued on next page**

**Table C.3 – Continued from previous page**

| ID | Requirement |
|---|---|
| UIA-R5 | Delete existing approach attributes<br>**Description:** The application should allow deleting existing values for approach subcategories. The deletion should only be possible if these attributes are not assigned to any existing approaches in the repository.<br>**Rationale:** Equivalently to UI-R14, removing existing subcategory values should be allowed for administrative users.<br>**Priority:** 4<br>**Dependencies:** JF:BE-R1, BE-R4, BE-R6, UIA-R4 |
| UIA-R6 | Export existing approaches<br>**Description:** The application should allow exporting all existing approaches in the repository to a file in JSON format.<br>**Rationale:** Exporting all migration approaches to a file would allow making bulk changes and transfer them to a different application instance. The functionality should be available through the UI for administrator users.<br>**Priority:** 2<br>**Dependencies:** BE-R4 |
| UIA-R7 | Import approaches from file<br>**Description:** The application should allow importing approaches from a JSON file to the application's repository.<br>**Rationale:** Equivalently to UI-R16, this functionality allows making bulk changes or transfer them to a different application instance. As well, the functionality provides an alternative and potentially more efficient way to add new approaches to the repository<br>**Priority:** 1<br>**Dependencies:** BE-R4 |
| UIA-R8 | Toggle administrator and user roles<br>**Description:** The UI should allow switching user roles, i.e., switch the administrator functionalities on or off. The administrator interface should be password protected.<br>**Rationale:** We want to provide a way to switch user roles in the UI, i.e. having a switch that makes the additional functionalities for administrative users available or hides them. This requirement prepares for implementing a user management in the future. During early usability evaluations, the functionality ensures a consistent behavior of the application as accidental changes to the repository can be prevented.<br>**Priority:** 3 |

**Continued on next page**

**Table C.3 – Continued from previous page**

| ID | Requirement |
|---|---|
| UIA-R9 | Tool management functionality<br>**Description:** As for refactoring approaches, the administrator interface should allow to add new tool, edit and delete existing tools, assign them to approaches, as well as export and import tools.<br>**Rationale:** We want to handle tools offered by refactoring approaches the same way we handle approaches. Hence, we require the basic functionality to maintain them. In addition, adding a new approach should allow assigning them one or more tools that the approach uses.<br>**Priority:** 3<br>**Dependencies:** UIA-R1, UIA-R2, UIA-R3, UIA-R6, UIA-R7 |

**Table C.4.:** Requirements for Quality Model (detailed)

| ID | Requirement |
|---|---|
| QM-R1 | Quality model representation<br>**Description:** The quality model including quality attributes, sub-attributes, system properties, and their relationships to each other should be implemented.<br>**Rationale:** At application start, the quality model represented by the quality attributes, sub-attributes, and system properties should be seeded into the database. The API should provide for Create, Read, Update and Delete (CRUD) functionality for the quality model.<br>**Priority:** 3<br>**Dependencies:** |
| QM-R2 | Quality assessment model representation<br>**Description:** The model for a quality assessment in phase one of the migration framework should be accommodated by the database and accessible via the application back-end API. The API should provide for Create, Read, Update and Delete (CRUD) functionality for the quality assessment model.<br>**Rationale:** A model for scenario-based quality assessment will be the foundation for a quality-driven migration. The model should reflect the quality assessment based on ATAM, including scenario name, description, importance and difficulty fields. A subset of the available quality attributes and sub-attributes is associated with a scenario, hence attributes and sub-attributes need to be assigned to scenarios. The API should provide for Create, Read, Update and Delete (CRUD) functionality for the scenarios and quality assignments.<br>**Priority:** 3<br>**Dependencies:** QM-RB2 |
| QM-R3 | Recommendations incorporating quality assessment<br>**Description:** The recommendations for migration approaches, patterns, and best practices should be extended to reflect the quality model. Recommendations will incorporate the calculated suitability score which incorporates a weighting of quality attributes according to the quality assessment model.<br>**Rationale:** A scenario-based workflow will reflect the quality-driven migration to microservices. Based on the quality model and quality assessment model, the recommendations for refactoring approaches, patterns, and best practices in phases two and three will incorporate the qualities elicited in phase one.<br>**Priority:** 3<br>**Dependencies:** QM-RB1, QM-RB2 |

**Continued on next page**

**Table C.4 – Continued from previous page**

| ID | Requirement |
|---|---|
| QM-R4 | Creation of scenarios<br>**Description:** In phase one, the UI should allow creating scenarios based on the quality assessment model. Accordingly, scenarios should be presented with a name, description, importance, and difficulty. Furthermore, it should be possible to edit, delete, and rearrange them. As well, quality attributes and sub-attributes should be assignable to the current scenario.<br>**Rationale:** This requirement represents the UI functionality in phase one for implementing the scenario-based workflow. As a first step, a number of scenarios will be entered with their associated details and assigned qualities. The scenarios represent the outcome of a quality assessment based on ATAM.<br>**Priority:** 3<br>**Dependencies:** QM-RB1, QM-RB2 |
| QM-R5 | Scenario-based workflow for approaches<br>**Description:** In phase two, the user should be able to decide between the existing manual filtering and the scenario-based workflow that incorporates the qualities and weightings of the scenarios in phase one.<br>**Rationale:** The scenario-based workflow combines the quality assessment of phase one with additional manual filtering functionality, resulting in a more comprehensive recommendation result view.<br>**Priority:**<br>**Dependencies:** QM-RB4, QM-RF1 |
| QM-R6 | Recommended approaches reflecting suitability<br>**Description:** When using the scenario-based workflow in phase two, the recommendation view for migration approaches should have additional columns showing the calculated suitability score based on the quality model.<br>**Rationale:** In the scenario-based workflow, the user should be able to distinguish between recommendations that origin from the scenarios in phase one, and recommendations that are based on the manual filter configuration.<br>**Priority:** 3<br>**Dependencies:** QM-RF2 |

**Continued on next page**

**Table C.4 – Continued from previous page**

| ID | Requirement |
|---|---|
| QM-R7 | Scenario-based workflow for patterns and best practices<br>**Description:** In phase three, the user should be able to decide between the existing manual filtering and the scenario-based workflow that incorporates the qualities and weightings of the scenarios in phase one. An additional manual filter configuration from phase two should be adopted.<br>**Rationale:** The recommendation views for patterns and best practices should be presented equally to the the approaches in phase two.<br>**Priority:** 3<br>**Dependencies:** UI-R10.1, QM-RF2 |
| QM-R8 | Configuration overview<br>**Description:** When navigating though the different phases and related views, the currently active configuration reflecting qualities and filter settings should be permanently visible to the user.<br>**Rationale:** This way, the user keeps track of the current filter configuration which economizes the navigation and reduces the need to double-check created scenarios and applied filter settings.<br>**Priority:** 2<br>**Dependencies:** QM-RB1, QM-RB2, QM-RF1 |