MIGRATING A MONOLITHIC ARCHITECTURE TO MICROSERVICES WITH LOW-CODE: A ROADMAP FOR FINANCIAL ORGANIZATIONS

MSc. Thesis

RESEARCHER & AUTHOR – STUDENT UNIVERSITY OF TWENTE
TOM ESSERS

SUPERVISORS – UNIVERSITY OF TWENTE
DR. LUCAS O. MEERTENS
DR. JOÃO L. REBELO MOREIRA

SUPERVISOR – KPMG NL
STEFAN REIJENGA

UNIVERSITY OF TWENTE.

11-07-2025

Preface

This thesis marks the final milestone of my time as a student at the University of Twente. Over the past five years, I have had the opportunity to develop myself in the field of Business Information Technology—a programme that focuses on tackling business challenges through the use of technology. This unique combination of business and technology has fascinated me from the beginning and was the main reason I chose this field of study. It has been a journey filled with personal growth, academic development, and many valuable experiences that I will carry with me into the next phase of my career.

I would like to express my sincere gratitude to Lucas and João for their guidance and support as my supervisors during the past months. Your feedback and critical insights have been essential in shaping this thesis.

A special thanks to Stefan for always being available to help with any questions, whether professional or personal. It has been a true pleasure working with you and relaxing afterwards with some enjoyable rounds of golf 😉. I also want to thank my other colleagues at KPMG for their input, insight, and assistance, which all greatly contributed to this research.

Thank you to everyone who participated in this thesis. Your experiences, perspectives, and openness have helped shape its content and direction. Hearing your passionate views on the topic strengthened my ambition to further pursue a career in this field.

Finally, I want to thank my family, girlfriend, and friends for their continued support throughout my time at university. These last years have been filled not only with study but mostly with unforgettable moments and great fun.

This thesis is the result of insights drawn from both academic literature and interviews with experienced professionals. I hope you find it as insightful and engaging to read as I found it to write.

Tom Essers
Utrecht, 11 July 2025

Management Summary

This research proposes a roadmap for financial organizations to migrate their monolithic architecture toward microservices using low-code.

Organizations in the financial sector are experiencing rapidly changing regulatory requirements and external market pressures that force them to adapt their business models and processes quickly. Yet, the IT landscapes of many financial organizations contain many legacy IT systems that have been operational for decades, which limits their ability to be agile. These systems, which often adopt a monolithic architecture, can be migrated toward a more flexible microservices architecture that can potentially provide more maintainability and scalability.

Performing this migration, however, is a complex task. In practice, many organizations still face challenges and frequently fail in this lengthy process. This research addresses this gap by proposing a roadmap to support financial organizations in managing the migration process. The roadmap explains how low-code, an emerging approach to software development that shares technical resemblance with microservices, can be used within this process. The roadmap also highlights challenges that financial organizations can face during the migration. By making practitioners aware of these challenges throughout the process, it enables them to find ways to mitigate their impact.

This research uses literature reviews and interviews as input for the roadmap. This input was used to design an initial version of the roadmap, which was later improved by a workshop. Through expert opinion interviews, the roadmap was validated, which resulted in a final, validated design of the roadmap.

The roadmap consists of three phases and various steps within each phase. The first phase '*Analyze',* is preparation for the migration. The second phase '*Execute*', highlights an iterative approach to decompose the monolithic system toward microservices with the Strangler Fig Pattern. The third phase '*Manage*', outlines parallel management processes that must be maintained throughout the migration process.

Within the migration process, low-code can be used as a prototyping tool for user-interface-driven systems or for backend processes, and as a tool to develop microservices. The use of low-code can potentially speed up the migration as its level of abstraction speeds up the software development process. However, using low-code for this purpose brings some risks. Financial organizations should always consider whether the business case for using low-code in their organization and migration process is justifiable.

Contents

List of Figures

List of Tables

List of Abbreviations

ADM – Architecture Development Method

AI – Artificial intelligence

CF – Content Framework

DDD – Domain Driven Design

DSEC – Design Science Engineering Cycle

DSRM – Design Science Research Methodology

DT – Digital Transformation

EA – Enterprise Architecture

EAI – Enterprise Architecture Integration

ECB – European Central Bank

ESB – Enterprise Service Bus

FinTech – Financial Technology

IoT – Internet of Things

IS – Information System

IT – Information Technology

LCDP – Low-Code Development Platform

MDA – Model-Driven Architecture

MDE – Model-Driven Engineering

MVP – Minimal Viable Product

POC – Proof Of Concept

RAD – Rapid Application Development

RPC – Remote Procedure Call

RPI – Remote Procedure Invocation

RQ – Research Question

SLR – Systematic Literature Review

SOA – Service-Oriented Architecture

TOGAF – The Open Group Architecture Framework

UI – User Interface

UML – Unified Modeling Language

1.Introduction

This Chapter is an introduction to this thesis. It aims to introduce the reader to the topic, shed light on the problem at hand, and elaborate on the objectives of this research. Section 1.1 explains the context for this research. Section 1.2 outlines the research objectives, including the problem statement, objectives, and scope. Section 1.3 outlines the structure of the remainder of this thesis.

1.1 Context: Enterprise Architecture and Low-Code

In today's rapidly evolving world, it is essential that organizations adapt quickly to stay competitive and relevant. Being able to fully use technologies is a critical success factor in this. Digital Transformation (DT) – the use and integration of technologies within an organization to create value and build a competitive advantage [1], [2] – plays an essential role in helping organizations achieve this by aiming to improve their efficiency and innovation speed to gain a competitive edge over competitors. DTs are often long-term efforts and can require organizations to fundamentally change the way they operate [2]. It sometimes requires organizations to rethink their business model or to change their current business processes. Due to its complex nature, consultancy organizations assist their clients in designing and implementing DTs [3].

DTs can include adopting new technologies such as Artificial Intelligence (AI), Big Data, and Internet of Things (IoT), but can also be concerned with modernizing organizational information technology (IT) landscapes. The latter is important, as having a modern, flexible, and scalable IT landscape allows companies to more quickly adapt to the rapidly evolving world, take advantage of emerging opportunities and market trends [4], and comply with regulatory requirements [5]. Enterprise architecture (EA) is the research field focused on modernizing organizational IT landscapes. The EA discipline supports DTs by evaluating and reconfiguring IT landscapes to better align with organizational goals [4]. Enterprise architects – professionals responsible for managing an organization's IT architecture – are therefore not only concerned with the technologies used within an organization but also with optimizing business processes and ensuring their alignment with organizational objectives.

Two well-known types of IT architectures are monolithic architecture and microservices. Both architectures take different approaches to the design, implementation, and management of IT systems. Monolithic architectures are a traditional approach to the development of software and are often associated with old legacy systems.

Microservices are a more recent approach, which is often viewed as a more future-ready [6].

An emerging development approach that plays an increasingly important role in DTs is low-code development [7]. Low-code development is a software development paradigm that builds on concepts of model-driven engineering and enables developers to design, build, and deploy software applications [8]. Low-code applications can be built using low-code development platforms (LCDP), which are visual tools containing predefined components, such as UI components, business logic components, and models to build a database. As these components are predefined and the LCDP also handles the deployment of the application [9], LCDPs minimize the amount of high-code that needs to be written by developers [8]. Due to these factors, low-code is often considered a rapid development method, allowing developers to quickly build applications. This also makes low-code a fitting tool to be used for prototyping [8]. Low-code has already shown potential in building microservices applications [10], due to its close technical resemblance [9], [11], [12].

1.2 Research Objectives

This Section elaborates on the problem statement in Section 1.2.1 and objectives in Section 1.2.2. Additionally, the scope and relevance of the research are described in Section 1.2.3.

1.2.1 Problem Statement

Recently, the financial sector has seen a large-scale transition toward a more digitally powered business model. The rise of financial technology (FinTech) organizations has caused large organizations to change how they operate. Such organizations seek to use technology to improve their financial activities [13]. Additionally, financial organizations are constantly challenged by new regulations (some examples are WTP, MiCAR, DORA, PSD2, and KYC [14]) and are forced to improve their efficiency and cut operational costs [15]. This requires them to improve automation and streamline processes by combining data insights from different systems [16]. Together, these developments require organizations within the financial sector to modernize their IT landscape.

The IT landscapes of many financial organizations are characterized by legacy systems, often over 15 years old, which pose significant challenges. These legacy systems – which generally embrace a monolithic architecture – are difficult to maintain, and building integrations to other systems becomes increasingly more complex as more systems are connected [17], [18], [19]. At the same time, employees who originally developed these systems are retiring or leaving the organization, leading to a gradual loss of critical system knowledge. These problems hinder organizational agility and slow down time to market for new features [18], making it harder for these organizations to respond to the rapidly changing market and regulatory requirements. Microservice architectures might provide flexibility and maintainability for the IT landscapes of financial organizations to improve organizational agility.

Migrating existing systems from a monolithic architecture to microservices, however, is a highly complex task. Organizations performing these migrations often still face practical challenges that need to be addressed. One issue is the lack of tools that can assist experts in these strategies [20], [21], [22]. This research aims to find how low-code development, a software development approach that shares technical resemblance with microservices, can be used to assist the migration process. This research addresses the gap by introducing a practical tool in the form of a roadmap to assist practitioners in financial organizations with the migration process from monolithic architectures to microservices. It aims to address challenges that can be faced and explain how low-code can be used during the migration process.

1.2.2 Objectives

Based on the problem statement, the main objective of this research is to investigate how financial organizations can use low-code in the migration process from monolithic architecture to microservices. The goal is to create a practical tool in the form of a roadmap, as this can be used to guide organizations in the migration process.

The main objective of this research is defined as follows:

- Design a roadmap for financial organizations to migrate a system from a monolithic architecture to microservices with low-code.

This is supported by the following sub-objectives:

- Provide a state-of-the-art and comprehensive overview of enterprise architecture, monolithic architecture and microservices, and low-code. This aims to provide the readers with a solid understanding of the background of this research.
- Provide a trade-off between monolithic architecture and microservices. This aims to inform financial organizations about the benefits and challenges of both styles of architecture.
- Investigate the process for financial organizations to migrate their existing monolithic architecture toward microservices and find challenges that are currently being experienced in this process.

- Investigate to what extent – and how – low-code can be used within the process of migrating from monolithic architecture toward microservices.
- Evaluate the effects of the designed roadmap by validation.

1.2.3 Scope & Relevance

This research falls within the scope of the financial sector. This scope is especially relevant, due to the large number of legacy systems which are present in this sector, relative to other sectors [18]. The reason for this is that these organizations are often large and old, and the systems used by these organizations were built decades ago. Many financial organizations are currently too slow and need to accelerate the modernization of their digital landscape to allow for more organizational agility [23].

The financial sector has a significant impact on society as its systems store sensitive data that are critical to individuals and organizations all over the world. Financial organizations store and handle tremendous amounts of money, transactions, and financial data for potentially millions of clients that date back to the day that the systems were operational. The impact of data corruption or systems not being operational while migrating toward a new architecture can be catastrophic and should be treated highly carefully [18]. While these are solely technical difficulties, many financial organizations are spread across borders and have to comply with regulatory requirements from governments. These factors make the financial sector relevant for this research.

1.3 Outline of Work

The remainder of this research is organized as follows:

- Chapter 2 presents the approach and design of this research, detailing the research questions and research methodology.
- Chapter 3 provides the background, introducing the reader to the topics of enterprise architecture, monolithic architecture, microservices, and low-code.
- Chapter 4 focuses on problem identification, clarifying the problem context relevant to this research.
- Chapter 5 covers the treatment design, describing the development of the roadmap.
- Chapter 6 addresses the treatment validation, outlining how the roadmap was validated.
- Chapter 7 concludes the work with a conclusion, discussion, limitations, implications, and directions for future research.

2. Research Approach

This Chapter aims to describe the approach of this research. In Section 2.1 the research questions are presented. In Section 2.2, the methodologies used to answer the research questions are further elaborated upon, and the research process is described in detail.

2.1 Research Questions

To achieve the objectives stated in Section 1.2.2, a main research question is formed and is defined as follows:

MQ. How can financial organizations use low-code to migrate their existing systems from a monolithic architecture toward microservices?

This research question will be answered by the following sub-questions:

SQ1. What are the trade-offs for monolithic architecture or microservices for financial organizations?

SQ2. What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?

SQ3. How can low-code be used in migrating a monolithic architecture to microservices?

SQ4. What steps should a financial organization take to perform the migration from a monolithic architecture to microservices using low-code?

2.2 Research Methodology

This research adopts a design science research methodology (DSRM). This methodology focuses on the design and validation of artifacts within a context to achieve a set of goals [24]. Wieringa (2014) proposes a guideline for this methodology, aiming to assist practitioners in performing design science research in the information systems (IS) field [25]. This guideline will be used within this research and is further explained in Section 2.2.1. Furthermore, to obtain academic input, a mixed-methodology literature review is adopted. This includes exploratory research and systematic literature review methodologies, which are further explained in Section 2.2.2. To obtain practical insights from professionals, interviews are conducted in various parts of this research. Section 2.2.3 outlines the interview procedure and data analysis approach.

Table 1 explains the methodologies for each of the research questions, their approach and results, and the Section that answers the question.

Question	**Approach & Result**	**Section**
MQ	Conclusion from the combination of answers to sub-questions. The result is a validated roadmap highlighting the process of migrating a monolithic architecture toward microservices with low-code	7.1
SQ1	Results from a literature review of preliminary research performed by the researcher. Lists of benefits and challenges of both monolithic architecture and microservices are combined with insights specific to the financial sector	4.1
SQ2	Results from literature review and interviews. Insights from both are used to define challenges that are currently being faced in the migration process in the financial sector. These are used as input for the requirements of the roadmap	4.2.3
SQ3	Results from literature review and interviews. Insights from both are used to define different ways low-code can be used within the migration process from a monolithic architecture toward microservices	5.3.2.3
SQ4	Results from all previous sub-questions, literature reviews, and interviews. Input from all these is used to define different phases and steps within the migration from a monolithic architecture toward a microservices	5.4

Table 1 – Approach and Results for Research Questions

2.2.1 Design Science Research Methodology

The goal of a design science project is to (re)design an artifact within a problem context to improve something for stakeholders [25]. Wieringa (2014) proposes the Design Science Engineering Cycle (DSEC) as a process guideline for practitioners for performing design science research-related projects [25]. The cycle consists of four phases, namely problem investigation, treatment design, treatment validation, and treatment implementation. The DSEC is displayed in Figure 1. It is important to mention that, while it is part of the DSEC, the treatment implementation phase is excluded, as implementation of the artifact within the problem context is outside of the scope of this research.

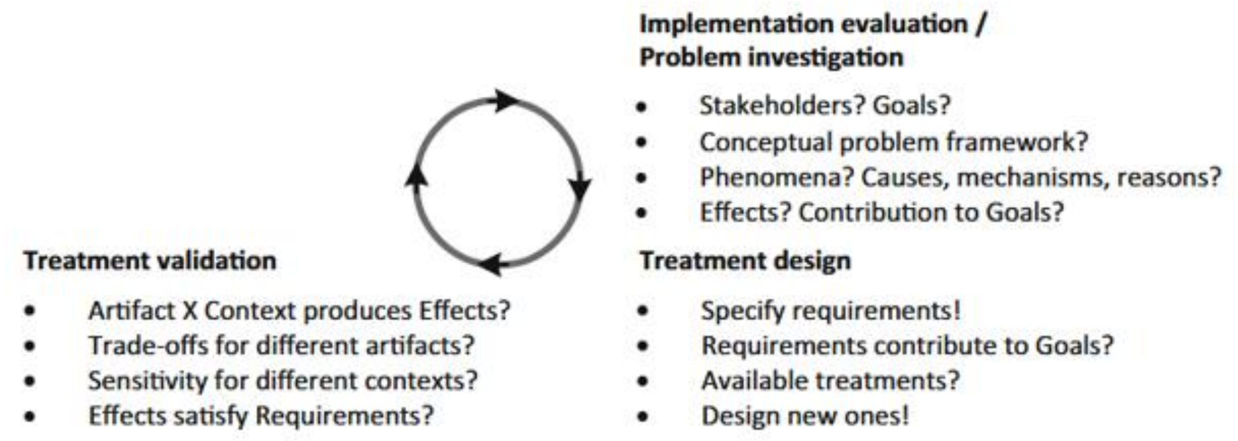
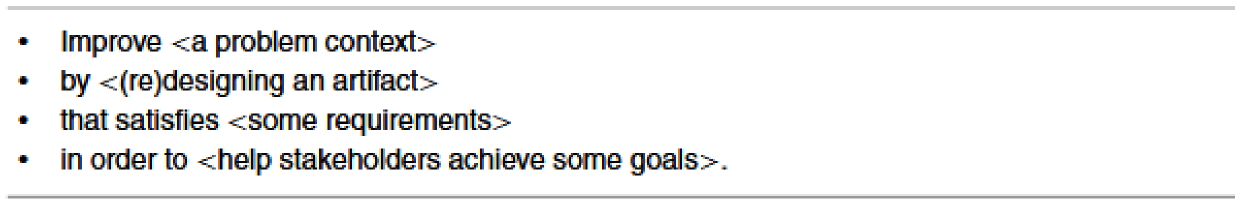

Implementation evaluation / Problem investigation
• Stakeholders? Goals?
• Conceptual problem framework?
• Phenomena? Causes, mechanisms, reasons?
• Effects? Contribution to Goals?
Treatment design
• Specify requirements!
• Requirements contribute to Goals?
• Available treatments?
• Design new ones!
Treatment validation
• Artifact X Context produces Effects?
• Trade-offs for different artifacts?
• Sensitivity for different contexts?
• Effects satisfy Requirements?

Figure 1 – Design Science Engineering Cycle (from [25])

The designed artifact in this research has the form of a roadmap. To fully formulate the design problem, the template proposed shown in Figure 2 is used. This aims to clarify the problem context for which the artifact is designed, and formulate the requirements and the goals it aims to achieve.



- Improve <a problem context>
- by <(re)designing an artifact>
- that satisfies <some requirements>
- in order to <help stakeholders achieve some goals>.

Figure 2 – Design Problem Template (from [25])

This research aims to:

Improve	*monolithic to microservices migration strategies in financial organizations*
by	*designing a roadmap*
that	*demonstrates how financial organizations can use low-code in the process and addresses challenges during the migration*
in order to	*support financial organizations migrate their legacy monolithic systems to microservices with low-code*

In Figure 3, the process of this design project and the tasks performed within each of the phases is outlined. Each of the three phases is further elaborated upon.

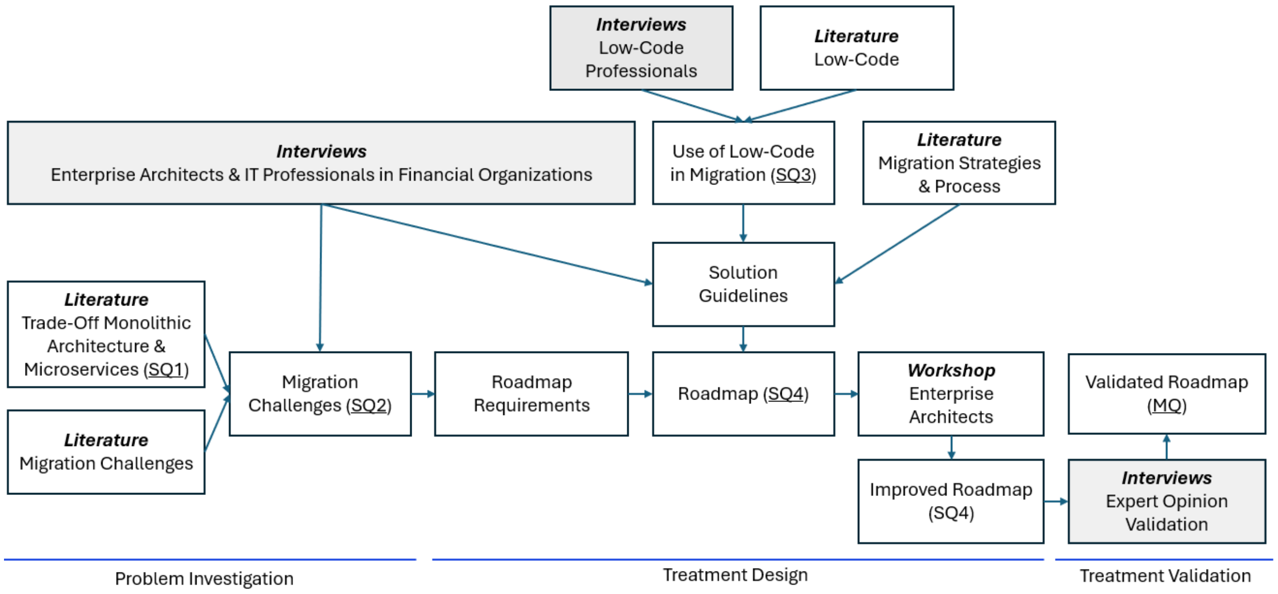


Figure 3 – Process for Design Research

Problem Investigation

The problem investigation is the first phase of the DSEC and aims to explore the problem context that the design research aims to solve. In this phase, knowledge questions – questions that ask for knowledge about something – help define the problem at hand [25]. SQ1 (*What are the trade-offs for monolithic architecture or microservices for financial organizations?*) is a knowledge question that will be answered by a literature review in the form of an exploratory research and a systematic literature review. This research question helps define the problem.

The second knowledge question that is answered during this phase is SQ2 (*What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?*). This question will be answered by performing an exploratory literature review, by using input from SQ1, and by conducting interviews with enterprise architects and IT professionals in financial organizations. Section 2.2.3 elaborates on how these interviews are conducted, and how the data gathered is analyzed.

The insights gathered in the literature reviews and interviews serve as input for creating a structured definition of the problem context [25] and also help define requirements for the design of the roadmap.

Chapter 4 elaborates on the problem investigation.

Treatment Design

The second phase of the DSEC, the treatment design, focuses on specifying the requirements and designing the artifact. In this phase, the roadmap is created from the roadmap requirements and solution guidelines.

Roadmap requirements are obtained from insights from the problem investigation phase. These requirements are based on challenges that are currently being faced by organizations in the financial sector in migrating from a monolithic architecture to microservices.

Solution guidelines serve as a foundation for the creation of the roadmap. These guidelines consist of four parts:

- Input from migration strategies and their process from an exploratory literature review.
- Input about migration strategies and their process from interviews with enterprise architects and IT professionals in financial organizations.
- Input about the use of low-code for the migration process from an exploratory literature review.
- Input about the use of low-code for the migration process from interviews with enterprise architects, IT professionals in financial organizations, and low-code professionals.

The latter two answer SQ3 (*How can low-code be used in migrating a monolithic architecture to microservices?)*.

The combination of the roadmap requirements in the form of challenges and the solution guidelines will together will be used together to define the phases and steps for the migration process. This answers SQ4 (*What steps should a financial organization take to perform the migration from a monolithic architecture to microservices using low-code?*). These phases and steps are combined into an initial version of a roadmap. To improve this roadmap, a workshop is held with enterprise architects. They are asked to provide feedback on the roadmap, which is used as input for a revised and improved version.

Chapter 5 elaborates on the treatment design.

Treatment Validation

Treatment validation is the last phase of the DSEC and aims to validate the artifact that has been designed in the previous phase by imagining what will happen when this artifact is used in the problem context [25]. To do this, expert opinions are used. In interviews, various experts are introduced to the improved design of the roadmap and are asked to imagine how this will interact within the defined problem context. [25]. The experts are asked to give their opinions on the roadmap. The roadmap is then validated, and potential points of improvement will be documented and can be used as input for future work.

Chapter 6 elaborates on the treatment validation.

2.2.2 Literature Review Methodology

The literature reviews incorporate both exploratory and systematic literature review methodologies. This Section outlines these methodologies in Section 2.2.2.1 and Section 2.2.2.2 respectively. Part of the performed literature review is part of preliminary work previously performed by the researcher.

2.2.2.1 Exploratory Research

According to Dash (2019), the intention of exploratory research methodology should be to explore [26] a topic at hand. This research methodology is used for obtaining the background for this research, exploring the field of EA and monolithic architecture and microservices, exploring challenges faced in the migration process, and exploring existing migration strategies. Additionally, exploratory research is used to obtain a state-of-the-art into low-code and its applications. This flexible research approach allows a comprehensive understanding of the aforementioned domains. The insights are used in the problem investigation and treatment design phase.

For the exploratory research, the following databases are used:

- Google Scholar;
- Scopus;
- IEEE Xplore.

These databases together provide a wide range of potentially relevant papers. Google Scholar indexes the most documents compared to any other database, but does not have advanced search facilities. Scopus does provide these search facilities, and together with IEEE Xplore, indexes much computer science-related research [27].

The search queries used within each of the databases for the exploratory research for each of the usages above can be found in Appendix A.

2.2.2.2 Systematic Literature Review

Okoli (2015) defines a systematic literature review (SLR) within the information systems field as “a systematic, explicit, [comprehensive,] and reproducible method for identifying, evaluating, and synthesizing the existing body of completed and recorded work produced by researchers, scholars, and practitioners.” [28]. The SLR will be used to assist in answering SQ1 (*What are the trade-offs for monolithic architecture or microservices for financial organizations?*). This research methodology allows for a systematic search through the literature to create a comprehensive overview of benefits and challenges for both types of architecture, which allows for a more accurate trade-off.

Okoli (2015) proposes an eight-step framework – based on different prominent publications within the IS field, such as Kitchenham et al. (2007, 2009 & 2013) publications on guidelines for performing an SLR for software engineering [29], [30], [31], and Wolfswinkel et al. (2013) publication on the Grounded Theory approach [32] – that is used as a guideline in this research. These eight steps are displayed in Figure 4 [28]. This SLR was performed in preliminary work, performed by the researcher, and its process and protocol are described in detail in Appendix B.

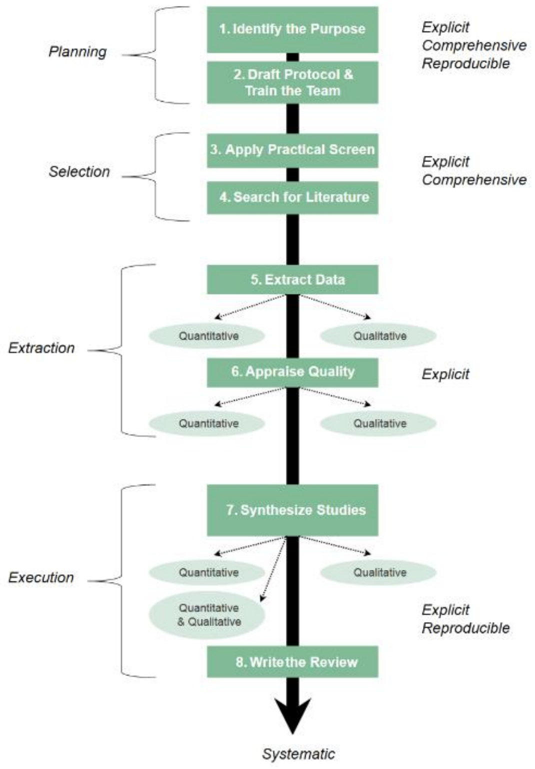


Figure 4 – Eight-Step Framework for Performing an SLR (from [28])

2.2.3 Interviews

Interviews are used to obtain input in various stages of this research. In Section 2.2.3.1 the goals and procedure for the input interviews are explained. The method used to analyze the input data is elaborated upon in Section 2.2.3.2. Section 2.2.3.3 describes the procedure of the workshop. Section 2.2.3.4 describes the procedure of interviews for the validation with expert opinions.

2.2.3.1 Interview Goals and Procedure

Interviews can be used to gather valid and reliable data from experts that is relevant to answering a research question [25], [33]. The interviews conducted in this research are semi-structured and one-on-one. Interviewees are carefully selected to have the required knowledge about the subjects discussed in this research. A predefined list of interview questions is defined to guide the interview, but the interviewer has the opportunity to explore particular themes or responses further. Interviews are conducted either in-person or online via Microsoft Teams.

The list of interview questions, together with an informed consent form, is sent to the interviewees before the interview. Interviewees are asked to sign this informed consent form and return it to the researcher. This research is approved and follows instructions from the BMS ethics committee of the University of Twente [34]. Before the interview, the interviewees are asked whether the interview could be audio-recorded, and depending on their answer, they were or were not. Interviews are transcribed using Transkriptor [35]. Transcriptions made with tools are validated by the researcher. Interviewers are referred to in text by their interviewee ID to ensure anonymity.

Interviews are held with industry experts from three different groups, namely enterprise architects, IT professionals in financial organizations, and low-code professionals.

As Figure 3 displays, interviews are conducted in the problem investigation and treatment design phases of this research. The goal of the interviews in the problem investigation phase is to obtain insights into the challenges found in migrating a monolithic architecture to microservices. These interviews are held with enterprise architects and IT professionals in financial organizations. Their insights are particularly valuable to this research, as they have hands-on experience with such migrations and have knowledge of existing systems within financial organizations. These interviews aim to identify practical challenges that are experienced by financial organizations that might not be fully captured in existing literature.

Table 2 displays the list of interviewees from the enterprise architect group. In Table 3, the interviewees from the IT professionals in financial organizations group are displayed. The tables include the sector they work in, their function role, and organization size (micro: < 10 employees, small: 10-49 employees, medium-sized: 50-249 employees, large: > 250 employees [36]). The latter is relevant, as experience with legacy monolithic systems is more likely to occur in large, older financial organizations, in comparison to smaller ones.

Interviewee ID	**Sector**	**Function Role**	**Size**
EA1	Consulting	Senior Manager Digital Architecture	Large
EA2	Consulting	Manager Digital Architecture	Large
EA3	Consulting	Manager Digital Architecture	Large
EA4	Consulting	Senior Consultant Digital Architecture	Large
EA5	Consulting	Senior Manager Software Architecture	Large
EA6	Consulting	Director Digital Enablement	Large

Table 2 – List of Interviewees in Enterprise Architects Group

Interviewee ID	**Sector**	**Function Role**	**Size**
FO1	Banking	Strategic Advisor (Recently Chief Architect)	Large
FO2	Banking	Lead Enterprise Architect	Large
FO3	Banking	Data Analyst	Large
FO4	Pension	IT Architect	Large
FO5	Insurance	Enterprise Architect	Large

Table 3 – List of Interviewees in IT Professionals in Financial Organizations Group

A predefined list of questions can be found in Appendices D and E.

After conducting several interviews for the problem identification, the interviewer notices that no new information is being obtained. Everything shared is already covered by previous participants. This suggests that a sufficient number of interviews have been conducted for this part.

The goal of interviews in the treatment design phase is to obtain insights into the process of migrating from monolithic architecture to microservices, which is part of the solution guidelines. The input for this is obtained during the same interviews as in the problem investigation phase for practical reasons. These interviews are held with the same professionals from the enterprise architects and IT professionals in financial organizations groups. Additionally, in this phase, interviews are conducted to obtain insights into the possible use of low-code in the monolithic to microservices migration process. These interviews are conducted with low-code professionals. Interviewees from the enterprise architects and IT professionals in financial organizations groups that have experience with low-code are, where possible, also asked for their opinions on this.

In Table 4, the interviewees from the low-code professionals group are displayed. The table includes the sector they work in, their function role, and experience with low-code. EA4 and FO3 are excluded from this part of the research, due to their lack of experience with low-code.

Interviewee ID	**Sector**	**Function Role**	**Low-Code Experience**
LC1	Low-Code Consulting	CTO	OutSystems Expert
LC2	Low-Code Consulting	Consultant & Architect	Mendix Expert
LC3	Low-Code Consulting	Senior Consultant & Solution Architect	Mendix Expert
LC4	Consulting	Head of IT Solutions	OutSystems Champion
LC5	Consulting	Manager Digital Enablement	OutSystems Associate

Table 4 – List of Interviewees in Low-Code Professionals Group

A predefined list of questions can be found in Appendices D, E, and F.

After conducting several interviews for the treatment design, the interviewer notices that no new information is being obtained. Everything shared is already covered by previous participants. This suggests that a sufficient number of interviews have been conducted for this part.

2.2.3.2 Data Analysis

The data collected within the interviews for input is analyzed using a thematic analysis approach, a common approach for analyzing qualitative interview data [33]. Maguire & Delahunt (2017) have a practical step-by-step coding guide for doing a thematic analysis of interview data [37]. This guide is used in this research and consists of the following six steps:

1) Become familiar with the (transcript) data.

2) Generate initial codes – Initial codes should be assigned to the raw data, following an open coding methodology.

3) Search for themes – Find patterns in the data and group them into themes (or categories) and similar concepts.

4) Review themes – Revise the found themes and sub-themes.

5) Define themes – Final refinement of the themes and concepts.

6) Write-up – Write down the final results [37]. The results of the interviews are discussed in Sections 4.2.2, 5.3.1.2, and 5.3.2.2.

2.2.3.3 Workshop

To refine the initial roadmap design, a workshop is conducted with 3 enterprise architects. The purpose of this session is to present the first version of the roadmap and the accompanying document, review the process in detail, and identify which elements are correct and which should be revised. The participants are first exposed to the full roadmap, after which each of the different phases and steps, and their descriptions, are presented. The feedback gathered from the participants is used in a second version of the roadmap, which is validated afterward. The workshop is transcribed using Transkriptor [35].

Throughout the workshop, for each of the phases and steps, they are asked the two questions: ‘What is correct about the process?’ and ‘What can be improved?’.

The participants for the workshop have been carefully selected for their knowledge and experience with similar migration processes. Table 5 displays the list of enterprise architects attending the workshop, together with the interviewee ID previously assigned to them. The workshop participants are referred to in the text by their workshop ID to ensure anonymity.

Workshop ID	**Sector**	**Function Role**	**Interviewee ID**
WS1	Consulting	Manager Digital Architecture	EA2
WS2	Consulting	Manager Digital Architecture	EA3
WS3	Consulting	Senior Consultant Digital Architecture	EA4

Table 5 – List of Participants for Workshop

2.2.3.4 Validation

To validate the roadmap, interviews are held with experts. These interviews are structured and one-on-one. The interviewees are asked to imagine the effects of the roadmap if it were to be used in practice. Feedback from the interviewees and potential points of improvement are documented and can be used as input for future work. The validation interviews are transcribed using Transkriptor [35].

The goal of the validation is to determine whether the designed artifact would contribute to the stakeholders when implemented in the problem context [25]. One way to validate the roadmap is by expert opinion. The design of the artifact is shown to experts who are asked to imagine how the artifact – the roadmap – will interact within the problem context, namely monolithic to microservices migration strategies in financial organizations [25]. By conducting one-on-one interviews with the experts, they can directly provide feedback on the roadmap. Using expert opinion for validation is a highly popular method, and most commonly used in design research to investigate the applicability and usability of the design [38]. Expert opinion interviews allow for an in-depth understanding of the reasoning behind an expert's opinion and enable the interviewer to clarify any misunderstandings. The advantage of using this approach is that it is easier to obtain open feedback on the roadmap and specific phases and steps. When using an expert opinion approach, it is critical that the experts fully understand the artifact [25]. To achieve this, before the validation interview, the experts are sent the roadmap together with the accompanying document, a significant time before the interview is conducted.

In the interviews, the experts are asked to comment on the designed roadmap across three categories defined by Wieringa (2014) [25]:

- Effect: What are the effects produced by the artifact within the context?
    - Does the roadmap assist migration strategies in financial organizations?
    - What changes should an organization make to adopt this roadmap in their current migration strategy?
- Sensitivity: What effects are produced by the artifact in another context?
    - Could this roadmap be used by organizations that are not in the financial sector?
    - Are there any organizational environments in which (e.g. size) the roadmap would have a different impact?
    - What type of legacy monolithic systems is this roadmap best suited for?
- Requirements satisfaction: Does the artifact satisfy the requirements (Section 5.2)?
    - Can the roadmap be perceived as easy to understand?
    - Do these phases and steps accurately highlight how financial organizations should perform the migration process?
    - Does the roadmap highlight the challenges that organizations can face during the migration process?
    - Does the roadmap provide a clear explanation of how low-code can be used within the migration process?

Wieringa (2014) also considers the 'trade-off' (How do different artifacts perform in the context?) as a validation category [25]. However, as one of the challenges defined in Chapter 4 is a lack of tools (thus other artifacts), this category is excluded from the questions asked in the validation interviews.

The list of questions that are asked to each of the experts is based on the abovementioned categories. A predefined list of these questions, based on the aforementioned questions, can be found in Appendix G. As these interviews are also semi-structured, the interviewer may deviate from these questions.

The list of participants for the validation can be found in Table 6. At least one participant from each of the three defined interview groups has been selected to have experience in all topics discussed in this research. As the roadmap is created for organizations within the financial sector, participants are primarily selected from this sector. Some validation interviews are conducted with people previously interviewed for input for the design of the roadmap. For these participants, the interviewee ID previously assigned to them is displayed. The validation participants are referred to in the text by their validation ID to remain anonymous. The organization size is also documented (micro: < 10 employees, small: 10-49 employees, medium-sized: 50-249 employees, large: > 250 employees [36]). This is relevant to compare differences in sensitivity based on organization size.

Validation ID	**Sector**	**Function Role**	**Size**	**Interviewee ID (if applicable)**
VD1	Consulting	Senior Manager Digital Architecture	Large	EA1
VD2	Low-Code Consulting	Senior Consultant & Solution Architect	Medium	LC3
VD3	Insurance	Lead Enterprise Architect	Large	-
VD4	Insurance	Enterprise Architect	Large	-
VD5	Manufacturing (previously insurance)	System Architect	Large	-
VD6	Banking	Lead Enterprise Architect	Large	FO2
VD7	Insurance	Integration Architect	Large	-
VD8	Leasing	Senior Enterprise Architect	Large	-

Table 6 – List of Interviewers for Expert Opinion Validation

3. Background: Enterprise Architecture and Low-Code

This Chapter aims to provide the reader with background information relevant to the topics discussed in this thesis. In Section 3.1, enterprise architecture and relevant methodologies and tools used within the field are discussed. Section 3.2 provides an introduction to monolithic architecture and microservices. In Section 3.3, an introduction to low-code is provided.

3.1 Enterprise Architecture

EA is a discipline within the field of computer science that focuses on designing and managing an organization's structure. IEEE Standard 1471-2000 defines 'architecture' as: *"the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principle guiding its design and evolution."* [39]. Building on this, EA addresses architecture at the level of an entire organization. This research adopts the definition of EA defined by Lankhorst et al. (2005): "*a coherent whole of principles, methods, and models that are used in the design and realisation of an enterprise's organisational structure, business processes, information systems, and infrastructure*." [39].

EA operates at the intersection of an organization's strategy and operations (Figure 5). This is important to consider, as focusing exclusively on optimizations of a specific operational process could lead to cost reductions, but could harm the performance of the organization as a whole. A good EA, therefore, balances the optimization of daily operations and aligns them with strategic organizational goals [40].

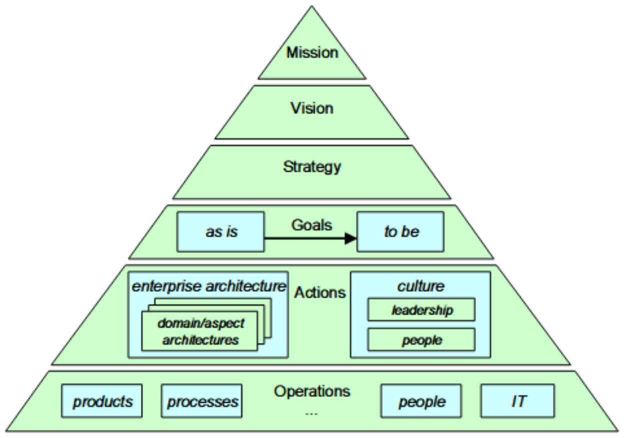


Figure 5 – Placement of Enterprise Architecture within an Organization (from [39])

EA is both a process and a product. The product – an important technical document in the form of an architecture – enables an organization's managers to align their business processes with their policies and strategy. The document should provide an integrated, holistic view offering a comprehensive overview of the whole organization. Many stakeholders are not directly interested in the architecture document itself, but rather in the effects of the architecture on the organization [40].

The role of an enterprise architect is to translate organizational requirements into architecture. To achieve this, input should be gathered from a diverse group of stakeholders across the organization, and potentially from external sources as well. Enterprise architects can make use of different artifacts such as the architecture document, models, views, presentations, and other analysis tools to communicate to the relevant stakeholders (Figure 6) [39].

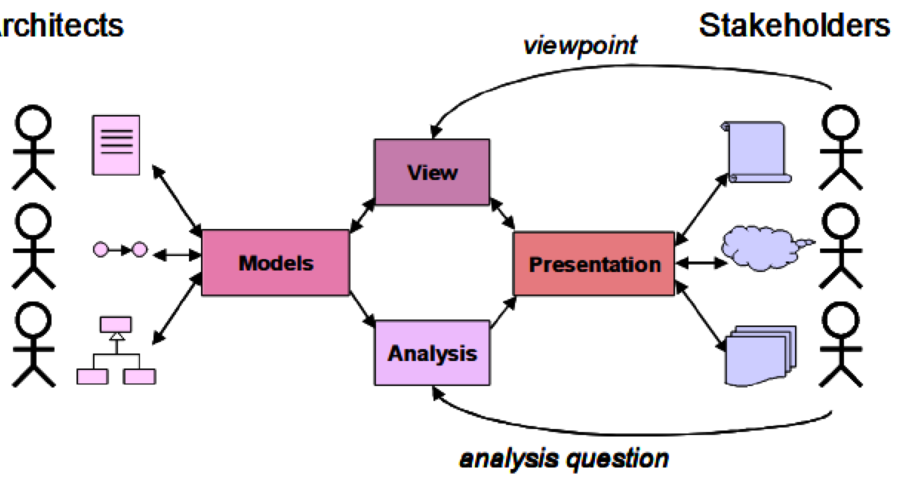


Figure 6 – Communication between Architects and Stakeholders (from [37])

Managing these artifacts is a dynamic process that, in this rapidly evolving environment, allows organizations to keep innovating and becoming more efficient. The architecture document, which describes an organization's current architecture, should always be kept up-to-date, as this allows enterprise architects to identify necessary changes quickly [40]. Additionally, an organization can create potential future architectures to explore various scenarios for their DTs. This is beneficial for the following reasons:

- EA can assist in assessing the organization's readiness for a DT [41].
- EA can assist in capturing the value of a future DT [4].
- EA can assist in reviewing the strategy of a future DT [41].
- EA can help promote a common, shared understanding of a DT at the enterprise level [4].

- EA can help systematically develop, plan, implement, and manage future structures [4], [41].

As the latter mentions, enterprise architects are not only involved in developing the (future) architecture documents, but are also involved in the planning, and implementation of the architecture. Executing EA-related DTs is a complex process. Different EA methodologies and tools have been proposed to assist in this process. The next Section will shed light on some of those methodologies and tools.

3.1.1 Methodologies and Tools

Over the years, different methodologies have been used by researchers and consultants to assist organizations in their EA. These frameworks describe methods for designing architectures and their alignment with organizations [39]. They realize both the creation of the product and the performance of the process of EA by:

- Facilitating as a component specification tool for building the architecture document (product).
- Serving as planning (process) and problem-solving tools for performing the DTs [42] (Figure 7).

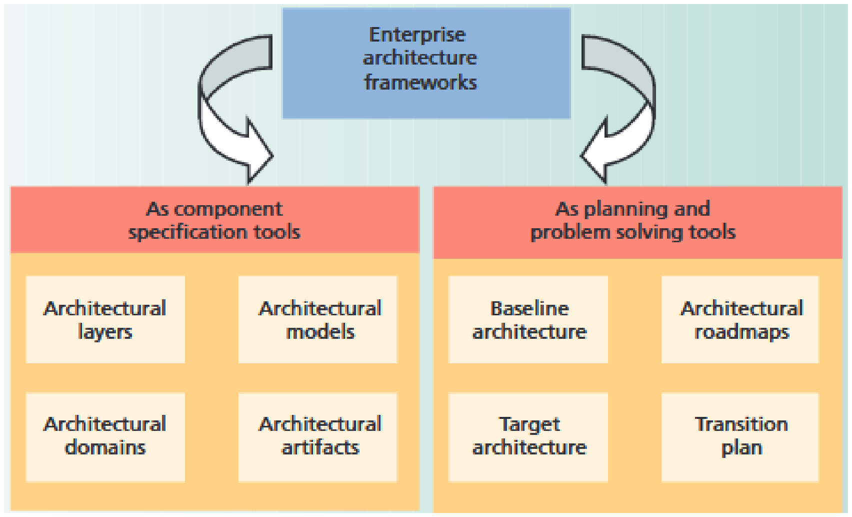


Figure 7 – Role of Enterprise Architecture Frameworks (from [42])

The component specification tool consists of architectural building blocks that can be used to create and present the architecture document. It consists of different architectural layers (often the business, application, and technology layers), models, domains, and artifacts that can be used by the enterprise architect [42]. These building blocks aim to represent different real-world objects and architectural elements, relevant to organizations, so that an enterprise architect can create a fitting representation of their organization.

The planning and problem-solving tool entails the baseline (current) architecture and possible target (often also called solution) architectures. It proposes a framework that can be used to transition from the baseline toward the target architecture by creating roadmaps and transition (or migration) plans. The roadmap highlights milestones, phases, and steps that need to be performed before reaching the target architecture [42].

The Zachman Framework, published by John Zachman in 1987 [43] is seen as one of the first EA frameworks to be used by EA professionals [42]. Over the years, the Zachman Framework was revised and updated. The framework was popular due to its ease of use and its independence from other tools and methodologies [39]. A drawback of the framework was its lack of commonly accepted standards for architectural descriptions [40].

Other frameworks, such as OMG's Model-Driven Architecture (MDA), aimed to solve this issue by complementing existing EA frameworks using modelling standards such as the Unified Modeling Language (UML) [39]. While initially, the OMG was primarily useful for software design, its framework was extended to also include business aspects of organizations and thus make more impact on an organizational level [39].

3.1.1.1 The Open Group Architecture Framework

Nowadays, the most well-known and most widely used EA methodology is The Open Group Architecture Framework (TOGAF). The TOGAF standard was developed by The Open Group and its members based on the Technical Architecture Framework for Information Management (TAFIM), developed by the US Department of Defense [44]. At the core of TOGAF lies the TOGAF Architecture Development Method (ADM) Cycle. The ADM serves as the planning and problem-solving tool of the TOGAF methodology. ADM is a generic method for architecture development and explains the EA process and the different stages and activities that an enterprise architect should follow [45]. It describes the process of moving from a baseline architecture toward a target architecture. Figure 8 displays the different stages of the ADM cycle.

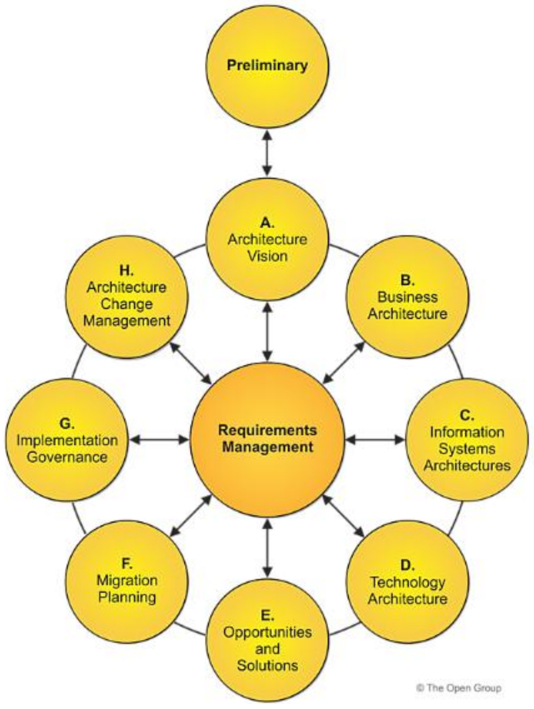

Preliminary
A. Architecture Vision
B. Business Architecture
C. Information Systems Architectures
D. Technology Architecture
E. Opportunities and Solutions
F. Migration Planning
G. Implementation Governance
H. Architecture Change Management
Requirements Management
© The Open Group

Figure 8 – TOGAF Architecture Development Method Cycle (from [45])

Next to the ADM, TOGAF provides a Content Framework (CF) that defines the structure for the component specification tool. The CF, displayed in Figure 9, is structured in accordance with the phases of the ADM. The CF consists of a collection of models that can describe an organization’s architecture. The Enterprise Metamodel defines the entities that appear in these models and the possible relationships between them.

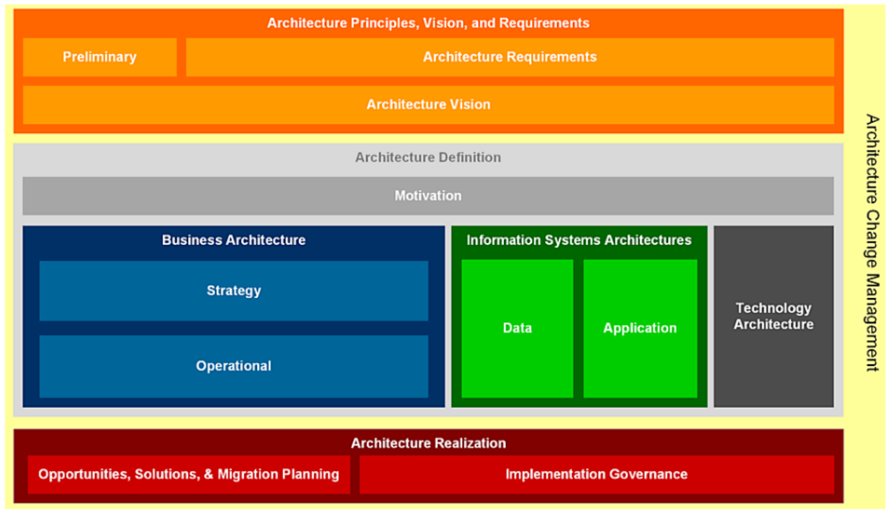

Architecture Principles, Vision, and Requirements
Preliminary
Architecture Requirements
Architecture Vision
Architecture Definition
Motivation
Business Architecture
Strategy
Operational
Information Systems Architectures
Data
Application
Technology Architecture
Architecture Realization
Opportunities, Solutions, & Migration Planning
Implementation Governance
Architecture Change Management

Figure 9 – TOGAF Content Framework (from [45])

The Architecture Definition Document is the main deliverable from the CF. It contains models from four different architecture domains from the architecture, namely business, data, application, and technology. It includes models from all relevant states, referred to as baseline, transition, and target architectures [45]. While the CF identifies relevant architectural building blocks, it does not provide a way of notating them [46]. To bridge this gap, an enterprise modeling language can be used.

3.1.1.2 Enterprise Modeling Language ArchiMate

While TOGAF aims not to constrain an enterprise's modeling language [45], the standardized and most popular language used to create the different models is the ArchiMate modeling language [47], [48]. As ArchiMate is an enterprise modeling language, it is positioned at the organizational level. It therefore provides a less detailed level of design compared to languages such as BPMN (which focuses on business processes) and UML (used for designing software applications) [46]. ArchiMate is aligned with the TOGAF CF and offers entities and relationships that can be used by an enterprise architect. The ArchiMate full framework is displayed in Figure 10.

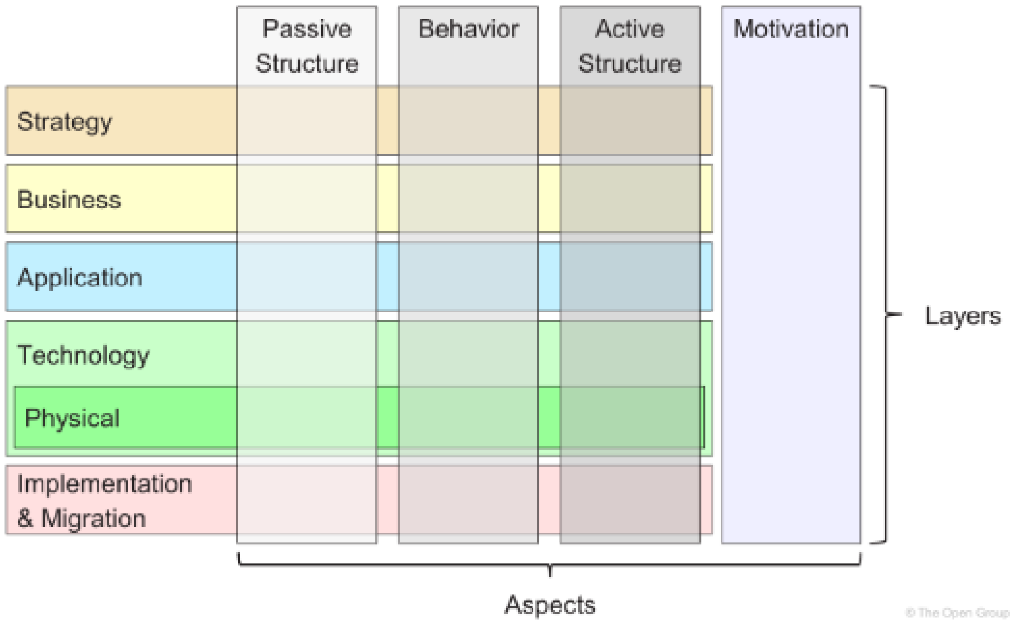


Figure 10 – ArchiMate Full Framework (from [48])

The ArchiMate full framework consists of two dimensions, namely layers and aspects. The layers describe the organizational level domains that can be modeled. While the core framework consists solely of the business, application, and technology layers, the full framework also includes the physical, strategy, and implementation & migration layers.

The aspect dimension describes the different facets of the architectural elements. The passive structure aspect focuses on elements that are being manipulated, created, or consumed. The behavior aspect focuses on activities, processes, and functions that occur. The active structure aspect focuses on elements that perform the behavior. Usually, an element from the active structure performs a behavior on an element from the passive structure. Additionally, a motivational aspect can be used throughout all layers of a model. This is useful as it allows an enterprise architect to explain the motivation or reasons for the design or change of an EA [47].

In addition to the different elements from the various aspects and layers, ArchiMate defines a core set of relationships that can connect a predefined source element to a target element. Relationships between elements in different layers are common.

TOGAF and ArchiMate together allow organizations to explore and implement different architectural approaches that suit the organization’s goals and principles. In Section 3.2, two architectural approaches, namely monolithic architecture and microservices will be explained, and examples of both architectures will be modeled using the ArchiMate modeling language.

3.2.2 Enterprise Architecture Integration

For systems to be useful within an organization, they need to be able to communicate with other systems. Communication between systems is realized by integrating them. Enterprise Architecture Integration (EAI) refers to the use of architectural patterns to integrate IT systems, applications, data, and processes within an organization. While there are many ways to integrate IT systems, there are four top-level EAI patterns: File transfer, shared database, remote procedure invocation (RPI), and messaging [49].

Enterprise architects are responsible for designing and implementing appropriate integration patterns to properly integrate different systems. This is important, as the choice of integration impacts the performance, scalability, and maintainability of the system. Each EAI pattern has its trade-offs, and enterprise architects must assess organizational goals and requirements to design effective integrations. ArchiMate can play a role in designing integrations within an organization’s IT landscape as it allows enterprise architects to model EAI patterns between systems. These are typically relationships between elements in the application layer.

The abovementioned EAI patterns are frequently used within monolithic architecture and microservices. The next Section will elaborate on how these patterns are implemented within both architectural types.

3.2 Monolithic Architecture and Microservices

This Section provides an explanation of monolithic architecture and microservices.

3.2.1 Monolithic Architectures

Monolithic architecture is a traditional architectural approach. In monolithic architectures, a system's components are combined into a single, inseparable infrastructure. Such systems often consist of 3 components, namely a user interface (UI), business logic, and database, which are all maintained and deployed as one system [50]. A monolithic system is referred to as a tightly coupled system because its different components are heavily dependent on one another [51]. Additionally, all code for this system is written in the same codebase. While this design approach is widely used in the industry, and many existing systems are built with a monolithic architecture, literature often refers to it as a legacy system's architecture [52]. Figure 11 displays an illustration of a monolithic architecture. Here, the resource management is realized by a DBMS, while in practice, monolithic architecture can also store their data in files.

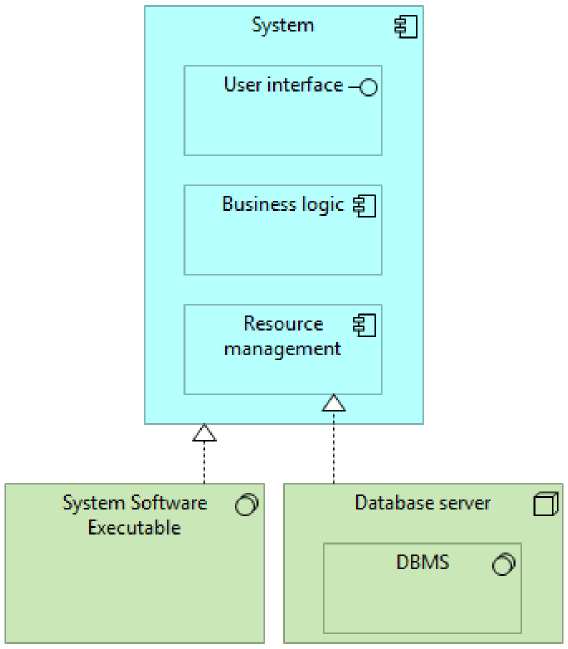


Figure 11 – Illustration of a Monolithic Architecture

When monolithic systems integrate with other systems, due to their tightly coupled design, they must interface as a single unit. Integrations with other systems are achieved through adapters, which facilitate communication between the systems [53]. As the ecosystems in which these monolithic systems are positioned grow, the number of integrations and adapters to other systems will increase. Because the entire system is interfaced as a single unit, a change to the monolithic system can result in required updates to all integrations and adapters.

An example of when this is happening is if monolithic architecture integrations are realized by a shared database. A shared database is a frequently used EAI pattern, where multiple applications make use of the same database. Because of this, they have access to the same data, which is consistent over the systems and can thus serve as a single source of truth [54]. The database designer is responsible for creating a database that suits all connected systems. If a change is made to the database, all systems that are connected to that database may require updates, making it a tightly coupled pattern [54].

Monolithic architecture also frequently uses the file transfer EAI pattern. File transfer allows files of data to be transferred from one application (producer) to another (consumer) [55]. Different underlying communication protocols, such as a network or even manual transportation between devices, realize this pattern. This pattern is quick and easy to set up, but its long-term robustness is questionable [55]. The reason for this is that they often process batches between intervals, instead of offering real-time insights, which often results in security vulnerabilities [55].

3.2.2 Microservices

Microservices are an architectural style that decomposes a system into a set of services called microservices [53]. This architectural style was first proposed in 2011 [56] and is built on the concepts of Service-Oriented Architecture (SOA). SOA was introduced in the early 2000s and aimed to modularize systems by splitting the logic of different business functions into services, which are programs that run independently from one another. The services, however, often used the same underlying database [57] and were integrated through an enterprise service bus (ESB) [58], which was responsible for the communication between different services. While SOA was expected to become the standard for building applications and replace the monolithic architecture, it received a lot of criticism due to its many abstractions, use of legacy protocols, and difficulties in maintaining the ESB [59].

Microservices architectures build upon the concepts of SOA but aim to reduce abstraction by more strictly defining their concepts [59]. Microservices are completely independent in development and deployment [56], and each microservice therefore has its own business logic and own database. Microservices are usually smaller than traditional services and aim to support interoperability by sending lightweight messages to one another, rather than connecting through an ESB [60]. While it is not strictly defined how large a microservice should be, microservices should only relate to one business capability or sub-domain. Each microservice should thus be able to work as an independent unit performing some business capability or sub-domain [59]. As each microservice can run independently from other microservices, they are more loosely coupled compared to SOA and monolithic architectures.

In reality, a system that uses many microservices is only functional if the microservices covering all business capabilities are well-integrated. As microservices work independently, the system that uses the different microservices becomes a distributed system with a network of microservices. Microservices often communicate with each other by performing RPI calls, also called a remote procedure call (RPC). In this EAI pattern, a system can invoke a publicly exposed service synchronously over a network. The client of a system performs a service call to a published service server [61]. The server then responds with data or a message depending on the action requested. In the case of microservices, the standard used approach is using RESTful APIs, which are a lightweight implementation of RPC that allows microservices to access other microservices' resources (data) [62]. Another EAI pattern used for microservices is messaging, which allows for asynchronous communication between the microservices [63]. These are often implemented with event-driven queues.

Users can access a microservices system through a UI. This UI is often connected to an API gateway that handles tasks such as load balancing and user authentication [53], making the implementation of a microservices architecture more practical. This gateway will direct the requests to the necessary microservices to perform the actions inputted by the user. Based on the action performed, one microservice might need to communicate with multiple other microservices. An illustration of a microservices architecture is displayed in Figure 12. While microservices frequently use a database for data persistence, this does not need to be the case. In practice, a microservices architecture also can – and often does have – multiple UIs.

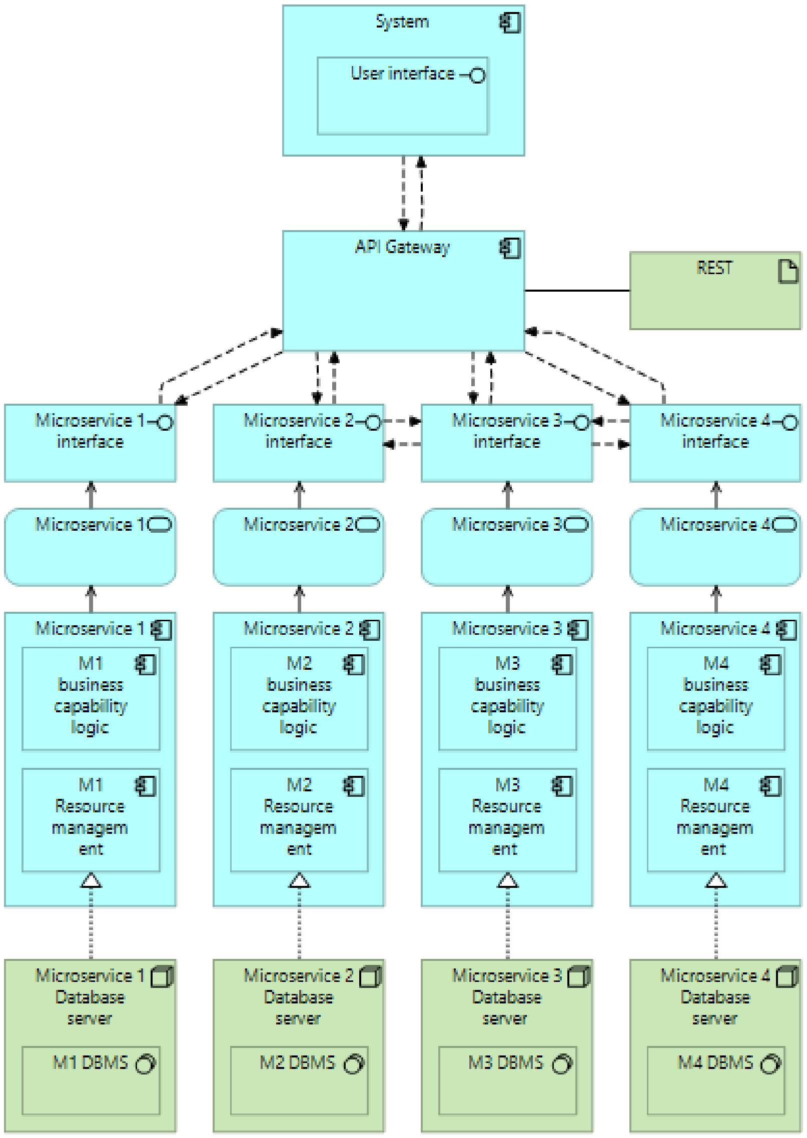


Figure 12 – Illustration of a Microservices Architecture

3.3 Low-Code

This Section provides an introduction to low-code, an important concept in software development and part of the treatment design in Chapter 5.

Many software applications are developed by writing code, referred to as high-code, using programming languages such as Python, Java, C#, or COBOL. High-code is written in human-readable plain text and allows developers to instruct a computer to perform tasks such as processing data, executing logic, and storing information [64]. These instructions form the technological backbone of many IT systems. Developers working with high-code must have an understanding of their programming language, including its specific concepts, structures, and syntax.

Low-code development is an emerging approach to software development that enables developers to create software applications without necessarily having to write high-code. Low-code applications, on the other hand, are built using LCDPs, which are development environment tools often sold as platform-as-a-service solutions [9]. They aim to significantly reduce the amount of high-code a developer needs to write (if any) to develop an application, as they use model-driven tools for the entire application's technology stack [9], [65]. Some popular LCDPs are Mendix, OutSystems, Microsoft Power Platform, and Appian (Figure 13).

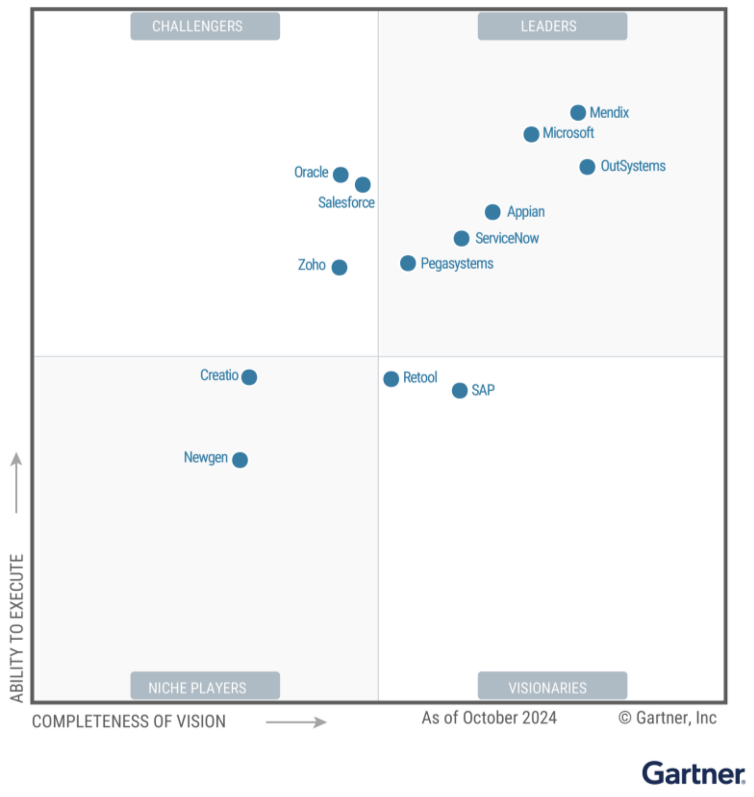


Figure 13 – Gartner's Magic Quadrant for Enterprise LCDPs (from [66])

LCDPs contain built-in features such as a graphical user interface, live collaborative development support, business logic mechanisms, deployment support, and monitoring support [12]. With this, LCDPs aim to be solutions for managing the entire software development lifecycle, including its development, deployment, and monitoring.

The first LCDPs can be traced back to 2011, but the official term low-code was not used until 2014 [67]. Low-code has emerged from different industry trends, such as Rapid Application Development (RAD) and Model-Driven Engineering (MDE). RAD was a response to traditional software development methodologies and put emphasis on the quick development of prototypes, giving early insights into the usability of software products. MDE emphasizes using models as first-class artifacts in the development lifecycle [12]. These models help in representing a system and can assist in various parts of software development. They increase productivity by automating these steps. MDE relies on domain-specific languages (DSLs) that can express high-level abstractions to the problem domain. Many LCDPs use models in the form of drag-and-drop components to facilitate software development in, for example, creating database models, defining logic, and creating UIs. These high-level abstractions are often translated to high-code by the LCDP, so the user of the platform does not have to write high-code themselves. An example of creating such a database model in Mendix (domain model) is displayed in Figure 14. Mendix draws on concepts from UML – in line with the MDA approach defined by OMG (see Section 3.1.1) – as its implementation of a DSL for defining database tables and their relationships, allowing for visual representations of the data structure.

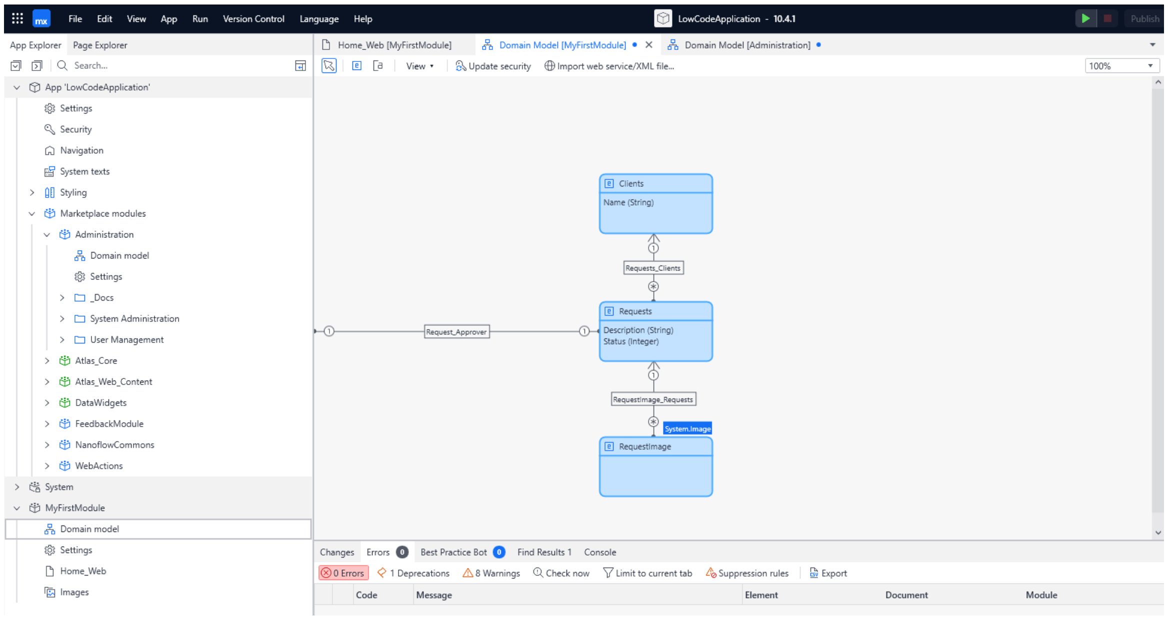


Figure 14 – Domain Model in Mendix Studio Pro (version 10.4.1)

Many researchers claim that due to the higher level of abstraction, low-code is generally easier to understand and use, allowing non-IT professionals (so-called citizen developers) to build applications [7], [9]. This can help organizations cope with building increasing demand for applications and automation, while not needing to hire qualified high-code developers [68]. Other researchers are more skeptical about this because they think that non-IT professionals lack the required knowledge to build such applications [69], or that this practice introduces significant risks, such as security risks and shadow IT [8], [70].

Despite this ongoing debate, low-code is gaining traction. Gartner estimated that by 2025, 70% of newly developed applications will use some form of low-code or no-code (LCDP without the possibility to write high-code) technology [71]. The primary reason for low-code adoption is that it speeds up the development of software systems [9], [72]. LCDPs offer everything needed to create a system in one platform, contain reusable components, and – while they do support it – minimize the need for using high-code. Due to these factors, companies building systems can quickly create prototypes, or minimum viable products (MVPs), to quickly gather customer feedback for future development, allowing organizations to more quickly react to the market demand and rapidly changing requirements [73]. To support this, the low-code development lifecycle is generally tightly aligned with agile principles [9].

4. Problem Investigation: Challenges in Migrating a Monolithic Architecture to Microservices

This Chapter elaborates on the problem context for this design research. The aim of this Chapter is to define the problem that the design research aims to solve. In Section 4.1, benefits and challenges of monolithic architecture and microservices are discussed, and a trade-off between them is created. In Section 4.2, challenges found in migrating a monolithic architecture to microservices are discussed.

The results from Section 4.1 and 4.2 will be used as input for the treatment design of the roadmap.

4.1 Trade-Off Between Monolithic Architectures and Microservices for Organizations in the Financial Sector

Monolithic architectures and microservices are inherently different from one another. Monolithic architectures are centralized, often tightly coupled applications, while microservices are distributed systems that are generally more loosely coupled. Both architectural styles provide their benefits and challenges, and a trade-off can be made between the two. In this Section, this trade-off is discussed, which answers SQ1 (*What are the trade-offs for monolithic architecture or microservices for financial organizations?*). The benefits and challenges have been obtained from a preliminary literature review performed by the researcher, namely an SLR, for which the process is explained in Section 2.2.2.

The benefits and challenges associated with monolithic architecture are displayed in Table 7. Each of the benefits and challenges is further elaborated upon in Appendix C.

Benefits	Governance	Easier development for small applications ([56], [58], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84])
		Quicker time-to-market ([76], [78], [80], [81], [82], [83])
		Requires less experienced developers ([76])
	Operations	Easier monitoring ([53], [60], [63], [64], [86])
	Single infrastructure	Fewer dependencies ([77])
		More code reuse ([74], [84])

		Same technology stack ([83])
	System performance and evolution	Better maintainable for small applications ([77], [79], [81], [83])
		No network overload ([56], [58], [76], [77], [83])
Challenges	Governance	Complexity due to codebase size ([56], [81], [86])
	Operations	Difficult troubleshooting & monitoring ([75], [86])
	Single infrastructure	Accidentally affect other components ([58], [77], [79], [84], [87])
		Application-wide-bugs and failures ([56], [74], [76], [77], [78], [81], [82], [85])
		Deployment in entirety ([56], [58], [74], [75], [76], [77], [79], [81], [82], [83], [84], [86], [87], [88])
		Language/technology lock-in ([56], [58], [74], [75], [76], [81], [85], [86])
	System performance and evolution	Limited scalability ([56], [74], [75], [76], [77], [78], [79], [81], [82], [87])
		Poor maintainability ([58], [74], [75], [76], [77], [84], [85], [87])
		Suboptimal performance ([26], [54], [60], [65])

Table 7 – Benefits and Challenges of Adopting a Monolithic Architecture

The benefits and challenges associated with microservices architecture are displayed in Table 8. Each of the benefits and challenges is further elaborated upon in Appendix C.

Benefits	Governance and agility	Alignment with CI/CD ([56], [58], [75], [80], [83], [84], [87], [88])
		Foster innovation ([74], [78], [79], [87], [89])
		Less individual complexity ([58], [81])
		Self-managed teams ([56], [74], [79], [84])
		Technology modularity ([56], [58], [75], [76], [77], [78], [79], [81], [84], [87], [88], [89])
	Microservice isolation	Data isolation ([81])
		Fault isolation ([56], [74], [76], [77], [79], [81], [84], [88], [89])
		Isolated audit, monitoring, releasing, and testing ([56], [74], [75], [76], [80], [87], [88])
	System performance and evolution	Better reliability ([81], [82], [86], [89])
		Better resilience ([74], [81], [85], [87], [89])
		Easier maintainability ([74], [75], [76], [77], [79], [82], [83], [85], [87], [89])

		Increased availability ([74], [82], [86], [87], [89])
		More flexibility ([56], [58], [75], [77], [80], [84], [89])
		More scalability ([56], [75], [76], [77], [78], [79], [80], [81], [83], [84], [86], [87], [88], [89])
Challenges	Complexity of distributed systems	Complex monitoring ([77], [79], [85], [88], [89])
		Complex security ([26], [51], [54], [57])
		Complex troubleshooting & testing ([56], [75], [77], [77], [79], [81], [85], [88])
		Data consistency & management ([74], [75], [77], [80], [82], [83], [85], [86], [87], [88], [89])
		Integration management ([56], [77], [88], [89])
	Governance	Complex governance ([75], [76], [77], [78], [79], [81], [88])
		Requires experienced developers ([75], [76])
	Operations	Operational overhead ([77], [81], [89])
	System performance	Network overhead ([56], [74], [75], [76], [77], [84])

Table 8 – Benefits and Challenges of Adopting a Microservices Architecture

Organizations within the financial sector are facing issues related to the agility, maintainability, and scalability of their IT systems. As identified in Table 7, these challenges are found to be associated with systems that adopt a monolithic architecture. Specifically, language/technology lock-in, limited scalability, and poor maintainability are challenges that financial organizations are facing. This can be explained as many operational systems in the financial sector have a monolithic architecture [16], [17], [18], [90].

The systems that financial organizations started developing decades ago now seem to be creating issues. One relevant example of this is the technology lock-in with monolithic architecture. Many financial monolithic systems have been built using high-code languages that were popular decades ago, such as COBOL and Fortran. From a survey among banks in the United States, 43% of banking systems are found to be built with COBOL [91]. However, this language is no longer frequently used, and knowledge about it is currently scarce, in comparison to newer programming languages like Java [75]. Experts who helped develop these systems are often no longer working at these financial organizations or are retired, making it very difficult for current employees to understand and update the existing systems.

Most benefits identified with monolithic systems are relevant for smaller and newer systems. These include ease of development, quick time-to-market, requiring less experienced developers, and better maintainability for small applications. It therefore made sense for financial organizations to use this architectural style for these systems decades ago. However, these benefits are currently largely irrelevant for as their IT landscapes predominantly consist of already existing, large, and complex systems. One notable benefit of monolithic applications experienced by financial organizations is the absence of network overload. This is especially relevant for financially based computations, as exposing these to latencies can lead to worries for customers using their systems.

Meanwhile, on the other hand, microservices seem to be a promising solution to address these challenges. Some of the benefits presented in Table 8 that are particularly relevant for issues faced in the financial sector are better maintainability, more flexibility, more scalability, and technology modularity. These benefits are directly associated with organizational agility and could help financial organizations future-proof their IT landscapes. By adopting microservices architecture, financial organizations can, therefore, transform their IT infrastructure to be able to more quickly adapt to compliance and external pressures.

Microservices, however, also bring challenges that need to be addressed. They introduce significant technical challenges, such as ensuring data consistency and managing integration over the microservices network, which requires a different design approach compared to monolithic systems. Operational tasks such as monitoring, troubleshooting, and testing require more complex setups to be able to handle the full network of microservices. The same is true for ensuring security across the network.

Besides technical difficulties, microservices introduce governance complexities. They support different governance of the teams working on the microservice compared to monolithic systems, and often require more skilled developers with expertise in designing and implementing the distributed system, as they are inherently more complex.

It needs to be mentioned that microservices are not a silver bullet [92], [93]. Adopting a microservices architecture does not automatically make your system more scalable or maintainable [93]. To make a microservices implementation viable, organizations need to carefully plan their migration and commit significant resources. Much research considers monolithic architecture as generally old and that it should be avoided, while microservices are the answer to the above-mentioned problems. However, not all financial organizations have the resources to develop, manage, and maintain microservices. For example, financial organizations with few experienced developers might experience issues with consistency and reliability when needing to build a distributed system [93]. Microservices will not benefit all financial organizations, and neither will it always be beneficial to migrate a monolithic system to microservices. Organizations also need to assess the alternatives for adopting microservices. For that reason, the business case for microservices needs to be clear [92].

Microservices thus seem to be an architectural style that organizations in the financial sector can benefit from. They allow financial organizations to be more organizationally agile and flexible, improve the maintainability of their IT landscape, and thus potentially solve the issues they are currently facing. However, microservices come with significant technical and organizational complexities that need to be taken into consideration and managed. This answers SQ1 (*What are the trade-offs for monolithic architecture or microservices for financial organizations?*).

4.2 Monolithic Architecture to Microservices: Migration Challenges

Due to the complexity of the migration, financial organizations migrating from a monolithic architecture to microservices will experience challenges. In this Section, these challenges are identified. The challenges are obtained from both literature reviews and interviews. This Section aims to answer SQ2 (*What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?*). The results from the literature review are discussed in Section 4.2.1. The results from the interviews are discussed in Section 4.2.2. Section 4.2.3 combines the insights from both literature and interviews into one holistic overview containing challenges in the migration from monolithic architecture to microservices.

4.2.1 Literature: Challenges in Migration

In this Section, challenges in migrating a monolithic architecture to microservices are identified from literature. This aims to assist in partially answering SQ2 (*What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?*).

Velepucha and Flores (2021) [22] explore different challenges that are experienced in the process of migrating an existing monolithic architecture to microservices. The results of their SLR are summarized in Table 9. Each of the challenges will be further elaborated upon.

Challenges in Migrating a Monolithic Architecture toward Microservices
Data Consistency Across Microservices
Governance Changes
Identification and Design of Microservices
Lack of Suitable Tools, Frameworks, and Methods
Validation of Microservices Architecture

Table 9 – Challenges in Migrating a Monolithic Architecture toward Microservices (from [22])

Data Consistency Across Microservices

One of the key challenges in designing a microservices network is data consistency. In a monolithic architecture, data is typically stored in a single, centralized database. However, after migrating to microservices, data becomes distributed across multiple databases. Where the system's data was first stored in a single database, it is now spread across various microservices [22] (Section 4.1). To ensure that data is consistent between all microservices, especially in scenarios involving network failures, microservice failures, or data corruption, it is essential to adopt a good design for data replication and redundancy [94].

To maintain data consistency, correctness, and reliability in databases, the ACID properties – Atomicity, Consistency, Isolation, and Durability – are essential. These principles define how data operations across one or more databases should be executed as a single unit of work (called a transaction) [95]. A common example in the financial sector is the transaction of money from one bank account to another. In such cases, it is crucial that all steps in the transaction are completed successfully or none at all. This ensures that all involved databases remain in a consistent state.

Literature proposes various approaches to address the ACID principles [94], however, this is inherently more difficult to design and implement in comparison to using a single database. Due to this, designing the data consistency approaches and data migration requires more experienced architects and developers.

Governance Changes

Organizations face governance challenges when migrating their monolithic architecture to microservices. The migration includes organizational obstacles, such as a change in mindset from centralized governance to governance of a distributed system [96]. In practice, this often implies a change of a traditional waterfall mindset toward working with autonomous teams applying agile methodologies [97]. This requires organizations to set up cross-functional teams, break horizontal barriers and internal silos, and require cultural changes [98]. It thus impacts the full organization. Monolithic systems are usually developed by multiple teams. Splitting up these teams and making different teams responsible for individual microservices also increases the importance of communication between the different teams, as the microservices are part of a network.

Additionally, as distributed systems require more knowledge about architecture and come with CI/CD, an organization might need to hire new, highly skilled architects, developers, or train their employees [22].

Identification and Design of Microservices

Another challenge is the identification of microservices and the design of the network. In the process of migrating a monolithic architecture to microservices, the large monolithic system is split up into microservices. To do this, the monolithic system needs to be analyzed, and the design of the new microservices system needs to be created [96]. Poniszewska-Maranda et al. (2023) conclude that especially the step of breaking up the monolith, so-called decomposing it (further elaborated upon in Section 5.3.1.1), is a highly complex task for organizations, especially when the database is tightly coupled [99]. Literature shows that finding the places to break up the monolithic application into individual microservices is often seen as the most complex task in the process [100], [101]. Properly defining how to decompose the monolith and design the microservices is difficult, but it has a great impact on the outcome of the system. Organizations also struggle with deciding which parts of the system to break up first [22].

Lack of Suitable Tools, Frameworks, and Methods

One of the biggest limitations in migrating a monolithic architecture to microservices is the lack of tools, frameworks, and methods to support the migration [20], [22]. This includes practical frameworks to assist practitioners within the process. Literature has aimed to bridge this gap by proposing standardized migration strategies, which are explored in Section 5.3.1.1, but there are inconsistencies between the different research. There is also a lack of automated tools to assist in finding where to decompose the monolithic system. In practice, this process is almost always done by the experience of the architects and developers [102]. While some advanced techniques and tools have recently been proposed in literature to mitigate this issue, the problem with these tools is that they frequently require experts to use and are often not practical (or validated in practice) [52], [103], [104]. Other stages in the migration that lack an absence of tools are the prototyping phase [105], and the validation phase.

Validation of Microservices Architecture

As microservices are not a silver bullet, organizations need to validate whether microservices can potentially solve the issues that they are currently facing [22]. Not every monolithic application should be migrated to microservices. Before initializing the migration, the organization must create a business case to determine whether migrating is the right decision. This validation is not straightforward but can be done by properly defining a business case [22]. Business drivers for migrating toward a microservices architecture for financial organizations include seamless integration of their service delivery platform, process and infrastructure flexibility, and efficiency gains [106]. Organizations, however, have to create their own business case to determine the value that microservices can bring to their organization.

4.2.2 Interview Results: Challenges in Migration

In this Section, challenges in migrating a monolithic architecture to microservices are identified from interviews. This aims to assist in partially answering SQ2 (*What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?*).

With a thematic analysis (Section 2.2.3.2) the challenges of migrating a monolithic architecture to microservices for financial organizations from the interviews are categorized into five themes. These are displayed in Table 10. Each of these themes and the challenges within them will be further elaborated upon.

Challenge Categories in Migrating a Monolithic Architecture toward Microservices
Business Case Definition
Complexity of Existing Monolithic System
Governance and Resistance
Lack of Knowledge and Support
Risks and Business Continuity

Table 10 – Challenges in Migrating a Monolithic Architecture to Microservices from Interviews

Business Case Definition

Creating a business case for migrating a monolithic architecture to microservices for financial organizations is a challenging task. This has several reasons.

The costs for performing the migration are often unclear. The reason for this is that it is hard to determine how much effort needs to be put into migrating a system, and what the costs are in the process (EA1). This is worsened, as there are many uncertainties during the process, which makes it more likely to become more expensive than initially estimated (EA1, EA6). Yet, the costs associated with the migration are often the most critical factor when it comes to creating a business case (FO5). Performing the migration takes serious effort and is likely to cost a lot of money (FO5).

On top of this, the need to migrate a working system without adding any value to the system is not always understood by the business. When migrating a monolithic system toward microservices, the functionalities of the system are not altered, and no value is added from the perspective of the system's users (EA1). The business, which is often responsible for project management office and portfolio management, might give a lower priority to such projects and frequently does not understand the need for a large budget to do this. EA2 points out that the business does not understand the need for the migration, and explains that they consider this an IT problem: Why rebuild an application that is already running? EA6 also discusses this. They explain that business sometimes compares it to the current coming out of the outlet: *'Why would it [the system] suddenly stop working?'*. Adding value to the system by adding or updating functionalities could improve the adoption of the business case.

Additionally, defining the requirements for the migration's business case and measuring them is difficult (EA5). The migration aims to make the organization's system more future-proof, but does not directly add value to the system. As explained in Section 4.1, microservices can bring more flexibility, maintainability, and scalability, compared to monolithic architectures. Several business drivers were also mentioned in the last Section. However, the effects of non-functional requirements are hard to measure and operationalize, making it difficult to measure the direct impact of the migration (EA1). Yet, knowing the impact of the migration is critical to determine whether the business case for microservices is feasible.

Lastly, many financial organizations have limited developer capacity to perform this migration. EA1 notes that developers already have to update the system to comply with regulations, receive new requirements for updates, and keep the systems operational. Because of these tasks, the backlog for these organizations is often already full. These tasks are often considered more important than performing the migration, making the business case less attractive.

Complexity of Existing Monolithic System

Many systems with a monolithic architecture within the financial sector are complex systems with many functionalities and integrations. Multiple interviewees used Gartner's three application layers [107] to differentiate between different types of systems within the financial sector (EA2, EA3, EA5, EA6, FO2), namely systems of innovation, systems of differentiation, and systems of record [107].

These interviewees point out that systems with a monolithic architecture are most frequently systems of record and sometimes systems of differentiation. These are information storage systems that serve as core systems for these organizations. An example is a core banking transaction system or insurance policy administration system.

These monoliths are complex systems that, in some cases, have been operational for decades (EA1, FO1, FO5). Often, they are classified as legacy systems. Due to the high pressures in regulation and natural organizational changes over time, these systems have been incrementally updated and patched (EA1). Because of this, the monoliths contain much functionality and many integrations with other systems. This makes them complex, which poses some issues during the migration.

In the process of migrating toward microservices, it frequently happens that a newly decomposed microservice needs to integrate with the old monolith. In practice, this often comes with complications, as monolithic legacy systems often do not have the proper interfaces to connect to newer systems. FO3 gives an example of an instance where the microservice required data from the monolithic system, which only supported physical file transfer. If this happens, updates to both systems might need to be made, just to facilitate the migration process. The same is true for the existing systems that are already integrated with the monolithic system. These integrations and interfaces all have to be updated, as updates are made to the monolith (FO1). During the migration, these integrations might need to be revised several times or phased out.

One of the major issues of a monolithic system is the tightly coupled components and integrations (FO1, FO2). Different functionalities within the same system can become so coupled that it becomes really complex to decompose the monolith. This situation is worsened when there are connections or integrations that are unknown to the system's architects and developers. FO2 explains that this is especially relevant for integrations that run incidentally, or infrequently, such as year reports. In those cases, it can happen that during the migration process, some system stops working at some point, and it can be difficult for the developers to discover how this issue has arisen.

Many legacy monolithic are difficult to test. When these systems were originally developed, automatic tests were often not implemented (FO2). As a result, during the migration process, making changes to the system can introduce faulty code without the awareness of the developers, or it requires developers to create tests during the migration process.

What further complicates the migration process is the large amount of data that needs to be migrated toward the microservices' databases. The existing monoliths often have been operational for decades and contain a lot of important data (EA2). Migrating the data that is stored in one single database to many databases can require a complex data migration.

Governance and Resistance

Migrating a monolithic system to microservices is likely to create governance challenges and resistance from within the organization. There are multiple reasons for this.

As a monolithic system is migrated to microservices, the governance of the system changes. Large monolithic systems are often developed by several development teams, who are working on the same system simultaneously (FO1, FO2, EA1). When the system is migrated toward microservices, the governance of the system needs to change. In practice, this frequently means that small teams become responsible for a set of microservices. However, this requires a restructuring of the IT department, as well as the teams that keep the existing systems operational (EA1).

Besides a change in governance for the finished microservices network, the migration process itself also needs different governance. During the migration, the monolith needs to remain operational and will likely need to be updated. A team of developers needs to be responsible for this. Next to that, other teams will need to start developing the microservices network (EA2), thus requiring a change of governance for only the migration process.

Adding functionality and value to the microservices system during the migration process improves the business case for the migration. However, when doing this, business processes might need to be revised (EA5). This adds a level of complexity, as it might have an impact on integrations with existing systems or might require a revision of the microservices design.

During the migration process, changes are made to the monolithic application and business processes. This will have an impact on how users interact with the system. This is likely to create resistance within the organization, as employees need to adapt their usual way of working to the new system (EA3, EA5, FO3). As the monolith is migrated to microservices, this way of working might change frequently, leading to more possible resistance in the process.

Creating a sense of urgency and understanding among the organization for the migration can be difficult (EA5). While the problems associated with legacy systems are often understood by the IT department, the business might not understand the need for the migration, making it harder to get support (EA4, EA5). A lack of understanding might increase the employee’s resistance to change making the migration more challenging.

EA6 and FO2 indicate that various architects and developers, within their experience at financial organizations, have indicated to be afraid to perform such migrations, as they do not want to be responsible for breaking anything in the operational monolithic system. This causes resistance from the IT department to performing the migration, which needs to be handled accordingly.

Lack of Knowledge and Support

Migrating a monolithic system to microservices has some challenges associated with a lack of knowledge and support.

Monolithic systems in the financial sector have, in some cases, been built decades ago. These systems have been designed and built by a select group of employees, sometimes a third party, who are often no longer working at the organization (EA4, EA5, EA6, FO1, FO2, FO3). During the migration process, some processes or parts of the system might not be understood by the organization’s current architects and developers. As the system’s initial creators might no longer be approachable, the reason behind these decisions can remain unclear, leading to problems in the migration process.

As the monolithic systems are often also poorly – if even – documented (EA5, EA6, FO2), this problem is worsened. EA5 gives an example of an employee at a financial organization who had turned off a certain business rule in a financial system, only for the organization to find out years later that certain transactions were no longer being validated. No one else in the organization had a significant understanding of the monolithic system to understand why this happened.

For some monolithic systems, the underlying software or hardware might no longer be supported by the suppliers as they have reached their end-of-life. Because of this, there is no help available from the suppliers to assist during the migration process (EA3, EA6). This can also mean that security vulnerabilities will no longer be patched, potentially leading to vulnerabilities in the monolithic system during the process (EA3).

Risks and Business Continuity

Migrating a monolithic system to microservices comes with several risks and business continuity challenges for financial organizations.

As most monolithic systems for financial organizations are systems that are critical for business continuity, the consequence of system downtime or failures can be catastrophic and irreversible (EA1, EA2). Due to this, the system needs to be operational at all times, as well as when updates are made. FO1 points out that confidentiality, integrity, and availability of the system must remain high at all times during the migration*: “We zijn tijdens het vliegen van het vliegtuig, de band van het vliegtuig aan het vervangen.”* (We are changing the plane’s tire during the flight). This is especially relevant for systems of record, such as the core banking transaction system.

A major risk for the migration of monolithic systems is the data migration. Existing data must be migrated from one database, toward the microservice’s databases, and needs to remain intact (EA2). Data corruption can have serious consequences, especially if the data is critical for the business. Data also needs to be consistent over the databases. Because of this, within the migration process, data migration needs to be done carefully.

There are also risks associated with updating code to a legacy system that has not been changed for a long time (FO2). This risk is increased when the initial developers of the system are no longer working at the organization, as the decision-making behind the code is not always clear.

EA5 points out that another risk that is frequently overlooked when migrating to microservices is the adherence to privacy and regulatory requirements. While privacy regulations for the monolithic system can be implemented at once by, for example, implementing database rules, a microservices network requires a more careful and planned approach. This impacts not only the migration but also the management once the microservices are operational.

4.2.3 Overview of Challenges

This Section combines the challenges of migrating a monolithic system to microservices from literature in Section 4.2.1 and from the interviews in Section 4.2.2 into one holistic overview. This full overview is displayed in Figure 15 and answers SQ2 (*What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?*).

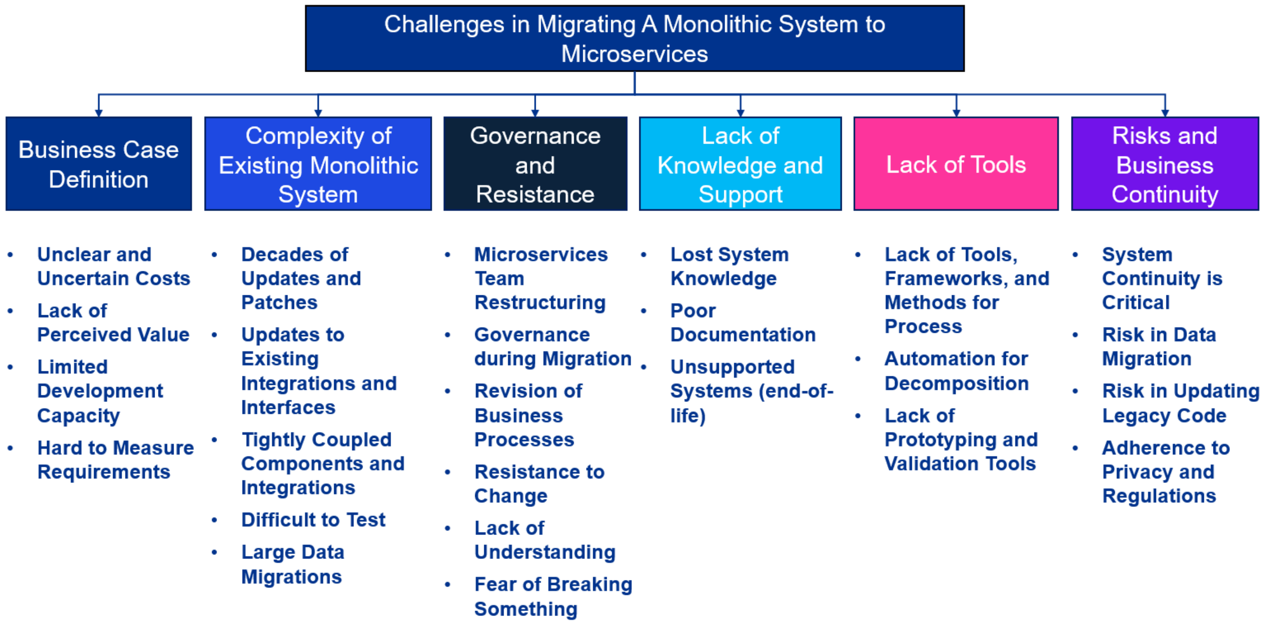

Challenges in Migrating A Monolithic System to Microservices
Business Case Definition
Complexity of Existing Monolithic System
Governance and Resistance
Lack of Knowledge and Support
Lack of Tools
Risks and Business Continuity
Unclear and Uncertain Costs
Lack of Perceived Value
Limited Development Capacity
Hard to Measure Requirements
Decades of Updates and Patches
Updates to Existing Integrations and Interfaces
Tightly Coupled Components and Integrations
Difficult to Test
Large Data Migrations
Microservices Team Restructuring
Governance during Migration
Revision of Business Processes
Resistance to Change
Lack of Understanding
Fear of Breaking Something
Lost System Knowledge
Poor Documentation
Unsupported Systems (end-of-life)
Lack of Tools, Frameworks, and Methods for Process
Automation for Decomposition
Lack of Prototyping and Validation Tools
System Continuity is Critical
Risk in Data Migration
Risk in Updating Legacy Code
Adherence to Privacy and Regulations

Figure 15 – Overview of Challenges in Migrating a System with a Monolithic Architecture to Microservices

5. Treatment Design: Migration to Microservices with Low-Code

This Chapter elaborates on the treatment design for this design research. The aim of this Chapter is to create the artifact for this design research, which is a roadmap for migrating monolithic architecture to microservices with low-code in the financial sector. In Section 5.1, the choice of roadmap as an artifact is justified, and the process of creating it is further elaborated upon. In Section 5.2, the requirements for the roadmap are discussed, which are obtained from the problem context in Chapter 4. In Section 5.3, the solution guidelines are discussed. In Section 5.4, the initial roadmap is designed, which is revised and improved through a workshop with enterprise architects in Section 5.5.

5.1 Roadmap

The artifact that is produced in the treatment design is a roadmap. As identified in Chapter 4, there is a lack of practical tools, methods, and frameworks that can be used by practitioners in the migration from monolithic architecture to microservices. Architectural roadmaps are a frequently used problem-solving tool in the field of EA (Figure 7) [42]. They highlight the milestones, phases, and steps that are needed to shift from a baseline architecture (monolithic architecture) toward a target architecture (microservices) [42]. The roadmap, therefore, explains the process of migrating a monolithic architecture to microservices. The use of low-code within this process is highlighted. The roadmap created in this research aims to standardize the migration process within the scope of the financial sector. It also aims to shed light on the different tasks that are performed within the process and their associated challenges.

The roadmap is created from requirements and solution guidelines. In Section 5.2, the requirements are specified. These consist of challenges in migrating a monolithic architecture to microservices experienced in the financial sector, which are identified in Chapter 4. In Section 5.3, the solution guidelines are specified. The solution guidelines consist of four parts:

- Input from migration strategies and their process from an exploratory literature review.
- Input about migration strategies and their process from interviews with enterprise architects and IT professionals in financial organizations.
- Input about the use of low-code for the migration process from an exploratory literature review.

- Input about the use of low-code for the migration process from interviews with enterprise architects, IT professionals in financial organizations, and low-code professionals.

5.2 Roadmap Requirements

This Section defines the requirements for the creation of the roadmap for migrating a monolithic architecture to microservices with low-code in the financial sector. The requirements consist of two parts:

- The roadmap demonstrates how financial organizations can use low-code in the migration process.
- The roadmap addresses challenges that can be faced during the migration.

These challenges are identified in Chapter 4. These requirements should make practitioners within these organizations aware of the challenges that they can encounter within the migration process, so they can make the necessary preparations to mitigate their impact of them. The goal of the roadmap is therefore not to mitigate the impact of the challenges themselves. The requirements that will be used for input in the roadmap are the challenges displayed in Figure 15. Within each of the phases and steps of the migration process, the challenges will be highlighted.

Besides these requirements, the roadmap should also be easy to understand by the financial organizations that are adopting it. This requirement is important as perceived ease of use is a factor in the intention to use a piece of technology [108].

5.3 Solution Guidelines

This Section defines the solution guidelines that are used to create the roadmap. The solution guidelines consist of four parts:

- Input from migration strategies and their process from an exploratory literature review. This is discussed in Section 5.3.1.
- Input about migration strategies and their process from interviews with enterprise architects and IT professionals in financial organizations. This is discussed in Section 5.3.1.
- Input about the use of low-code for the migration process from an exploratory literature review. This is discussed in Section 5.3.2.

- Input about the use of low-code for the migration process from interviews with enterprise architects, IT professionals in financial organizations, and low-code professionals. This is discussed in Section 5.3.2.

The latter two answer SQ3 (*How can low-code be used in migrating a monolithic architecture to microservices?*).

5.3.1 Migration Strategies and Process

This Section aims to explore strategies for migrating from a monolithic architecture toward microservices. Section 5.3.1.1 explores existing migration strategies and processes from literature. Literature input in this Section is partially obtained by preliminary research performed by the researcher. The results of the interviews are discussed in Section 5.3.1.2.

5.3.1.1 Literature: Migration Strategies and Process

Migrating a monolithic architecture toward microservices is a topic that is explored within literature. One of the topics discussed is migration strategies that can be used by professionals in practice. Tuusjärvi et al. (2024) perform an SLR to understand the state-of-the-art of migration strategies within the industry [20].

The most researched field of migration strategies is the transformation phase and process [20]. This includes workflows and processes that help organizations change their EA from monolithic applications to microservices. It aims to practically help managers in performing the migration [20]. Insights from these workflows and processes will be used as input for the solution guideline to assist in creating a migration roadmap.

Examples of transformation-focused research are Premchand et al. (2016), who created a roadmap for financial organizations to manage architecture simplification [18], Pinos-Figueroa et al. (2023) propose a migration process from monolithic architecture to microservices for a tourism services company [87], and Wolfart et al. (2021) propose a roadmap for modernizing legacy systems with microservices [21]. While these studies provide useful insights into the process of migrating from monolithic architecture to microservices, they present different results. Tuusjärvi et al. (2024), therefore, also indicate that it is essential that more tools and methods that can help the process of migration are created, to promote experiences, best practices, and for developing standards [20]. In Table 11, the migration processes of some research are presented.

Research	Migration Process
Premchand et al. (2016) [18]	Assessment (planning, analysis, validation) → Business case → Define Technical Approach → Define integration model → Define non-functional requirements → Manage implementation
Pinos-Figueroa et al. (2023) [87]	Definition → Implementation → Test
Wolfart et al. (2021) [21]	Analyze driving forces → [Understand legacy system → Decompose → Define microservice architecture → Execute → Integrate microservices → Verify and validate microservices → Monitor] (between '[' and ']' is an iterative process)
Poniszewska-Maranda et al. (2021) [99]	Team building → Extraction and implementation of services → Service discovery → Communication between services → Testing → Implementation → Scaling → Support and maintenance
Lorenzo De Lauretis (2019) [109]	Function analysis → Business functionalities identification → Business functionalities analysis → Business functionalities assignment → Microservices creation

Table 11 – Comparison of Monolithic to Microservices Architecture Migration Process

Most existing migration strategies start by analyzing the business case or driving forces for changing from a monolithic architecture to microservices. This is an important step, as a microservice architecture is not a silver bullet to achieve flexible IT landscapes for all organizations [20].

After the business case has been defined, and is beneficial, the strategies propose an analysis of the existing system and create a proposal for the to-be-created microservices system. Wolfart et al. (2021) explain that an important step in defining the to-be-created microservices is determining the granularity of the microservices [21]. Smaller microservices result in more integrations, while larger microservices lead to more functionalities within individual microservices.

When this analysis is defined, the execution of the transformation starts. What Premchand et al. (2016) [18], Pinos-Figueroa et al. (2023) [87], and Poniszewska-Maranda et al. (2021) [99] have in common is that they propose the migration from monolithic architecture to microservices in one 'big boom'. This is realized by developing the full microservices system next to the existing monolithic system. To do this, within the process, the organization needs to propose a target architecture that explains how the network of microservices is integrated and develop the full application from the ground up. An issue with this approach is that creating this target architecture is hard and brings challenges that are difficult to establish at an early stage [100]. It is also an expensive approach, as both the monolithic and microservices systems should be operational and maintained until the full microservices system is operational.

Other strategies take a more incremental approach, by migrating small parts of the application at the same time [100]. This approach is referred to as the Strangler Fig Pattern [110], which is also adopted by Wolfart et al. (2021) [21]. Over time, as the Strangler system implements more functionality and the number of microservices increases, until the full monolithic application has been migrated [53].

Within the Strangler Fig Pattern, decomposing is the process that aims to break up monolithic systems into components fit for microservices. Within this step, developers of the system investigate the system more deeply, which allows them to find places to break up the system into components. These components, which contain a part of the functionality of the monolithic system, can be put into individual microservices and are then removed from the monolithic system. The two most common patterns used to decompose a monolith into a microservice architecture are [53], [111] decomposition by business capability [84], [112] and decomposition by sub-domain [99], [113], [114].

The first considers the scope of a single microservice as the business capability that the microservice should perform, while the second considers the scope as the domain of the business functionality and uses Domain-Driven Design (DDD). Decomposition by business capability starts by identifying and analyzing the purpose of an organization and what the business does [53]. Each capability is then isolated in a microservice. This is a business-oriented approach.

Decomposition by subdomain starts by defining the application’s problem space (the business) as the domain. This domain is divided into subdomains following a DDD approach [115]. Subdomains have clearly bounded contexts and are often more granular compared to business capabilities. They are often also less dependent on other subdomains and thus require fewer integrations compared to business-capability decomposed microservices.

While these two patterns take a different approach, they can end up with similar results [53]. Decomposing iteratively with a Strangler Fig Pattern is visualized in Figure 16.

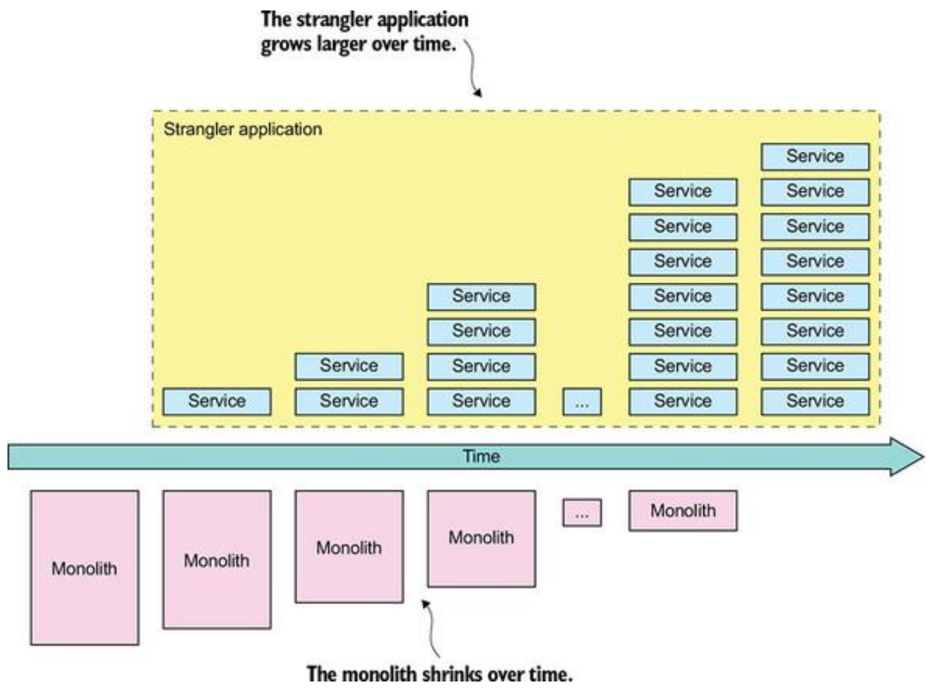

The strangler application grows larger over time.
Strangler application
Service
...
Time
Monolith
The monolith shrinks over time.

Figure 16 – Strangler Fig Pattern (from [53])

Whichever strategy is chosen, decomposing is not a straightforward task, especially for large and complicated monolithic systems. It requires organizations to have experienced developers who are very familiar with the system and have experience with distributed systems [52]. As Section 4.2.1 explains, different researchers have proposed classifications [105] and automated tools [103], [116], [117] to do this. These tools, however, are often not practical or validated, and in practice, this is often done by the experience of the enterprise architects and developers [52], [103], [104].

After the granularity and scope of the microservices are determined using one of the strategies above, they should be implemented and integrated. The microservice and integration with the network should then be tested and validated. As the microservices are part of a network, several things have to be tested. First, the internal implementation details of the microservice need to be tested [118]. As microservices are naturally isolated, this testing involves the functionality of the microservice itself. However, as microservices are part of a network and integrate with various other microservices, this behavior also needs to be tested. These integration tests verify the correctness of the microservice interacting with other microservices [118]. In microservices networks, integration testing is critical to ensure that the full network operates as it should.

Premchand et al. (2016) and Poniszewska-Maranda et al. (2021) perform testing and validation before the full system is deployed [18], [99]. Pinos-Figueroa et al. (2023) do this after execution of the full migration [87]. The latter is an interesting approach, as flaws in the architecture and system can be better found before the system becomes operational. Wolfart et al. (2021) propose that testing and validation should be done after every iteration of implementing a microservice to ensure requirement satisfaction [21].

One observation is that many migration processes fail to take into consideration the organizational and governance-related changes that need to be made during the transformation [99]. As Section 4.2 identifies, this is seen as one of the challenges for migrating toward microservices [22]. There are multiple stages throughout the entire migration process where governance is critical. As these process strategies are aimed at managers [20], this seems to be a crucial part of the migration that is missing.

Once the full monolithic application is migrated to microservices, the system should be maintained and monitored [18], [99]. The linear approach from Premchand et al. (2016) and Poniszewska-Maranda et al. (2021) considers this as soon as the full system is operational [18], [99]. According to the iterative approach from Wolfart et al. (2021), monitoring of the microservices system should start as soon as the first microservice becomes operational [21].

5.3.1.2 Interview Results: Migration Strategies and Process

In this Section, steps in the process of migrating a monolithic architecture to microservices are identified from interviews. From the thematic analysis, as Section 2.2.3.2 explains, the migration process is divided into three phases: Analyze, Execute, and Manage. Each of these phases contains various steps. These phases and steps are defined in Table 13. Each of the phases and steps will be discussed in detail in the following Sections.

Analyze

The first phase in the migration process is to analyze the current situation of the organization and define the future vision and plans. Many interviewees agree that the first step of the analysis should be to create a business case (EA1, EA2, EA4, EA6, FO5). This business case should outline whether the migration to a microservices architecture is feasible and justifiable. It should determine whether the benefits that are achieved with the new architecture outweigh the cost. It should also take into consideration the requirements of the organization and other internal and external factors (EA4). When creating such a business case, an organization should also consider options other than migrating the system. EA6 explains that stretching the support or retiring the system and developing a new one are sometimes suitable alternatives. Microservices are not a silver bullet, and the business case is a critical step to determine whether they could solve the issues an organization is facing. It might also be possible that the organization is forced to perform such a migration, due to changes in regulation (FO2). If this business case is not feasible or necessary, the migration should be terminated.

Once the business case is validated, an organization should create a baseline architecture, target architecture, and perform a gap analysis. To accomplish this, EA5 and FO5 recommend following the TOGAF ADM cycle, as they mention that this is often used within the financial sector. In each of the three steps, layers across the organization — including business, data, application, and technology — should be carefully assessed and documented. Creating a baseline analysis should include identifying the functionalities and integrations that the system has (EA4). This allows the organization to determine what parts of the architecture need to be updated and what parts can be reused (EA5). Multiple interviewees agree that ArchiMate is a popular modeling language for creating architecture documents (EA2, EA5, FO5). Figure 11 is an example of how architects can illustrate the application and technology layer of the baseline architecture of their monolithic system using ArchiMate.

The organization should then create a target architecture based on the requirements defined in the business case and the organization's strategic vision. Creating a target architecture defines the organizational and technical changes that need to be made to migrate toward the microservices architecture (EA5). EA3 and EA4 also explain that the definition of 'hygiene' rules, such as API standardization, security (including authentication and authorization), and logging, are important within the target architecture. EA6 explains that properly defining a target architecture is important to always keep in mind the end goal, but acknowledges that it can be difficult to work out all details, given the lack of information known about these legacy systems: "*Als je de himalaya gaat beklimmen moet je weten wat je moet meenemen, anders kun je halverwege teruglopen of je komt niet verder* ". In the target architecture, revised business processes should also be outlined (EA4). Figure 12Figure 11 is an example of how architects can illustrate the application and technology layer of the target architecture of their microservices system using ArchiMate.

When the baseline architecture and target architecture are defined, a gap analysis and migration planning should be created (EA4, EA5). The creation of the gap analysis outlines the steps that the organization needs to take to achieve the target architecture from the baseline architecture. The organization performing the migration needs to realistically define these steps, as directly achieving the target architecture is often not realistic (EA5).

When the baseline architecture, target architecture, and gap analysis and migration planning are all approved by the stakeholders, the development and execution phase can start (EA4).

Execute

The second phase of the migration process is the execution phase. Here, the monolithic system will be migrated toward microservices. Multiple interviewees agree that an iterative approach (Strangler Fig Pattern) is the best way to execute the migration (EA1, EA4, EA5, EA6, FO1, FO2). EA5 explains that using a Big Bang approach always creates a second-system effect, where the initial legacy system is never phased out. EA6 elaborates on a case where the Big Bang scenario worked well, but explains that this will not work for all migrations.

To determine which components to extract from the monolith first, the organization should create a trade-off between what is easiest and most valuable (EA1). In practice, it is easiest to start at the newest created components (FO1). The advantage of this is that you gradually decrease the number of integrations to the monolithic system and increase the size of the microservices network. However, as the microservices cannot operate in isolation, it will happen that newly created microservices need to re-integrate with the monolithic system. This brings extra complexity, as information is being 'ping-ponged' between the different systems (EA3, EA5, EA6). EA3 explains that you want to minimize this, as employees are likely to resist having to use multiple systems simultaneously. Functional sub-domain decomposition, therefore, is preferred over decomposition by business capabilities (EA3, EA5, EA6, FO1), as these minimize the cross-system interactions. An example of this is provided by FO5, who explains how their organization used functional decomposition to decompose their document management system into different microservices that took the functions of document creation, version management, document collaboration, and document archiving. During this decomposition analysis, new insights about the systems might be found that were not found in the previous analysis phase. In this case, the target architecture might need to be revised.

Another thing to consider when extracting a component is the size of the microservice. During the interviews, the definition of how large a microservice should be was often not clear. Multiple interviewees point out that making microservices too small leads to performance issues, governance issues, and data consistency problems (EA5, EA6, FO4, FO5). Therefore, the size of a microservice needs to be considered carefully, also for future maintainability.

As the scope of a to-be-extracted component is defined, the organization needs to consider which technology it wants to use to develop the microservice. Multiple interviewees agree that, when possible, SaaS is preferred over custom-developing the microservice (EA2, FO4, FO5). If the software is not directly available on the market, the microservice should be custom-built. After the microservice is purchased or built, it should be integrated with the already existing microservices and possibly the monolithic system (EA5, EA6, FO2). EA4 explains that organizations frequently use the DTAP (development, test, acceptance, and production) street to test and validate new functionalities. Once the microservice has gone through this street, the microservice is deployed, and the iteration starts over.

Manage

During the migration process, several things are important to manage throughout the entire migration. Governance plays a crucial role in ensuring the migration process is structured and aligns with business strategy and goals.

From the first phase of the process, it is important that all stakeholders, that are to some extent impacted by this migration, are well-informed (EA4). A sense of urgency needs to be spread across the organization. It is possible that throughout the migration, employees might resist changes being made (EA3, EA4). It is important to manage these employees.

As soon as the business case is approved, teams that will be responsible for the execution of the migration need to be created (EA4). This includes enterprise architects, solution architects, and developers. During the migration, some teams need to be set up to develop the new microservices architecture (EA2), while other teams need to maintain the existing monolithic system. This is important, as the monolithic system should remain operational during the migration process (EA5). During the migration, it might also be possible that the monolith has to be updated, for which a maintenance team is responsible.

As the migration proceeds, the team structure will need to change as more microservices are being deployed. Besides the developers, an organization might also need to create a project management team that handles employees' questions regarding the system (EA4).

5.3.2 Low-Code for Migrating

In this Section, the use of low-code in the migration of a monolithic architecture to microservices is described. This Section aims to answer SQ3 (*How can low-code be used in migrating a monolithic architecture to microservices?*). The results from literature are discussed in Section 5.3.2.1. The results from the interviews are discussed in Section 5.3.2.2. Section 5.3.2.3 combines the insights from both literature and interviews into one holistic overview explaining how low-code can be used in the migration process.

5.3.2.1 Literature: Low-Code for Migrating

In this Section, the use of low-code in the migration of a monolithic architecture to microservices is described from literature. This aims to assist in partially answering SQ3 (*How can low-code be used in migrating a monolithic architecture to microservices?*).

Low-code can be used to develop software applications. While literature does not explicitly define the context for which low-code development is most applicable, four frequently mentioned cases for using low-code are:

- Innovation – Low-code can be used to quickly develop new ideas, prototype, or create a proof of concept (POC).
- Customer engagement – Improve customer experience by creating application portals that they can connect to.
- Operational efficiency – Use low-code to create business-process/workflow-specific applications.
- Legacy modernization – Use low-code to replace or improve a legacy system [119].

Within the context of migrating an existing monolithic application toward microservices, especially cases ‘Innovation’ and ‘Legacy modernization’ are relevant. How low-code can be used in the migration process for each of these will be further elaborated upon.

Innovation

Due to the rapid development power that low-code gives, organizations can use low-code to quickly develop new software ideas, prototype them, and gather feedback from potential users of the software [120]. This is an iterative approach and allows low-code developers to better align the system with the needs of the users. In the context of migrating an existing monolithic system, low-code can be particularly useful for prototyping microservices that involve direct user interaction through a UI. However, it is less relevant for microservices that operate solely as backend processes, as end users are likely not acquainted with non-visual systems.

Legacy Modernization/Migration

Low-code can be used to replace or improve legacy software solutions. This is important for the context of this research. While low-code can be used to directly migrate legacy applications, it can also be used as a ‘skin’ on top of the legacy system [119]. The core legacy system will remain operational, while the low-code software is integrated with it. LCDP vendors such as Outsystems and Mendix, and low-code consulting organizations often claim that low-code is applicable for this [121], [122], [123]. Nevertheless, in literature, the use of low-code for this purpose is barely researched.

5.3.2.2 Interview Results: Low-Code for Migration

In this Section, the use of low-code in the migration of a monolithic architecture to microservices is described from interviews. This aims to assist in partially answering SQ3 (*How can low-code be used in migrating a monolithic architecture to microservices?*).

Within the migration process, interviewees mention that low-code can be used for prototyping, and for developing microservices. The use of low-code in both of these and the associated benefits and risks will be further elaborated upon.

Prototyping Microservices

Using low-code for prototyping implies that a microservice – before its implementation and deployment – is built using an LCDP. This approach enables architects and developers of the microservice to assess the appropriate granularity of the microservice, identify potential issues, evaluate its fit within the broader process, and verify the functionality of its integrations. Several interviewees agree that low-code is a suitable tool for this purpose (EA1, EA3, EA6, FO4, LC3, LC4, LC5).

Prototyping microservices with low-code is effective due to two factors. First, it is relatively simple to quickly create a microservice that integrates to other microservices over an API (EA3). The already built-in API connectors allow low-code developers to save considerable time in setting up the integrations (LC5).

Second, low-code facilitates the rapid development of applications that need to be tested with potential users and customers. Particularly, user-interface-driven prototypes can easily be built by using predefined components (EA6). This allows the developers to quickly gather feedback from the end-users, customers, or clients. Using low-code for this is especially valuable for process-oriented systems, intended for internal use by employees, or external use by customers (LC3).

In practice, however, LCDPs are usually too expensive for solely being used for prototyping purposes. Platforms, such as Mendix, have license models that allow organizations to either develop only one application or unlimited applications [124]. For organizations not already using low-code for broader development purposes, the need to acquire an unlimited license solely for prototyping may lead to a business case that is not justifiable (FO4).

Developing Microservices

Low-code can be used to develop individual microservices. Low-code inherently shares many technical similarities with microservices architectures. They are, depending on the LCDP, frequently hosted in the cloud, as the LCDPs are responsible for the deployment of the application. They also contain predefined REST or event-driven integration connections, allowing them to integrate the different microservices. An example of a low-code-powered microservice integrated within a network of microservices is displayed in Figure 17. In this example, low-code is used in microservice 4. This example is similar to the microservices network presented in Figure 12, but the core difference is that the deployed system and database server are managed by the low-code supplier, rather than by the developers themselves. Another way low-code can be used within a microservices network is as a UI.

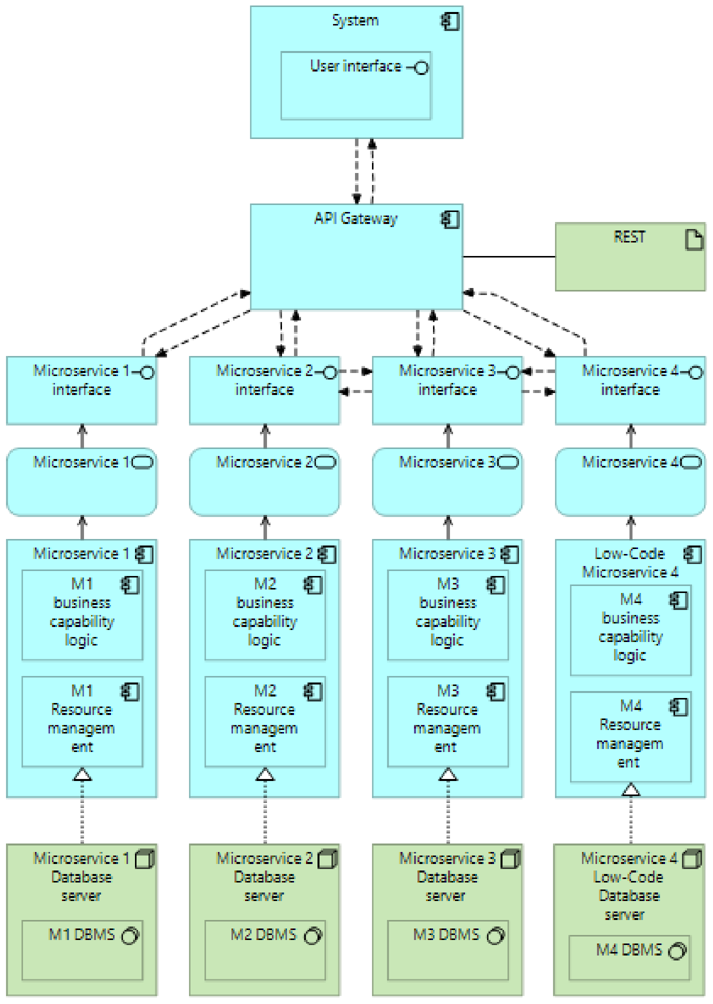


Figure 17 – Illustration of a Microservices Architecture with Low-Code

Nearly all interviewees identify increased development speed as the primary advantage of using low-code for developing systems (EA1, EA3, EA5, EA6, FO1, FO4, FO5, LC1, LC2, LC3, LC4, LC5). During the migration from a monolithic architecture toward microservices, this would allow developers to more rapidly develop individual microservices and integrate them into the microservices network compared to high-code. This could thus accelerate the – often long-lasting – migration process.

Another advantage of using low-code is that low-code fosters collaboration with other stakeholders, as LCDPs make the development of software more comprehensible and make interaction with users of the application more effective (LC3, LC4).

Today, low-code can be used to create virtually any piece of software (LC1, LC3). Many interviewees agree that low-code would be especially useful for processes that have a significant amount of user interaction and systems that require rapid innovation. Low-code brings the flexibility and development speed that is useful for these types of systems, often referred to as systems of innovation (EA2, EA3, FO2, LC2). The availability of built-in components allows developers to rapidly adapt the software to the organization's or user's needs. Low-code is also useful for systems with many small processes (EA5, FO1). For instance, FO1 described the use of low-code in a banking application to support updating pay limits and changing residential addresses. FO1 and EA5 explain that their organizations primarily use low-code for small stepwise processes within a large network of microservices.

However, there might also be cases where an organization should refrain from using low-code to migrate a monolithic system to microservices. When using low-code in this context, there are some challenges and risks that need to be addressed. These are categorized into the following themes defined in Table 12. Each will be elaborated upon.

Risk Categories in Using Low-Code for Migrating a Monolithic Architecture to Microservices
High and Unclear License Costs for LCDPs
Limitation of LCDP's Functionality
Performance due to Abstraction
Vendor Trust & Lock-In
Ease of Development & Citizen Development

Table 12 – Risks in Migrating a Monolithic Architecture to Microservices using Low-Code from Interviews

High and Unclear License Costs for LCDPs

The costs for using an LCDP are high and often unclear (EA1, EA5, FO4, FO5, LC3, LC4). Organizations using an LCDP pay for the full service that the supplier provides. These total costs often exceed those of a custom-built high-code solution hosted in the cloud. Many low-code suppliers also impose extra charges based on the number of users accessing the developed systems (LC2, LC3). LC2 notes that this could be problematic for financial systems with many users. FO4 and LC5 agree that for multiple low-code suppliers, the license cost structures are vague, not allowing their organizations to accurately forecast the costs of using the platform. LCDP license fees may increase over time, increasing the uncertainty of costs made in the long term (FO4, FO5).

Limitation of LCDP's Functionality

LCDP also presents functional limitations. Low-code developers are limited by the capabilities the platform provides (EA1, FO2, LC4). Additional functionalities may need to be built in high-code. While LC2 points out that the ability to combine low-code and high-code is what makes LCDPs ideal, others argue that writing high-code within a low-code microservice removes the added value of using an LCDP in the first place (EA1, EA6, FO4). This hybrid approach requires developers to have the skills to write low-code and high-code.

Performance due to Abstraction

Almost all interviewees mentioned that low-code is not useful for processes and systems that require substantial processing power for a significant amount of data (EA1, EA2, EA3, EA5, EA6, FO4, FO5, LC1, LC2, LC3, LC4, LC5), processes that often occur within systems of record. In the financial sector, examples of such a system could be a core transactional banking system (LC3) or a microservice that requires much data to be aggregated into a report (LC2). Unlike high-code environments, where algorithms can be optimized manually, LCDPs rely on predefined components that generate code that cannot be altered. For processes that have high performance as a requirement, high-code is therefore preferred due to its greater options for optimization (FO5, LC5). In this context, FO4 is also concerned about the stability of the LCDPs, particularly for business-critical microservices, where minimal system downtime is crucial.

Vendor Trust & Lock-In

A frequently mentioned risk when using low-code is the dependency on the vendor of the LCDP (EA1, EA3, FO4, LC4). Once an LCDP is in use, the organization can face problems in transitioning to another platform. Organizations can face challenges when trying to migrate the data and when transferring the code to another vendor. The latter is, in almost all cases, impossible. If an LCDP goes out of business, this can have serious consequences for an organization using the platform for its microservices. In this case, organizations may be forced to redevelop all microservices from scratch. Especially for systems that are critical to an organization's strategy, such as a core transactional banking system, as mentioned before, this is a large risk (EA3).

Ease of Development & Citizen Development

While low-code makes software development more comprehensible and accessible, this can have negative consequences. Many LCDPs are – or were historically – marketed toward citizen development. This has attracted less experienced developers to develop low-code solutions and identify themselves as low-code developers, while they are not software development experts (EA6, LC3, LC5). However, the use of citizen developers in the migration from a monolithic architecture toward microservices should not be considered (EA1, FO5). Monolithic systems in the financial sector are often complex systems that require a deep knowledge of software development to migrate, something that citizen developers lack (EA1). FO1 explains that many of their high-code developers have more experience developing compared to their low-code developers: *“De high-code developer is verder ontwikkeld in zijn ontwikkeljourney dan de low-code developer … omdat hij vaker op z’n bek is gegaan”*. Additionally, as LCDPs simplify the development of applications, in practice, low-code systems are often developed more rapidly compared to high-code systems. As a result, in practice, it is easier to develop bad software with low-code. This issue is not inherent to low-code itself but is frequently observed by professionals and is noted by several interviewees (FO1, LC1, LC3). Another drawback of the simplicity of low-code is the ease of breaking something during the development of a microservice. LC3 explains that it can be as easy as the click of a button to break an application in an LCDP by “*Just adding an index to a database column*”. Although this can also occur in high-code development, the abstraction layers that low-code platforms provide can make them more susceptible to such mistakes. Nevertheless, when properly designed, a system built with low-code can, just like high-code, function as proper software (EA3).

Overall, the opinions on using low-code in the migration from a monolithic architecture toward microservices were quite different. While some interviewees definitely see the use of low-code in developing microservices and its advantages, others do not. EA6 and FO4 argue that an experienced developer would often be able to develop a microservice as quickly as a low-code developer, thus removing its main advantage.

When deciding whether to use high-code or low-code to develop a microservice, an organization should take into account the abovementioned factors. What is also important to take into consideration is the availability of resources within the organization and the organization’s strategy. When an organization has already adopted an LCDP for broader purposes and has many developers available, it might be more beneficial to use low-code than an organization that has yet to start using low-code and needs to train its employees (LC1). In the latter case, using high-code might be more effective.

In the end, low-code is just another “*available tool in the toolbox*” (LC3, LC4). Deciding whether to use high-code or low-code should be done by creating a business case (LC4). Here, the organization should weigh the benefits of using high-code or low-code against the drawbacks and make a decision accordingly.

5.3.2.3 Overview of Low-Code Use for Migrating

This Section combines insights into the use of low-code for migrating a monolithic architecture to microservices from literature in Section 5.3.2.1 and from the interviews in Section 5.3.2.2 into one holistic overview. This answers SQ3 (*How can low-code be used in migrating a monolithic architecture to microservices?*).

Within the process of migrating a monolithic architecture to microservices, low-code can be used for two reasons:

- Prototyping microservices;
- Developing microservices.

Low-code fosters innovation by allowing organizations to quickly create prototypes. These prototypes can be used to quickly gather feedback about the system from end-users. While literature primarily discusses the use of low-code for UI-driven systems, as these can be quickly developed using existing components, multiple interviewees also think low-code could be useful for backend processes, due to the simplicity of developing integrations between multiple systems. The latter allows developers to test a microservice within the microservices network. However, due to the licensing structure for most LCDPs, using low-code solely for developing prototypes might not be beneficial.

Low-code can be used to modernize legacy systems by extracting components from existing monolithic systems and developing them as a low-code-powered microservice. Literature has barely discussed the use of low-code for this. Multiple interviewees agree that the use of low-code for this can be beneficial, especially as it can speed up the migration process. Some interviewees are skeptical about the use of low-code for this, as they argued that low-code is rather expensive and brings some risks that high-code does not bring. Organizations should always create a business case to determine what technology to use to implement microservices and determine whether high-code or low-code is the right tool for them to use within their migration process.

5.4 Initial Roadmap Design

In this Section, the phases and steps that need to be taken by a financial organization to migrate a monolithic architecture to microservices with low-code are described. This Section combines the insights from Sections 5.2 and 5.3 (i.e. the roadmap requirements and solution guidelines) and answers SQ4 (*What steps should a financial organization take to perform the migration from a monolithic architecture to microservices using low-code?*). Together these phases and steps are combined into an initial roadmap.

The insights for the migration strategies and process from both literature and the interviews in Section 5.3.1 are structured into different phases and steps that together make up the full migration process for migrating a monolithic architecture to microservices. As literature only defines steps in this process, the phases defined in Section 5.3.1.2, namely ‘*Analyze’*, ‘*Execute’*, and ‘*Manage’* are the phases used for the roadmap. The steps are put into chronological order. Steps within ‘*Execute*’ are performed iteratively, and steps within ‘*Manage’* are parallel steps during the migration.

In the process of defining steps, interview data is prioritized over literature. An example of this, is the choice for a sub-domain decomposition analysis, rather than a business-capability decomposition analysis. While literature discusses both options, the interviewees point out that decomposition through sub-domains results in less integrations between the microservices and the old monolith and is therefore preferred.

Section 5.3.2 defines two ways low-code can be used within the migration process. These findings are incorporated as two steps to the migration process, namely prototyping and developing microservices. The first is marked as an optional step, due to the large differences in opinions on this with the interviewees.

The other requirements defined in Section 5.2, namely the challenges financial organizations can face during the migration process, are added below ‘Keep in mind’ within the description of each step. This clearly indicates the challenges that can be expected throughout the full process and each of the steps.

The final result, namely the phases and steps, together with their description, is displayed in Table 13.

Phase (P)	**Step (S)**	**Description**
P1: Analyze	S1: Create Business Case	• The first step in the migration should be to create a business case • The business case to migrate to microservices needs to be clear • External regulation might force organizations to migrate • Investigate alternatives to migrating the monolithic system • Keep in mind: o Costs associated with migration are often unclear and uncertain

		○ Non-functional benefits might be difficult to measure (flexibility, scalability, maintainability, etc.) ○ Lack of perceived value from the business ○ Financial organizations have limited development capabilities due to full backlogs
	S2: Determine Feasibility Business Case	• The benefits of the microservices should outweigh the costs • If the business case is not feasible, the migration should be terminated
	S3: Create Baseline Architecture	• A baseline architecture should be created • Includes an analysis of the system and its integrations • Should follow TOGAF ADM guidelines • Keep in mind: ○ Difficult to fully understand the existing monolithic system ○ Decades of updates and patches make understanding the system difficult ○ Tightly coupled components and integrations ○ Difficult to test existing functionalities ○ Lost systems knowledge ○ Poor documentation ○ Unsupported end-of-life systems
	S4: Create Target Architecture	• A target architecture should be created • Use the requirements from the business case • Align with organization's strategy • Potentially revise business processes • Define hygiene rules (e.g. API standardization, security, logging) • Should follow TOGAF ADM guidelines

	S5: Create Migration Analysis & Planning	• A migration analysis & planning should be created • Start with the easiest and/or most valuable piece of functionality • Should follow TOGAF ADM guidelines
P2: Execute	S6: Analyze Sub-domain Decomposing	• A decomposition analysis should be done by defining sub-domains for components in the monolith • Iterative Strangler Fig Pattern approach: Decompose the monolith per component • This analysis is more detailed compared to the target architecture, as the system is deeply analyzed • Create a design for the microservice and a detailed description of the integrations in the network • Keep in mind: ○ Lack of practical tools for automating decomposition
	S7: Revise Target Architecture	• Changes might need to be made to the target architecture, depending on the findings in the '*Analyze Sub-domain Decomposing'* (S6) step
	S8: (Optional) Prototype	• Organizations can use low-code to prototype the designed microservice ○ Useful for prototyping UI-driven systems to quickly gather user feedback ○ Useful for integration prototyping • Optional step, as organizations should not use low-code solely for this purpose
	S9: Perform Market Analysis & Purchase SaaS	• A market analysis should be performed to investigate whether the microservice is purchasable as a SaaS • If this is the case, the microservice should be purchased

		• Keep in mind: ○ Adherence to privacy and regulations for microservices
	S10: Create Business Case & Develop Microservice	• If the microservice is not available as a SaaS, it should be custom-developed • Organizations need to create a business case to determine whether to use high-code or low-code as a tool to develop the microservice • Depending on the results of the business case, the microservice should be developed in the selected tool • Low-code can speed up the migration process, but there are risks associated with the use of low-code that need to be assessed (Section 5.3.2.2) • Keep in mind: ○ Adherence to privacy and regulations for microservices
	S11: Integrate Microservice	• The purchased or custom-built microservice should be integrated with the microservices network, and sometimes the monolithic system • Integrations should always first be performed in development environments • Data migrations should be prepared • Keep in mind: ○ Updates might need to be made to existing integrations and interfaces ○ Large data migrations might need to be created
	S12: Test & Validate Microservice	• The microservice and its integrations in the network should be tested and validated • Microservices should advance the DTAP phases

	S13: Deploy Microservice & Phase-out	• When the microservice is tested and validated, it should be deployed • Old functionality from the monolithic system should be phased out • Keep in mind: ○ System continuity is critical ○ Risk in updating legacy code ○ Risk in data migration
P3: Manage	S14: Maintain	• Throughout the migration process, both the existing monolithic system and the microservices need to be maintained • New functionality might need to be added during the migration to either of the systems due to external factors • Keep in mind: ○ System continuity is critical ○ Risk in updating legacy code
	S15: Monitor	• During the migration, both the monolithic and microservices network need to be monitored • The microservices need to be monitored as soon as the first microservice has been deployed • Error handling needs to be well-defined, as the organization transitions from one system to many systems

	S16: Govern	• Migrating a monolithic system to microservices requires good governance • Communication with all stakeholders throughout the full process is critical • Keep in mind: o Microservices require team restructuring o Governance to restructure teams also happens during the migration o Resistance from stakeholders due to the revision of business processes o Resistance from stakeholders to change to new systems o Lack of understanding from the business o Fear of architects and developers to break something

Table 13 – Phases and Steps for Migrating a Monolithic Architecture to Microservices with Low-Code

The phases and steps defined in Table 13 can be combined into an initial process roadmap. This roadmap is displayed in Figure 18. The description of each of the steps is not incorporated into the roadmap. Therefore, Table 13 will serve as the '*accompanying document'* that gives more information on the details of the migration roadmap.

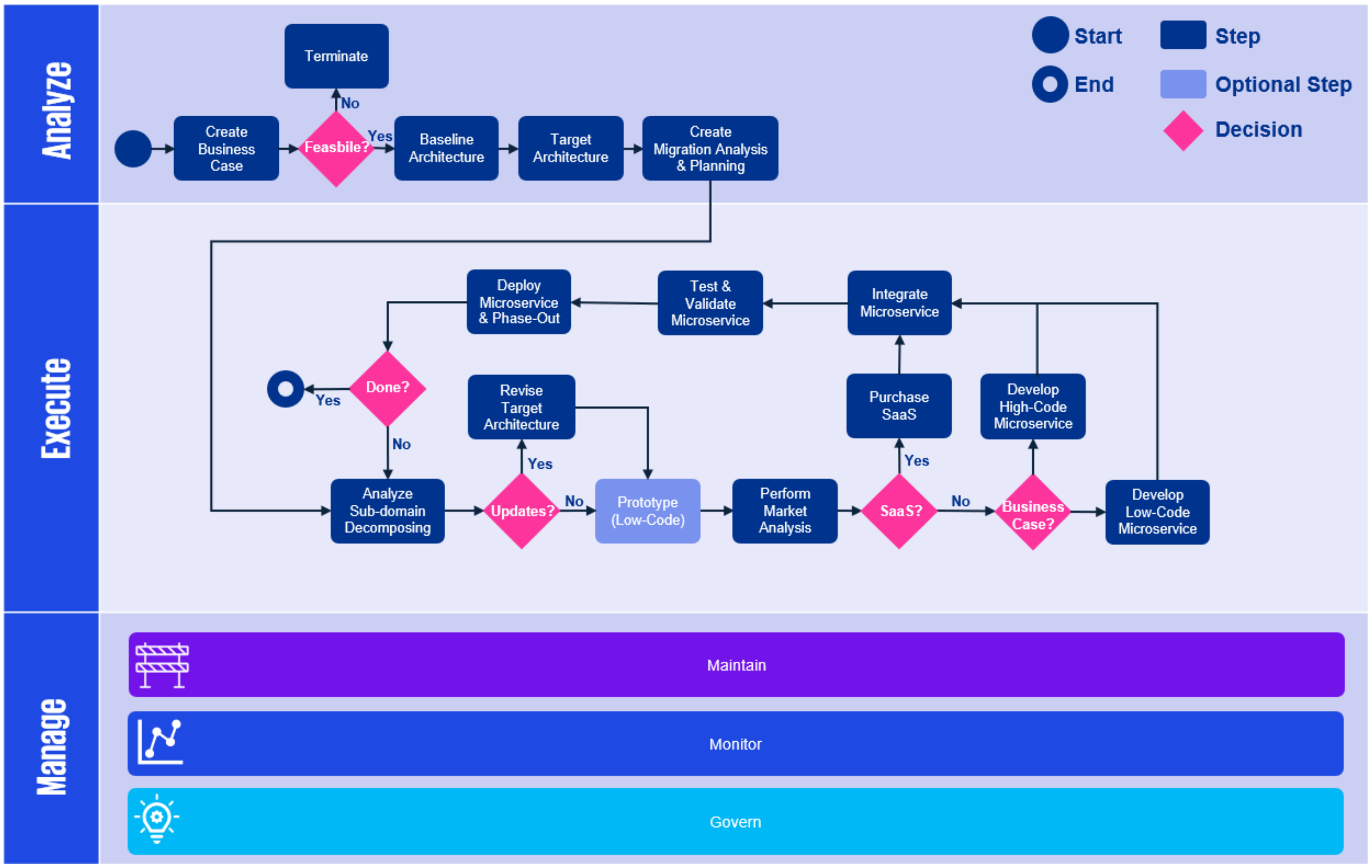


Figure 18 – Initial Roadmap for the Process of Migrating a Monolithic Architecture to Microservices with Low-Code

5.5 Improved Roadmap: Workshop with Enterprise Architects

To improve the roadmap proposed in Section 5.4, a workshop with three enterprise architects is held. The results from the workshop are discussed in Section 5.5.1. The changes and revised, improved roadmap are presented in Section 5.5.2.

5.5.1 Workshop Results: Feedback on Roadmap

In this Section, the outcomes of the workshop are presented. Overall, the participants respond positively to the initial version of the roadmap. They agree with the structure of the three defined phases as well as with the visual representation of the roadmap. They do, however, see some points of improvement for steps within the phases which will be further elaborated upon.

Analyze

Overall, the workshop participants are positive about how this phase was structured. WS1 points out that an important step, before creating any architecture, is defining the architectural principles, which is part of the preliminary step in the TOGAF ADM Cycle. They agreed that this should be further elaborated upon with an extra explanation of the roadmap within the baseline architecture step. WS2 explains that, while a good preparation is critical in these migrations, it often happens that both the baseline architecture, target architectures, and migration analyses are revised during the process. The roadmap currently does not clearly showcase that.

Execute

The workshop participants agree that for the execution of the migration, an iterative approach is best-suited. However, WS2 points out that with the current design of the roadmap, it seems that only one microservice can be decomposed and deployed simultaneously. All three participants agree that this is often done in parallel with multiple teams. They also mention that with the current design, microservices can only be deployed individually, while in practice this is almost always done in releases with multiple microservices at a time. To facilitate this change, the roadmap should, as WS2 points out, have a box around the iterative part, and put a parallel sign inside it. The process of decomposing a microservice should, as the release is not yet finished, iterate again. If the release is ready, the set of microservices can be deployed and part of the monolith phased out. Additionally, WS3 explains that both the baseline and target architectures are often revised during the process in an agile way, instead of only after the analyze sub-domain decomposing step. WS2 elaborates that, while it usually happens during the entire process, it is most frequently revised between releases. This way, organizations can create a migration analysis and plan for each release.

Manage

All participants agree with the design of the manage phase, and the three parallel processes that take place during the entirety of the migration process. WS1 and WS2 expressed that a critical governance step is requirement management, to ensure that the requirements that were set up in the business case are achieved. This should be added to the explanation of the governance step.

5.5.2 Final Roadmap

Using the feedback from the workshop, the initial version of the roadmap, displayed in Figure 18, has been revised. These revisions are further elaborated upon, and the impact on the defined phases and steps in Table 13 – the accompanying document – is also highlighted by their identifier. The following revisions are made to the roadmap:

- Decomposition of microservices in the execute phase is now a parallel process, as multiple teams can perform this process simultaneously. *S6* and *S8-S12* are grouped.
- Target architecture is no longer only revised after the sub-domain decomposing step. *S7* is removed.
- Microservices are no longer deployed individually at the end of the decomposition process, but rather in releases. Releases can consist of a single or more microservices. *S13* is updated.
- The baseline architecture, target architecture, and migration analysis & planning are revised before every new release.
- Requirement management is added to governance. *S16* is updated.

These revisions result in an improved, revised roadmap, displayed in Figure 19.

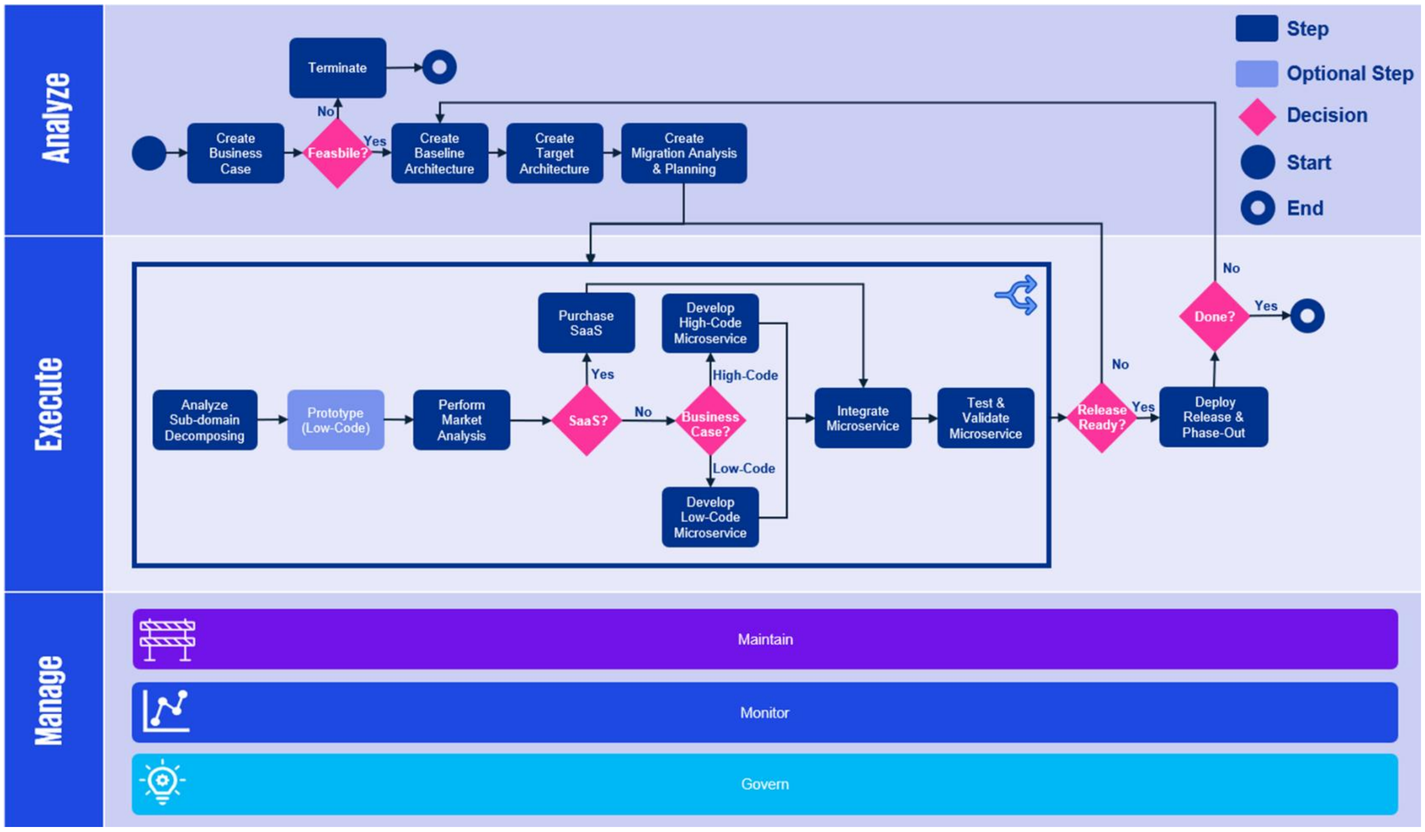


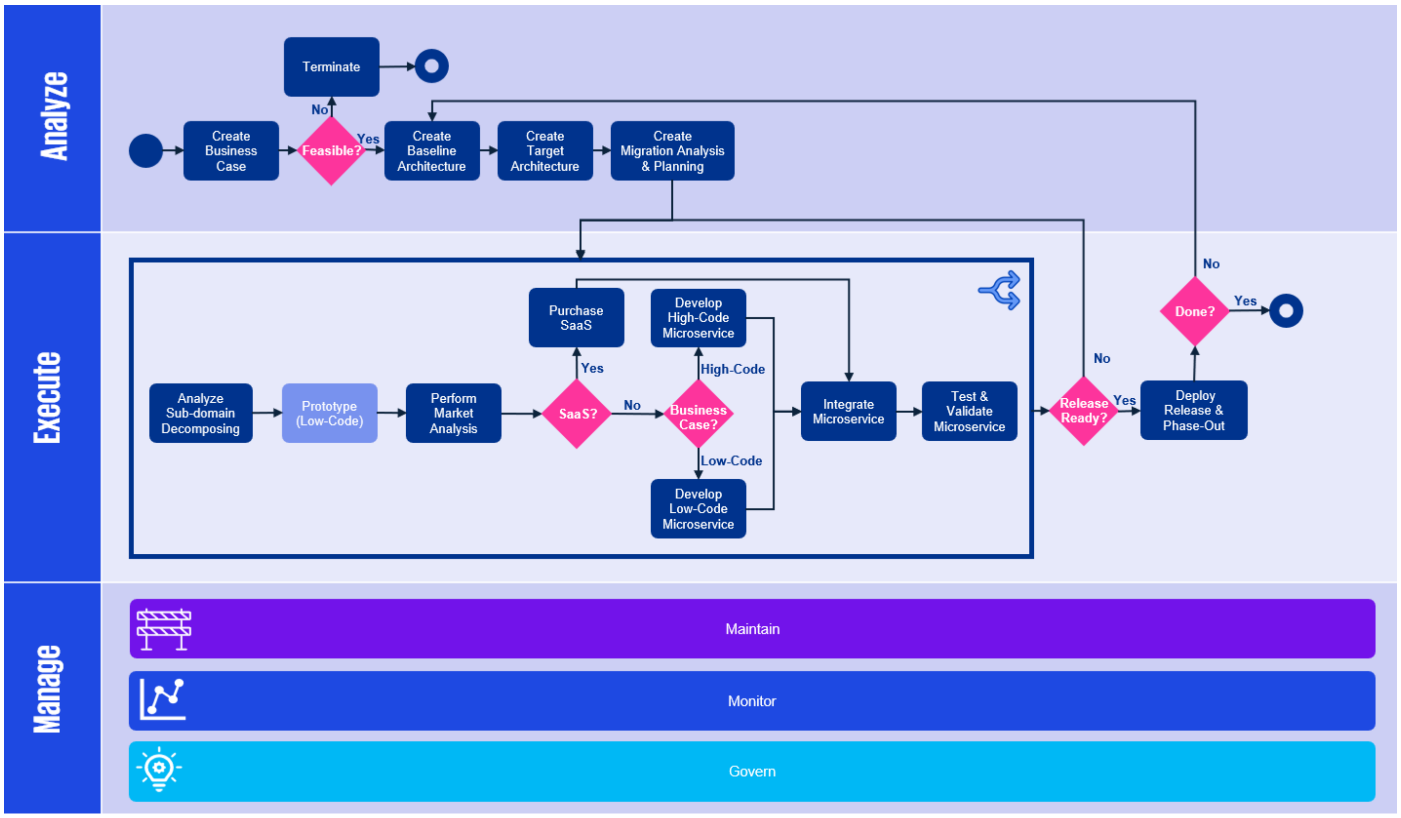
Figure 19 – Revised Roadmap for the Process of Migrating a Monolithic Architecture to Microservices with Low-Code

6.Treatment Validation: Expert Opinions

This Chapter elaborates on the treatment validation for this design research. The aim of this Chapter is to validate the roadmap for migrating monolithic architecture to microservices with low-code in the financial sector. Section 6.1 displays the final roadmap, which will be presented to the experts. Section 6.2 elaborates on the results of the validation interviews with these experts.

6.1 Final Roadmap

Figure 20 displays the final roadmap for migrating a monolithic architecture to microservices with low-code. This roadmap is displayed to experts through expert interviews.



Analyze
Execute
Manage
Terminate
No
Create Business Case
Feasible?
Yes
Create Baseline Architecture
Create Target Architecture
Create Migration Analysis & Planning
Purchase SaaS
Develop High-Code Microservice
Yes
High-Code
Analyze Sub-domain Decomposing
Prototype (Low-Code)
Perform Market Analysis
SaaS?
No
Business Case?
Low-Code
Develop Low-Code Microservice
Integrate Microservice
Test & Validate Microservice
No
Release Ready?
Yes
Deploy Release & Phase-Out
No
Done?
Yes
Maintain
Monitor
Govern

Figure 20 – Final Roadmap for the Process of Migrating a Monolithic Architecture to Microservices with Low-Code

6.2 Treatment Validation Results: Expert Opinion Interviews

In this Section, the results from the expert opinion validation interviews are discussed. The results are divided into the three categories defined in Section 2.2.3.4, namely, effect, sensitivity, and requirement satisfaction.

6.2.1 Effect

The goal of effect validation is to discover the effects produced by the artifact within the context. The experts are asked how they think the roadmap will impact the migration strategies for migrating a monolithic architecture toward microservices in financial organizations.

Overall, the interviewees expressed a positive view of the proposed roadmap, acknowledging it as a clear and structured guide for financial organizations migrating their legacy monolithic systems to microservices (VD1, VD2, VD4, VD5, VD6, VD7). They generally perceive the roadmap as a valuable contribution to existing migration strategies. It highlights key challenges that organizations may encounter during the migration process. As VD5 explained, the roadmap supports financial organizations in “extracting layers from legacy systems” and incrementally modernizing them through microservices, thereby enabling these components to, among others, become more scalable.

VD3 and VD7 believe that this roadmap can help organizations structure the complex migration process. VD3 notes that the roadmap’s visualization can aid in showing the business of the organization where they currently stand within the process. VD7 agrees that there is currently a lack of tools and frameworks for this migration process, and that the roadmap can make an impact here.

Most experts agreed that the level of granularity in the roadmap is appropriate, offering sufficient flexibility to accommodate organization-specific implementations for each of the phases and steps while avoiding too much abstraction (VD1, VD2, VD4). This balance increases adaptability, as too many details may hinder an organization’s individual migration process, whereas too much abstraction may fail to provide actionable guidance. VD4 emphasized that the current level of detail is suitable and suggested that organizations should further refine the details within the roadmap’s steps to fit their specific contexts. VD7 agrees that this migration process suits the current approach that their organization is taking toward modernizing their monolithic systems.

VD1 explains that the roadmap aligns well with the agile methodology. This is important, as more organizations within the financial sector are moving toward agile software development practices (VD1), and microservices also foster continuous development.

VD8 explains that the roadmap requires their organizations to make changes to their current process, and that they will not likely adopt the current state of the roadmap due to this.

6.2.1.1 Organizational Adaptations

Several steps of the roadmap may require organizations to adapt their existing practices. First, multiple interviewees note that their organizations do not currently adopt TOGAF ADM as their architecture development methodology. As such, the use of this roadmap may influence how these organizations typically approach architecture from a methodological perspective (VD1, VD4, VD3, VD5, VD6, VD7, VD8). Although the roadmap, particularly in the '*Analyze'* (P1) phase, suggests the use of TOGAF ADM guidelines, these are not deeply embedded throughout the entire process. This provides flexibility for financial organizations to select an alternative architectural framework, though this choice may lead to slight changes in how the roadmap is implemented. VD7 explains that their organization uses Novius as an EA framework, but agrees that this can replace TOGAF within the process and roadmap without any changes.

Second, some interviewees indicate that their organizations update the baseline and target architectures more frequently during the migration process than the roadmap suggests (VD2, VD3). While the roadmap currently places this activity in the '*Analyze'* (P1) phase, in practice, some organizations continuously maintain their architectures throughout the migration process. This appears to be organization-specific and reflects different approaches to architectural governance, just like the use of TOGAF ADM or different frameworks. VD7, namely, argues that the roadmap's process correctly highlights when their organization updates their baseline and target architectures.

Third, multiple interviewees highlight that, in their experience, the business case for migration is often revisited multiple times throughout the migration (VD2, VD3, VD4). In its current form, the roadmap positions the business case as a preliminary step that determines the feasibility of the migration. However, in practice, organizations may revise the business case iteratively. For example, VD3 and VD4 mention that their organization would not be able to develop a single business case for the entire migration. Instead, the migration would be divided into multiple sub-projects, each requiring its own business case. VD2 also points out that the business case may be revised during the migration. For these organizations, the business case can be part of the '*Governance'* (S16) step within the roadmap.

VD8 explains that within their organization, the baseline architecture is continuously updated and therefore does not need to be reconsidered after the business case is created. In addition, their organization develops a target architecture that looks several years ahead. Based on this, they create a migration plan using a roadmap for the coming years. As a result, VD8 argues that, in their case, the '*Create Business Case'* (S1) step takes place after the '*Create Migration Analysis & Planning'* (S??) step, since they only develop the business case for implementation before executing the plan. VD8 does acknowledge that this can be organization-specific.

6.2.2 Sensitivity

The goal of sensitivity validation is to discover the effects produced by the artifact within different contexts. The experts are asked how they think the roadmap will impact the migration strategies for migrating a monolithic architecture toward microservices in organizations in different sectors, in different environments, and for different types of systems.

6.2.2.1 Sector

Multiple interviewees agree that the roadmap is applicable beyond the financial sector and can be integrated into migration strategies across various industries (VD2, VD4, VD6). VD6 emphasizes the roadmap's general applicability, suggesting that although some challenges are more frequently experienced in the financial sector, organizations in other sectors also encounter similar issues. Sectors such as energy, logistics, and healthcare are mentioned. For instance, VD6 identified the Tax Authority (*Belastingdienst*) as a public organization with numerous legacy monolithic systems and significant regulatory pressure to modernize, making it a suitable candidate for applying the roadmap. Similarly, VD2 shares that organizations in sectors including healthcare, logistics, and construction could also benefit from adopting the roadmap. VD7 explains from their experience that the roadmap can be used within the financial sector and the funeral sector, as their organization is situated within both. However, VD1 and VD4 note that while the roadmap is broadly applicable, it may be particularly advantageous for financial organizations. These organizations were among the first to invest in IT systems and often operate decades-old IT landscapes. Moreover, regulatory requirements force long-term data retention, resulting in extensive data accumulation, such as transactional records, life insurance policies, and mortgage loan histories. According to VD6, financial organizations also tend to have sufficient financial resources necessary to undertake large-scale migration projects, increasing the feasibility of roadmap adoption in this sector. VD1 mentions that financial organizations are more likely to have the development capacity to be able to develop custom software, and that other sectors might not have this, therefore making this roadmap more applicable to the financial sector.

In contrast, VD5 indicates that the roadmap has limited relevance within their current organization, which primarily has newer, cloud-based systems and does not face the same legacy challenges. Nevertheless, they acknowledged that in their previous roles within the financial sector, the roadmap would have been significantly more relevant. VD5 further argued that the financial sector has a greater need to adopt microservices architectures, largely due to volatile scalability demands. For example, insurance organizations often experience a large increase in claims following extreme weather events, such as storms. In such cases, a microservices-based architecture enables dynamic scaling, which is crucial for managing sudden increases in workload efficiently.

Regulatory changes are often a key driver for initiating such migrations. Therefore, sectors that operate under strict regulatory oversight may particularly benefit from the adoption of this roadmap (VD4). VD6 explains that regulatory changes can be a key driver for performing a migration.

6.2.2.2 Organizational Age and Size

The roadmap is especially relevant for old organizations that maintain legacy systems within their IT landscape. In contrast, newer organizations, such as VD5's organization, often already have embraced SOA and microservice principles and developed modular, cloud-native systems. Legacy monolithic architectures are more commonly found in older organizations that began developing software prior to the emergence of these architectural paradigms. Therefore, the age of an organization appears to impact the roadmap's applicability and relevance.

Furthermore, the ability to carry out a migration of this scale requires sufficient organizational resources. This does not only include financial resources but also the availability of skilled personnel and time. VD4 highlights that their organization lacks the internal capacity to develop all software in-house, which requires them to rely more on commercial off-the-shelf or SaaS solutions. VD7 and VD8 also argue that their organization primarily purchases software, and developing in-house rarely happens. This has a direct impact on the organization's migration strategy and impacts the roadmap and the migration process. Overall, the roadmap better applies to larger organizations with more resources available.

VD8 argues that the size of the organization also has an impact on how EA is done within the organization. Within larger organizations, enterprise architects might have different responsibilities compared to smaller organizations. VD6, for example, explains that as enterprise architects, they are not involved in the buy or build decision that is done within the '*Execute'* (P2) phase of the roadmap, as this is the task for the solution architect. VD8, however, explains that within their organization, which is significantly smaller than the organization of VD6, as an enterprise architect, they are involved in this decision-making, and the decision to purchase a SaaS is, in practice, often already done when creating a target architecture. The size of the organization can thus impact the order in which the steps are executed within the migration process roadmap.

Overall, the roadmap better suits old and very large organizations.

6.2.2.3 Country

Another factor influencing the roadmap's relevance is the geographic context in which an organization operates. According to VD6, the IT landscapes within the Dutch financial sector are relatively mature compared to those in many other countries. This maturity implies that the roadmap may hold even greater relevance for financial organizations in other countries, particularly in regions where legacy systems are still more prevalent or where modernization efforts are at an earlier stage. In the United States of America, for example, IT landscapes in the financial sector are frequently more legacy compared to Europe [125].

However, an important consideration in this context is the willingness to adopt low-code technologies. As VD2 notes, organizations in the Netherlands demonstrate a higher level of maturity and openness in adopting LCDP compared to those in other countries. For example, VD2 mentions that in countries such as Australia and Germany, traditional high-code solutions are still more commonly preferred over low-code. This influences how organizations in different countries can use the roadmap, especially in contexts where low-code is not yet widely accepted or trusted.

6.2.2.4 Low-Code Development Platforms

Although this research focuses on the use of low-code in general and does not evaluate any specific LCDP, VD2 highlights that not all low-code platforms are suitable for enterprise-level software development. In their view, only enterprise-grade low-code platforms possess the necessary capabilities and functional sophistication to support the development of microservices, which is essential to the migration process. By contrast, no-code solutions are often too limited in functionality and flexibility to meet the demands of complex enterprise systems. As such, these might need to be considered outside the scope of this roadmap.

6.2.2.5 System

The type of system for the migration is a factor in determining the usability of the proposed roadmap. While the scope of this research is centered on the migration of legacy monolithic systems to microservices, not all monolithic systems – nor all of their components – are suitable for this. For instance, VD5 notes that certain legacy systems, particularly those running on mainframe infrastructure, may not technically support a microservices architecture. VD3 and VD8 argue that microservices often are not feasible for an organization’s core systems of record, but can be used for the systems of differentiation and systems of innovation around the core, following the definitions of Gartner [107]. VD7 also highlights challenges their organization has with microservices.

Moreover, beyond technical feasibility, not all systems are appropriate candidates for low-code microservices from a strategic or functional perspective. VD3 and VD6 emphasize that some migration efforts, especially those involving systems of record, may not allow for low-code solutions. Some of these decisional factors are mentioned in Section 5.3.2.2.

Despite these constraints, interviewees identified several use cases where the roadmap offers clear value. VD5, for example, references the migration of a large monolithic ERP system, where specific components needed to be extracted to improve scalability. Similarly, VD4 explains an example of an insurance policy administration system as a relevant case in which the roadmap could be effectively applied.

The ‘*Create Business Case’* (S1) step is an important step that determines whether it is a feasible option to migrate a monolithic system toward microservices. VD6 thinks that this step also adds value by filtering out systems that should not be migrated to microservices. Organizations must make informed assumptions about the scope and limitations of legacy IT migration, taking into account both the system architecture and the suitability of low-code technologies.

6.2.3 Requirement Satisfaction

The goal of requirement satisfaction validation is to determine whether the artifact satisfies the requirements. The requirements for this research are defined in Section 5.2. The experts are asked to comment on the satisfaction with the roadmaps’ requirements.

6.2.3.1 Ease of Use

All experts indicate that they find the roadmap easy to understand (VD1, VD2, VD3, VD4, VD5, VD6, VD7). VD4 and VD7 explain that the accompanying document of the roadmap definitely adds value and makes the roadmap easier to understand. VD7 does acknowledge that this roadmap is especially useful for people with a technology background and that it might need more elaboration if it were presented to the business. VD8 argues that adding definitions to certain steps, such as the '*Create Baseline Architecture'* (S3) step, could better clarify the scope of these steps.

6.2.3.2 Accuracy

Most experts agree that the roadmap accurately highlights the migration process for migrating an existing monolithic system to microservices (VD2, VD3, VD4, VD5, VD6, VD7). VD6 explains that the current state of the roadmap explains well how financial organizations can migrate their existing monolithic systems to microservices using low-code. VD7 agrees that this roadmap is very similar to the process they use for modernization of their monolithic systems toward microservices.

As explained in Section 6.2.1.1, the roadmap's level of granularity allows organizations to tailor specific steps to their internal processes or implementation preferences. Some steps might not fully suit every organization's migration process. One example of this is the '*Test & Validate Microservice'* (S12) step within the migration process. VD6 explains that their organization adopts a test-driven development approach, in which tests are defined prior to the development of the microservice. Organizations using this approach can integrate this practice into the roadmap with minimal impact on the overall migration flow. VD4 supports this flexibility, noting that the roadmap reflects a realistic view of how testing practices can vary across organizations. They agree that while test activities may differ in implementation, the placement of the '*Test & Validate Microservice'* (S12) step within the roadmap is appropriate. Relocating or removing this step could risk making the roadmap overly abstract and thus less actionable in practice.

VD6 highlights that, although the '*Deploy Release & Phase Out'* (S13) step is correct, it might not always be possible to phase out components in the monolithic system. As the monolith is not always developed in a modular manner, phasing parts of the code out might then not be possible, as they are too tightly coupled. Here, other alternative strategies, such as disabling parts of the code, might be better alternatives.

VD1 and VD3 explain that they disagree with the revision of baseline architecture and target architecture after every release. According to them, it is more likely that these are clearly defined before the migration starts, and after that, continuously updated. VD3 argues that the baseline architecture might be slightly revised to stay in line with the updates made to the IT landscape, but this can be a minor change. VD8 argues that mostly the solution and system architectures are updated here, and not the EA.

Other phases and steps that might require organizations to tailor the roadmap are the continuous architectural and business case revisions during the full migration process, as highlighted in Section 6.2.1.1.

6.2.3.3 Challenges

Multiple experts agree that the roadmap's accompanying document clearly highlights the challenges that financial organizations can face during the migration process (VD1, VD4). Other experts also mention that some of these challenges are generic and not only specific to the financial sector (VD2).

VD3 explains that their organization uses the ISO25010 framework to determine the quality of their systems and potentially microservices network. They therefore do not experience the challenge 'Hard to measure requirements'.

VD8 argues that the challenge 'Adherence to privacy and regulation' should also be incorporated in the *'Create Target Architecture*' (S4) step, as the enterprise architects should already take privacy and regulatory concerns into account when developing the target architecture.

6.2.3.4 Low-Code

Most experts indicate that the roadmap currently highlights well how low-code can be used within the migration process (VD2, VD4, VD5, VD6, VD8). VD2 explains that they agree that a business case should determine what technology is most fit to use for building a microservice. VD2 argues: "*Low-code is just another tool in the toolbox,*" and the organization should decide which tool is most fit for each microservice.

VD6 points out that the roadmap currently highlights well how low-code can be used for the migration process. They do argue that the roadmap can better elaborate on the decision between high-code and low-code. This is currently discussed in Chapter 5.3.2, but not elaborated upon within the accompanying document. VD6 explains that their organization uses the total cost of ownership to make this decision.

VD1, VD3, and VD8 argue that the use of low-code for prototyping is only beneficial if organizations also build their microservices using low-code. They explain that it is not likely that an organization would build a prototype with low-code and purchase a SaaS solution after that. Instead, they indicate it is more likely that an organization would prototype a microservice after deciding to custom-develop it.

7.Conclusion, Discussion, and Future Work

This Chapter concludes this research. In Section 7.1, the conclusion is presented. Section 7.2 discusses the findings of this research and their relevance. Section 7.3 highlights potential biases and limitations of the research. Sections 7.4 and 7.5 discuss the practical and theoretical implications respectively. Section 7.6 discusses potential areas for future work.

7.1 Conclusion

To conclude, financial organizations can use low-code as a tool for prototyping and developing microservices for migrating their existing systems from a monolithic architecture to microservices. Low-code can be used as a prototyping tool for user-interface-driven systems to quickly gather user feedback, or for backend processes to test integrations of the microservice within the microservices network. This allows developers and architects to gather feedback or find flaws before the microservice is purchased or developed. Due to the licensing fees of low-code development platforms, the use of low-code for solely this purpose is usually not beneficial. Low-code can be used as a tool to develop microservices, which can potentially speed up the migration process, due to the increased development speed that low-code brings. However, using low-code for this purpose brings some risks. Financial organizations should always consider whether the business case for using low-code is justifiable.

This answers the main research question: *How can financial organizations use low-code to migrate their existing systems from a monolithic architecture toward microservices?*

This research adopts a design science research methodology with the primary objective of developing a roadmap that defines the full process for migrating a monolithic architecture toward microservices with low-code. Four sub-questions are formulated and answered using a mixed-methodology approach, combining literature reviews and interviews with 16 experts. This input is used to create an initial version of a roadmap. This roadmap is revised after obtaining feedback from three enterprise architects in a workshop. The revised version of the roadmap is validated with eight expert opinion interviews.

The final version of the roadmap is displayed in Figure 20. The roadmap displays the phases and steps for the full migration process and explains how low-code can be used within this process. In the accompanying document (Table 13), these phases and steps are further elaborated upon, and challenges that financial organizations can face during the process are highlighted. Some steps are revised after the workshop, which are discussed in Section 5.5.

Overall, the migration process is organized into three phases. The first phase, ‘*Analyze’ (P1),* prepares the migration. The second phase, ‘*Execute*’ (P2), highlights an iterative approach to decompose the monolithic system toward microservices with the Strangler Fig Pattern. The third phase, ‘*Manage*’ (P3), outlines parallel management processes that must be maintained throughout the migration process.

The research goal for this research is defined as follows:

Improve	*monolithic to microservices migration strategies in financial organizations*
by	*designing a roadmap*
that	*demonstrates how financial organizations can use low-code in the process and addresses challenges during the migration*
in order to	*support financial organizations migrate their legacy monolithic systems to microservices with low-code*

The roadmap in Figure 20 together with Table 13 achieve this research goal.

7.2 Discussion

This Section discusses the relevance of the findings that have not been considered yet, and their interpretation and potential implications.

7.2.1 Microservices

This research explores how financial organizations can migrate their legacy monolithic systems to microservices. Microservices represent an architectural style that promotes modularity by decomposing applications into independently deployable services. This modularity enhances flexibility and scalability, allowing organizations to select the most appropriate technologies – whether high-code or low-code – for each individual microservice, thereby aligning technological choices with strategic goals.

One key advantage of microservices is technological flexibility. However, adopting a microservices architecture does not inherently guarantee better flexibility, maintainability, or scalability, nor are microservices always better compared to monolithic architectures. While academic research often characterizes monolithic systems as legacy and microservices as the preferred solution for scaling applications, this view can be overly simplistic. Microservices introduce substantial complexity, especially in development and management, and come with serious complications due to their distributed nature. These complexities are worsened when migrating legacy systems that have been operational for decades. Various experts indicate that microservices would be too complex for their organization.

It is essential to recognize that a successful microservices implementation requires significant organizational effort, including investment in tooling, governance, and development practices. As Brooks (1987) famously stated, *“There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity”* [117]. This observation underscores the notion that no technological solution – including microservices – should be regarded as a universal solution to architectural problems. In this context, Brooks’s quote seems especially relevant, as microservices are often considered a silver bullet.

One frequently discussed topic during the interviews conducted in this research is the granularity of microservices. In principle, the size of a microservice is defined by a single business capability or sub-domain. However, this definition remains subjective, as organizations have the autonomy to determine the scope of what constitutes a business capability or sub-domain within their own context. This means that the size of a microservice is not strictly fixed.

Despite this, the term *microservices* can be misleading, as it implies that these services should be very small in scale. Excessive decomposition into overly fine-grained microservices can result in a high number of services within the network, leading to increased complexity in system management and potential performance issues, such as network latency. Several interviewees indicate that their organizations are hesitant to adopt microservices due to concerns about the architectural complexity introduced by managing a highly distributed system.

It is important to recognize that microservices, contrary to what the term might suggest, are not necessarily “micro” in size. In some cases, incorporating a broader range of functionality within a single microservice can simplify network integrations and reduce operational overhead. This can significantly influence how IT professionals in the financial sector perceive and evaluate microservices as an architectural approach, emphasizing the importance of a balance between modularity and manageability.

7.2.2 Low-Code

One notable observation made by the researcher during the interviews was the variation in opinions regarding low-code development. While some organizations appear to be gradually moving away from low-code solutions, others are actively incorporating them into their strategies. The underlying reasons for this are not directly identified within the scope of this research. However, several potential contributing factors are observed.

One possible explanation may relate to the marketing strategies employed by low-code platform vendors. For a long period, many low-code providers positioned their platforms primarily for use by citizen developers, emphasizing the ease of creating business applications without formal software engineering skills. This approach was often framed as a solution for the high use of Excel in organizations. Although vendors such as Mendix have more recently shifted away from this citizen development model [126], the perception that low-code is intended for non-professionals continues to influence how it is received in some organizations. Indeed, several interviewees express skepticism about the quality of applications developed using low-code, with some describing these low-code applications as inferior to high-code and often lacking in adherence to non-functional requirements.

It is important to note, however, that the quality of software created with low-code is more related to the skills of the developers involved rather than the platform itself. Regardless of whether high-code or low-code is used, the design and development of high-quality software still requires good architectural knowledge and high technical expertise.

Another factor that may shape organizational attitudes toward low-code is their prior experience with specific platforms. Negative outcomes associated with a particular low-code solution may lead to broader skepticism toward low-code, even if such issues are not representative of all development platforms.

Ultimately, as multiple interviewees say, low-code should be regarded as one of many tools available within the software development toolbox. Like any other development approach, its adoption should be guided by assessment of its benefits, limitations, and risks, relative to the specific context and objectives of the organization.

7.2.3 AI within the Migration Process

Although the role of AI in the migration process is not considered within the scope of this research, it emerged as a recurring topic in discussions with interviewees. One challenge in the migration process is the lack of practical tools for automatic system decomposition. As Chapter 4 explains, approaches, including machine learning-based tools, are proposed in the literature. These are often impractical or have not been tested in practice.

Several interviewees point to the potential of generative AI as a promising development in this area. In particular, generative AI may offer significant advantages in analyzing legacy codebases, identifying system integrations, and even generating code. One interviewee notes that their organization has a dedicated research team to explore the application of generative AI within their modernization strategy.

Given its capabilities in code comprehension and generation, generative AI can play a role in automating parts of the migration process that are currently manual, time-consuming, and error-prone. Especially within the '*Analyze Sub-domain Decomposition'* (S6) step, it would therefore be valuable for future research to investigate how such technologies can be effectively integrated into migration strategies and what implications this might have for the roadmap proposed in this research.

7.3 Limitations and Biases

Like any other research, this research is bound to some biases and limitations. These biases and limitations are discussed in this Section.

7.3.1 Bias

The main research methodologies used in this research are literature reviews and interviews. The literature reviews consist of exploratory reviews and a systematic literature review. All of these methodologies are bound to bias.

Exploratory reviews are used in various parts of this research, including the answering of SQ2, SQ3, and SQ4. Exploratory reviews limit the findings of research in a specific field to a small sample of research that is found by the researcher [26]. Due to this, sampling bias is introduced, which can lead to the researcher missing out on relevant research, and them taking a specific view on the topic at hand. Due to this, not all research on these topics is considered. While the search queries and databases are well documented for transparency (Appendix A), a systematic approach might give additional interesting insights that this research has not been able to provide.

A systematic literature review was performed to answer SQ1. This methodology aims to lower sampling bias by creating a rigorous and reproducible process that will lead to more reliable results [127]. This research methodology, however, increases reviewer bias, which is the effect of the subjectivity of the researcher on the systematic literature. An example is the effects of the interpretation of the researcher on the results, especially in the synthesizing step [28]. This research attempted to minimize reviewer bias by comprehensively documenting the research methodology process and by also including the query selection process (Appendix B). It is still acknowledged that, when written by another researcher, it would have likely resulted in similar, but different, results.

Interviews are conducted in all phases of this design research and are used to answer SQ2, SQ3, and SQ4. Interviews are inherently bound to various biases. First, interviews are bound to sampling bias. A relevant example of this, within the context of this research, is the likelihood that professionals working with low-code are more positively inclined toward using low-code for the purposes of migrating legacy systems. This research tries to mitigate this bias by selecting a balanced group of professionals for all three interview groups. This was also done for the workshop and validation interviews.

One potential source of selection bias arises from the composition of participants in the input and validation interviews. The majority of input interviews were conducted with individuals from (very) large organizations, as well as consultants with primary experience in such organizations. In contrast, the validation interviews also included experts from smaller organizations. Within the scale that is used, these organizations still classify as large organizations, but the researcher still observes differences in large organizations and very large organizations. Very large organizations, for example, have more resources available for custom software development. These findings are discussed in Section 6.2.2.2. Conducting input interviews with participants from smaller organizations and using a different scale for measuring the organization size can help to mitigate this bias.

Interviews can also be biased due to interviewer bias. Here, the interviewer consciously or unconsciously influences the participant's responses by asking questions that suggest a correct answer. By asking neutral, open-ended questions and informing interviewees about this, the researcher tried to mitigate this bias. Bias can also occur in the thematic analysis process. As this is a subjective process, different results might be found by different researchers. Because of this, other researchers using the same interview data might end up with a different roadmap.

7.3.2 Classification of LCDPs

One limitation of this research is a lack of classification of LCDPs for the use of low-code in the migration process. As various experts indicate, not all LCDPs should be considered for migrating a monolithic architecture to microservices. The use of a classification of platforms could better define what LCDP should and should not be used for this migration process.

7.3.3 Treatment Implementation

One of the key limitations of this research is that the roadmap was not implemented in practice. The migration of a monolithic system to a microservices architecture is a complex and lengthy process that can span several years, making it infeasible to evaluate within the scope of this study. As a result, the validation of the roadmap relied solely on expert opinion interviews, in which participants were asked to reflect on and imagine its potential application in practice. This is the most popular and frequently used validation method in design research [38].

While expert opinion validation offers valuable insights, applying the roadmap in a practical case study would be valuable. Such an implementation would allow for the observation of real outcomes and provide empirical evidence regarding the roadmap's effectiveness, feasibility, and impact. Furthermore, a practical implementation would enable the definition and use of measurable criteria to assess the success of the migration process – a gap also highlighted by Tuusjärvi et al. (2024) [20].

7.3.4 Absence of Scientific Methodology for Architectural Roadmap Creation

Another limitation of this research is the absence of a standardized scientific methodology for the development of the architectural roadmap. Architectural roadmaps highlight milestones, phases, and steps that need to be performed for migrating a baseline architecture toward a target architecture [42].

The roadmap is designed based on insights gathered from literature and interviews, with the researcher using this input to design a set of phases and steps. As discussed in Section 7.3.1, this process is inherently subjective, which negatively impacts reproducibility and transparency. It is likely that a different researcher, working with the same data, might have designed a different roadmap.

Despite efforts to identify a suitable methodological framework for designing an architectural process roadmap, no widely accepted scientific methodology is found. The lack of such a framework highlights a gap in the literature and practice. Establishing a standardized methodology for roadmap development could enhance the reproducibility of similar research efforts, but also strengthen EA frameworks such as TOGAF. TOGAF highlights the architectural roadmap as one of the key deliverables in the ADM Cycle, but does not specify how it should be created [45].

7.4 Practical Implications

This research has several practical implications for organizations.

First, this research proposes a roadmap for financial organizations to migrate their existing monolithic systems toward microservices with low-code. Organizations within the financial sector are increasingly facing external pressures such as evolving regulatory requirements and competition from FinTech, which force them to modernize their IT landscapes. These landscapes are often composed of legacy monolithic systems, which can be migrated to microservices in order to enhance flexibility, maintainability, and scalability. However, the migration from monolithic architectures to microservices is inherently complex and, depending on the system, may take several years to complete. Despite this complexity, there is a lack of practical tools to support practitioners in navigating such migrations. This roadmap solves this gap. It does this by displaying the process of how to migrate a monolithic system toward microservices. Financial organizations can use this roadmap to plan their migrations. The roadmap represents a step toward the standardization of modernization strategies, offering a structured yet adaptable framework to guide organizations through the complexities of legacy system migration. Organizations in other sectors than the financial sector might also benefit from this roadmap, as multiple experts have indicated that it is generic enough for this purpose.

Second, this research discusses the role of low-code in the modernization of legacy IT landscapes within the financial sector. This research provides benefits, limitations, and potential risks that can be associated with using low-code for migrating monolithic systems to microservices. This allows practitioners within financial organizations to make a more informed decision when creating a business case for either using low-code or high-code solutions.

Third, this research creates a comprehensive overview of the challenges that financial organizations can face while migrating their monolithic architecture to microservices. This overview is useful, as it can make practitioners in financial organizations aware of the potential challenges they can face during the migration process and mitigate their impact before they occur.

Fourth, this research provides a trade-off for financial organizations between monolithic architecture and microservices. It gives a comprehensive overview of monolithic architectures and microservices, and their benefits and challenges. This can be used by organizations to consider migrating to microservices or to stay with their monolithic architecture.

7.5 Theoretical Implications

This research has several implications for theoretical academic literature.

First, this research provides a comprehensive list of benefits and challenges that are associated with monolithic architecture and microservices. While existing research very frequently mentions a few of these benefits and challenges, they do not paint the full picture of the implications of both architectural styles. The danger of this is that research often only looks into specific benefits and challenges, which results in them often considering microservices as a silver bullet, while it clearly also has practical disadvantages. This research aimed to bring together the existing scattered research into one comprehensive result, which can be used by researchers as a foundation for research on migrating from monolithic architecture to microservices.

Second, this research investigates challenges that practitioners can face during the migration from monolithic architecture to microservices. This complements the already existing literature on this topic. Future research can use these challenges to get a more comprehensive overview of the problems that organizations might face within their migration strategies and processes.

Third, this research aims to contribute to the academic paradigm of low-code by investigating the role it can play in migrating monolithic systems to microservices. Although low-code platforms are gaining popularity in both industry and practice, the academic literature about the use of low-code remains limited. While more research is being done, there is still a lack, especially on how organizations can use low-code. On top of this, the low-code industry itself seems to play a prominent role in low-code development research, leading to possibly biased results [128]. This research took into consideration experiences with low-code from a big group of professionals from both the low-code industry as well as enterprise architects and IT professionals in the financial sector. This balanced approach aims to mitigate bias and provide a more objective perspective, avoiding the self-serving literature captured by the well-known Dutch expression: “Wij van WC Eend adviseren WC Eend” (we recommend our own products).

Lastly, this research proposes a roadmap that consists of a generic component and a component specific to the financial sector. The generic component consists of the phases and steps to take to migrate a monolithic architecture to microservices. The specific component consists of the trade-off between monolithic architecture and microservices, and the challenges that financial organizations can face in the migration process. Research can replace the component specific to the financial sector with any other sector to change the scope of this research. Other researchers can thus make a trade-off specific to another sector and find challenges that organizations in these sectors might experience during the migration process. Multiple experts indicate that the roadmap can be useful for other sectors, indicating that this will likely add value to these sectors as well.

7.6 Future Work

This Section highlights potential areas of future work.

7.6.1 Treatment Implementation in Case Study

One of the key limitations of this research is that the roadmap was not implemented in practice. Such an implementation would allow for the observation of real outcomes and provide empirical evidence regarding the roadmap's effectiveness, feasibility, and impact. Furthermore, a practical implementation would enable the definition and use of measurable criteria to assess the success of the migration process – a gap also highlighted by Tuusjärvi et al. (2024) [20]. Future research could use the roadmap in a case study to validate whether the expected effects from the expert opinions in the validation interview are valid in practice.

7.6.2 Decision-Making Model between High-Code and Low-Code

While this research identifies various factors that may influence the decision between using high-code or low-code for migrating monolithic systems to microservices, it does not provide a definitive framework for determining when one should be preferred over the other. Future research could focus on developing a decision-making model that more clearly defines the conditions under which each approach is most appropriate. As one expert notes during the validation interview, a valuable starting point for such a model would be an analysis of the total cost of ownership.

7.6.3 Decision-Making Model for LCDP

A limitation of this research is that it does not consider specific LCDPs for the migration process, although experts think that not all LCDPs are suited for this. Future research could explore creating a decision-making model to decide which LCDP is suitable for the migration process. Sahay et al. (2020) [129] create a comparison between different LCDPs and their features, which can be used as a foundation for this model. This research, however, is already relatively old and needs to be refined [129], as many low-code suppliers have updated and improved their platforms. They also do not compare the EAI patterns, which is critical for LCDP decision-making for microservices.

7.6.4 Roadmap within Other Sectors

As several experts mentioned in the validation interviews, future research could explore the application of the roadmap beyond the financial sector. Experts mention that the energy sector, the healthcare sector, the logistics sector, the construction sector, and the tax authority (*Belastingdienst*) would be interesting parties for a similar roadmap.

7.6.5 Generative AI in Decomposing

As Section 7.2.3 highlights, several experts indicate that generative AI has the potential to play a significant role in the migration process by automating aspects of system decomposition. Given the limited research currently available on this emerging topic, further investigation into the practical applications and limitations of generative AI in this context would be highly valuable. Exploring its role could discover new opportunities for accelerating and optimizing the migration from monolithic architectures to microservices and modernizing legacy IT landscapes.

7.6.6 Scientific Methodology for Creating Architectural Roadmaps

A limitation of this research is the absence of a standardized scientific methodology for the development of the architectural roadmap. Future research could focus on defining a standardized methodology for developing architectural roadmaps to improve transparency and reproducibility of similar design research. This can also be used by EA frameworks such as TOGAF to better specify how enterprise architects can create architectural roadmaps within their organizations.

7.6.7 Addressing the Challenges during the Migration Process

One of the roadmap's requirements is to highlight the challenges financial organizations may encounter during migration. The purpose of this is to raise awareness among practitioners, enabling them to identify strategies to mitigate potential impacts. Plenty of literature already exists on how to manage several of these challenges. Future research could build on this by linking the challenges identified in this study with approaches for addressing them, thereby further supporting financial organizations throughout the migration process.

References

[1] S. Kraus, P. Jones, N. Kailer, A. Weinmann, N. Chaparro-Banegas, and N. Roig-Tierno, “Digital Transformation: An Overview of the Current State of the Art of Research,” *Sage Open*, vol. 11, no. 3, Jul. 2021, doi: 10.1177/21582440211047576.

[2] “What is digital transformation?,” McKinsey & Company. Accessed: Jan. 01, 2024. [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-digital-transformation

[3] “Digital Transformation Delivery Services - KPMG Netherlands,” KPMG. Accessed: Dec. 20, 2024. [Online]. Available: https://kpmg.com/nl/en/home/services/advisory/technology/digital-transformation-delivery-services.html

[4] J. J. Korhonen and M. Halén, “Enterprise Architecture for Digital Transformation,” in *2017 IEEE 19th Conference on Business Informatics (CBI)*, Jul. 2017, pp. 349–358. doi: 10.1109/CBI.2017.45.

[5] S. Balasubramanian, “Digital transformation for the risk and compliance functions.” Deloitte, 2018. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/finance/us-digital-transformation-for-the-risk-and-compliance-functions.pdf

[6] “Monolithic vs Microservices - Difference Between Software Development Architectures- AWS.” Amazon Web Services, Inc., 2023. Accessed: Dec. 28, 2024. [Online]. Available: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/

[7] S. A. A. Naqvi, M. P. Zimmer, P. Drews, K. Lemmer, and R. C. Basole, “The Low-Code Phenomenon: Mapping the Intellectual Structure of Research,” presented at the Proceedings of the Annual Hawaii International Conference on System Sciences, 2024, pp. 7800–7809.

[8] K. Rokis and M. Kirikova, “Challenges of Low-Code/No-Code Software Development: A Literature Review,” in *Perspectives in Business Informatics Research*, Ē. Nazaruka, K. Sandkuhl, and U. Seigerroth, Eds., Cham: Springer International Publishing, 2022, pp. 3–17. doi: 10.1007/978-3-031-16947-2_1.

[9] K. Rokis and M. Kirikova, “Exploring Low-Code Development: A Comprehensive Literature Review,” *Complex Syst. Inform. Model. Q.*, no. 36, pp. 68–86, Oct. 2023, doi: 10.7250/csimq.2023-36.04.

[10] G. Morais, M. Adda, and D. Bork, “Breaking Down Barriers: Building Sustainable Microservices Architectures with Model-Driven Engineering,” in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, in MODELS Companion ’24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 528–532. doi: 10.1145/3652620.3687799.

[11] REST, “Publish a REST Service.” Mendix Documentation, Dec. 2024. Accessed: Jan. 30, 2025. [Online]. Available: https://docs.mendix.com/refguide/publish-a-rest-service/

[12] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, “Low-code development and model-driven engineering: Two sides of the same coin?,” *Softw. Syst. Model.*, vol. 21, no. 2, pp. 437–446, Apr. 2022, doi: 10.1007/s10270-021-00970-2.

[13] P. Schueffel, “Taming the Beast: A Scientific Definition of Fintech,” *J. Innov. Manag.*, vol. 4, no. 4, Art. no. 4, 2016, doi: 10.24840/2183-0606_004.004_0004.

[14] “Digital Transformation and Innovation | Deloitte Nederland.” Accessed: Dec. 20, 2024. [Online]. Available: https://www.deloitte.com/nl/nl/services/risk-advisory/services/digital-transformation-and-innovation.html

[15] “State of the Banks FY24 (EN).” Accessed: May 16, 2025. [Online]. Available: https://indialogue.io/clients/reports/public/authenticate?returnUrl=%2Fclients%2Freports%2Fpublic%2F678a175e8c18f73564db32ac%2F

[16] E. C. Bank, “Digital Finance: does it change the trade-off between risk and resilience?,” European Central Bank - Banking supervision. Accessed: Jan. 01, 2024. [Online]. Available: https://www.bankingsupervision.europa.eu/press/speeches/date/2024/html/ssm.sp240322~7bbfe50962.en.html

[17] T. Daru, “Strategic Product Modernization,” Deloitte United States. Accessed: Nov. 18, 2024. [Online]. Available: https://www2.deloitte.com/us/en/pages/consulting/articles/digital-transformation-in-banking.html

[18] A. Premchand, S. M, and S. Sankar, “Roadmap for simplification of enterprise architecture at financial institutions,” in *2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)*, Apr. 2016, pp. 043–051. doi: 10.1109/ICCPEIC.2016.7557221.

[19] “Disruption from Within: De-risking Banking Transformations at Speed,” Bain. Accessed: Jan. 01, 2024. [Online]. Available: https://www.bain.com/how-we-help/disruption-from-within-de-risking-banking-transformation-at-speed/

[20] K. Tuusjärvi, J. Kasurinen, and S. Hyrynsalmi, "Migrating a Legacy System to a Microservice Architecture," *E-Inform. Softw. Eng. J.*, vol. 18, no. 1, 2024, doi: 10.37190/e-Inf240104.

[21] D. Wolfart *et al.*, "Modernizing legacy systems with microservices: A roadmap," presented at the ACM International Conference Proceeding Series, 2021, pp. 149–159. doi: 10.1145/3463274.3463334.

[22] V. Velepucha and P. Flores, "Monoliths to microservices - Migration Problems and Challenges: A SMS," in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, Mar. 2021, pp. 135–142. doi: 10.1109/ICI2ST51859.2021.00027.

[23] "CFO Agenda for Elevating Finance." KPMG, 2023. Accessed: Dec. 24, 2024. [Online]. Available: https://kpmg.com/us/en/articles/2023/cfo-agenda-for-elevating-finance.html

[24] P. Johannesson and E. Perjons, *An Introduction to Design Science*. Cham: Springer International Publishing, 2014. doi: 10.1007/978-3-319-10632-8.

[25] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer, 2014. doi: 10.1007/978-3-662-43839-8.

[26] P. Dash, "Analysis of Literature Review in Cases of Exploratory Research," Dec. 17, 2019, *Social Science Research Network, Rochester, NY*: 3555628. doi: 10.2139/ssrn.3555628.

[27] A. Valente, M. Holanda, A. M. Mariano, R. Furuta, and D. Da Silva, "Analysis of Academic Databases for Literature Review in the Computer Science Education Field," in *2022 IEEE Frontiers in Education Conference (FIE)*, Oct. 2022, pp. 1–7. doi: 10.1109/FIE56618.2022.9962393.

[28] C. Okoli, "A Guide to Conducting a Standalone Systematic Literature Review," *Commun. Assoc. Inf. Syst.*, vol. 37, Nov. 2015, doi: 10.17705/1CAIS.03743.

[29] B. Kitchenham and S. M. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," ResearchGate. Accessed: Nov. 26, 2024. [Online]. Available: https://www.researchgate.net/publication/302924724_Guidelines_for_performing_Systematic_Literature_Reviews_in_Software_Engineering

[30] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – A systematic literature

review," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 7–15, Jan. 2009, doi: 10.1016/j.infsof.2008.09.009.

[31] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2049–2075, Dec. 2013, doi: 10.1016/j.infsof.2013.07.010.

[32] J. F. Wolfswinkel, E. Furtmueller, and C. P. M. Wilderom, "Using grounded theory as a method for rigorously reviewing literature," *Eur. J. Inf. Syst.*, vol. 22, no. 1, pp. 45–55, Jan. 2013, doi: 10.1057/ejis.2011.51.

[33] M. N. K. Saunders, P. Lewis, and A. Thornhill, *Research methods for business students*, Ninth edition. Harlow, England: Pearson, 2023.

[34] "Ethics (BMS/domain HSS) | Faculty of Behavioural, Management and Social sciences (BMS)," Universiteit Twente. Accessed: Feb. 19, 2025. [Online]. Available: https://www.utwente.nl/en/bms/research/ethics-domainHSS/

[35] "Transkriptor: Transcribe Audio to Text - AI Transcription," Transkriptor. Accessed: Mar. 20, 2025. [Online]. Available: https://transkriptor.com

[36] "Glossary:Enterprise size." Accessed: Mar. 24, 2025. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Enterprise_size

[37] M. Maguire and B. Delahunt, "Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars.," *Irel. J. High. Educ.*, vol. 9, no. 3, Art. no. 3, Oct. 2017, doi: 10.62707/aishej.v9i3.335.

[38] M. Eisenmann, Grauberger ,Patric, Üreten ,Selin, Krause ,Dieter, and S. and Matthiesen, "Design method validation – an investigation of the current practice in design research," *J. Eng. Des.*, vol. 32, no. 11, pp. 621–645, Nov. 2021, doi: 10.1080/09544828.2021.1950655.

[39] M. Lankhorst *et al.*, *Enterprise architecture at work: Modelling, communication, and analysis*. in Enterprise Architecture at Work: Modelling, Communication, and Analysis. 2005, p. 334. doi: 10.1007/3-540-27505-3.

[40] H. Jonkers, M. M. Lankhorst, H. W. L. Ter Doest, F. Arbab, H. Bosma, and R. J. Wieringa, "Enterprise architecture: Management tool and blueprint for the organisation," *Inf. Syst. Front.*, vol. 8, no. 2, pp. 63–66, Feb. 2006, doi: 10.1007/s10796-006-7970-2.

[41] D. Simon, K. Fischbach, and D. Schoder, "Enterprise architecture management and its role in corporate strategic management," *Inf. Syst. E-Bus. Manag.*, vol. 12, no. 1, pp. 5–42, 2014, doi: 10.1007/s10257-013-0213-4.

[42] H. Shah and M. El Kourdi, “Frameworks for Enterprise Architecture,” *IT Prof.*, vol. 9, no. 5, pp. 36–41, Sep. 2007, doi: 10.1109/MITP.2007.86.

[43] J. A. Zachman, “A framework for information systems architecture,” *IBM Syst. J.*, vol. 26, no. 3, pp. 276–292, 1987, doi: 10.1147/sj.263.0276.

[44] “TOGAF.” www.opengroup.org, 2025. Accessed: Jan. 02, 2025. [Online]. Available: https://www.opengroup.org/togaf

[45] “TOGAF® Standard — Introduction - Introduction.” Opengroup.org, 2022. Accessed: Jan. 02, 2025. [Online]. Available: https://pubs.opengroup.org/togaf-standard/adm/chap01.html#tag_01

[46] H. Jonkers, E. Proper, M. M. Lankhorst, D. A. C. Quartel, and M.-E. Iacob, “ArchiMate(R) for Integrated Modelling Throughout the Architecture Development and Implementation Cycle,” in *2011 IEEE 13th Conference on Commerce and Enterprise Computing*, Sep. 2011, pp. 294–301. doi: 10.1109/CEC.2011.52.

[47] T. O. Group, “Introduction: ArchiMate® 3.2 Specification.” Opengroup.org, 2023. Accessed: Jan. 06, 2025. [Online]. Available: https://pubs.opengroup.org/architecture/archimate3-doc/index.html

[48] “The ArchiMate® Enterprise Architecture Modeling Language.” www.opengroup.org, 2025. Accessed: Jan. 06, 2025. [Online]. Available: https://www.opengroup.org/archimate-forum/archimate-overview

[49] G. Hohpe and B. Woolf, *Enterprise integration patterns: designing, building, and deploying messaging solutions*, 17. print. in The Addison-Wesley signature series. Boston Munich: Addison-Wesley, 2013.

[50] GeeksforGeeks, “Monolithic Architecture System Design.” GeeksforGeeks, Apr. 2024. Accessed: Jan. 06, 2025. [Online]. Available: https://www.geeksforgeeks.org/monolithic-architecture-system-design/

[51] C. Bandara and I. Perera, “Transforming monolithic systems to microservices - An analysis toolkit for legacy code evaluation,” presented at the 20th International Conference on Advances in ICT for Emerging Regions, ICTer 2020 - Proceedings, 2020, pp. 95–100. doi: 10.1109/ICTer51097.2020.9325443.

[52] J. Kazanavičius and D. Mažeika, “Migrating Legacy Software to Microservices Architecture,” in *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, Apr. 2019, pp. 1–5. doi: 10.1109/eStream.2019.8732170.

[53] R. Chris, *Microservices Patterns : With Examples in Java.* Manning, 2019. [Online]. Available:

http://ezproxy2.utwente.nl/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2949079&site=ehost-live&ebv=EK&ppid=Page-__-1

[54] “ArchiMate Community EAI-Shared-Database.” Archimate-community.org, 2025. Accessed: Jan. 13, 2025. [Online]. Available: https://www.archimate-community.org/#!patterns/EAI-Shared-Database.md

[55] “ArchiMate Community EAI-File-Transfer.” Archimate-community.org, 2025. Accessed: Jan. 13, 2025. [Online]. Available: https://www.archimate-community.org/#!patterns/EAI-File-Transfer.md

[56] N. Dragoni *et al.*, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

[57] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, “Microservices: architecture, container, and challenges,” in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Dec. 2020, pp. 629–635. doi: 10.1109/QRS-C51114.2020.00107.

[58] D. Saxena and B. Bhowmik, “Paradigm Shift from Monolithic to Microservices,” presented at the RASSE 2023 - IEEE International Conference on Recent Advances in Systems Science and Engineering, Proceedings, 2023. doi: 10.1109/RASSE60029.2023.10363466.

[59] S. V. Zykov, “Agile services,” *Smart Innovation, Systems and Technologies*, vol. 92. pp. 65–105, 2018. doi: 10.1007/978-3-319-77917-1_3.

[60] A. Bucchiarone *et al.*, Eds., *Microservices: Science and Engineering*. Cham: Springer International Publishing, 2020. doi: 10.1007/978-3-030-31646-4.

[61] “ArchiMate Community EAI-RPC.” Archimate-community.org, 2025. Accessed: Jan. 13, 2025. [Online]. Available: https://www.archimate-community.org/#!patterns/EAI-RPC.md

[62] “ArchiMate Community EAI-REST-API.” Archimate-community.org, 2025. Accessed: Jan. 13, 2025. [Online]. Available: https://www.archimate-community.org/#!patterns/EAI-REST-API.md

[63] “ArchiMate Community EAI-Messaging.” Archimate-community.org, 2025. Accessed: Jan. 13, 2025. [Online]. Available: https://www.archimate-community.org/#!patterns/EAI-Messaging.md

[64] “Source code,” *Wikipedia*. Jun. 25, 2025. Accessed: Jun. 30, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Source_code&oldid=1297289301

[65] “Enterprise Low-Code Application Platforms Reviews and Ratings 2025 | Gartner Peer Insights.” Accessed: Jul. 01, 2025. [Online]. Available: https://www.gartner.com/reviews/market/enterprise-low-code-application-platform

[66] “Magic Quadrant for Enterprise Low-Code Application Platforms,” Gartner. Accessed: Apr. 07, 2025. [Online]. Available: https://www.gartner.com/en/documents/5844247

[67] “New Development Platforms Emerge For Customer-Facing...,” Forrester. Accessed: Feb. 12, 2025. [Online]. Available: https://www.forrester.com/report/new-development-platforms-emerge-for-customer-facing-applications/RES113411

[68] J. Hintsch, D. Staegemann, M. Volk, and K. Turowski, “Low-code Development Platform Usage: Towards Bringing Citizen Development and Enterprise IT into Harmony,” *ACIS 2021 Proc.*, Jan. 2021, [Online]. Available: https://aisel.aisnet.org/acis2021/11

[69] B. Binzer, E. Elshan, D. Fuerstenau, and T. J. Winkler, “Establishing a Low-Code/No-Code-Enabled Citizen Development Strategy,” *ResearchGate*, no. MIS Quarterly Executive 23(3), pp. 253–273, Sep. 2024, doi: 10.17705/2msqe.00097.

[70] D. Hoogsteen and H. Borgman, “Empower the Workforce, Empower the Company? Citizen Development Adoption,” *Hawaii Int. Conf. Syst. Sci. 2022 HICSS-55*, Jan. 2022, [Online]. Available: https://aisel.aisnet.org/hicss-55/in/human-centricity/9

[71] “Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences,” Gartner. Accessed: Feb. 21, 2025. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences

[72] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, “Characteristics and Challenges of Low-Code Development: The Practitioners’ Perspective,” in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, in ESEM ’21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 1–11. doi: 10.1145/3475716.3475782.

[73] S. Käss, S. Strahringer, and M. Westner, “Drivers and Inhibitors of Low Code Development Platform Adoption,” in *2022 IEEE 24th Conference on Business Informatics (CBI)*, Jun. 2022, pp. 196–205. doi: 10.1109/CBI54897.2022.00028.

[74] V. Velepucha and P. Flores, “A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges,” *IEEE Access*, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.

[75] S. Baškarada, V. Nguyen, and A. Koronios, “Architecting Microservices: Practical Opportunities and Challenges,” *J. Comput. Inf. Syst.*, vol. 60, no. 5, pp. 428–436, 2020, doi: 10.1080/08874417.2018.1520056.

[76] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges When Moving from Monolith to Microservice Architecture,” presented at the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2018, pp. 32–47. doi: 10.1007/978-3-319-74433-9_3.

[77] U. Chouhan, V. Tiwari, and H. Kumar, “Comparing Microservices and Monolithic Applications in a DevOps Context,” presented at the 2023 3rd Asian Conference on Innovation in Technology, ASIANCON 2023, 2023. doi: 10.1109/ASIANCON58793.2023.10270721.

[78] M. Villamizar *et al.*, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” presented at the 2015 10th Colombian Computing Conference, 10CCC 2015, 2015, pp. 583–590. doi: 10.1109/ColumbianCC.2015.7333476.

[79] M. Kyryk, O. Tymchenko, N. Pleskanka, and M. Pleskanka, “Methods and process of service migration from monolithic architecture to microservices,” in *2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET)*, Feb. 2022, pp. 553–558. doi: 10.1109/TCSET55632.2022.9767055.

[80] K. Hmue, M. P. Phyu, and A. M. M. Paing, “Microservices vs Monolith: A Comparative Analysis and Problem-Solving Approach in Web Development Area,” in *2024 5th International Conference on Advanced Information Technologies (ICAIT)*, Nov. 2024, pp. 1–5. doi: 10.1109/ICAIT65209.2024.10754922.

[81] J. Doležal and A. Buchalcevová, “MIGRATION FROM MONOLITHIC TO MICROSERVICE ARCHITECTURE: RESEARCH OF IMPACTS ON AGILITY,” presented at the IDIMT 2022 - Digitalization of Society, Business and Management in a Pandemic: 30th Interdisciplinary Information Management Talks, 2022, pp. 401–411. doi: 10.35011/IDIMT-2022-401.

[82] K. Gos and W. Zabierowski, “The Comparison of Microservice and Monolithic Architecture,” in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, Apr. 2020, pp. 150–153. doi: 10.1109/MEMSTECH49584.2020.9109514.

[83] D. Gravanis, G. Kakarontzas, and V. Gerogiannis, “You don’t need a Microservices Architecture (yet): Monoliths may do the trick,” presented at the ACM

International Conference Proceeding Series, 2021, pp. 39–44. doi: 10.1145/3501774.3501780.

[84] S. Newman, “Monolith to Microservices,” vol. 1st Edition, Dec. 2019.

[85] J. Zaki, S. M. R. Islam, N. S. Alghamdi, M. Abdullah-Al-Wadud, and K.-S. Kwak, “Introducing Cloud-Assisted Micro-Service-Based Software Development Framework for Healthcare Systems,” *IEEE Access*, vol. 10, pp. 33332–33348, 2022, doi: 10.1109/ACCESS.2022.3161455.

[86] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar, “Microservices: Migration of a Mission Critical System,” *IEEE Trans. Serv. Comput.*, vol. 14, no. 5, pp. 1464–1477, 2021, doi: 10.1109/TSC.2018.2889087.

[87] B. A. Pinos-Figueroa and G. A. Léon-Paredes, “An Approach of a Migration Process from a Legacy Web Management System with a Monolithic Architecture to a Modern Microservices-Based Architecture of a Tourism Services Company,” presented at the Proceedings - 2023 11th International Conference in Software Engineering Research and Innovation, CONISOFT 2023, 2023, pp. 9–17. doi: 10.1109/CONISOFT58849.2023.00012.

[88] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang, “Microservice Architecture in Reality: An Industrial Inquiry,” in *2019 IEEE International Conference on Software Architecture (ICSA)*, Hamburg, Germany: IEEE, Mar. 2019, pp. 51–60. doi: 10.1109/ICSA.2019.00014.

[89] G. Pathak and M. Singh, “A Review of Cloud Microservices Architecture for Modern Applications,” in *2023 World Conference on Communication & Computing (WCONF)*, Jul. 2023, pp. 1–7. doi: 10.1109/WCONF58270.2023.10235199.

[90] “Operationele uitdagingen: De impact van de nieuwe Pensioenwet.” Accessed: Jan. 15, 2025. [Online]. Available: https://www.banken.nl/nieuws/25606/operationele-uitdagingen-de-impact-van-de-nieuwe-pensioenwet

[91] “COBOL blues,” Reuters. Accessed: Jan. 15, 2025. [Online]. Available: http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html

[92] A. Carrasco, B. V. Bladel, and S. Demeyer, “Migrating towards microservices: migration and architecture smells,” in *Proceedings of the 2nd International Workshop on Refactoring*, Montpellier France: ACM, Sep. 2018, pp. 1–6. doi: 10.1145/3242163.3242164.

[93] J. Vučković, “You Are Not Netflix,” in *Microservices: Science and Engineering*, A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A.

Sadovykh, Eds., Cham: Springer International Publishing, 2020, pp. 333–346. doi: 10.1007/978-3-030-31646-4_13.

[94] M. V. Aikins, “DISTRIBUTED STORAGE SYSTEMS AND HOW THEY HANDLE DATA CONSISTENCY AND RELIABILITY,” *ResearchGate*, vol. 5, no. 1, pp. 84–90, Dec. 2023.

[95] C. Britton and P. Bye, *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems (2nd Edition)*. Pearson Addison Wesley, 2004.

[96] Sh. Salii, J. Ajdari, and Xh. Zenuni, “Migrating to a microservice architecture: benefits and challenges,” presented at the 2023 46th ICT and Electronics Convention, MIPRO 2023 - Proceedings, 2023, pp. 1670–1677. doi: 10.23919/MIPRO57284.2023.10159894.

[97] 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)*, Sep. 2019, pp. 481–490. doi: 10.1109/ICSME.2019.00081.

[98] “Digital Finance Transformation,” Gartner. Accessed: Jan. 15, 2025. [Online]. Available: https://www.gartner.com/en/finance/topics/digital-finance-transformation

[99] A. Poniszewska-Maranda, J. Macloch, B. Borowska, and W. Maranda, “Mechanisms for Transition from Monolithic to Distributed Architecture in Software Development Process,” presented at the Proceedings - IEEE Computer Society’s Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS, 2021. doi: 10.1109/MASCOTS53633.2021.9614287.

[100] A. Henry and Y. Ridene, “Migrating to Microservices,” in *Microservices: Science and Engineering*, A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, Eds., Cham: Springer International Publishing, 2020, pp. 45–72. doi: 10.1007/978-3-030-31646-4_3.

[101] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, “Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages,” presented at the ACM International Conference Proceeding Series, 2017. doi: 10.1145/3120459.3120483.

[102] H. Hassan, M. Abdel-Fattah, and W. Mohamed, “Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review,” *ResearchGate*, Nov. 2024, doi: 10.14569/IJACSA.2024.0151013.

[103] A. Bucchiarone, K. Soysal, and C. Guidi, “A Model-Driven Approach Towards Automatic Migration to Microservices,” in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds., Cham: Springer International Publishing, 2020, pp. 15–36. doi: 10.1007/978-3-030-39306-9_2.

[104] I. Trabelsi *et al.*, “From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis,” *J. Softw. Evol. Process*, vol. 35, no. 10, 2023, doi: 10.1002/smr.2503.

[105] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, “From Monolith to Microservices: A Classification of Refactoring Approaches,” vol. 11350, 2019, pp. 128–141. doi: 10.1007/978-3-030-06019-0_10.

[106] H. Luthria and F. A. Rabhi, “Building the Business Case for SOA: A Study of the Business Drivers for Technology Infrastructure Supporting Financial Service Institutions,” in *Enterprise Applications and Services in the Finance Industry*, D. Kundisch, D. J. Veit, T. Weitzel, and C. Weinhardt, Eds., Berlin, Heidelberg: Springer, 2009, pp. 94–107. doi: 10.1007/978-3-642-01197-9_7.

[107] “Accelerating Innovation by Adopting a Pace-Layered Application Strategy,” Gartner. Accessed: Apr. 13, 2025. [Online]. Available: https://www.gartner.com/en/documents/1890915

[108] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis, “User Acceptance of Information Technology: Toward a Unified View,” Sep. 01, 2003, *Social Science Research Network, Rochester, NY*: 3375136. Accessed: May 22, 2025. [Online]. Available: https://papers.ssrn.com/abstract=3375136

[109] L. De Lauretis, “From monolithic architecture to microservices architecture,” presented at the Proceedings - 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops, ISSREW 2019, 2019, pp. 93–96. doi: 10.1109/ISSREW.2019.00050.

[110] C.-Y. Li, S.-P. Ma, and T.-W. Lu, “Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System,” in *2020 International Computer Symposium (ICS)*, Dec. 2020, pp. 519–524. doi: 10.1109/ICS51289.2020.00107.

[111] J. Kazanavičius and D. Mažeika, “An Approach to Migrate from Legacy Monolithic Application into Microservice Architecture,” in *2023 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, Apr. 2023, pp. 1–6. doi: 10.1109/eStream59056.2023.10135021.

[112] A. Premchand and A. Choudhry, “Architecture simplification at large institutions using micro services,” presented at the Proceedings of the 2018 International

Conference On Communication, Computing and Internet of Things, IC3IoT 2018, 2018, pp. 30–35. doi: 10.1109/IC3IoT.2018.8668173.

[113] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, “Microservices migration patterns,” *Softw. Pract. Exp.*, vol. 48, no. 11, pp. 2019–2042, 2018, doi: 10.1002/spe.2608.

[114] H. Knoche and W. Hasselbring, “Using Microservices for Legacy Software Modernization,” *IEEE Softw.*, vol. 35, no. 3, pp. 44–49, May 2018, doi: 10.1109/MS.2018.2141035.

[115] “Microservices Pattern: Pattern: Decompose by subdomain,” microservices.io. Accessed: Apr. 18, 2025. [Online]. Available: http://microservices.io/patterns/decomposition/decompose-by-subdomain.html

[116] L. Qian, J. Li, X. He, R. Gu, J. Shao, and Y. Lu, “Microservice extraction using graph deep clustering based on dual view fusion,” *Inf. Softw. Technol.*, vol. 158, 2023, doi: 10.1016/j.infsof.2023.107171.

[117] M. Dehghani, S. Kolahdouz-Rahimi, M. Tisi, and D. Tamzalit, “Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning,” *Softw. Syst. Model.*, vol. 21, no. 3, pp. 1115–1133, Jun. 2022, doi: 10.1007/s10270-022-00977-3.

[118] A. Schaffer, “Testing of Microservices,” Spotify Engineering. Accessed: Jul. 01, 2025. [Online]. Available: https://engineering.atspotify.com/2018/1/testing-of-microservices

[119] S. Kumar and A. Brouwer, “Breaking the deadlock for low-code on the Dutch market,” Compact. Accessed: Mar. 05, 2025. [Online]. Available: https://www.compact.nl/articles/breaking-the-deadlock-for-low-code-on-the-dutch-market/

[120] E. Elshan, D. Germann, E. Dickhaut, and M. Li, “FASTER, CHEAPER, BETTER? ANALYZING HOW LOW CODE DEVELOPMENT PLATFORMS DRIVE BOTTOM-UP INNOVATION,” *ECIS 2023 Res.--Prog. Pap.*, May 2023, [Online]. Available: https://aisel.aisnet.org/ecis2023_rip/82

[121] “Navigating Legacy Systems With Low Code.” Accessed: Apr. 28, 2025. [Online]. Available: https://www.clevr.com/blog/navigating-legacy-systems-with-low-code

[122] “Low-code for legacy modernization.” Accessed: Apr. 28, 2025. [Online]. Available: https://www.outsystems.com/initiatives/legacy-modernization/

[123] “Low-Code for Legacy Modernization | Mendix.” Accessed: Apr. 28, 2025. [Online]. Available: https://www.mendix.com/strategies/legacy-modernization/

[124] “Mendix Pricing | Enterprise Application Development Platform.” Accessed: Apr. 24, 2025. [Online]. Available: https://www.mendix.com/pricing/

[125] “European banks have to upgrade their core tech. Are U.S. banks next?,” American Banker. Accessed: May 29, 2025. [Online]. Available: https://www.americanbanker.com/news/european-banks-have-to-upgrade-their-core-tech-are-u-s-banks-next

[126] F. Nick, “Is it the End of Citizen Development? | Mendix.” Accessed: May 26, 2025. [Online]. Available: https://www.mendix.com/blog/end-of-citizen-development/

[127] S. Boell and D. Cezec-Kecmanovic, “ARE SYSTEMATIC REVIEWS BETTER, LESS BIASED AND OF HIGHER QUALITY?,” *ECIS 2011 Proc.*, Oct. 2011, [Online]. Available: https://aisel.aisnet.org/ecis2011/223

[128] D. Pinho, A. Aguiar, and V. Amaral, “What about the usability in low-code platforms? A systematic literature review,” *J. Comput. Lang.*, vol. 74, p. 101185, Jan. 2023, doi: 10.1016/j.cola.2022.101185.

[129] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, “Supporting the understanding and comparison of low-code development platforms,” in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 171–178. doi: 10.1109/SEAA51224.2020.00036.

[130] Y. Levy and T. J. Ellis, “A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research,” *Informing Sci. Int. J. Emerg. Transdiscipl.*, vol. 9, pp. 181–212, 2006, doi: 10.28945/479.

[131] A. Carrera-Rivera, W. Ochoa, F. Larrinaga, and G. Lasa, “How-to conduct a systematic literature review: A quick guide for computer science research,” *MethodsX*, vol. 9, p. 101895, Jan. 2022, doi: 10.1016/j.mex.2022.101895.

[132] M. Khalifa and M. Albadawy, “Using artificial intelligence in academic writing and research: An essential productivity tool,” *Comput. Methods Programs Biomed. Update*, vol. 5, p. 100145, Jan. 2024, doi: 10.1016/j.cmpbup.2024.100145.

[133] A. Tang, K.-K. Li, K. O. Kwok, L. Cao, S. Luong, and W. Tam, “The importance of transparency: Declaring the use of generative artificial intelligence (AI) in academic writing,” *J. Nurs. Scholarsh.*, vol. 56, no. 2, pp. 314–318, 2024, doi: 10.1111/jnu.12938.

[134] “use-of-ai-in-education-at-the-university-of-twente.pdf.” Accessed: Jun. 11, 2025. [Online]. Available: https://www.utwente.nl/en/learning-teaching/Expertise/ai-in-education/use-of-ai-in-education-at-the-university-of-twente.pdf

Appendix A: Exploratory Review Search Queries

A.1 Enterprise Architecture, Monolithic Architecture, and Microservices

Search Query	**Database**
“Enterprise Architecture”	Google Scholar, Scopus, IEEE Xplore
“Enterprise Architecture” AND “Monolithic” AND “Microservice”	Google Scholar, Scopus, IEEE Xplore
“Enterprise Architecture” AND “Monolithic*” AND “Microservice*”	Google Scholar, Scopus, IEEE Xplore
“Enterprise Architecture” AND “Methodolog*”	Google Scholar, Scopus, IEEE Xplore
“Enterprise Architecture” AND “Model*”	Google Scholar, Scopus, IEEE Xplore

Table 14 – Search Queries for Enterprise Architecture

A.2 Low-Code

Search Query	**Database**
“Low-Code” OR “Low code”	Google Scholar, Scopus, IEEE Xplore
“Low-Code” AND “Digital Transformation”	Google Scholar, Scopus, IEEE Xplore

Table 15 – Search Queries for Low-Code

A.3 Migration Strategies for Monolithic Architecture to Microservices

Search Query	**Database**
“Monolith*” AND “Microservice*” AND “Transition”	Google Scholar, Scopus, IEEE Xplore
“Monolith*” AND “Microservice*” AND “Transition” AND “Strategy”	Google Scholar, Scopus, IEEE Xplore

“Monolith*” AND “Microservice*” AND “Migration”	Google Scholar, Scopus, IEEE Xplore
“Monolith*” AND “Microservice*” AND “Migration” AND “Strategy”	Google Scholar, Scopus, IEEE Xplore

Table 16 – Search Queries for Migration Strategies for Monolithic Architecture to Microservices

Appendix B: SLR Process and Protocol

B.1 SLR Process

This SLR uses the eight steps defined by Okoli (2015) [28]. Each of the eight steps is further elaborated upon.

Purpose

The first step of a SLR is to clearly define its purpose [28]. For this review, the purpose of the SLR is to create a state-of-the-art comparison between monolithic architectures and microservice architectures. It will provide benefits and challenges for both architectures. The SLR is used to assist answering SQ1 (*What are the trade-offs for monolithic architecture or microservices for financial organizations?*).

Draft Protocol & Train The Team

The second step of Okoli's eight-step model addresses the importance of creating a protocol and aligning the different team members with this protocol when performing an SLR [28]. As this SLR will only be performed by one person, the latter is not applicable.

The protocol serves as a roadmap toward answering the research questions [28]. The protocol follows the approach of Kitchenham and Charter (2007) can be found in Appendix B.2.

Apply Practical Screen & Appraise Quality

The third step of Okoli's eight-step model is the application of the practical screen. The goal of this step is to decide on the criteria for selecting and excluding papers from the SLR [28]. This step does not consider the quality of the papers (as this is part of step 7 Appraise Quality), but rather tries to increase the amount of relevant papers included and aims to reduce the number of papers to a number that can practically be managed by the researcher(s) [28]. The list of criteria is part of the protocol that can be found in Appendix B.2.

In the first round (round 1) of the practical screen, the selected papers will be scanned, and their title and abstracts will be read [28]. Papers that do not suffice the inclusion criteria, or that do suffice the exclusion criteria, will be excluded from the review. In the second round (round 2), the full text of the papers that have been included after round 1 will be read.

The guidelines of Okoli (2014) treat quality appraisal as a separate step performed after data extraction. However, in this research quality appraisal is included in round 2 of the practical screen as this reduces redundant work. This follows the approach of Kitchenham and Charter (2007) [29].

Search For Literature

The fourth step of Okoli's eight-step model is to search for the literature. Using the search query and the criteria defined in the protocol, papers should be retrieved and reviewed [28].

After completing the search and the process mentioned in Apply Practical Screen & Appraise Quality, a "backward" and "forward" search will be used to ensure that all sources have been identified [28]. The backward search is performed by scanning citations used within the selected and relevant reviewed literature. The forward search can only be performed in certain databases that allow us to find all papers that have since cited the reviewed literature. The papers found from this search will go through the same process as the initially screened papers. When repeated searches no longer give new results, the backward and forward searches are completed [28], [130].

The full process of the literature search is displayed in the PRISMA flow in Figure 21. In total, including the papers identified from backward and forward searches, 18 papers are included in the SLR.

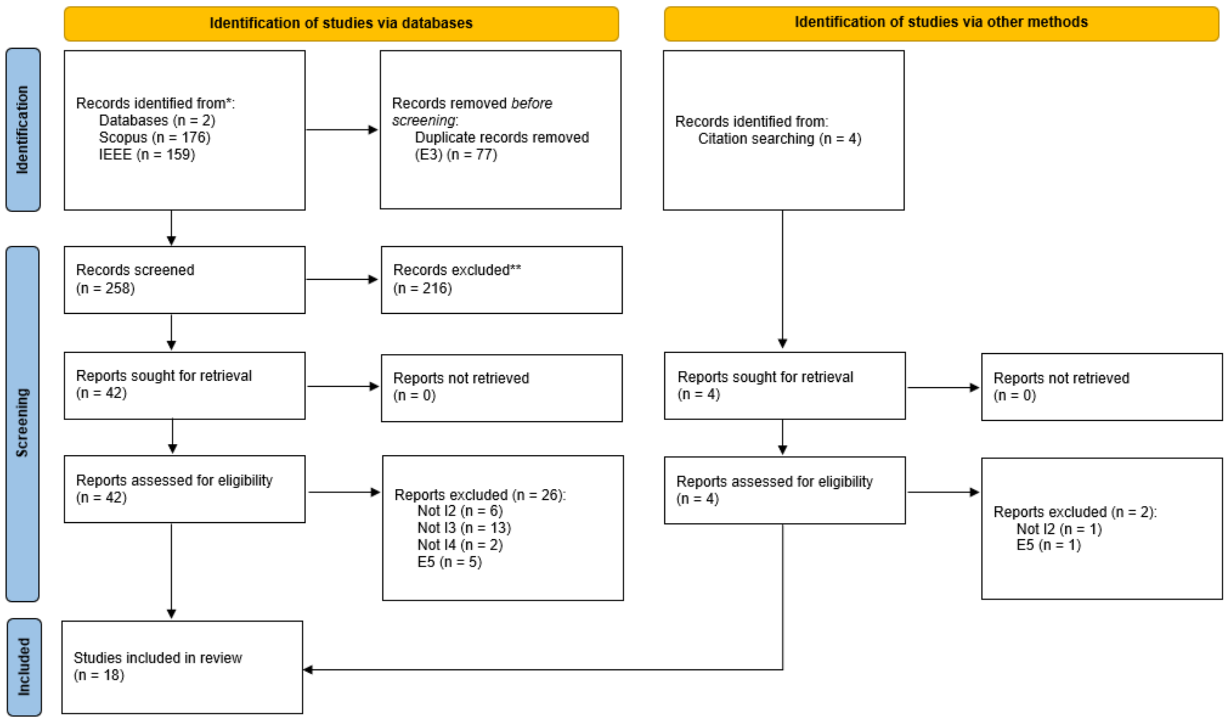

Identification of studies via databases
Identification of studies via other methods
Identification
Screening
Included
Records identified from*:
Databases (n = 2)
Scopus (n = 176)
IEEE (n = 159)
Records removed *before screening*:
Duplicate records removed (E3) (n = 77)
Records identified from:
Citation searching (n = 4)
Records screened (n = 258)
Records excluded** (n = 216)
Reports sought for retrieval (n = 42)
Reports not retrieved (n = 0)
Reports sought for retrieval (n = 4)
Reports not retrieved (n = 0)
Reports assessed for eligibility (n = 42)
Reports excluded (n = 26):
Not I2 (n = 6)
Not I3 (n = 13)
Not I4 (n = 2)
E5 (n = 5)
Reports assessed for eligibility (n = 4)
Reports excluded (n = 2):
Not I2 (n = 1)
E5 (n = 1)
Studies included in review (n = 18)

Figure 21 – PRISMA Flow Diagram for SLR

Extract Data & Synthesize Studies

The fifth and seventh steps of Okoli's eight-step model are extracting the data and synthesizing the studies [28]. The goal of data extracting is to take the raw material from all selected papers, based on the protocol defined. This data needs to be combined to make comprehensive sense, which is done by aggregation in the synthesizing step.

In this review, data extracted will be benefits and challenges for both monolithic and microservices architectures. Wolfswinkel et al. (2013) Grounded Theory approach will be used for aggregating the data [32]. This approach consists of three parts, namely 1) Open coding, 2) Axial coding, and 3) Selective coding [32].

In open coding, benefits and challenges (referred to as concepts) that have been found during the reading of the papers are extracted [32]. Through an iterative method, all papers will be read and all individual concepts will be identified. This process is iterative, as already-defined concepts can be reconsidered or renamed [32]. The result of open coding is a set of benefits and challenges for monolithic and microservices architectures. In axial coding, the concepts and categories defined should be further developed. Sub-categories can be defined that lower the amount of abstraction across the categories [32]. Additionally, similar concepts with different names are grouped here. In selective coding, the categories are further refined, integrated, and finalized [32]. This results in the final list of categorized benefits and challenges for monolithic and microservices architectures.

Write The Review

The last step of the SLR is to write the review using the findings from the above-presented methodology. This will include the findings from the data extraction and synthesis. These findings will be presented in Section 4.1 and are part of the problem investigation of the design research.

B.2 SLR Protocol

This protocol is based on Kitchenham and Charter (2007) [29]. It should include the following:

- Research questions to be answered;
- Strategy for searching literature (search query, database locations, etc.);
- Inclusion and exclusion (acceptance or eligibility) criteria for papers to be included;
- Checklists and procedures [29].

Research questions to be answered

This SLR aims to assist in answering SQ2 (*What are challenges that financial organizations face when migrating from a monolithic architecture to microservices?*). The result of the SLR is a comprehensive list of benefits and challenges of monolithic and microservices architectures.

Strategy for searching literature (database locations, search query, etc.)

The databases locations used for this research are Scopus and IEEE Xplore. Multiple researches show how that Scopus is one of the strongest databases in the computer science research discipline [27], [131]. Additionally, SLRs could benefit from including databases relevant to only the to-be-researched field, in this case, computer science [131]. For that reason, IEEE Xplore is included [131].

The search queries used in each of these database locations are displayed in Table 17.

Search Query	**Database**	**# of Results**
TITLE-ABS-KEY ((monolith) AND (microservice*) AND (architecture OR landscape) AND (driver* OR benefit* OR limitation* OR challenge* OR advantage* OR disadvantage* OR problem*) AND (organi* OR compan* OR business* OR firm*))* AND PUBYEAR > 2010 AND PUBYEAR < 2025 AND (LIMIT-TO (LANGUAGE , "English")) AND (LIMIT-TO (PUBSTAGE , "final"))	Scopus	176
(monolith) AND (microservice*) AND (architecture OR landscape) AND (driver* OR benefit* OR limitation* OR challenge* OR advantage* OR disadvantage OR disadvantages OR problem OR problems) AND (organi* OR compan* OR business* OR firm OR firms)*	IEEE Xplore	159

Table 17 – Search Queries and Results per Database for SLR

Inclusion and exclusion (acceptance or eligibility) criteria for papers to be included. The full list of inclusion and exclusion criteria used in round 1 and round 2 of this SLR are listed in Table 18. To increase the relevancy of the papers, inclusion criteria I1, I2, I3, and I4 have been added. Besides increasing relevancy, this step also aims to reduce the number of papers to a number that can be practically managed by the researcher(s) [28]. For that reason, exclusion criteria E1, E2, and E4 have been added.

The PRISMA framework in Figure 21 shows how many papers are excluded from the review due to each criterion.

Inclusion criteria	**Exclusion criteria**
I1: Search string found in Title-Abstract-Keywords I2: Needs to mention monolithic and microservice architectures I3: Needs to mention benefits or challenges of the abovementioned architecture types I4: Needs to be an organizational-level context	E1: Papers published before 2011 E2: Papers published after 2024 E3: Duplicate papers E4: Papers not written in English E5: Papers are not of sufficient quality

Table 18 – Inclusion and Exclusion Criteria for SLR

Guidelines show that both quantitative (e.g. rating with the Likert scale) and qualitative methods are suited for quality appraisal [28], [29], [131]. For reproducibility, reasons for papers excluded from the SLR due to quality appraisal should be well documented. Quality appraisal criteria E5 consists of the following qualitative exclusion criteria: E5-1 Papers without a well-established design or methodology [28], [29], E5-2 Papers without sufficient elaboration on the reliability of the measurements (includes reporting on bias(es)) [28], [29], and E5-3 Papers with an insufficient language level (grammar and/or spelling).

Checklists and procedures

The above process is executed and well documented. Please reach out to the Researcher to access this document.

Appendix C: Benefits and Challenges of Monolithic and Microservices Architectures

C.1 Benefits of Monolithic Architecture

The benefits of monolithic architecture are structured in 4 categories, namely governance, operations, single infrastructure, and system performance and evolution.

Governance

Adopting a monolithic architecture brings some governance benefits. This includes easier development for small applications, quicker time-to-market, and requiring less experienced developers.

Easier development for small applications

Monolithic architectures are the simpler choice when developing smaller applications, especially in the early stages. With the entire system's source code in a single codebase, developers can more easily understand and manage the system end-to-end. This centralized setup reduces complexity, makes coupling between components stronger [58], and makes design, development, and deployment more manageable [53], [56].

For projects with a limited scope or a small and stable number of users, monoliths are practical. They avoid the design and development of distributed systems which are inherently more complex [78], [80]. Plus, having a centralized database ensures data stays uniform across the system, which further simplifies development [84].

Quicker time-to-market

As monolithic applications are easier to develop when they are smaller and have fewer technical difficulties, they often have a quicker time-to-market compared to distributed systems [78], [81], [83]. As all components are in a single codebase, developers can more easily design and develop the software [80]. Later as their codebase grows the development speed often slows down [76].

Requires less experienced developers

The development and deployment of centralized systems are less complex than those of distributed systems. Because of this, systems built with a monolithic architecture often require less experienced and skilled developers and DevOps professionals [76].

Operations

Systems that are built with a monolithic architecture tend to be easier to monitor compared to distributed systems.

Easier monitoring

As all code written in a monolithic architecture is present within a single codebase, it allows developers to focus on a single set of metrics and logs. During runtime, if the system has a failure or performs incorrect behavior, developers do not need to piece together information from different systems in a distributed system to resolve the failures. Rather, there is only one system unit to investigate [81]. This architectural style thus simplifies monitoring as it offers a centralized view of an application [74], [85], [86].

Single Infrastructure

Monolithic systems are built within a single infrastructure. Due to this, they tend to have fewer dependencies, facilitate more code reuse, and use only one technology stack.

Fewer dependencies

Monolithic applications generally use fewer dependencies compared to distributed systems. Different services in distributed systems may rely on their own external frameworks and libraries. In a monolithic system, all dependencies are used within the same codebase and are therefore not duplicated. This simplifies the initial development and reduces the chance of complications due to dependency management [77].

More code reuse

Reusing code is easier when all code is stored within a single codebase [84]. Different components, such as classes, functions, and modules, can be used across the entire application. An example of this is data validation functions, which in a distributed system are more complicated to maintain. For that reason, developers have more control over the changes made [74].

Same technology stack

Due to only using a single codebase, systems with a monolithic architecture use the same technology stack for the entirety of the application. During troubleshooting, this can be a major benefit, as the developers working on it understand the technology that was used to develop the application [83].

System Performance and Evolution

The architectural style of a system can have an impact on its performance, and also its evolution toward the future. Monolithic architectures are usually better maintainable when the applications are smaller, and they also do not experience a network overhead, when compared to distributed systems.

Better maintainable for small applications

As small monolithic applications are relatively easy to develop, have fewer dependencies, and use the same technology stack, they are easier maintainable when their size remains small [79], [81], [83]. This makes sense, as small applications have fewer components, compared to larger applications which are harder to maintain [77]. Smaller applications which are built as a distributed system, are more difficult to maintain as they have their code and data spread across different systems.

No network overload

One big advantage of using a monolithic architecture is the absence of integrations that run over a network. Communication between different parts of the software is done internally as all code is stored in the same codebase, and therefore monolithic architectures do not experience network overhead [76]. This has a large positive impact on the performance of the applications, as distributed systems have latency when information has to flow between services [77]. Additionally, distributed systems can even experience bad performance, or unexpected behavior, when, for example, networks are slow or down [83]. For monolithic systems, this saves complexity and time during runtime [58].

C.2 Challenges of Monolithic Architecture

The monolithic architecture style also poses some challenges. These are structured into the same 4 categories as the benefits, namely governance, operations, single infrastructure, and system performance and evolution.

Governance

Adopting a monolithic architecture also brings a challenge in governance, namely the handling of the complexity due to the increase of its size.

Complexity due to codebase size

As systems with a monolithic architecture grow, the coupling between its large components becomes more complex. As components are tightly coupled, it becomes harder to govern the application. Large groups of developers need to be managed, which is more difficult when they are working on the same codebase [84], [86]. New team members might also have difficulties understanding the codebase contains all the business logic of the whole system [81]. Different developers might want to change the same piece of code, which causes problems at deployment [84].

Operations

Operating a monolithic system during runtime can become more challenging as troubleshooting and monitoring can become more difficult.

Difficult troubleshooting & monitoring

While easier monitoring was seen as one of the benefits of having a monolithic architecture, centralization can also cause troubleshooting and monitoring to become more complicated. This is especially true when monolithic systems become larger [75]. As the codebase grows, and different components of the application become more tightly coupled, it can become more complicated to find the source of a failure or performance issue [86].

Single Infrastructure

As monolithic systems are developed in a single infrastructure they face some challenges, such as accidentally affecting other functions, application-wide bugs and failures, deployment in its entirety, and language/technology lock-in.

Accidentally affect other components

When new components are added, or existing components are updated, the tightly coupled existing components might be accidentally affected, resulting in unwanted functionality changes [77], [79], [87]. This is especially the case when many components of the system are used in multiple places, and the amount of dependencies grows within the system [58]. This problem increases as more developers are responsible for the application, increasing the probability of affecting each other’s components [84].

Application-wide bugs and failures

Having a single infrastructure can cause problems when the system faces bugs and failures. Failures within one component of the application can cause the entire application to crash or malfunction [74]. This can, for example, happen when updating dependencies [56] or in case of malfunctioning of the database management system (DBMS). The latter causes the system to fail, as well as any integrations integrated through a shared database. Due to this centralization, it makes monolithic applications a potential single point of failure [77], [78], [85]. This is often considered as one of the main disadvantages of using a monolithic architecture [81], [82].

Deployment in its entirety

Deployment of a monolithic system is always done in its entirety. Updates to the system to any component, small or large, require the full application to be redeployed [58], [76], [79], [83]. As the application grows, this happens more frequently, which impacts the availability of the system [77], [81], [82] – especially for large systems [56] - and also slows down development cycles [75]. Managing version management is further negatively impacted when multiple development teams are working on the same application [86].

Language/technology lock-in

While using the same technology stack was listed as a benefit of monolithic systems, it can also be considered a challenge [74]. As they only have one codebase, monolithic applications are generally built using a single technology stack or programming language. Especially older monolithic systems are built using relatively old legacy technologies, such as COBOL, which generally, are programming languages not known well by developers [75]. Such systems might not provide the best system performance or user experience [75]. Maintaining them is difficult due to the lack of developers experienced with them, and replacing them with more well-known, modern programming languages would now require extensive resources and time, especially for larger systems [58], [76], [81]. This negatively impacts the organization’s development speed and agility [86]. This issue, known as technology lock-in, is especially relevant for monolithic applications, as they use a single infrastructure [56], [85].

System Performance and Evolution

Systems with a monolithic architecture can experience limited scalability, poor maintainability, and suboptimal performance compared to distributed systems.

Limited scalability

One of the biggest challenges for monolithic systems is scalability. As the application grows and increases in features, it is not possible to scale components of the monolith individually [75], [81], [87]. This is an issue, as a monolithic architecture does not allow to only scale the part of the application that demands more resources [76]. As a consequence of the monolithic being deployed in its entirety, the entirety of the application has to be scaled, which can become costly [74], [77], [78], [82]. Applications can be scaled by adding more resources (vertical scaling), but also by deploying multiple copies of the entire application (horizontal scaling) [74]. Vertical scaling is limited to the resource capacity of the machine that the application is deployed on and if the system expects to grow significantly, it is not a reliable option [56], [79]. Horizontal scaling is especially costly, as full copies of the entire system need to be deployed to different servers. Additionally, as horizontal scaling makes the system distributed, this option introduces problems such as database consistency, and load balancing [74].

Poor maintainability

Large monolithic applications are difficult and expensive to maintain [74], [75], [87]. While in the early stage of the application, a single codebase is easier for developers, as the system evolves it gets much more difficult to manage. As the application becomes more tightly coupled, and multiple teams are responsible for the development, maintaining a monolith becomes harder. On top of that, the possibility of accidentally affecting other components of the applications slows down the maintenance process [58], [87].

Suboptimal performance

Monolithic systems might experience suboptimal performance, as they are deployed in their entirety. Developers must always choose a one-size-fits-all configuration, which can result in suboptimal performance for individual components of the application [56], [75]. Additionally, as a consequence of technology lock-in, monolithic systems built with relatively old technologies generally have a slower performance in tasks such as performing database queries [86].

C.3 Benefits of Microservices Architecture

Adopting a distributed system architecture with microservices has some benefits over a centralized approach. These benefits are structured in governance and agility, microservice isolation, and system performance and evolution.

Governance and Agility

Adopting a microservices architecture brings some governance benefits as well as increasing agility. This is due to better alignment with CI/CD practices, microservices better fostering innovation, less complexity of individual microservices, better alignment with self-managed teams, and the possibility of technological modularity.

Alignment with CI/CD

Continuous integration and continuous delivery (CI/CD) is a set of practices and tools designed to improve the software deployment process. Microservices architectures are generally independently deployed in containers, with tools such as Docker and Kubernetes, which are often used in CI/CD. Microservices thus align better with the modern CI/CD process [87]. Individual microservice deployment accelerates the process of delivery, as these are generally smaller applications than monolithic ones [75], [80]. Additionally, as CI/CD is isolated for each of the microservices, rollbacks for individual microservices are generally less complicated [83]. Microservices also allow for gradual transitions to new versions. New versions of a microservice can be deployed, while also maintaining older versions. This allows services that depend on the microservice to gradually modify their software, which fosters CI [56]. It needs to be noted, however, that monolithic applications can also be deployed in containers and microservices do not necessarily make the process easier [88].

Foster innovation

As microservices are small components that are deployed independently, releasing a new version of a component can often be done more quickly [74]. Additionally, when microservices are deployed with gradual transitions to new versions, it is more convenient to try out new features without impacting existing integrations [56]. Both these factors foster innovation, as it allows organizations to more quickly experiment with their software [78], [89].

Less individual complexity

As microservices only contain the code and logic for one specific business capability, the complexity of the code of the individual microservice is often lower compared to a large monolith. This allows new developers to more easily investigate and start working on a specific microservice [81]. Additionally, as the data management is isolated around the specific business capability, the database will become less extensive and easier to understand [58].

Self-managed teams

One of the challenges of a monolithic architecture was the governance of different teams working on the same codebase. As microservices have fully independent codebases, it is more easy to assign individual microservices to specific teams. Self-managing teams can autonomously be responsible for maintaining individual microservices. These teams can be smaller than the teams working on a monolithic application, as they are working on a smaller codebase. This allows the developers to become more specialized in their microservice, the business capability they are responsible for, and deliver more frequent updates [56], [74], [84].

Technology modularity

Being able to use various technologies and programming languages is one of the benefits of using microservices with independent code bases. One advantage of doing this is that teams and developers can find technologies that are well-suited to the goal of the microservice [58], [78], [79], [81], [84], [89]. Additionally, as new microservices are added, developers can choose to embrace more modern technologies. If eventually a microservice needs to be replaced with a new technology, it is easier as the codebases are smaller [76]. Overall, technology modularity allows organizations to stay more flexible and avoid technology lock-in [56], [75]. In practice, however, while microservices do support it, organizations often do not encourage using a wide variety of technologies as this requires their employees to become acquainted with all of them [88].

Microservice Isolation

Microservice architectures isolate individual microservices within a network of microservices. This modular approach has some advantages including data isolation and fault isolation. It also brings operational advantages in isolated audits, monitoring, releasing, and testing.

Data isolation

As microservices are isolated from one another, it is more complicated for attackers to carry out system-wide attacks. If a database gets breached, this isolation will ensure that attackers do not gain access to the entire application's data, but rather only the data stored by the individual microservice [81].

Fault isolation

Microservice architectures isolate a specific business capability from a network of microservices. If one microservice fails, the rest of the microservices, and thus the application are not essentially directly impacted [74], [79], [84], [89]. This allows the developers to investigate the failure, and deploy a fix while the rest of the application is still operational thus reducing application downtime [77], [81].

Isolated audit, monitoring, releasing, and testing

Managing operations such as auditing, monitoring, releasing, and testing, is easier for isolated individual microservices compared to large monoliths. In line with the autonomous self-managing teams, the individual components can be audited, monitored, released, and tested independently by the team assigned to the microservice [74]. Releasing and testing are increasingly automated, as the different components are not tightly coupled to other components within the codebase of the microservice itself [75], [87]. Developers can also test and investigate the functionality of a microservice in isolation to the full network of microservices, which can make it easier to find bugs [56].

System Performance and Evolution

Microservices architectures have some benefits for system performance and evolution over monolithic architectures. This includes better reliability, better resilience, easier maintainability, increased availability, more flexibility, and more scalability.

Better reliability

Reliability is the ability of all parts of a system to work together correctly and in coordination, ensuring that the application is available. Microservice architectures enhance reliability by isolation of faults, as when one microservice fails, other components of the system will still be available [82], [86], [89]. This reduces the probability of downtime of the full application [81].

Better resilience

A system's resilience is its ability to recover quickly from an error and minimize downtime. The isolation of microservices allows developers to recover a microservice more quickly, as monitoring the individual microservice is easier [81], [87], [89]. As the codebase is independent and smaller, it is generally easier for developers to locate the point of failure and find a potential fix. Additionally, as microservices can be deployed gradually, a new version of a microservice can more easily be deployed and tested on a small group of users before rolling it out for all integrations, further enhancing its resilience [74].

Easier maintainability

Microservice architectures encourage the maintainability of applications, as microservices are smaller, independent, and have a well-defined scope based on the business capability they encompass. Compared to a monolithic architecture, microservices are less tightly coupled. These factors make individual microservices easier to understand and maintain [75], [79], [85], [87], [89]. It also makes it easier to navigate through the codebase in case an update needs to be made, or a bug needs to be addressed [74], [82].

Increased availability

As a microservice architecture improves the reliability and resilience of a network of microservices by isolating individual microservices, these systems will also experience better availability [74], [87]. Additionally, unlike for monolithic applications, updates do not cause the entire system to experience downtime, rather only the microservice that is being updated [82], [87].

More flexibility

The modular approach of microservice architectures allows teams and developers to deploy new or updated functionality independently [58], use different technologies and programming languages [77], and enable more innovation [56], [89]. They also allow organizations to more easily replace microflows with newer technologies. Overall, these points make a microservice architecture more flexible compared to a monolithic architecture, and allow organizations to make changes to the microservices more easily in the future [75], [80], [84].

More scalability

Microservices are part of a distributed system, and each microservice is deployed independently. Because of this, each microservice can be scaled independently, which is a big advantage compared to monolithic systems. A microservice that requires significantly more resources, can now be individually scaled vertically and horizontally [77], [79], [81], [87], [88]. This results in significantly better resource utilization [75], and lower costs compared to monolithic architectures, that can only be scaled in their entirety [76], [78], [89]. At the same time, microservices that require fewer resources can also be scaled down. This advantage is seen as one of the main benefits of microservice architectures [83].

C.4 Challenges of Microservices Architecture

Microservice architectures also bring challenges that need to be considered. These are structured into the complexity of distributed systems, governance, operations, and system performance.

Complexity of Distributed Systems

Microservices architectures introduce complexities because they are a distributed system. Managing distributed systems is inherently more difficult compared to centralized systems. It introduces complexities in monitoring, complexities in managing security, complexities in troubleshooting and testing, difficulties in data consistency and management, and complexities in managing integrations between microservices and external systems.

Complex monitoring

While one of the benefits of microservices architecture was its ease of monitoring individual microservices, the complexities of monitoring the entire microservices network is one of its challenges [79]. To gain insights into the behavior of the full system and find performance issues or faults, organizations need to apply distributed tracing, logging, and centralized monitoring tools [89]. Each microservice requires different resources to stay operational, making the creation of monitoring metrics and well-defined performance threshold settings more difficult [88]. Overall, the monitoring process is complex, especially when a network consists of a significant amount of microservices [77], [85].

Complex troubleshooting and testing

Compared to centralized applications, distributed systems can be more complicated for troubleshooting and testing. While testing an individual, isolated microservice is relatively easy, as it only encompasses testing of a specific business capability, performing end-to-end testing is much more complicated [56], [77], [85], [89]. In end-to-end tests, behavior between the different microservices is tested. Updates to a single microservice require the integrations to all other microservices to be tested ensuring that the full application is still intact [75]. Setting up these tests is inherently more difficult than for monolithic applications [75], [79].

If a bug or error is found within a microservices application, troubleshooting can be more difficult, as it includes identifying the source of the malfunction [81]. In a network of many microservices, this can be a complex problem, which is further complicated by the need for communication between the self-managing teams for each microservice [88].

Complex security

Security for distributed systems is a serious concern. Networks of microservices have a much larger attack surface compared to monolithic architectures, as microservices communicate over a network [77], [80]. Every microservice must apply robust security measures such as authentication and authorization to ensure the application is secure [56]. In practice, to simplify this, microservices networks often use an API gateway. Meanwhile, as monitoring a network of microservices is more complex, security incidents might be harder to detect [80].

Data consistency and management

Data consistency and management is one of the major issues in microservices compared to centralized monolithic applications [80]. Monolithic applications have only one database, which can serve as a single source of truth. Microservices have isolated databases that only include data from the specific business capability [89]. To orchestrate the flow of data over the different microservices, data needs to be duplicated, and data consistency mechanisms, such as database transactions, need to be implemented [74], [87]. Designing and implementing such mechanisms is a complex process, that will likely introduce problems such as data that is stored in different formats, inconsistencies in identifiers, and challenges in synchronizing data across the microservices [75], [77], [88]. As data communication between microservices is done over a network, this can introduce additional complications as networks might experience latencies or downtime [83].

Integration management

Managing interconnections between the microservices of the network is a complex task [88], [89]. As the size of a microservice is not strictly defined, the developers of a network are responsible for designing the scope and coordination of the microservices. While some integrations might need a synchronized data flow, others can use asynchronous integration. As the network of microservices grows, designing and implementing these integrations can become more challenging and more prone to faults [56], [77].

Governance

Distributed systems are more complex to govern, and require more experienced developers to design and implement, compared to a monolithic architecture.

Complex governance

Distributed systems more clearly set the boundaries of individual microservices and the teams that manage them. Governance of the network of microservices, however, becomes more complicated as different microservices can become dependent on one another [81]. Updates to one microservice might have significant effects on the microservice network and thus need to be coordinated [75]. For large microservice networks with multiple self-managing teams, good governance – while being complex – is critical and requires careful coordination [79].

Microservices also require different compositions of teams compared to monolithic architectures. Monolithic applications typically are built by a group of developers, QA, and operations that work in separate teams [76]. Microservices, however, require cross-functional teams as each team is a self-managing autonomous actor responsible for a specific business capability. To facilitate this, the organization needs to break horizontal borders and change its organizational team communication protocol [76].

Requires experienced developers

Developing a well-implemented distributed system requires significantly experienced developers, especially for development and DevOps [75], [76]. While the functionality of an individual microservice might be easier to implement, the architectural design of the network needs a careful approach and requires developers to have significant experience with microservice tools such as Docker, Kubernetes, and more [75].

Operations

Microservices architectures might lead to additional operational overhead during the runtime of the application.

Operational overhead

A microservices architecture increases the operational overhead of a running application. Each microservice needs to be monitored which requires significantly more resources in a large network of microservices [89]. While operation overhead is especially large when microservices are being monitored independently, tools exist that assist organizations in monitoring the full network [77], [81].

System Performance

As microservices are a distributed system, integrations introduce network overhead. Due to this, this architectural style can have some performance challenges.

Network overhead

Unlike monolithic applications that do not experience communication over a network, microservices experience network overhead [74], [76]. Especially when microservices are small, and applications are complex, many synchronous network (typically RESTful API calls) calls need to be made, which can significantly impact the performance of the system due to latency [56], [75], [77], [84].

Appendix D: Interview Template for Professionals Financial Sector

Introduction interviewer

1. Give a brief introduction of myself, the research and the context

Introduction interviewee

2. Could you briefly introduce yourself?
3. Your position within {...} is {...}. Could you explain to me what your daily activities are?

Motivation for transition

4. Research has shown that many financial organizations still have large monolithic "legacy" systems that have often been operational for more than 15 years (sometimes for even more than 35 years). Is this also the case for your organization?
    a. What kind of monolithic systems do you have?
    b. How are these types of systems currently hosted? On-premise, or in the cloud?
    c. How do these systems play a role in your IT landscape? Are these stand-alone or do they have integrations to other systems?
5. Monolithic systems are often associated with being inflexible and unscalable. What is your take on this?
    a. What are problems you have experienced with monolithic systems?
    b. With your monolithic systems, do you have trouble adapting quickly? How does the rapidly changing market and legislation play a role here?
6. Do you have business drivers to replace/update these types of systems?
    a. What are those drivers?
    b. Are steps being taken from within the organization to do this?
    c. Are you experiencing external factors that are slowing down/accelerating this process?
7. Research also shows that microservices, although a lot more complex for implementation and maintenance, could solve many of the problems of monoliths. What is your take on this?
    a. Could this also be the case for your organization? Why?
    b. What might be problems in adopting microservices?

Transition process & issues

8. Converting existing monoliths to a microservices architecture is, according to literature, a complex and often lengthy process. Have you ever experienced such a process?
    a. Could you tell me roughly what such a process looks like? How long did it take in total?
        i. How did you come to this process?
    b. What are the different steps in this process?
        i. What are the technical steps?
        ii. What are the organizational/human steps?
    c. What kind of tools did you use in this process?
    d. How did you choose these tools?
    e. How did you experience these tools?
    f. What are problems you have had/can have in this process?
    g. How did you address these issues?
9. What are success factors for the transition?

Potential of low-code for transition

10. Low-code is an emerging software development trend that on certain points has correlation with microservices. Do you know low-code?
11. Have you ever deployed or experimented with low-code yourself?
12. Do you see low-code as a tool that could be used in the transition process?
    a. If so, how?
        i. What types of applications could low-code play a role in?
        ii. What exactly is that role? (For example, building/prototyping/validating)
    b. If not, why?
    c. Do you see any risks in applying low-code in the process?

Appendix E: Interview Template for Enterprise Architecture Specialists

Introduction interviewer

1. Give a brief introduction of myself, the study and the context

Introduction interviewee

2. Could you briefly introduce yourself'
3. Your position within {...} is {...}. Could you explain to me what your daily activities are?

Motivation for transition

4. Within your work area, how often have you had to deal with old "legacy" systems?
    a. What kind of systems were these?
    b. What were the problems parties encountered with these systems?
5. Monolithic systems are often associated with being inflexible and unscalable. What is your opinion of this?
    a. What are problems you have experienced with monolithic systems?
    b. Are there industries or types of customers where these types of systems were more common?
6. What are the business drivers for replacing/updating these types of legacy systems?
7. Research also shows that microservices, although they can be a lot more complex to implement and maintain, could solve many of the problems of monoliths. What is your take on this?
    a. Have you ever implemented a microservices architecture?
        i. What was your experience with it?

Transition process & issues

8. According to the literature, converting existing monoliths to a microservices architecture is a complex and often lengthy process. Have you ever experienced such a process?
    a. Could you tell me roughly what such a process looks like? How long did it take in total?
        i. How did you come to this process?
    b. What are the different steps in this process?
        i. What are the technical steps?
        ii. What are the organizational/human steps?

    c. What kind of tools did you use in this process?
    d. How did you choose these tools?
    e. How did you experience these tools?
    f. What are problems you have had/can have in this process?
    g. How did you address these issues?
9. What are success factors for the transition?

Potential of low-code for transition

10. Low-code is an emerging software development trend that has consistency with microservices in places. Do you know low-code?
11. Have you ever deployed or experimented with low-code yourself?
12. Do you see low-code as a tool that could be used in the transition process?
    a. If so, how?
        i. What types of applications could low-code play a role in?
        ii. What exactly is that role? (For example, application build/prototype/validate)
    b. If no, why?
    c. Do you see any risks in applying low-code in the process?

Appendix F: Interview Template for Low-Code Professionals

Introduction interviewer

1. Give a brief introduction of myself, the research and the context

Introduction interviewee

2. Could you briefly introduce yourself?
3. Your position within {...} is {...}. Could you explain to me what your daily activities are?
4. What is your experience with low-code platforms and application development?

Motivation for transition

5. Monolithic applications often present challenges such as scalability, maintenance and flexibility. Research has shown that microservices can potentially solve many of these problems. How do you see this?
    a. Have you ever built a monolith yourself?
6. Microservices seem very much related to low-code. After all, they are both in the cloud and low-code development platforms often provide built-in REST connections to integrate microservices. Do you have experience building a microservices application?
    a. How has using low-code helped you do this?
7. Do you have experience using low-code to convert an existing monolithic system to microservices?
    a. Would you perhaps provide an example of such a project?
8. In what ways has low-code added value in this process?
9. In what ways do you think low-code added less value to this process?

Transition process

10. The process of converting from a monolith to a microservice architecture can take a long time. There are many technical but also often organizational factors involved in this process. Where in the process have you applied low-code? (Prototype, MVP, Final Product)
    a. Did low-code also impact other phases of the transition?
    b. Do you see any potential of low-code to be used in other phases than the one you applied it to?
11. Do you see any risks in using low-code to transition?
12. Are there any situations where you would not use low-code?

Appendix G: Interview Template for Validation Expert Opinions

Introduction interviewer

1. Give a brief introduction of myself, the research and the context

Introduction interviewee

2. Could you briefly introduce yourself?
3. Your position within {...} is {...}. Could you explain to me what your daily activities are?

Effect

4. To what extent could the roadmap help your/financial organization(s) in the migration process?
    a. Would this roadmap fit within your organization's current approach to application modernization? Would it require changes to be made?
5. What outcomes do you foresee if your organization would use this roadmap?

Sensitivity

6. Could this roadmap be used for organizations that are not in the financial sector?
    a. If not, what adjustments should be made?
7. Are there any organizational environments in which (size, country, etc.) the roadmap would have a different impact?
8. What type of legacy monolithic systems is this roadmap best suited for?

Requirement satisfaction

9. Do you perceive the roadmap as easy to understand?
10. Do these phases and steps accurately highlight how financial organizations should perform the migration process?
11. Does the roadmap highlight the challenges that organizations can face during the migration process?
12. Does the roadmap provide a clear explanation on how low-code can be used within the migration process?
13. Are there any phases or steps that are not accurate or your organization would do differently?
14. Does this roadmap align with your expectations before I showed it to you?

Appendix H: Use of AI

AI has significant potential in academic research and writing [132]. Particularly generative AI can assist researchers in various stages throughout the research process. Transparently declaring use of generative AI usage is essential to uphold academic integrity and credibility [133]. As this research prioritizes transparency, it is important to clarify the role that generative AI has played.

Tang et al. (2024) identify six core domains in which AI can contribute to academic research and writing [132]. In this reserach, generative AI has been used exclusively in Domain 2: ‘improving content and structuring.’ The use of generative AI aim to improve the readability of the research.

This vision is shared by the University of Twente and requires researchers to state the following [134]:

During the preparation of this work the researcher used Grammarly and ChatGPT in order to improve content, structuring, and readability. After using this tool/service, the researcher reviewed and edited the content as needed and takes full responsibility for the content of the work.