CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The Data Management of a Microservices Migration of Embedded Software

Strategies and Challenges in Migrating Embedded Software to Microservices Architecture

Master's Thesis in Computer Science and Engineering

Oscar Hedenäs Bennet
Alexander Jyborn

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

The Data Management of a Microservices Migration of Embedded Software

Strategies and Challenges in Migrating Embedded Software to Microservices Architecture

Oscar Hedenäs Bennet
Alexander Jyborn

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

The Data Management of a Microservices Migration of Embedded Software
Strategies and Challenges in Migrating Embedded Software to Microservices Architecture
Oscar Hedenäs Bennet
Alexander Jyborn

Supervisor: Miroslaw Staron & Hamdy Michael Ayas, Computer Science and Engineering
Advisor: Mikael Krekola, David Windehem & Björn Östlund, Ericsson
Examiner: Hans-Martin Heyn, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

The Data Management of a Microservices Migration of Embedded Software
Strategies and Challenges in Migrating Embedded Software to Microservices Architecture
Oscar Hedenäs Bennet
Alexander Jyborn
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Technology advances at a rapid pace and migrating from traditional embedded systems to a microservices architecture (MSA) has become essential for organizations aiming to enhance scalability, maintainability, and operational efficiency. This thesis investigates the data management challenges associated with such a migration and explores effective strategies to address these challenges. The effectiveness of the strategies are assessed by the tradeoff between latency, time to recover and the implementation complexity. Through a mixed-methods approach, including a systematic mapping study, semi-structured interviews, and proof-of-concept implementations, we identify key challenges such as handling statefulness, ensuring data consistency, managing data synchronization, data splitting and data denormalization.

Our findings highlight the importance of prototyping and iterative testing, revealing that strategies like Database per Service, Private Tables per Service, Schema per Service, Stateful Messaging Pattern, and State Repository Pattern can effectively address the data management challenges of an MSA migration, though their suitability depends on the system's unique constraints. We also emphasize the need for robust data synchronization mechanisms and found the Saga and CQRS patterns as potential solutions.

The research highlights the importance of a flexible, iterative approach to migration, allowing for the gradual transition of the system and alignment with customer demands. By implementing intermediate designs, organizations can manage the complexities of migration more effectively, ensuring a smoother process and achieving a scalable, maintainable system architecture.

Overall, this thesis provides valuable insights and practical recommendations for organizations undertaking the migration of embedded systems to microservices, contributing to the broader field of software engineering and system architecture.

Keywords: Microservices Architecture, Embedded Systems, Data Management, Cloud Migration, Stateless, Strategies, Challenges, Prototyping, Scalability

Acknowledgements

We want to thank our supervisors Miroslaw Staron and Hamdy Michael Ayas for their active supervision in this thesis and for their guidance and support throughout the whole project. We also want to thank our advisors at Ericsson, Mikael Krekola, David Windehem and Björn Östlund for supporting us and providing us with all the resources and help needed for a successful master thesis. Lastly, we want to thank the employees at Ericsson that took the time to participate in the interviews.

Oscar Hedenäs Bennet & Alexander Jyborn, Gothenburg, 2024-06-03

Contents

List of Figures

List of Tables

1

Introduction

In an era of rapid technological advancements, the migration from traditional embedded systems to a cloud-based microservices architecture (MSA) stands out as a significant evolution in software engineering practices. Organizations across various sectors are increasingly recognizing the benefits of MSA, which include enhanced scalability, maintainability, and operational efficiency of software systems [1].

Embedded systems, often characterized by their real-time performance, reliability, and efficiency, have been the backbone of many applications ranging from industrial automation to consumer electronics [2]. These systems are designed to perform specific tasks and are usually optimized for performance and resource efficiency. However, the monolithic nature of these systems presents challenges in terms of flexibility and scalability. As the demand for more complex functionalities, higher performance, and faster time-to-market increases, the limitations of monolithic architectures become more pronounced.

Monolithic systems, where all components are interconnected and interdependent, can become difficult to manage as they grow [3]. Any modification in one part of the system might necessitate extensive changes throughout the entire codebase, leading to longer development cycles. This architecture also limits the ability to scale parts of the system independently based on demand, often resulting in inefficient resource utilization and potential performance bottlenecks.

The transition to a cloud-based MSA offers a promising solution to these challenges. By decomposing a monolithic system into smaller, independently deployable services, organizations can achieve a higher degree of modularity [3]. Each microservice can be developed, tested, deployed, and scaled independently, which enhances the agility and responsiveness of the development process. This modularity not only simplifies maintenance but also allows for the targeted scaling of individual services, leading to more efficient use of resources.

However, the migration from a monolithic architecture to MSA is not without its challenges. This process involves significant architectural changes, particularly in how data is managed across the system [4]. In a monolithic system, data management is typically centralized, making it straightforward to ensure data consistency

and integrity. In contrast, a microservices architecture inherently involves a distributed system where data is managed by multiple services. This distribution introduces complexities related to data consistency, synchronization, and integrity, which must be carefully addressed to realize the full benefits of MSA [4]. Additionally, embedded systems limitations on memory and computing ability restrict the range of feasible cloud solutions that can be adopted [2].

To navigate these challenges, organizations must adopt effective strategies for service decomposition, data management, and inter-service communication. This thesis aims to explore these challenges and provide actionable insights and strategies for migrating embedded systems to a microservices architecture. By addressing these challenges, the research seeks to facilitate a smoother transition to MSA.

1.1 Research Context

Organizations with legacy embedded systems face the need to modernize their software infrastructure to maintain a competitive edge and meet evolving demands. These legacy systems, while robust, often lack the flexibility and scalability provided by modern microservices architectures [2]. Additionally, there is a need to develop solutions compatible with both traditional embedded systems and cloud environments.

Ericsson, a leader in telecommunications technology, exemplifies these challenges. The company's legacy embedded system, while reliable, does not offer the same level of flexibility and scalability as modern microservices architectures. This study seeks to provide actionable guidance for Ericsson and other industry stakeholders, assisting in the strategic decision-making process regarding migration approaches for common data management challenges. Additionally, it aims to enrich the existing body of knowledge concerning the migration to microservices, offering strategies and insights.

By addressing these challenges, the research aims to facilitate a smooth migration to microservices architecture, enhancing scalability, maintainability, and operational efficiency. By examining Ericsson's embedded system as a case study, it provides a concrete example of these challenges and solutions. Additionally, through semi-structured interviews and a systematic mapping study, the research offers valuable insights that can be applied across various industries and use cases.

1.2 Research Questions

The primary aim of this thesis is to systematically explore the challenges and strategies associated with migrating large-scale embedded systems to microservices, focusing on data management issues that arise during this transition. To guide this exploration, the research is structured around three research questions:

- **RQ1:** *What are the technical challenges of data management when migrating a large embedded system to a microservices architecture?*
  This question seeks to identify and characterize the data-related challenges that typically emerge during the migration from embedded systems to microservices.

- **RQ2:** *What are the most effective strategies for addressing the identified data management challenges?*
  This question aims to evaluate various strategies and technical solutions that can mitigate the identified challenges. The effectiveness of the strategies are evaluated by analysing the balance between three factors: latency, time to recover, and implementation complexity, where implementation complexity is assessed both by the quantity of implementation steps required, and the effort demanded by each step.

- **RQ3:** *What practical insights can be derived from the implementation of these strategies?*
  The final question focuses on applying the identified strategies in a practical scenario at Ericsson, aiming to gain actionable insights and document the lessons learned during the implementation. This includes evaluating the effectiveness of different migration strategies and identifying potential pitfalls and best practices.

2

Background

This chapter outlines the theoretical foundations and related research relevant to this thesis. It begins by exploring the principles of Microservice Architecture (MSA) and the migration process, with a particular focus on transitioning from embedded systems. Additionally, this chapter will delve into concepts related to data management.

2.1 Microservice Architecture

The concept of MSA entails the development of software systems as a collection of small, autonomous services, each operating within its own process and interacting through lightweight mechanisms [5]. MSA is an architecture style derived from the paradigm of distributed systems and is specifically a style of Service Oriented Architecture (SOA) [6]. The key principle of microservices is that they should be independently deployable [5]. To achieve this, microservices need to be loosely coupled and have as few dependencies to other artefacts as possible. Loosely coupled services require well-defined interfaces that create clear boundaries towards other services. Defining these boundaries and what should or should not be its own microservice is challenging and referred to as the service granularity problem [7]. These independently deployed services then work together to form the complete system, compared to a monolithic architecture where the system is built like a single unit, often being tightly integrated and deployed as one. When achieving MSA with microservices that are independent, each service can be independently scaled to handle varying workloads. The system will also be resilient due to fault isolation. Fault isolation is the aspect that a failure in a service does not affect other parts of the system, creating a resilient system. To achieve a highly available system, it is also important for microservices to be able to recover from failure effectively and automatically. To fully take advantage of all the benefits available in a cloud environment, microservices must be stateless [8]. The transition to MSA has due to the mentioned and more benefits been increasingly recognized as advantageous for organizations as it enhances scalability, deployment independence, and system resilience [5],[9].

2.2 Migrating to Microservice Architecture

The shift to MSA is driven by the desire to improve system agility, scalability, and maintainability, a trend that is gaining momentum across various organizations [10]. However, transitioning from a monolithic architecture to MSA is fraught with complexity and introduces a plethora of challenges, primarily due to the fundamental changes it necessitates [5]. The migration process to MSA has been thoroughly investigated, with classifications into systemic and technical changes outlined in recent research [1]. Systemic migration addresses the migration's structural, organizational, and business-oriented dimensions, whereas technical migration deals with the direct technical alterations required, such as system analysis, data splitting, and DevOps integration. Despite a framework for migration being proposed, there is no one-size-fits-all solution, due to each system being unique and requires a tailored migration strategy [11].

The journey towards MSA is particularly intricate when applied to embedded systems. Embedded systems are distinct in their integration with physical processes, imposing unique requirements such as real-time performance, reliability, and efficiency [12]. These systems are designed to perform dedicated tasks on specific hardware, making optimal use of limited resources like processing power, energy, and memory. Given these constraints embedded systems typically operate in a stateful manner to save resources. This statefulness, where system output is influenced by previously generated states, contrasts with the stateless ideal of microservices, which benefits from cloud computing's elasticity, load balancing, and redundancy for high availability and reliability [8]. Transitioning from a stateful to a stateless architecture necessitates significant modifications, highlighting the complex nature of migrating legacy systems to MSA. Further studies investigating the migration journey focusing on embedded system should be conducted, which is lacking in current research.

2.3 Data Management

Decentralization is favourable in all aspects of microservices due to the increased scalability, enhanced resilience and greater flexibility, and data management is no exception [13]. Data management involves challenges such as data consistency, synchronization, and partitioning of data in a monolith. While current literature presents a high level representation of data management challenges and solutions, it often lacks information on technical implementations, missing the complexities and implementation activities of the solutions.

2.3.1 CAP and PACELC Theorem

The CAP theorem also known as Brewer's Theorem, where CAP stands for consistency, availability and partition tolerance [14], states that a distributed system can only deliver two of those three characteristics [15]. Understanding this theorem is essential for microservice oriented data management as it is by definition a distributed system. Data consistency is a data quality aspect that refers to if the same

data kept at different places match or not. Availability in the CAP theorem refers to the ability that every request receives a response, but it is not guaranteed to be the most recent data, depending on the consistency. In the context of the CAP theorem, partition refers to a communications break in the system, a lost or delayed connection between nodes can be such a break. Partition tolerance means that the system has to work even though these breaks occur. Databases can therefore be classified as CP, AP or CA databases, depending on which characteristics they supply. The CAP theorem has since been extended because in reality CAP's limitations on the system only apply when facing certain failures and does not limit the system during normal operation, and therefore it misses a major part of the system's operation [16]. The extended theorem is called PACELC and adds an additional trade off between latency and consistency that is separated from the CAP trade offs because it is applied when the system functions normally. Adding the additional trade off into design considerations for the system is bringing the discussion closer to the state of the art.

2.3.2 Data Consistency

Any MSA system that have distributed data through the system, inherently introduces additional complexity related to data consistency [17]. Data consistency issues are therefore a challenge many systems specifically face when migrating from single threaded system that do not have any data consistency issues to microservices that have multiple instances interacting in parallel [8].

Consistency can also be described as eventual consistency, which is a consistency model that is used in distributed systems that informally guarantees that if no new updates are executed to a data item, the data will be consistent. There are multiple consistency models that can be used, the most common ones being either ACID or BASE based. Atomicity, consistency, isolation and durability are the aspects that make up the acronym ACID [18]. Where atomicity refers to the requirement that every database transaction is treated like a single action that either succeeds or fails completely. Consistency refers to the ability to ensure that the database can only go from one consistent state to another, which means that any reads from the database always returns the last updated data or an error. Isolation refers to ensuring that concurrent transactions leaves the database in the same state it would be if the transactions were executed in sequence. Durability is a guarantee that when a transaction has been committed it will not be undone even by system failure. This means the data is stored in some type of non-volatile memory.

The consistency model called BASE is the opposite of ACID, and following the BASE model suits microservices well because it achieves higher levels of scalability than ACID in exchange for lower levels of consistency [4]. Basically available, soft state and eventual consistency are the aspects that make up the acronym BASE [18]. Where Basically available is the accessibility aspect of BASE and means that the database is available at all times even with many concurrent transactions, even though there might be a slight delay. Soft state refers to the fact that data can have

temporary states over time and that the state is not final before all transactions are completed. Eventual consistency means that over time and when all concurrent transactions have completed, the database can be considered consistent. This can be done by propagating changes made to the database through the system to make sure the latest data is present everywhere.

2.3.3 Data Synchronization

Data synchronization is the processes and logic that needs to be in place, to establish consistency between data sources. Choreography and orchestration are the two main patterns that are used in distributed systems to handle communication across data sources [4]. The choreography pattern is built on the idea that the components of the system interact with each other through well-defined interfaces without needing a central coordinator [19]. A system using this pattern is by definition event driven, as the services sends an event adhering to the interface and does not know when that event is processed. Following this pattern allows for loosely coupled services that can work completely independently, allowing the system to be more scalable. The asynchronous communication of event driven designs can also be more scalable but can affect consistency.

Orchestration, on the other hand, is a pattern that has a central coordinator that is responsible for managing the communication between services. Systems that use this pattern are not event driven, but instead command driven. This means that the sender wants something to happen, and the receiver does not know who sent the message. Orchestration is often easier to implement and maintain because with a central controller it is easier to monitor the services interactions and get a holistic view of the system. The drawback of using orchestration is that it leads to tighter coupling and a single point of failure in the system.

3

Methods

This thesis adopts a mixed-methods investigation to explore the migration of embedded systems to MSA. The methodology includes a systematic mapping study, and a case study drawing upon the foundational work of Robert K. Yin [20]. According to Yin, a case study is defined as an empirical inquiry that investigates a contemporary phenomenon within its real-life context, particularly when the boundaries between the phenomenon and context are not clear. This research method is particularly well-suited for this study due to its ability to incorporate various data collection and analysis methods, offering a nuanced understanding of the subject. This approach leverages multiple sources of evidence to ensure a comprehensive exploration of the migration strategy, its implementation, and outcomes, thereby enhancing the study's validity and reliability through methodological triangulation [21].

The methodologies used in this study can be seen in Figure 3.1. This research implements a systematic mapping study [22], as a foundational step of identifying relevant literature and theoretical frameworks, ensuring the research is grounded in current academia on MSA and migration strategies. By mapping the literature to certain topics, an overview of current research helps to identify research gaps and trends in the research field. Semi-structured interviews [23], were also conducted as a data collection method, offering in-depth insights into the experiences, challenges, and strategies from those involved with the migration process. The first two interviews, referred to as the initial interviews, involved a team that had already made a similar migration to cloud. These interviews aimed to identify similarities between their system and the system currently being studied.

The findings from the systematic mapping study and the interviews were used to decide on two strategies to implement in a prototype of the system as a proof of concept. This method allows for the testing and evaluation of migration strategies in a controlled environment, providing practical insights that complement the theoretical and qualitative findings. Subsequently, two evaluation interviews were conducted with the team of the system under study after each proof of concept implementation. These interviews aimed to evaluate the implementation and determine how well the solutions could be integrated into the actual system, if they had any concerns with the solution, and to estimate the complexity of implementing this solution.

Employing a case study methodology, as detailed by Yin and supported by the principles of triangulation highlighted by Seaman, facilitates a comprehensive examination of the migration process at Ericsson [20], [21]. Combining a systematic mapping study with the case study, ensures that the findings are not only relevant and applicable to Ericsson but also contribute meaningfully to the broader field of software engineering and system architecture.

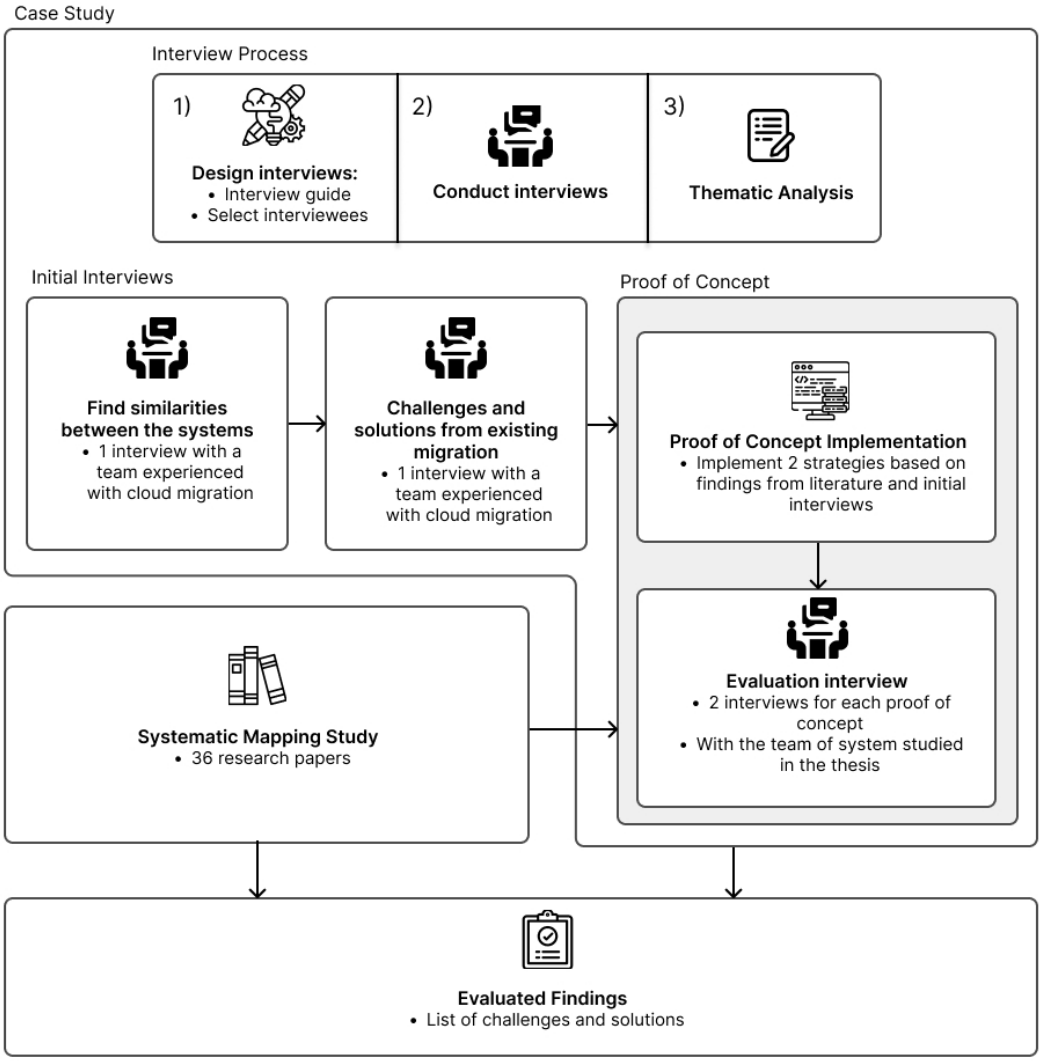

Figure 3.1: An overview of the research methodology.

3.1 Systematic Mapping Study

To gather knowledge and data about the areas and topics relevant for this study, a systematic mapping study was conducted. The systematic mapping study in this thesis is inspired by the methodology described in [22]. To ensure relevant literature was found, a search based method was applied to gather a set of academic papers that were used as primary papers. The literature search was done by using Google Scholar with the search patterns seen in Figure 3.1. The first pattern was used to find general information about MSA, the second one was used to find papers related

to the migration towards microservices, and the third one was to find microservice architecture related to embedded systems. These search strings allowed for different formats of the desired keywords. The use of Google Scholar introduces reliability issues, as its search algorithm may give different results at different times, even when the same search terms are used. To address this, all the papers analysed in the final set are listed in Table A.1 in Appendix.

Search Strings

(microservice* OR micro-service*)
(microservice* OR micro-service*) AND (migration* OR "migrating")
(microservice* OR micro-service*) AND "embedded"

Table 3.1: Table of search strings used when finding starting literature.

The chosen primary papers were dynamically screened by us through manually reading the papers, with a focus on the abstract and introduction sections. Inclusion and exclusion criteria which can be seen below were applied to evaluate each paper's relevance to this thesis area of research and its potential to address the research questions RQ1 and RQ2. This dynamic screening process allowed us to select papers that would be included in the primary set until a sufficient number of papers were found to comprehensively cover the research questions.

Inclusion Criteria

- Papers that present microservices architecture and the migration towards it.
- Papers that present challenges or solutions related to microservices.
- Case studies adopting a microservices architecture.

Exclusion Criteria

- Papers without reference to microservices.
- Papers that have not been peer reviewed.

After this search and screening for primary papers, a snowballing method to find further information by examining interesting papers cited in the primary papers was employed. Similarly to the screening process, the decision to include a paper in the final set were based on how the papers help to answer RQ1 and RQ2.

The final set of literature was analysed and categorized into topics based on similarities between them and the aspects they discuss. Categorizing the papers enabled an organized structure of existing research and served as an indicator of whether further snowballing was needed for underrepresented topics.

In addition to the systematic mapping study, further information of microservice concepts, technologies, and design patterns were found in grey literature. Adding grey literature was justified by the unique value grey literature, such as technical reports, industry white papers and blog posts, brings in providing up-to-date information and practical examples not yet available in academic journals [24]. Our addition of

grey literature was mainly to gain a deeper understanding of topics that were only briefly described in the academic literature, making sure we had a comprehensive understanding of current practices and up-to-date information.

3.2 Semi-Structured Interviews

This research utilized semi-structured interviews, which combines structured, predefined questions to bring out foreseen information and open-ended question, allowing the interviewee to add unexpected insights [23]. In this thesis, two different types of interviews were conducted: initial interviews and evaluation interviews. The goal of the initial interviews was to find foreseen challenges and solutions for the migration, while the evaluation interviews focused on evaluating the result of the proof of concept implementations.

3.2.1 Participants

The initial interviews were conducted with another team at Ericsson, referred to as T1, which had previously completed a cloud migration for a system sharing similarities with the one under study in this thesis. It was deemed sufficient to interview a single member from T1 to learn about the challenges they faced with the migration and the strategies they employed to overcome these obstacles, assuming that responses would be consistent across the team due to the completed migration. The interviewee was chosen with a combination between purposeful sampling and convenience sampling. The criteria of the interviewee were specific, he or she needed to be a software architect who had actively participated in a cloud migration, from start to finish. Fulfilling these criteria, the interviewee was then found through convenience sampling by an advisor at Ericsson, who connected us with a system architect.

The evaluation interviews were conducted with the team responsible for the system under investigation, called T2. The study targeted two system architects associated with the system to be migrated. The interviewee's in this phase were also found using a combination of purposeful sampling and convenience sampling. The criteria for this sampling was that the interviewees must be software architects working on the system investigated in this study. They were then identified by the supervisors at Ericsson.

Background information of all interview participants can be read in Table 3.2 and 3.3.

3.2.2 Protocol

Each interview session was designed to last approximately 45 minutes, starting with three introductory queries to understand the interviewee's background, role, and experience. This was followed by 4-7 main questions directed towards achieving the objectives of the interview. The session concluded with 1-2 closing questions,

Initial Interviews Participant

ID	Team	Role	Years in current role	Cloud Experience
I1	T1	System Architect	6	Completed a cloud migration.

Table 3.2: Background information of the participants of the initial interview.

Evaluation Interviews Participants

ID	Team	Role	Years in current role	Cloud Experience
I2	T2	System Architect	5	Participated in an attempt of creating a microservices architecture.
I3	T2	System Architect	2	Limited experience, currently learning.

Table 3.3: Background information of the participants of the evaluation interviews.

providing an opportunity for interviewees to share important insights not previously asked. Consent for recording was sought from all interviewees, with an assurance that the transcript would not be disclosed in the thesis. An overview of the interview guides can be seen in Table 3.4.

The first interview involved identifying similarities between the two systems. An interview with a member from T1 was arranged to gain a comprehensive understanding of their system's state before, during, and after the migration, as well as the migration process and the reasons behind the migration. The interviewee was requested to provide system artefacts to assist in analysing the system and was informed that the artefacts would not be published in the report. The insights from this interview were later shared with the architects of the system to be migrated, to pinpoint similarities and differences, aiding in the creation of an interview guide tailored to exploring specific challenges and solutions. The result was also compared with the findings of the systematic mapping study to see how their migration journey aligns with the literature.

With the new-found insights of the similarities between the two systems, it was possible to define questions for the second interview with T1 that would also be relevant for T2. The goal of this interview was to ask more specific questions about the challenges they faced during the migration, as well as their solutions. With the findings from this interview, combined with the systematic mapping study, it was possible to decide on a strategy to be tested as a proof-of-concept in the prototype

	Questions	Purpose
Initial Interview 1	9	Understand T1's system before, during and after migration. Find out how they migrated to cloud: what strategies and techniques that they used. Also, identify the reason behind the migration, the requirements, and drivers.
Initial Interview 2	6	Identify challenges and solutions that are relevant for the system under study. The result from this will help with deciding on a strategy to test in the proof-of-concept implementation.
Evaluation Interview 1	9	Evaluate how well the first proof-of-concept solution would fit in the real system, find foreseen challenges implementing this in the real system, find solutions for the challenges encountered, estimate the implementation complexity and identify additional implementation steps. Also, gather data for the next proof-of-concept implementation.
Evaluation Interview 2	5	Evaluation of the second proof-of-concept implementation. This includes how well the solution fits in the real system, foreseen challenges, solutions for the encountered challenges, estimation of the implementation complexity and identification of additional implementation activities.

Table 3.4: Overview of the interview guides.

version of the system, called MigLab. This is further explained in Section 3.3.

After each proof-of-concept implementation, two evaluation interviews were conducted with the software architects of the system under study. The goal of these interviews were to assess the compatibility of the solution with the actual system, gather insights on potential solutions for the challenges encountered, estimate the complexity of implementation, and identify any additional steps required for implementing this in the real system. The first round of evaluation interviews was also used to decide on the next strategy to test for the second proof-of-concept implementation.

3.2.3 Analysis

Thematic analysis was applied to analyse the data from the interviews. "Thematic analysis is a method for identifying, analysing, and reporting patterns (themes) within data" [25]. There are six steps of this method: familiarization with data, coding, generating themes, reviewing themes, defining and naming themes, and producing the report. The process of how this was done in the thesis is described

below.

Data familiarization. The analysis commenced with a thorough review of the interview transcripts
Coding. Text that were relevant for the research was highlighted and coded. An important aspect of this is that the codes should not constraint the boundaries, so new codes were added when needed [25].
Generating themes. All codes that was previously created were mapped into themes based on similarity.
Reviewing themes. The themes were reviewed and improved until changes no longer affected the result.
Defining and naming themes. The goal of this stage is to name and define each theme so that it is easier to understand the data. This involved formulating exactly the meaning behind the theme.
Producing the report. The final step involved weaving the thematic analysis into the narrative of the thesis, relating the findings back to the research questions and existing literature on microservices migration challenges and solutions.

3.3 Proof-of-Concept Implementations

Two strategies identified in the systematic mapping study and semi-structured interviews were implemented in the case study, to assess their effectiveness and to understand how they could be integrated into the specific context of this research. Given the expansive nature of the system under study, which exceeded the scope manageable within this master thesis, a prototype version was developed, which was named "MigLab", short for Migration Laboratory. This methodology is similar to an experimental simulation where the actors are natural, but the setting has specifically been created to fit the purpose of the study, which allows for a high level of obtrusiveness while compromising the realism of the context [26]. The difference from an experimental simulation is that no hypothesis was established before the implementation. This method will help answer Research Question 3 (RQ3), where the goal is to gain insights into a practical migration based on the strategies and patterns found in previous steps of the study.

The selection of strategies to implement in the proof of concepts was informed by insights from both the systematic mapping study and the interviews. This approach ensured the testing of strategies most relevant to our research objectives. Specifically, strategies identified in both the systematic mapping study and the interviews were considered highly relevant for further investigation. To elaborate, the systematic mapping study provided an overview of existing challenges and strategies, while the semi-structured interviews offered contextual insights and practical considerations from experts involved in similar migrations. By combining these two sources of information, we identified the most promising strategies that could address the identified challenges and fit within the constraints of the MigLab prototype. This combined approach not only grounded the selection in theoretical evidence but also ensured the relevance and applicability of the strategies in a practical, real-world

setting.

3.3.1 Prototype

The prototype system, MigLab, captures the main relevant characteristics of the actual system in a manageable way and was developed in collaboration with domain experts, who are working on the system studied in the thesis.

The prototype system is composed of four primary components: devices, resource consumers, broker, and a PUB/SUB event bus, as depicted in Figure 3.2. The Broker, highlighted in blue within the figure, acts as the core of the system's functionality, responsible for the allocation of appropriate resources between resource consumers and available devices, and subsequently recording these transactions in a local state. The remaining components provide essential context for the system's operation.

The selection of technologies is driven by the attributes of the actual system. For the PUB/SUB event bus, NATS is employed. For inter-process communication between services, Ericsson's developed XCM, is utilized. Additionally, Google Protocol Buffer (GPB) is used in conjunction with XCM for the serialization of structured data.

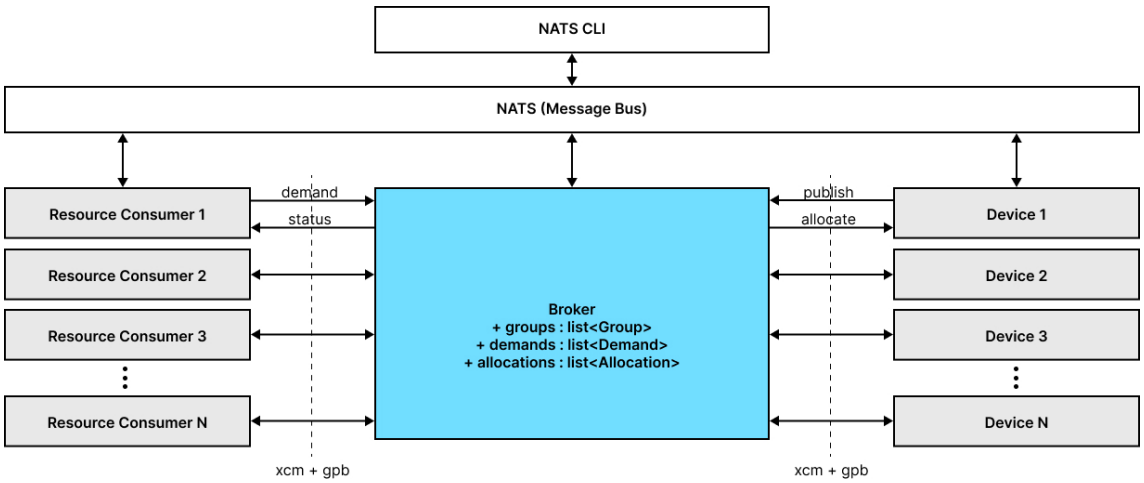

Figure 3.2: System architecture of the prototype system, MigLab.

```
struct Demand {
    int32_t local_demand_id;
    std::string allocator_name;
    std::int32_t capacity_demand;
    std::string capacity_group;
    Allocation_WPtr allocation;
    bool is_allocated() const;
};
```

Figure 3.3: Demand object used in prototype.

As seen in Figure 3.2 the broker have three internal states: groups, demands and allocations. The groups state is from configuration and is sent to the broker by

```
struct Allocation {
    enum class State {
        OngoingAllocation,  // AllocateCapacityRReq has been sent
        Allocated,          // AllocateCapacityRCfm was received, local_allocation_id is valid
        Rejected,           // AllocateCapacityRRej was received (should not happen. we should retry soon)
        OngoingDeallocation // DeallocateCapacityRReq has been sent
    };
    Allocation(std::int32_t capacity, std::string_view allocator_name, Demand_WPtr const & demand);
    State        state;
    std::int32_t capacity;
    std::string  allocator_name;
    Demand_WPtr  demand;
    std::int32_t local_allocation_id;
};
```

Figure 3.4: Allocation object used in prototype.

NATS. A group consists of a group name and device names that belong to that group. The structure of a demand can be seen in Figure 3.3 and is based on the information the broker receives from a resource consumer. The allocation object is created by the broker by running an allocation algorithm on events that checks if there are any non-allocated demands left. The allocation object can be seen in Figure 3.4.

3.3.2 Evaluation

The effectiveness of the strategies is assessed using the following metrics: Latency, which measures the response time; Time to recover, measuring the time it takes for the system to recover from a failure; and Implementation Complexity, determining the implementation difficulty of these features. The evaluation draws upon experimental results, comparisons with the actual system, and insights from relevant literature.

When measuring latency, several procedures within the system were evaluated. The baseline system of the prototype was tested both locally and within a Kubernetes cluster running locally on the computer with Minikube. The following metrics were recorded for this version: “Register Capacity Confirm”, which measures the time it takes for a consumer to register a demand with the broker and receive a confirmation. “Register Device Confirm” tracks the time it takes for a device to register itself with the broker and receive a confirmation. “Capacity Demand Status” refers to the time from when a consumer registers a demand to when it is allocated by the broker, representing a full round-trip time. An example of a sequence of these procedures are visualized in Figure 3.5, where “Register Capacity Confirm” is sequence 3 to 4, “Register Device Confirm” is 1 to 2, and “Capacity Demand Status” is 3 to 7. Additionally, for the proof-of-concept implementations, “Stable Recovery Time” was be measured. This metric indicates the time it takes for a broker to return to its stable state after a crash. This measurement is exclusive to the proof-of-concept implementations, as the base prototype does not support recovery from failure. A timer has been implemented to measure these durations, starting when the procedure begins and stopping when it ends. The goal of measuring latency is to assess how different implementations affect the systems latency and ensure it does not increase

excessively. The threshold for acceptable latency increase was determined through interviews with the team responsible for the system under investigation.

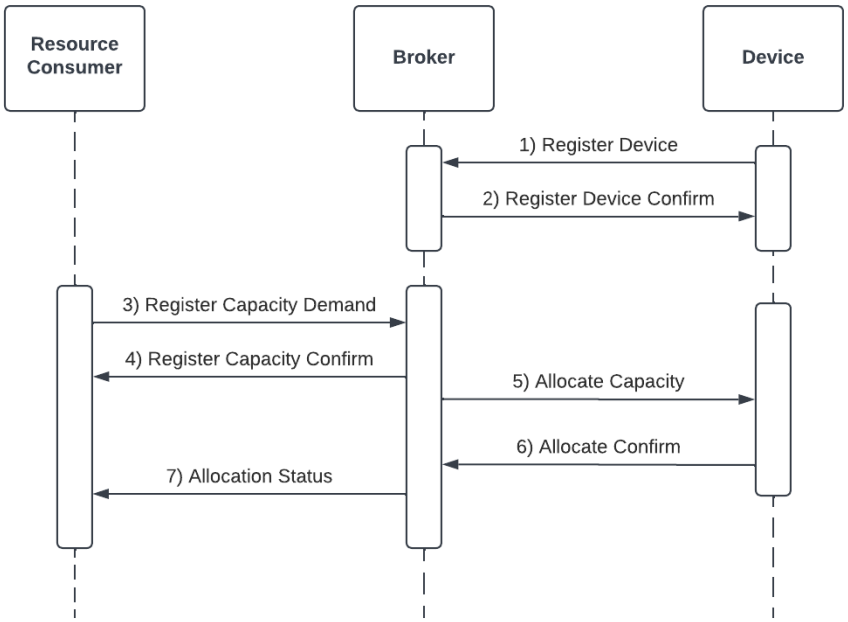

Figure 3.5: Sequence diagram of the procedures measured for the evaluation.

The measurement process involved implementing a timer within the system that recorded the duration of specific procedures in microseconds and logged this data to a log file. The same scenario was tested in each version of the system. A startup script initialized one broker, five devices, and five consumers, ensuring the devices' capacity matched the consumers' demand. Each procedure was measured 25 times, using the average value for comparative analysis. During the measurements, care was taken to keep unrelated modules operational to isolate the impact to only the module under test. For instance, during the "Register Device Confirm" measurement, it was crucial that both the consumer and broker were active to prevent their startup times from influencing the results. In the case of "Stable Recovery Time", only the broker was restarted, as it is the sole component designed to recover from failures in this scenario.

To assess the implementation complexity of the solutions, the steps required to implement the solution in the prototype were documented, along with any necessary decision-making processes. To ensure the relevance of these findings to the actual system, the solutions were further examined during the evaluation interviews. This discussion helped identify any additional steps needed to successfully integrate the solution into the real system.

4

Results

This chapter outlines the key findings and insights derived from the methodologies employed in this research. For ease of navigation, the sections are organized according to the distinct methodologies utilized: systematic mapping study, semi-structured interviews, and proof of concept implementations. Additionally, identified challenges and solutions are presented in individual sections to avoid duplications from the systematic mapping study and semi-structured interview sections.

4.1 Systematic Mapping Study Findings

The systematic mapping study aimed to address research questions RQ1 and RQ2 by exploring the challenges and solutions associated with data management during the migration to an MSA. The literature search resulted in 13 primary papers that covered a range of four main topics: Migration to MSA, MSA Challenges, MSA Solutions, and Embedded, all of which are relevant to identifying the state of the art related to this thesis's research questions. By further snowballing, an additional 23 papers were found and analysed. The classification of these papers is divided into main topics, which are further subdivided into subtopics. The number of papers in each classification and the division of topics can be seen in Figure 4.1, while Table A.1 in Appendix provides the identified papers and their topics.

As illustrated in Figure 4.1, the total number of papers assigned to the topics exceeds the number of analysed papers. This is because many papers address multiple topics. Although these topics are interrelated, each one also presents its own distinct and interesting aspects.

Migration to MSA describes the migration journey of a migration towards MSA. The topic is divided into three sub topics: Migration Processes, Decision Frameworks and Benefits/Driving Forces.

- **Migration Processes** includes papers that discuss the various processes involved in the migration journey and the activities that must be executed when migrating to an MSA.

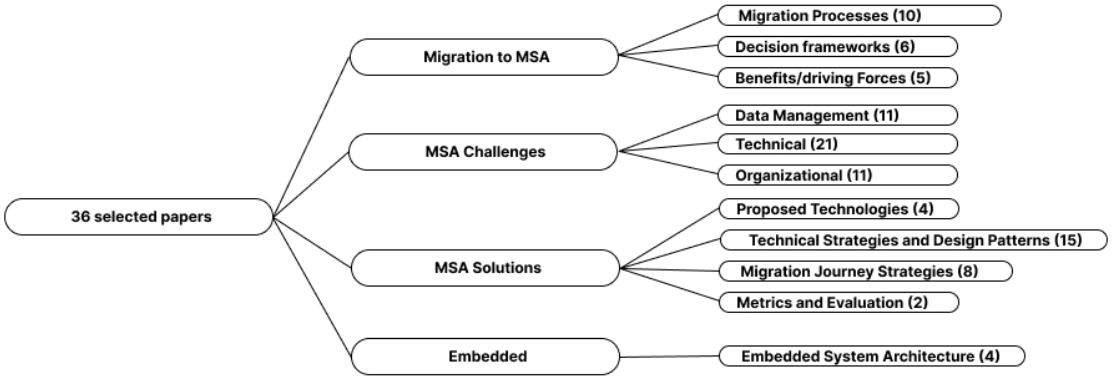

Figure 4.1: An overview of the topics of literature found in the systematic mapping study.

- **Decision Frameworks** are papers that propose frameworks for making decisions either during or before a migration.
- **Benefits/Driving Forces** are papers that discuss the benefits and driving forces behind moving towards an MSA.

MSA Challenges relates to challenges found in MSA migrations and is divided into three sub topics: Data Management, Technical and Organizational.

- **Data Management** are papers that mention specific challenges related to the data management aspects of migrations.
- **Technical** cover general technical challenges encountered during migrations.
- **Organizational** cover general organizational challenges encountered during migrations.

MSA Solutions were divided into four sub topics: Proposed Technologies, Technical Strategies and Design Patterns, Migration Journey Strategies, and Metrics and Evaluation.

- **Proposed Technologies** are papers discussing a certain technology and its effectiveness in an MSA.
- **Technical Strategies and Design Patterns** are papers that discuss specific architectural migration strategies or design patterns.
- **Migration Journey Strategies** focus on the overall migration journey, without focusing on specific patterns like the previous topic.
- **Metrics and Evaluation** include papers discussing how to measure a system

based on MSA and what metrics are interesting to monitor.

Embedded cover papers specifically related to the characteristics of embedded systems. Allowing us to get a better understanding of the domain and the requirements of embedded systems, thereby broadening the study beyond only focusing on how Ericsson's system operates. It is particularly useful for understanding various architectural designs such as unstructured monolithic, layered, and event-driven architectures [27].

4.2 Themes in the Semi-Structured Interviews

This section details the themes derived from the semi-structured interviews conducted in this thesis. The initial interviews were held with Team 1 (T1), who had previously undertaken a similar migration process. The objective of the first interview was to draw parallels between the system intended for migration (referred to as the target system) and a system that T1 had already migrated. The insights gained from this discussion were later shared with Team 2 (T2), the team responsible for the target system, to verify the similarities. Beyond confirming that the two systems share characteristics, the interview also contributed to the answer of research questions RQ1 and RQ2. The migration experiences of T1 proved highly relevant to T2, as the systems shared similar challenges and potential solutions. Both teams were also driven by the desire to enhance system availability and resilience through cloud migration. A thematic analysis was conducted on the interviews, where 8 themes were identified, as seen in Table 4.1.

Initial Interviews Analysis

Themes	**Summary**
Challenges	The challenges that T1 faced during their migration.
Solutions	Solutions that T1 adopted in their migration.
Requirements & Qualities	Requirements for the migration to be deemed successful and qualities they achieved.
Technical	Answers that describe the technical aspects of their system before, during and after the migration.
Organizational	Explanations of how the decisions were affected by organizational reasons.
Migration Activities	The process of migrating to cloud. This includes the different steps and activities needed, as well as intermediate designs.

Table 4.1: Themes found in thematic analysis of the initial interviews.

The evaluation interviews helped in evaluating the solution's effectiveness in the real system, as well as identifying a few additional implementations steps that are required in the actual system. The themes identified can be seen in table 4.2.

Evaluation Interview Analysis

Themes	Summary
Challenges	Challenges that may occur when implementing the solution in the real system.
Solutions	Possible solutions for the pitfalls of the system that we identified when implementing the proof-of-concept implementations.
Requirements & Qualities	Requirements and qualities for the solution to work in the real system and business area. This also includes restrictions from the embedded system, such as strict resource consumption.
Technical	Technical aspects of the system. Includes system characteristics and technologies utilized.
Migration activities	The migration activities necessary for implementing the proof-of-concept implementation in the real system.
Concerns	Concerns regarding how well the solution would fit in the real system.
Implementation Complexity	The implementation complexity of implementing the solution in the real system.

Table 4.2: Themes found in thematic analysis of the evaluation interviews.

Challenges discusses responses related to obstacles encountered during cloud migrations. In the initial interviews, this theme centers on the specific challenges faced by T1 during their migration process, including challenges explicit for their system but also challenges relevant to the system being studied. In the evaluation interviews, the focus shifts to potential challenges anticipated in applying the proof-of-concept implementation to a real system.

Solutions details the strategies and practices adopted to overcome challenges in an MSA cloud migration. The interviewee from the initial interviews shared solutions implemented during their migration, covering aspects like inter-service communication and approaches to statelessness. The evaluation interviews explored potential solutions for the pitfalls of the system that we identified during the proof-of-concept implementations.

Requirements & Qualities elaborates on the prerequisites and desired attributes for a successful cloud migration, including constraints related to existing legacy systems, such as resource consumption restrictions.

Technical categorizes responses detailing the technical aspects of the system before, during, and after the cloud migration. This includes technology usage, system architecture, and message flow. Responses from the initial interviews helped identify parallels with the studied system, whereas those from the evaluation interviews aided in refining the understanding of the target system and evaluation of the proof-

of-concept implementation.

Organizational delves into the dynamics of the teams workflow and how organizational factors influenced architectural decisions.

Migration Activities classifies the answers regarding the different migration activities it took T1 to migrate their system and the migration activities needed to implement the proof-of-concept implementations in T2's system.

Concerns captures concerns from evaluation interviewees regarding the suitability of the proof-of-concept solutions for their embedded system and business area.

Implementation Complexity addresses the perceived challenges in implementing the proof-of-concept implementation within T2s system, as discussed by interviewees in terms of the complexity of each migration activity.

4.3 Identified Challenges

Specifically focusing on data management challenges, we identified five key challenges from the literature, which was broadened by insights from the interviews. The identified challenges are, handling statefulness, data splitting, data consistency, data synchronization, and data normalization and are summarized in Table 4.3.

Identified Data Management Challenges

Challenge	Description
Handling Statefulness	Transforming the system to a stateless system and deciding on patterns and strategies.
Data Splitting	Breaking down a monolithic database into smaller databases, tables or schemas, and decoupling internal states.
Data Consistency	Achieving consistency with distributed data management.
Data Synchronization	Propagating data changes through the system.
Data Denormalization	Creating optimized data structures for each service.

Table 4.3: Overview of main data management related challenges encountered during an MSA migration.

Handling Statefulness is a challenge encountered when trying to migrate the system to MSA and allowing the system to benefit from cloud aspects like horizontal scaling, resilience and independent recovery. Achieving stateless and independent services is a crucial step for taking advantage of those benefits [8]. A service that is stateless does not retain information between requests, and each request should contain all the information required for the service to function. For this to work, any state that need to be persisted between requests must be stored externally, which separates the application code and the data management. When transitioning

a system towards being stateless and thereby implementing patterns to rely on external data, a considerable amount of code may need to be refactored, increasing the migration complexity. Therefore, deciding what pattern to use, where and how, requires a thorough analysis of existing code and system requirements. The initial interview held with T1 further supports the difficulty of this challenge, as they also faced this challenge during their cloud migration.

Data Splitting involves decomposing a monolithic database into smaller, service-specific databases [1]. In an embedded system, this also involves analysing internal states to make sure these comply with the new service granularity. The microservice principle of isolation which is key to microservices cannot be achieved without each service maintaining its own data. Relationships between the previous internal state of the system need to be adapted to work in the new split. The data splitting challenge also involves making decisions about what data engines are most suitable for each microservice, which requires careful analysis and consideration of system requirements and existing technologies.

Data Consistency is a challenge that is often mentioned in literature when dealing with distributed data management, which is a core aspect of MSA. Data consistency refers to the aspect that data in the system should be correct and uniform across the system, regardless of when or where it is accessed. Moving from a system following a strict consistency model to eventual consistency is complex and require changes to application logic that previously was based on the notion that the data used always was correct and consistent with the rest of the system. Many concurrent services updating data in parallel requires new solutions to handle this new consistency model without limiting the performance.

Data Synchronization is a challenge also found in literature as a significant challenge due to the need to ensure data consistency across independent and loosely coupled microservices. Microservices often manages its own data and therefore introduces complexities of keeping these data sources synchronized. The main challenge is implementing and establishing inner system communication mechanisms that keep data consistent across the system without tightly coupling the services. Many papers describe implementing inner system communication mechanisms as being a challenge, but it is often analysed on a more general level of inner system communication, but we decided to include it as a data management challenge as data synchronization is a part of those mechanisms.

Data Denormalization in microservices migration involves restructuring data to optimize performance and autonomy within a distributed system. Unlike normalized data, commonly used in monolithic architectures, denormalized data is duplicated across services, enabling each microservice to operate independently and allows for faster system operation by not having to fetch data from other services [1]. Data denormalization in microservices migration is challenging due to the need to maintain data consistency across multiple services with duplicated data. Synchronizing changes in real-time can be complex and introduce latency, and eventual consis-

tency models can lead to temporary inconsistencies. Additionally, managing redundant data increases storage costs and complexity, making it difficult to ensure data integrity and consistency throughout the system.

4.4 Identified Solutions

The subsequent section provides a brief overview of the solutions identified in the systematic mapping study and interviews that address the specified challenges. Six software design patterns were discovered for handling statefulness, and two for managing data synchronization and data consistency. Challenges related to data splitting and normalization are not addressed through software design patterns, but are rather resolved using strategies tailored to the specifics of the system undergoing migration.

4.4.1 Solutions for Handling Statefulness

This section provides a concise description of each solution for managing statefulness, highlighting their benefits and drawbacks. An overview of these solutions is presented in Table 4.4. All the patterns presented are aimed at making the services stateless.

Database per Service is an architectural pattern commonly used in microservices architectures [4], where each microservice has its own dedicated database, as seen in Figure 4.2. It enhances autonomy by allowing each microservice to manage its own database, which facilitates independent deployment, scaling, and management without affecting other services [28]. Additionally, it allows for varied database technologies across services to optimally address different needs. This setup promotes better data encapsulation since services expose data through APIs, making internal changes to the database invisible to other services. This architecture also improves fault tolerance, as the isolation between databases helps to prevent failures in one service from cascading to others.

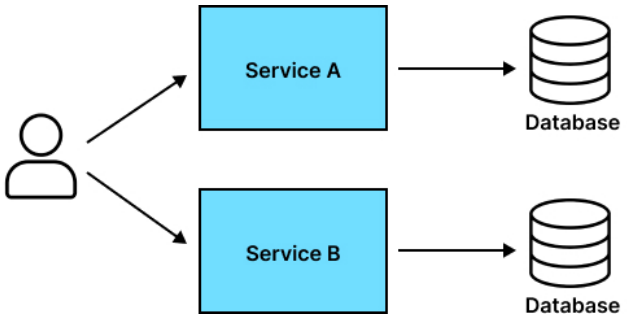

Figure 4.2: Example of the Database per Service pattern.

However, this approach is not without its drawbacks. Managing multiple databases increases complexity, especially concerning data consistency. It can also lead to data duplication and inconsistencies if not carefully managed. Moreover, handling

transactions that span multiple services and databases can become complex and may require advanced strategies like implementing distributed transaction patterns like Saga and CQRS. Lastly, maintaining several databases can lead to higher costs and increased resource consumption, adding to the infrastructure overhead.

Private Tables per Service is another pattern used by industry practitioners [29]. This pattern offers a middle ground between a shared database and the Database per Service pattern, and can be seen in Figure 4.3. In this pattern, each service owns a set of tables that are private to it in a shared database system. This method simplifies overall database management, while still providing a degree of isolation by restricting services from accessing each other's tables. However, it limits flexibility as all services must use the same database technology and can lead to resource shortages where high demand from one service may affect others.

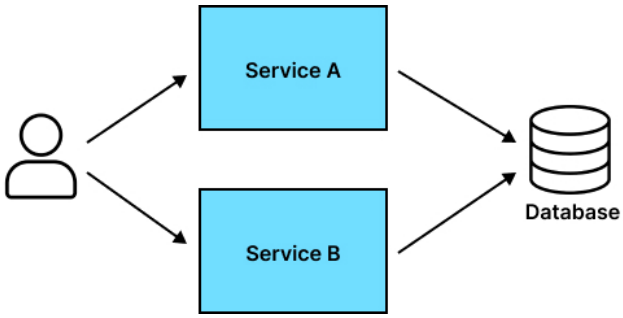

Figure 4.3: Example of the Private Tables per Service pattern and Schema per Service pattern. The difference is within the database.

Compared to the Database per Service pattern, Private Tables per Service offers easier management and potentially lower costs, but at the expense of reduced isolation and flexibility. This makes it a viable option for scenarios where moderate isolation is acceptable, and maintenance simplicity is prioritized.

Schema per Service is another alternative to Database per Service and Private Tables per Service and is frequently used in industry [29]. This pattern follows the same architecture as Private Tables per Service shown in Figure 4.3. It is a pattern in microservices architecture that involves each service operating its own schema within a shared database. This pattern is similar to Private Tables per Service, but allocates a distinct schema to each microservice instead of a table. This schema includes not just tables, but also relationships, views, and stored procedures that are specific to a service, effectively encapsulating the service's data and database logic. This isolation by schema allows for clearer boundaries between services compared to merely using private tables, as it groups all database objects related to a service together, offering a more robust form of isolation within the shared database environment. While offering higher isolation, it also comes with higher complexity than Private Tables per Service.

Stateful Messaging Pattern is a service-oriented architecture pattern that is also applicable for microservices architecture [8]. This pattern can be employed to transition a stateful service to a stateless one by relocating the internal states to a different service and embedding the entire state within the messages exchanged between services. While this method enables the targeted service to become stateless, it introduces technical debt to the service that now has to store the state, as it must manage the additional burden of another service's state. Additionally, this approach has a low memory usage because it does not rely on a database. However, this pattern also introduces challenges such as increased message complexity and size, which can lead to higher network traffic and slower processing times.

State Repository Pattern is another SOA pattern, applicable for MSA [8]. This approach allows microservices to operate in a stateless manner by offloading their state management to a stateful service designed specifically to store and manage the states. Figure 4.4 presents a simple example with 2 stateless services, Service A and Service B, and a stateful state repository service. This pattern does not use a database, which lowers the memory usage. However, this approach can introduce latency due to dependencies on the stateful service and may complicate state management. Additionally, it risks creating a single point of failure if the stateful service is not designed with robust failover mechanisms.

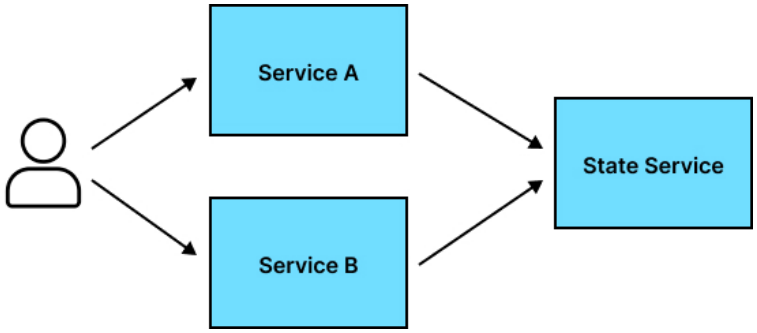

Figure 4.4: Example of the State Repository pattern.

4.4.2 Data Consistency & Data Synchronization

Two patterns were identified related to the challenges data consistency and data synchronization. These are two patterns that deals with the transaction of data in the system, as seen in Table 4.5.

The Saga pattern is a design pattern commonly used in MSA [29]. The pattern is used to manage distributed transactions [28]. It breaks a transaction into a series of steps, each handled by a different service. Each step has a corresponding compensating transaction to undo the operation in case of failure. This ensures data consistency and fault tolerance.

Solutions for Handling Statefulness

Solution	Benefits	Drawbacks
Database per Service	• Autonomy • Enhanced fault tolerance • Varied database technologies	• Data consistency challenges • Complexity managing multiple databases • Multiple service queries • Higher costs
Private Tables per Service	• Simplified database management • Some isolation without separate databases	• Limited flexibility • Possible resource bottlenecks • Less autonomy than Database per Service
Schema per Service	• Robust isolation • Clear boundaries and encapsulation	• Higher complexity than private tables • Less autonomy than Database Per Service • Database scalability
Stateful Messaging Pattern	• Low memory usage • Low implementation complexity	• Increased message complexity • Higher network traffic
State Repository Pattern	• Low memory usage • Low implementation complexity	• Single point of failure

Table 4.4: Identified solutions for dealing with the handling stateful challenge.

There are two types:

1. **Choreography-based Sagas:** Each service listens for events and triggers the next step.

2. **Orchestration-based Sagas:** A central orchestrator directs the transaction flow.

A simple transaction illustrating the Saga Pattern is shown in Figure 4.5. This figure applies to both types of Sagas and demonstrates the transactions involved in allocating capacity. If the process fails at any step, that action is reverted by a compensating transaction. For instance, if the allocation fails, it will be undone by a deallocation transaction. While effective, the Saga pattern adds complexity due to the requirement for compensating transactions and the management of asynchronous communication.

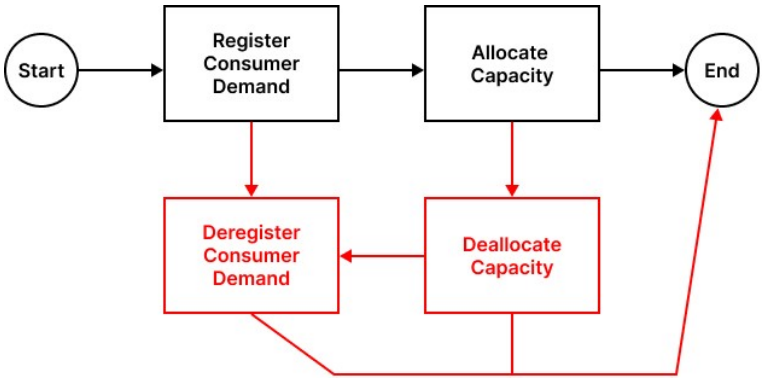

Figure 4.5: Simple example of a Saga transaction with compensating transactions.

Command Query Responsibility Segregation (CQRS) is a design pattern in software architecture that separates data modification operations (commands) from data retrieval operations (queries). Figure 4.6 illustrates this concept, showing how Service A writes to a dedicated write database via commands and reads data from a separate read database through queries. The databases are synchronized with eventual consistency, allowing independent scaling of read and write operations, which enhances performance and efficiency, especially during concurrent operations. However, this separation can introduce replication lag and adds complexity due to the need for managing and synchronizing the command and query models separately.

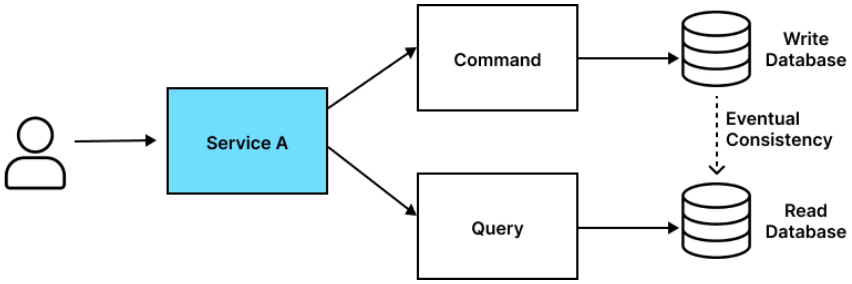

Figure 4.6: Example of the CQRS pattern.

Solutions Data Consistency and Data Synchronization

Solution	Benefits	Drawbacks
Saga	• Consistency across services • Enhanced fault tolerance	• Increased complexity
CQRS	• Independent scaling of read and write operations • Improved performance and efficiency	• Increased complexity • Data replication lag

Table 4.5: Identified solutions for dealing with the data consistency and synchronization challenge.

4.4.3 Data Splitting and Data Denormalization

Effective data splitting and data denormalization ensures that each microservice has access to the data it needs while minimizing dependencies on other services [1]. Specific solutions for data splitting were not identified in the systematic mapping study. From the interviews, it was identified that this problem requires unique solutions for each part of the system. Finding the optimal solution is a complex task in a large embedded system, involving a careful analysis of the data dependencies. This analysis can be overwhelming and as indicated by interviews that it can be shortened by prototyping different data splits and then evaluating the data split's effect on the system.

4.5 Findings from Proof of Concept

Two strategies have been implemented in the prototype system as a proof of concept. The choice of the strategies to be implemented was decided on relevancy, drawn by the result from the systematic mapping study and the interviews. Solutions that have been proven to be successful in similar systems, and solutions that fit the requirements and restrictions of the target system, were deemed highly relevant. The proof of concept of the strategies also aims to evaluate the complexity of implementation. The following sections describe the solutions, their implementation activities, and an evaluation of the result.

4.5.1 Stateless Architecture with Redis Clusters

The findings from the systematic mapping study and interviews emphasize the need for the system to transition from a stateful to a stateless architecture to support horizontal scaling. The system currently maintains a substantial amount of internal states, which limits its ability to scale horizontally. It is essential to adopt an appropriate solution for this problem. An interview with T1 revealed their use of

Redis Clusters to overcome this challenge. Given the success of this approach in T1's system and that the use of Redis Clusters aligns with solutions identified in the systematic mapping study (Database per Service, Private Table per Service and Schema per Service), this approach is considered highly relevant to our investigation. Consequently, it has been chosen as the initial strategy for implementation in the proof of concept.

The cluster is composed of several Redis instances, referred to as nodes, enabling the horizontal scaling of the Redis Cluster based on demand. In this particular setup, six nodes were employed, matching the number used in the example provided in the Redis Cluster documentation [30]. Redis Cluster incorporates numerous built-in features that aligns with cloud principles. For instance, data is sharded across the nodes, enhancing performance by distributing the load and reducing pressure on any single node. Additionally, Redis Clusters implements data replication using the master-replica model, which improves availability. Should a master node fail, the cluster remains operational. The architectural design includes three master nodes and three replica nodes, mirroring the configuration outlined in the Redis documentation example. The architectural design implementing this solution is illustrated in Figure 4.7.

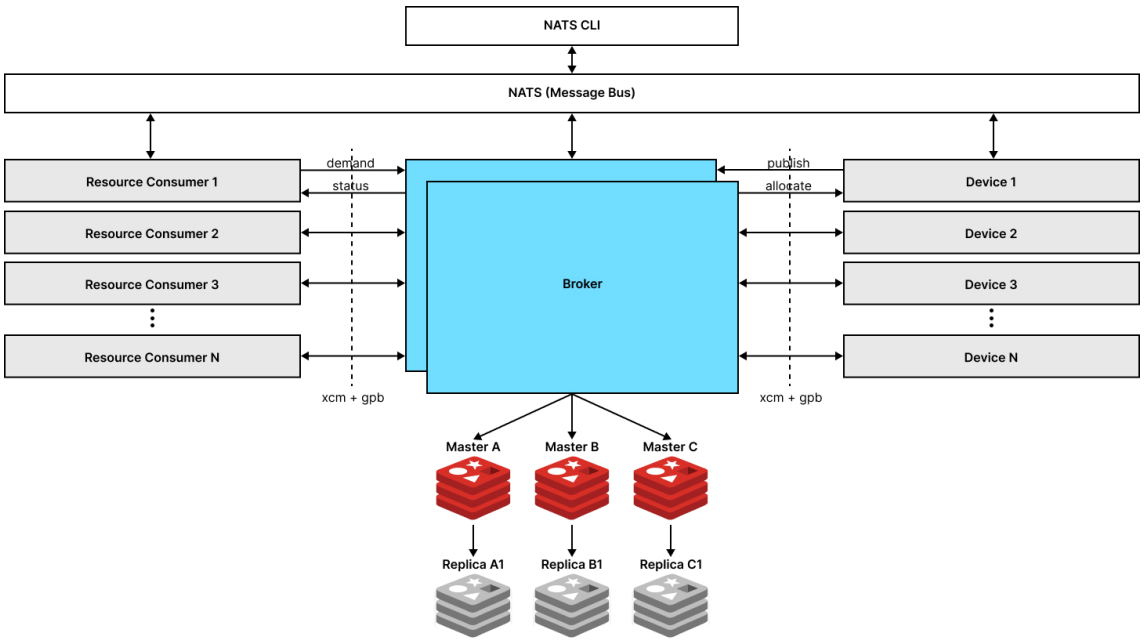

Figure 4.7: System Architecture with Redis Cluster.

4.5.1.1 Implementation Activities

There are 4 activities required to implement this solution in the real system, identified not only during the implementation step, but also from the literature and the interviews. The list of activities can be seen in Table 4.6.

Replace Communication Protocol is the first activity that should be considered before when dealing with the stateless challenge. This is an activity that was

Proof of Concept 1 Activities

Activity	Description
Replace Communication Protocol	Socket communication is suboptimal for MSA. This should be replaced with a technology that supports stateless communication.
Data Structuring	Carefully design the data structures to optimally function in the new architecture. This includes data splitting and data normalization/denormalization.
Redis Cluster Setup	Setup the Redis Cluster in Kubernetes. This includes designing the configuration files: StatefulSet, ConfigMap, and Service.
Refactor	Refactor all instances of the code that uses internal states to use the Redis Cluster instead.

Table 4.6: Identified activities for moving the internal states to a Redis Cluster.

not performed in the proof of concept, but instead identified during the proof-of-concept implementation and confirmed in the evaluation interviews. During the implementation, we realized that the new architecture was not as decoupled as initially expected. The microservices interact with each other using XCM over sockets. As a result, once a device or resource consumer connects to a specific broker instance, it continues to communicate exclusively with that instance. This setup could create load balancing challenges, where a few instances end up handling the majority of the transactions. To achieve full horizontal scaling, it is necessary to replace this or create a workaround. Drawing from the interviews, it is recommended to replace the communication protocol with one more fitting for MSA. T1 used HTTP and REST for their system, but other communication protocols like gRPC are also suitable. The different communication protocols can have an impact on designing the most optimal data structures. The activity of replacing the communication protocol should therefore be done before the upcoming activities.

Data Structuring is a crucial activity for successfully migrating a stateful system to stateless. This was also emphasized in the initial interviews. This activity includes the challenges data splitting and data normalization that were identified in the systematic mapping study. All interviewees also stated the importance of making sure that the data is backwards compatible to make sure that new versions of services can handle data produced by older versions to prevent disruptions in the system.

Redis Cluster Setup involves setting up the Redis Cluster in the Kubernetes deployment. This includes writing the configuration files to fit the use case and make the decisions of how many master and replica nodes to deploy and their timeout time. In this implementation, there are 3 master nodes and 3 replica nodes with a timeout of 5000 ms.

Refactor is the last activity of the migration. This involves refactoring all the instances of the code that uses internal states to instead use the Redis Cluster. This can have varied implementation complexity depending on the system architecture. In this case, the system already utilized a state manager that handled all the states, so the refactors were quite isolated.

4.5.1.2 Strategy Evaluation

While not flawless, this solution facilitates horizontal scaling, enabling the system to recover from broker crashes. Multiple instances of the broker can be deployed for enhanced availability. However, these advantages come at the cost of increased latency. Metrics indicate a latency increase by 7.5 times compared to the base prototype in the local Kubernetes cluster, as shown in Table 4.7. This notable increase warrants careful consideration during implementation. Fortunately, for the studied system, this is not a concern, as confirmed in the evaluation interviews. Time-critical actions are not a focus, and database reads and writes mainly occur at startup. Yet, interviewees expressed worries about resource consumption when using Redis Clusters. The embedded system's limited resources necessitate efficient utilization. Inconsistency concerns also arose with eventual consistency. One interviewee suggests prioritizing strong consistency to mitigate unexpected issues, even at the expense of performance. On the other hand, another interviewee proposes a simpler approach, employing fault handlers to restart affected components in case of data inaccuracies. Although not optimal performance-wise, given the solution's low complexity and the system's lack of time-critical procedures, this approach may suitably address the system's needs.

Proof of Concept 1 Metrics

Procedure	**Mean time [ms]**	**Comparison with base prototype**
Register Capacity Confirm	16.4	7.5x
Capacity Demand Status	24.1	7.5x
Register Device Confirm	15.2	8.7x
Broker Recovery Time	525	-

Table 4.7: Metrics from proof of concept 1.

4.5.2 Stateless Architecture using Stateful Messaging

Similar to the first proof of concept, there was a need to transition from a stateful to a stateless system to facilitate horizontal scaling and independent service recovery from failures. Another significant challenge for Ericsson's system is the requirement to develop solutions compatible with both the embedded platform and a cloud environment. Due to this requirement, there was a desire for a solution with minimal implementation overhead that could serve as an intermediate design, gradually migrating the system towards an MSA.

Therefore, a pattern that would allow the broker in our prototype to become stateless without adding a database was chosen. The chosen pattern is the Stateful Messaging Pattern. This pattern allowed the internal broker state to be externalized to the device and the Resource Consumer, while still allowing the state data to be retrieved and updated as messages over the XCM sockets already in place. The demand and allocation data was externalized to the resource consumer and device services, as seen in Figure 4.8, while groups remained in the broker as they are only used for configuration and were read from a file on start.

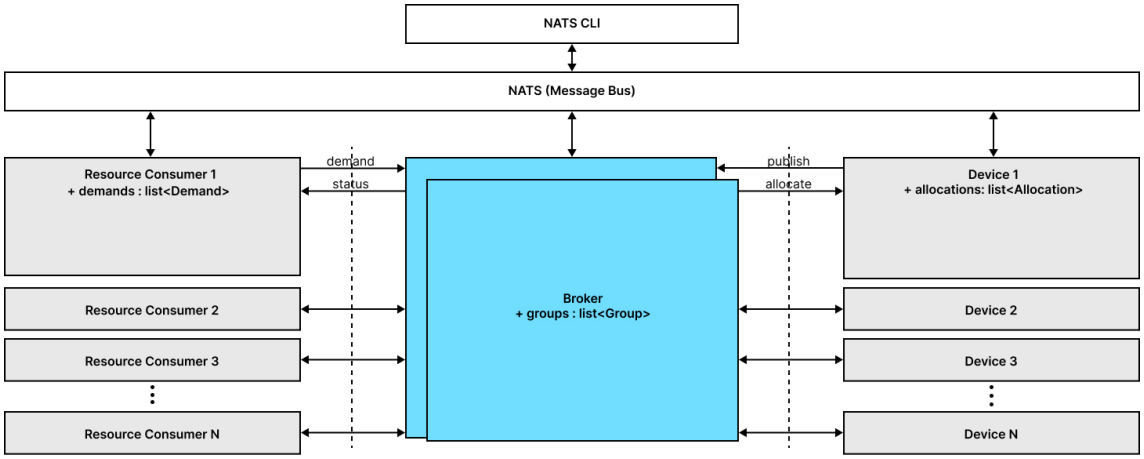

Figure 4.8: System using stateful messaging.

4.5.2.1 Implementation Activities

Implementing this solution in the real system requires three activities identified from implementing the proof of concept and during the evaluation interviews held afterwards. These activities are similar to what we found in the first proof of concept but with some slight differences and can be found in Table 4.8.

Proof of Concept 2 Activities

Activity	Description
Replace Communication Protocol	Socket communication is suboptimal for MSA. This should be replaced or adapted with a technology that supports stateless communication.
Data Structuring	Carefully design the data structures to optimally function in the new architecture. This includes data splitting and data normalization/denormalization.
Updating interfaces	Update message interfaces to handle sending and receiving state.

Table 4.8: Identified activities for moving the internal states to other services and using the Stateful Messaging Pattern.

Replace Communication Protocol is, like in the previous proof of concept, an important activity to validate whether the current communication protocol is suitable for the new design. In this proof-of-concept implementation, the use of sockets

remained unchanged, despite its discovered limitation on horizontal scaling. Due to time constraints, changing to a more suitable protocol was considered too time-consuming, but the solution can still be evaluated without addressing this aspect. However, as previously mentioned, this activity is key to fully unlock horizontal scaling and leverage the full benefits of the cloud.

Data Structuring was emphasized during the validation interviews as the most critical activity during the migration. Similar to its description in the first proof of concept, this activity is essential for addressing the challenges of data splitting and data normalization. Analysing which stateful data is needed for specific procedures and determining how to store that data is important for making the resulting system efficient and avoiding complexities. This activity also includes the externalization of the state, which in the proof of concept involved moving the allocation and demand data to the Device and Resource Consumer, respectively.

Updating Interfaces involves modifying the message interfaces between services to ensure that the newly externalized stateful data can be accessed and utilized effectively. In the proof-of-concept implementation, this required updating the GPB protocols to include all necessary data and modifying the endpoints to handle the received data correctly. An example of the GPB protocol used when a resource consumer connects to the broker is shown in Figure 4.9. In the updated design, the device and resource consumer send their allocation and demand data when connecting to the broker, allowing the broker to make decisions based on the previous data and update them accordingly.

```
message RegisterCapacityDemandReq {
    string allocator_name = 1;
    string capacity_group = 2;
    int32 capacity_demand = 3;
    int32 demand_id = 4;
    int32 allocation_id = 5;
}
```

Figure 4.9: GPB protocol, describing information sent to the broker from resource consumer when registering.

4.5.2.2 Strategy Evaluation

The Stateful Messaging Pattern enabled the broker to recover from failures independently, and except for the previously mentioned socket problem, it achieved horizontal scaling. This was accomplished by ensuring that the broker became stateless. This solution is less resource-intensive since no additional database needs to be added compared to the Database per Service pattern, thereby reducing the overhead of analysing additional technology choices and implementing them. The recovery time with this pattern which can be seen in Table 4.9 were 337ms which is slightly shorter but in the same magnitude as the Redis solution. Notably, there were no

significant increases in latency using this strategy compared to the base program, as the system performs similar operations as before, just with larger messages.

Proof of Concept 2 Metrics

Procedure	**Mean time [ms]**	**Comparison with base prototype**
Register Capacity Confirm	2.7	1.2x
Capacity Demand Status	5.1	1.6x
Register Device Confirm	2.6	1.5x
Broker Recovery Time	337	-

Table 4.9: Metrics from proof of concept 2.

However, the solution is not without flaws and comes with drawbacks. While it achieves statelessness where applied, applying this pattern on the entire system would introduce unbearable complexity by deferring the state to even more external services. Leading to highly complex couplings spanning over multiple services, breaking the microservice principle of independence. Moreover, using this solution results in an increase in the number of messages sent between services. Increasing the risk of an already existing problem of having too many messages being sent between services in the system causing failure, which was highlighted in the evaluation interviews as a concern. Distributing the internal state to another service also increases the coupling between these services, making maintenance and deployment more challenging.

Therefore, the most crucial activities involve carefully analysing the system to identify which data must be stored and determining when it needs to be sent over the communication channel. Evaluation interviews revealed that this solution could serve as an intermediate step toward a pure microservice architecture. It offers a way to externalize state without making major changes to the system, and allows for updating interfaces that can later be utilized in a more robust solution.

While evaluating the strategy, it was found in the interviews that all data that was previously in state is perhaps not necessary to store externally and can instead be calculated from other data. The activity of deciding what data is absolutely crucial to recover the state, should be considered while solving the data splitting and data normalization challenges.

5

Discussion

In this chapter, we address the research questions of this thesis, focusing on the challenges and strategies involved in migrating embedded systems to MSA, particularly focusing on data management. We discuss our own findings while reflecting on the existing research, aiming to provide a comprehensive understanding of the complexities involved in this migration. Additionally, we reflect on threats to validity of our study, and offer suggestions for future research.

5.1 Research Question 1

> **RQ1:** What are the technical challenges of data management when migrating a large embedded system to a microservices architecture?

The migration of embedded systems to a microservices architecture presents several technical challenges, particularly in the realm of data management.

5.1.1 Handling Statefulness

One of the primary challenges identified in the literature, and confirmed by our study, is transitioning from a stateful to a stateless architecture. Embedded systems are typically designed to perform specific tasks on dedicated hardware, leading to a stateful operation model. This statefulness goes against the ideally stateless nature of microservices, which leverage the elasticity and redundancy of cloud environments for high availability and reliability. In our study, the challenge of handling statefulness was significant, as it required decoupling the system state and ensuring that services could operate independently without relying on shared state information. This decoupling is crucial for achieving horizontal scaling and independent service recovery but introduces complexities in managing state information effectively.

5.1.2 Data Splitting

Migrating to a microservices architecture necessitates the decomposition of a monolithic database or a collection of internal states into smaller, service-specific databases.

This data splitting is critical for achieving service isolation and independent scalability. The process of data splitting involves determining the optimal way to partition data such that each microservice has access to the data it needs while minimizing dependencies on other services. Existing literature emphasizes the complexity of data splitting, which requires careful consideration of the data relationships and access patterns within the system. The challenge is further complicated by the need to maintain data integrity and consistency across multiple services. Additionally, we found that choosing the right data engine is more challenging for an embedded system than in a cloud environment, as different data engines consume varying amounts of resources, particularly memory. It is essential to find a data engine that meets performance demands without consuming too much resources, especially in resource-constrained embedded systems.

5.1.3 Data Denormalization

Data denormalization involves restructuring data to optimize performance and autonomy within a microservices architecture. We have found that this is not only a challenge of denormalizing the data, but also finding a balance between normalized and denormalized data, which involves trade-offs between data integrity, performance, and complexity. Normalization minimizes redundancy and ensures data integrity, which is beneficial for maintaining consistency and reducing storage costs in a centralized database. Denormalization, while introducing redundancy, enhances performance by allowing each microservice to access the data it needs without relying on others. This reduces latency and simplifies queries, but complicates data synchronization and consistency management. Existing work shows that in a microservices architecture, the benefits from a data denormalized solution often outweighs the benefits of a normalized solution [1]. In contrast, we have found that embedded systems with limited resources may not accommodate redundant data, and a denormalized data solution should be approached with caution.

5.1.4 Data Consistency

Ensuring data consistency in a distributed microservices architecture is inherently challenging. Unlike monolithic systems, where a single instance of the database ensures strong consistency, microservices require managing consistency across multiple distributed services. This challenge arises from the need to ensure that all instances of a particular data item are updated correctly across the system. The distributed nature of microservices means that achieving strong consistency can be difficult, leading to potential issues with data synchronization and eventual consistency. Managing distributed transactions and ensuring that all services have a consistent view of the data adds significant complexity to the system.

5.1.5 Data Synchronization

Data synchronization involves ensuring that changes to data are propagated consistently across all relevant services. This is particularly challenging in a distributed environment where services may be independently scaled and deployed. The need

for robust synchronization mechanisms to maintain data consistency was highlighted in our case study. Synchronization must ensure that data updates are propagated in a timely manner and that all services remain consistent. This is complicated by the decentralized nature of microservices and the potential for network delays and service outages.

5.1.6 Summary

The main challenges found are Handling Statefulness, Data Splitting, Data Denormalization, Data Consistency and Data Synchronization which highlights the complexity of data management in the context of migrating embedded systems to microservices architecture. Each challenge requires careful consideration and tailored approaches to ensure a smooth and effective transition. Our findings emphasize that addressing these challenges is crucial for leveraging the full benefits of a microservices architecture while maintaining the integrity and performance of the system.

5.2 Research Question 2

> **RQ2:** What are the most effective strategies for addressing the identified data management challenges?

This section answers RQ2 by exploring the most effective strategies for addressing the data management challenges encountered when migrating embedded systems to a microservices architecture.

5.2.1 Solving Statefulness

One of the main challenges when migrating embedded systems to a microservices architecture is handling statefulness. Ideally, the **Database per Service** pattern is considered the optimal solution for managing state in a microservice environment, according to existing work. This pattern provides each microservice with its own dedicated database, thereby enhancing service autonomy, fault tolerance, and scalability. By isolating the data, it allows services to be independently deployed, scaled, and managed without affecting others. When we implemented Redis Clusters using this pattern, additional latency was introduced but provided a robust and stateless solution, operating with high availability and resilience.

However, we found that while Database per Service is the ideal solution, it is not always the most practical or optimal choice for every system due to the unique characteristics and resource constraints of each system. Implementing this pattern increases complexity in managing multiple databases and ensuring data consistency across them. It also requires substantial changes to the existing system, which may not be feasible given resource constraints or the importance of maintaining existing operations during the migration process. Additionally, the memory consumption

associated with maintaining multiple databases can be too high, especially for embedded systems with limited resources.

For systems where Database per Service is not suitable, other strategies identified in existing literature can be employed:

Private Tables per Service simplifies database management while providing some degree of isolation. This approach is effective for systems where resource constraints or inter-service data dependencies make full database isolation impractical. It also consumes less memory compared to maintaining separate databases for each service.

Schema per Service balances the need for isolation with the practicalities of a shared database. It is suitable for systems that require clear service boundaries but cannot support the overhead of completely separate databases. This approach helps manage resource consumption efficiently. Both this pattern and Private Tables per Service rely on a shared database. This can easily introduce indirect coupling and does not allow the complete independence of services, which goes against core microservice principles and should be implemented with caution.

Stateful Messaging Pattern is particularly useful for systems where maintaining a low resource footprint is critical. This strategy however increases message complexity and network traffic but reduces memory usage by not relying on a database.

State Repository Pattern centralizes state management, which can introduce latency and potential single points of failure if not designed with robust failover mechanisms. This pattern is beneficial for systems that need to centralize state management for consistency but can tolerate the additional latency. It also reduces the memory load on individual services by consolidating state management.

5.2.2 Data Splitting & Data Denormalization Strategies

Successful implementation of the above patterns relies on carefully structuring and designing the data to deal with the challenges of data splitting and data denormalization. No specific solutions or strategies for these activities were found in the existing literature, but we have found that these challenges are instead solved on a case-by-case basis. From the interviews, we also identified another activity when performing data splitting that should be considered. Some data is not necessary to save, and can instead be calculated from other data. This simplifies the data splitting and data denormalization challenges. Finding the optimal solutions for these challenges are complex tasks, especially in a large embedded system, involving a careful analysis of the data dependencies. We have found that this process can be mitigated by shortening the analysis process and test different data splits through prototyping and evaluate its effect on the system.

5.2.3 Data Consistency & Data Synchronization Solutions

Ensuring data consistency and effective data synchronization across microservices are fundamental challenges in MSA. These challenges become particularly apparent when moving to a stateless architecture, as state management and data coordination must be handled across multiple services. This shift requires careful consideration of migration complexity, as updating code to accommodate eventual consistency can prove highly intricate. The patterns found from literature to handle these challenges are **Saga pattern** and **Command Query Responsibility Segregation (CQRS)**. Implementing these patterns can require much effort and a lot of refactoring in the existing code. However, we suggest that PACELC, with the tradeoff between consistency and latency, should be considered when solving these challenges. We have identified from interviews that if latency does not impact the system that much, easier solutions could be applied, solutions like locking the data when using it to make sure it stays consistent. This solution however introduces scalability issues and goes against core MSA principles, so this should be done with caution and be seen as an intermediate solution.

5.2.4 Intermediate Designs

To effectively implement any of these solutions, we have found that an iterative approach is the most suitable way of doing it. Designing intermediate designs that solve challenges for smaller parts of the system and gradually evaluate and migrate more of the system is much more manageable than trying to design the optimal system from the beginning. This has been further supported by responses during interviews where the risk of getting stuck in an analysis phase was discussed as a concern. Implementing smaller, and more manageable steps that make the migration process more effective by iteratively moving towards a solution that fulfils all requirements. A system like Ericsson's, that should support the ability to be deployed on both embedded and cloud platforms, can benefit from employing intermediate designs during the migration process. These designs may not be perfect for the cloud initially, but they still function on the embedded platforms. Intermediate designs allow for a gradual transition of the system towards an MSA capable of utilizing cloud aspects, thereby making the migration process more manageable. Following this approach can also allow for reaching a system that satisfies the customer requirements with less effort, as the system will evolve with customer demands instead of aiming to create a perfect system for the distant future.

5.2.5 Implemented Solutions

We implemented two solutions as a proof of concept in a small prototype of the real system and evaluated their effectiveness on embedded systems.

5.2.5.1 Proof of Concept 1: Database per Service with Redis Cluster

The first solution followed the ideal pattern of Database per Service, utilizing a Redis Cluster as the database. This approach enabled service autonomy, horizontal

scaling, and system recovery, leveraging the high availability and fault tolerance features inherent in Redis Clusters.

This solution introduced significant additional latency due to the overhead of managing an external database compared to the internal state handling of the base system. However, latency and recovery time were found to be non-critical for the embedded system under study. The system's operations were not highly time-sensitive, and database interactions primarily occurred at startup. While the implementation complexity in the prototype was manageable, evaluation interviews and existing literature indicated that applying this solution to the real system could entail greater implementation complexity. The Database per Service pattern introduces data synchronization and consistency challenges, requiring complex data handling strategies between services. Additionally, Redis Clusters use data replication for higher availability and resilience, which introduces replica lag that must be addressed.

5.2.5.2 Proof of Concept 2: Stateful Messaging Pattern

The second proof of concept utilized the Stateful Messaging Pattern to externalize the states by embedding it in the messages exchanged between services. This approach aimed to maintain a low resource footprint and avoid the complexity of managing multiple databases.

This solution introduced very low latency overhead because the data remained managed within the services without relying on an external database. While it increased message complexity and network traffic, resulting in a slight increase in latency, it had a significant lower latency than the Redis Cluster solution and had lower implementation complexity. It required less refactoring and was easier to integrate into the existing system, making it a more practical solution for the initial phase of migration.

However, while this solution presents several benefits, we recommend it only as an intermediate design due to the technical debt it introduces. The states of the services are moved to nearby services, which violates the microservice principle that each service should be responsible for its own data. Implementing this solution for several services can quickly increase the complexity of the system, as the state must be transported through multiple services before reaching the service that needs it. This added complexity can complicate maintenance and scalability in the long term.

5.2.5.3 Comparison and Findings

The proof of concept implementations highlighted the trade-offs between different strategies for handling statefulness and data management in a microservices architecture. While Database per Service with Redis Cluster demonstrated the potential for high availability and fault tolerance, its complexity and resource demands made it less suitable for immediate implementation. The second proof of concept with Stateful Messaging Pattern provided a more manageable approach with lower implementation complexity, making it a practical intermediate step.

Our findings suggest that an iterative approach, starting with simpler solutions like Stateful Messaging or State Repository pattern, and gradually evolving towards more complex patterns like Database per Service, appears to be the most effective strategy. This iterative approach aligns with the need to support both embedded and cloud platforms during the migration process, ensuring a smoother transition and better resource management.

5.2.6 Summary

Addressing data management challenges when migrating a large embedded system to a microservices architecture requires analyzing the system's unique characteristics and constraints. While the Database per Service pattern is ideal for handling statefulness, resource limitations and system complexity may necessitate alternative strategies like Private Tables per Service, Schema per Service, Stateful Messaging, and State Repository patterns.

Effective implementation depends on careful data structuring, thorough system analysis and prototyping. The Saga pattern and CQRS are effective for data consistency and synchronization but come with trade-offs in consistency, latency, and complexity. An iterative approach, starting with simpler solutions like Stateful Messaging or State Repository patterns and gradually moving towards more complex patterns like Database per Service, is found to be an effective strategy. This iterative approach facilitates a gradual and manageable transition to a microservices architecture that can leverage cloud capabilities.

5.3 Research Question 3

RQ3: What practical insights can be derived from the implementation of these strategies?

Implementing the strategies for migrating an embedded system to a microservices architecture has provided several practical insights. These insights shed light on the compatibility of existing technologies and patterns with the new architecture, as well as the processes necessary for a successful migration.

5.3.1 Prototyping and Testing

One important activity that we identified during the proof-of-concept implementations and discussed in interviews is the importance of prototyping and testing strategies on smaller parts of the system or a smaller prototype system, similar to this study, before a full-scale implementation. The evaluation interviews highlighted that analysing and planning for a large embedded system is complex, making it challenging to initiate the migration process. Prototyping and evaluating strategies make it easier to find suitable solutions, as opposed to attempting to design the perfect system from the outset. Incrementally developing and testing solutions allows for

early identification of potential issues and improvements of solutions, enabling a smoother migration process. If choosing to test strategies on a prototype system, it is essential to ensure that the prototype system is large enough to capture the system's characteristics. Otherwise, unforeseen issues may arise when implementing the strategy in the real system. In our case, the MigLab prototype system is a simplified and small version of the real system. Implementing strategies in this system led to findings that would have been difficult to uncover through literature alone, like the finding that the existing communication pattern worked poorly in the MSA solution. However, due to the prototype's small size, it was challenging to capture all the characteristics of the real system, and some important challenges and activities might have been missed as a result.

5.3.2 Identifying Existing Pitfalls

When implementing the strategies, compatibility issues with already used technologies and patterns can be discovered. Therefore, it is important to implement solutions on a smaller part of the system to identify these pitfalls early to avoid becoming too invested in the new design, and decrease the time spent on theoretical analysis and design. In our case, we found one major pitfall in the first proof of concept. Which was that the current implementation of XCM sockets that were used are not well suited for microservices as they hindered the system from achieving full scalability. This led us to the insight that establishing effective communication patterns that are well-suited for microservices, is an important activity to do before the data migration. Replacing or adapting existing communication techniques and patterns to align with MSA principles allow the system to take advantage from all cloud benefits.

Another significant pitfall we encountered was that integrating new technologies and design patterns into legacy systems often required more extensive refactoring than initially anticipated. For example, in the first proof of concept where we implemented Redis Cluster, we initially believed that the existing allocation algorithm would function correctly within the MSA solution. However, during the implementation, we realized that this was not the case. The algorithm had to be refactored to fetch and combine data differently due to the new data split. This required reworking the algorithm to handle the distributed nature of the data, which was more complex and time-consuming than expected. The need for extensive refactoring can significantly impact project timelines and company resources. Thus, having a flexible and iterative approach to prototyping and testing can help manage these integration challenges more effectively.

5.3.3 Summary

In summary, the practical insights gained from implementing migration strategies for an embedded system to a microservices architecture highlight the critical importance of prototyping and testing different approaches. We found that prototyping on smaller parts of the system or within a prototype version of the real system,

helps identify suitable solutions and potential issues early, making the overall migration process smoother and more manageable. Additionally, it reveals compatibility issues with existing technologies and patterns, and the potential extensive refactoring required when integrating new technologies. These findings underscore the value of an iterative and flexible approach, enabling the identification and resolution of challenges before full-scale implementation. By addressing these issues early in the prototype phase, we can ensure a more efficient and successful migration, fully leveraging the benefits of MSA.

5.4 Future Research

The findings and insights from this study open up for opportunities for future research. While this thesis provides a comprehensive exploration of the data management of migrating embedded systems to microservices architecture, there are areas that are interesting to further investigate.

5.4.1 Investigate More Solutions

This study focused on a few selected solutions. While these solutions provided valuable insights, there are many other design patterns and strategies for a migration to microservices architecture that remains to be investigated. Future work could involve testing and evaluating a broader collection of patterns, such as CQRS and Saga, which were never implemented in the proof-of-concept implementations.

5.4.2 Data Splitting & Data Denormalization Strategies

Data splitting and data denormalization emerged as common challenges during the migration process. These challenges require further investigation to develop robust strategies that ensure consistency, performance, and scalability. Future research could focus on identifying best practices for data partitioning and denormalization, because this was never found in our study. Furthermore, developing guidelines and frameworks to assist practitioners in making informed decisions about data management during migration could facilitate the migration process.

5.5 Validity Threats

In conducting this study, it is important to critically examine the threats to the validity of the research. These threats, if not sufficiently addressed, could undermine the study's credibility, reliability, and applicability. Below, the primary threats to validity within this study are outlined, in the same structure recommended by Robert K. Yin [20].

5.5.1 Construct Validity

Construct validity aims to ensure that the methodology of the thesis accurately measures the concept it was designed to evaluate [20]. There is a risk that these methods does not fully encapsulate the complexities of migrating to MSA. To mitigate this, the study uses methodological triangulation, leveraging multiple sources of evidence to ensure an accurate result [20], [21]. Additionally, we developed a small prototype based on the real system from Ericsson to study the system in a practical setting, enhancing the construct validity by providing a practical context for evaluation. This approach also allowed for additional findings, not found in existing literature.

5.5.2 External Validity

External validity concerns arise from the study's reliance on a single case study at Ericsson, raising questions about the generalizability of the findings. The unique context and characteristics of this case may limit the applicability of the conclusions drawn to other organizations facing a similar migration journey. To mitigate this, the findings are compared with existing literature in the field, aiming to increase the generalizability of the found strategies. Moreover, the literature study conducted provides an understanding of how specific or generic the findings are, based on the context of the studied example. Further research is encouraged to test the study in broader environments.

5.5.3 Internal Validity

Internal validity concerns relate primarily to the casual relationships identified in the study [20]. It is not possible for the prototype system to perfectly replicate the real system, and the selection of migration strategies were constrained by time and resources. This study transparently disclaims this threat and have systematically chosen strategies based on relevance and insights gained from literature and interviews. Additionally, the interviewees were selected using a combination of purposeful and convenience sampling, which could introduce bias as the participants may not represent the broader population of experts in the field. This potential bias is mitigated by cross-referencing the interview findings with those from the systematic mapping study.

5.5.4 Reliability

Ensuring that the study can be replicated with the same results is crucial for its reliability [20]. The personalized nature of semi-structured interviews and the specifics of the case study's context pose challenges to achieving consistent findings in future replications. To mitigate this, the study includes comprehensive documentation of the research process, including the development of the interview guide, criteria for literature selection, and detailed descriptions of the proof-of-concept implementations. This aims to enable other researchers to replicate the study under similar conditions. Moreover, for the systematic mapping study, we utilized Google Scholar to identify relevant literature. However, it is important to acknowledge that relying

on Google Scholar introduces a potential threat to the study's reliability. Due to its search algorithm, the results may not be the same for everyone, which can affect the replicability of the literature search process.

We also used ChatGPT for grammar and language correction to ensure clarity and coherence in our documentation. While this helps in maintaining the readability and professional presentation of the thesis, there is a potential risk that multifaceted meanings or technical terms might be unintentionally altered. To mitigate this risk, we reviewed all changes made by ChatGPT to ensure the correctness of the text.

5.6 Conclusion

Migrating a large embedded system to a microservices architecture presents a complex but necessary evolution to meet modern demands for scalability, maintainability, and operational efficiency. This thesis has explored the challenges and effective strategies associated with such a migration, with a specific focus on data management.

The research highlighted the challenges of handling statefulness, ensuring data consistency, managing data synchronization, data splitting and data denormalization in a microservices migration. Through a systematic mapping study, semi-structured interviews, and proof-of-concept implementations, we identified several strategies to address these challenges, including the Database per Service, Private Tables per Service, Schema per Service, Stateful Messaging Pattern, and State Repository Pattern.

Key findings from the study emphasize the importance of prototyping and iterative testing. Prototyping smaller parts of the system or a simplified prototype, as done with MigLab, allows for the early identification of potential issues and the refinement of solutions before full-scale implementation. This approach also helps uncover compatibility issues with existing technologies and patterns, reducing the risk of extensive refactoring later in the process.

We discovered that while the Database per Service pattern is ideal for managing state in microservices, it is not always feasible for systems with significant resource constraints. Alternative strategies like Private Tables per Service and Schema per Service offer practical solutions by balancing data isolation with resource efficiency. The study also underscored the need for robust data synchronization mechanisms and highlighted the Saga and CQRS patterns as effective, though complex, solutions for maintaining data consistency across distributed services.

Moreover, our research emphasized the significance of a flexible and iterative approach to migration. By implementing intermediate designs, organizations can gradually transition to MSA, which not only facilitates a smoother migration process but also ensures that the system remains functional and efficient throughout the transition.

In conclusion, migrating an embedded system to MSA requires careful planning, a thorough understanding of the system's unique constraints, and a commitment to iterative development and prototyping. By leveraging the strategies and insights gained from this research, organizations can navigate the complexities of microservices migration more effectively, ultimately achieving a more scalable, maintainable, and efficient system architecture.

Bibliography

[1] H. M. Ayas, P. Leitner, and R. Hebig, "An empirical study of the systemic and technical migration towards microservices," *Empirical Software Engineering*, vol. 28, p. 85, 2023. DOI: 10.1007/s10664-023-10308-9. [Online]. Available: https://doi.org/10.1007/s10664-023-10308-9.

[2] S. Wang, C. Du, J. Chen, Y. Zhang, and M. Yang, "Microservice architecture for embedded systems," in *2021 IEEE 5th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, IEEE, vol. 5, 2021, pp. 544–549.

[3] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, vol. 137, p. 106 600, 2021, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106600. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000793.

[4] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data management in microservices: State of the practice, challenges, and research directions," *arXiv preprint arXiv:2103.00170*, 2021.

[5] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st. O'Reilly Media, Feb. 2015, p. 280, ISBN: 978-1491950357.

[6] T. Salah, M. Jamal Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2016, pp. 318–325. DOI: 10.1109/ICITST.2016.7856721.

[7] S. Hassan, R. Bahsoon, and R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1651–1681, 2020. DOI: https://doi.org/10.1002/spe.2869. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2869. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2869.

[8] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency," *IEEE Software*, vol. 35, no. 3, pp. 63–72, 2018. DOI: 10.1109/MS.2017.440134612.

[9] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019, ISBN: 9781492047841. [Online]. Available: https://books.google.se/books?id=iul3wQEACAAJ.

[10] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015. DOI: 10.1109/MS.2015.11.

[11] M. Waseem, P. Liang, A. Ahmad, M. Shahin, A. A. Khan, and G. Márquez, "Decision models for selecting patterns and strategies in microservices systems and their evaluation by practitioners," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 135–144. DOI: 10.1145/3510457.3513079.

[12] P. Marwedel, *Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things*. Springer Nature, 2021.

[13] M. Wu, Y. Zhang, J. Liu, *et al.*, "On the way to microservices: Exploring problems and solutions from online q&a community," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 432–443. DOI: 10.1109/SANER53432.2022.00058.

[14] S. Simon, "Brewers cap theorem," *CS341 Distributed Information Systems, University of Basel (HS2012)*, 2000.

[15] IBM, *What is the cap theorem?* https://www.ibm.com/topics/cap-theorem, Accessed: 2024-02-20, 2024.

[16] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[17] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–2909. DOI: 10.1109/ICSA.2018.00012.

[18] Amazon Web Services, *Acid vs base databases*, https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/, Accessed: 2024-02-20, 2024.

[19] Thomas Heinrichs, *Microservice orchestration vs choreography whats the difference?* https://camunda.com/blog/2023/02/orchestration-vs-choreography/, Accessed: 2024-02-22, 2024.

[20] R. Yin, *Case Study Research and Applications: Design and Methods* (Supplementary textbook). SAGE Publications, 2017, ISBN: 9781506336152. [Online]. Available: https://books.google.se/books?id=fHE3DwAAQBAJ.

[21] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999. DOI: 10.1109/32.799955.

[22] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, 2008, pp. 1–10.

[23] S. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, 2005, 10 pp.–23. DOI: 10.1109/METRICS.2005.24.

[24] F. Kamei, I. Wiese, C. Lima, *et al.*, "Grey literature in software engineering: A critical review," *Information and Software Technology*, vol. 138, p. 106 609, 2021, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.

106609. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000860.

[25] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, pp. 77–101, Jan. 2006. DOI: 10.1191/1478088706qp063oa.

[26] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018, ISSN: 1049-331X. DOI: 10.1145/3241743. [Online]. Available: https://doi.org/10.1145/3241743.

[27] J. Beningo, "Embedded software architecture design," in *Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques*, Springer, 2022, pp. 29–53.

[28] Chris Richardson, *Microservices.io*, https://microservices.io/, Accessed: 2024-02-16, 2024.

[29] E. Ntentos, U. Zdun, K. Plakidas, D. Schall, F. Li, and S. Meixner, "Supporting architectural decision making on data management in microservice architectures," in Sep. 2019, pp. 20–36, ISBN: 978-3-030-29982-8. DOI: 10.1007/978-3-030-29983-5_2.

[30] Redis, *Redis cluster documentation*, https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/, Accessed: 2024-04-15, 2024.

[31] M. F. Gholami, F. Daneshgar, G. Beydoun, and F. Rabhi, "Challenges in migrating legacy software systems to the cloud an empirical study," *Information Systems*, vol. 67, pp. 100–113, 2017, ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2017.03.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437917301564.

[32] A. Ahmad and M. A. Babar, "A framework for architecture-driven migration of legacy systems to cloud-enabled software," in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14 Companion, Sydney, Australia: Association for Computing Machinery, 2014, ISBN: 9781450325233. DOI: 10.1145/2578128.2578232. [Online]. Available: https://doi.org/10.1145/2578128.2578232.

[33] J. Kazanaviius and D. Maeika, "Migrating legacy software to microservices architecture," in *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, 2019, pp. 1–5. DOI: 10.1109/eStream.2019.8732170.

[34] J.-F. Zhao and J.-T. Zhou, "Strategies and methods for cloud migration," *Int. J. Autom. Comput.*, vol. 11, no. 2, pp. 143–152, Apr. 2014, ISSN: 1476-8186. DOI: 10.1007/s11633-014-0776-7. [Online]. Available: https://doi.org/10.1007/s11633-014-0776-7.

[35] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Sep. 2019. DOI: 10.1109/icsme.2019.00081. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2019.00081.

[36] H. M. Ayas, P. Leitner, and R. Hebig, "Facing the giant: A grounded theory study of decision-making in microservices migrations," *CoRR*, vol. abs/2104.00390,

2021. arXiv: 2104.00390. [Online]. Available: https://arxiv.org/abs/2104.00390.

[37] W. Assunção, J. Krüger, and W. Mendonça, "Variability management meets microservices: Six challenges of re-engineering microservice-based webshops," Oct. 2020, pp. 1–6. DOI: 10.1145/3382025.3414942.

[38] K. Brown and B. Woolf, "Implementation patterns for microservices architectures," in *Proceedings of the 23rd Conference on Pattern Languages of Programs*, ser. PLoP '16, Monticello, Illinois: The Hillside Group, 2016.

[39] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proceedings of the 2nd International Workshop on Refactoring*, ser. IWoR 2018, Montpellier, France: Association for Computing Machinery, 2018, pp. 1–6, ISBN: 9781450359740. DOI: 10.1145/3242163.3242164. [Online]. Available: https://doi.org/10.1145/3242163.3242164.

[40] F. Auer, M. Felderer, and V. Lenarduzzi, "Towards defining a microservice migration framework," in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, ser. XP '18, Porto, Portugal: Association for Computing Machinery, 2018, ISBN: 9781450364225. DOI: 10.1145/3234152.3234197. [Online]. Available: https://doi.org/10.1145/3234152.3234197.

[41] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017. DOI: 10.1109/MCC.2017.4250931.

[42] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, 2018. DOI: 10.1109/MS.2018.2141035.

[43] V. N. Saa Bakarada and A. Koronios, "Architecting microservices: Practical opportunities and challenges," *Journal of Computer Information Systems*, vol. 60, no. 5, pp. 428–436, 2020. DOI: 10.1080/08874417.2018.1520056. eprint: https://doi.org/10.1080/08874417.2018.1520056. [Online]. Available: https://doi.org/10.1080/08874417.2018.1520056.

[44] M. Wu, Y. Zhang, J. Liu, *et al.*, "On the way to microservices: Exploring problems and solutions from online q&a community," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2022, pp. 432–443.

[45] X. Zhou, S. Li, L. Cao, *et al.*, "Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry," *Journal of Systems and Software*, vol. 195, p. 111 521, 2023, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111521. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222001972.

[46] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.09.082. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218302139.

[47] C. Berger, B. Nguyen, and O. Benderius, "Containerized development and microservices for self-driving vehicles: Experiences best practices," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 7–12. DOI: 10.1109/ICSAW.2017.56.

[48] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar, "Microservices: Migration of a mission critical system," *IEEE Transactions on Services Computing*, vol. 14, no. 5, pp. 1464–1477, 2021. DOI: 10.1109/TSC.2018.2889087.

[49] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2021.

[50] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city internet of things platform with microservice architecture," in *2015 3rd International Conference on Future Internet of Things and Cloud*, 2015, pp. 25–30. DOI: 10.1109/FiCloud.2015.55.

[51] T. Hänisch, "A case study on using microservice patterns in an embedded system,"

[52] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch, "How to adapt applications for the cloud environment: Challenges and solutions in migrating applications to the cloud," *Computing*, vol. 95, pp. 493–535, 2013.

[53] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering*, I. Garrigós and M. Wimmer, Eds., Cham: Springer International Publishing, 2018, pp. 32–47, ISBN: 978-3-319-74433-9.

[54] A. Henry and Y. Ridene, "Migrating to microservices," in *Microservices: Science and Engineering*, A. Bucchiarone, N. Dragoni, S. Dustdar, *et al.*, Eds. Cham: Springer International Publishing, 2020, pp. 45–72, ISBN: 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_3. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_3.

A

Appendix 1

	Processes during migration	Decision frameworks	Benefits/driving forces	Data management	Technical	Organizational	Proposed technologies	Technical strategies and Design Patterns	Migration journey strategies	Metric and evaluation	Embedded system architecture
An empirical study of the systemic and technical migration towards microservices [1]	X					X		X			
Challenges in migrating legacy software systems to the cloud an empirical study [31]	X					X					
A Framework for Architecture-driven Migration of Legacy Systems to Cloud-enabled Software [32]	X					X			X		
Migrating Legacy Software to Microservices Architecture [33]					X				X		
Strategies and Methods for Cloud Migration [34]							X	X	X		
From monolithic systems to Microservices: An assessment framework [3]		X								X	
Migrating Towards Microservice Architectures: An Industrial Survey [17]	X					X			X		

Microservices Migration in Industry: Intentions, Strategies, and Challenges [35]			X			X			X		
Facing the Giant: a Grounded Theory Study of Decision-Making in Microservices Migrations [36]	X	X							X		
Variability Management meets Microservices: Six Challenges of Re-Engineering Microservice-Based Webshops [37]					X	X					
Implementation Patterns for Microservices Architectures [38]								X			
Migrating towards Microservices: Migration and Architecture Smells [39]				X	X			X			
Towards Defining a Microservice Migration Framework [40]		X									
Processes Motivations and Issues for Migrating to Microservices Architectures An Empirical Investigation [41]				X	X	X			X		
Migrating Enterprise Legacy Source Code to Microservices [8]		X	X	X	X			X			
Using Microservices for Legacy Software Modernization [42]	X							X			
Architecting Microservices: Practical Opportunities and Challenges [43]						X			X		
On the Way to Microservices: Exploring Problems and Solutions from Online Q&A Community [44]	X				X	X		X			
Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry [45]					X	X					
Microservice transition and its granularity problem: A systematic mapping study [7]	X				X						
Decision Models for Selecting Patterns and Strategies in Microservices Systems and their Evaluation by Practitioners [11]		X			X			X			
The pains and gains of microservices: A Systematic grey literature review [46]					X						

Microservice Architecture for Embedded Systems				X	X					X	X
Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices [47]					X		X				
Microservices: Migration of a Mission Critical System [48]				X	X		X	X			
Data Management in Microservices: State of the Practice, Challenges, and Research Directions [4]	X		X	X	X			X			
Supporting Architectural Decision Making on Data Management in Microservice Architectures [29]		X		X	X			X			
Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum [49]					X		X	X			
Designing a Smart City Internet of Things Platform with Microservice Architecture [50]			X		X						
A Case Study on using Microservice Patterns in an Embedded System [51]			X					X			X
Embedded Software Architecture Design [27]											X
Embedded Software Architecture Design Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things [12]											X
How to adapt applications for the cloud environment [52]				X	X			X			
Consistency Tradeoffs in Modern Distributed Database System Design [16]				X	X						
Challenges When Moving from Monolith to Microservice Architecture [53]				X	X						
Migrating to Microservices [54]	X			X	X	X		X			

Table A.1: Table of all the literature analysed in the final set of the systematic mapping study and which topics they involved