CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Microfrontends in Practice: Adoption Challenges and Architectural Trade-Offs

Master's Thesis in Computer science and engineering

POUYA SHIRIN SOKHAN

ANDREAS WICKSTRÖM JOHANSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Microfrontends in Practice: Adoption Challenges and Architectural Trade-Offs

POUYA SHIRIN SOKHAN
ANDREAS WICKSTRÖM JOHANSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Microfrontends in Practice: Adoption Challenges and Architectural Trade-Offs
POUYA SHIRIN SOKHAN & ANDREAS WICKSTRÖM JOHANSSON

Supervisor: Philipp Leitner, Department of Computer Science and Engineering
Examiner: Gregory Gay, Department of Computer Science and Engineering
Examiner in practice: Linda Erlenhov, Department of Computer Science and Engineering

Master's Thesis 2025
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 2025

POUYA SHIRIN SOKHAN & ANDREAS WICKSTRÖM JOHANSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As frontend applications grow increasingly complex, micro-frontends have been proposed as a solution to improve scalability, flexibility, and team autonomy by decomposing monolithic frontend architectures into independently developed and deployed modules. However, despite their potential, micro-frontends have seen slower adoption compared to microservices. This study investigates the challenges developers face when adopting micro-frontends, the problems they aim to solve, and the solutions proposed in practice. A grounded theory approach was applied to developer discussions on Stack Overflow, focusing on real-world experiences during both early and late stages of adoption. The findings reveal that while micro-frontends offer important benefits such as independent deployment, technology flexibility, and organizational scalability, they also introduce a range of technical and theoretical complexity–including challenges in integration, migration processes, modularization strategies and infrastructure setup. Solutions proposed by developers highlight the importance of adhering to integration patterns, managing dependencies, and establishing clear architectural guidelines. This study aims to provide a better understanding of the trade-offs involved in adopting micro-frontends and offers practical insights for organizations considering this architectural approach.

Keywords: Micro-frontends, Adoption challenges, Adoption motivations, Web development, Grounded theory, Stack Overflow, Migration strategies, Integration complexity, Qualitative coding, Card sorting.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Philipp Leitner for their invaluable guidance and support throughout the development of this thesis. We would also like to extend our appreciation to our examiner, Linda Erlenhov, for the time to review our work and provide constructive suggestions, which have greatly contributed to the final result.

Andreas Wickström Johansson & Pouya Shirin Sokhan, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

The rapid evolution of web development has brought challenges in designing, maintaining, and scaling modern applications [1]. As web applications become more complex, traditional monolithic front-end architectures often struggle to keep pace and meet the demands of scalability, flexibility, and maintainability. In a monolithic approach, all components of an application, from the user interface to business logic and data access, are tightly coupled and deployed as a single application [2]. This approach, while sufficient for smaller applications, introduces substantial drawbacks as the application grows. Despite modern web development often separating the frontend and backend, monolithic architectures still result in tightly coupled deployments, making scaling specific parts of the application inefficient. For example, the entire system may need to scale or redeploy, even if only a small module of the system requires an update.

To address these issues, the microservices architecture was introduced. Microservices break down the back-end of an application into smaller, independently deployable services [3]. Each service is responsible for a specific piece of functionality and can be developed, deployed, and scaled independently. This modularity enables teams to work autonomously with tools best suited to their needs. By decoupling services, organizations gain the flexibility to update or replace individual components without affecting the entire system.

While microservices is a new architectural approach to the back-end, the front-end remained largely monolithic. The latter often suffered from the same issues that plagued monolithic back-end systems [1]. In response, these limitations have prompted the rise of micro-frontends. In 2016, ThoughtWorks Technology Radar introduced the term *micro-frontends*, though the software architecture did not gain significant traction until 2019, following an article by Cam Jackson [4, 5]. Jackson's article provided an in-depth exploration of Micro-frontends, and a timeline shared by ThoughtWorks indicated that adoption of the approach began to rise later that year [6], along with industry leaders such as DAZN, IKEA, and Starbucks [7]. Micro-frontends extends the principles of microservices to the front-end and decomposes the user interface into smaller independently managed modules [4]. This approach allows teams to build, deploy and maintain each module separately, solving many of the same problems as microservices did for the back-end. The benefits and issues of micro-frontends are similar to those of microservices [7]. Like microservices,

micro-frontends enable individual teams to work with their preferred frameworks and tools, promising increased developer productivity, faster development cycles, and improved system resilience. However, both architectural approaches may also increase the payload size of the application, code duplication and coupling between teams.

1.1 Problem Description

Microservices have revolutionized back-end development by providing modular, scalable, and independent systems, yet the same principles have not gained equivalent traction in front-end architectures. Micro-frontends extend microservices' modularity to the user interface in order to solve key problems in front-end development. Despite this potential, their adoption remains limited compared to microservices. Between 2019 and 2021, a surge of articles and blog posts suggests the growing popularity of Micro-frontends. However, more recently, discussions about the architecture appear to have waned.

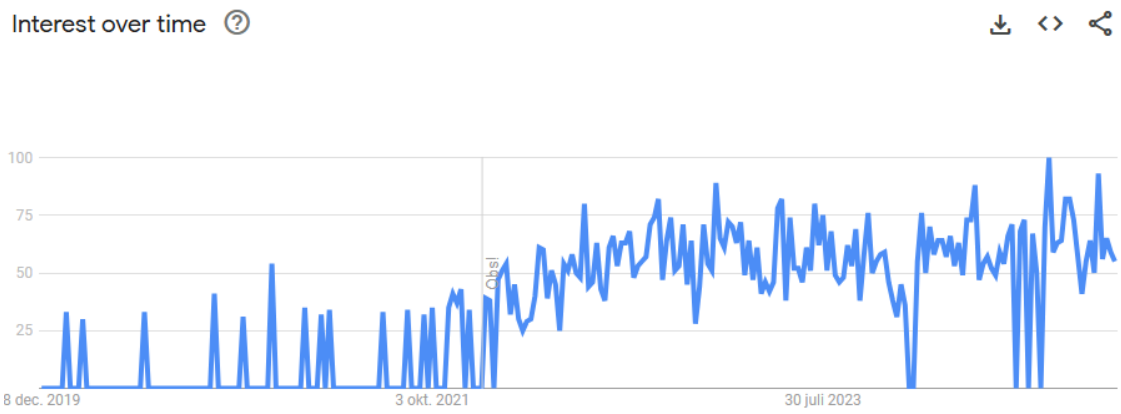

Figure 1.1: Google search interest for "Micro-frontends" from December 2019 to late 2024.

Figure 1.1, illustrates the growing interest in micro-frontends over time, with notable increases in search activity starting in 2021 and stabilizing at higher levels in subsequent years. Figure 1.2 compares the popularity of microservices and Micro-frontends, where the popularity of the former outperforms the latter. While this is arguably not conclusive evidence, this trend aligns with the perception of micro-frontends' limited adoption.

In a survey done by Software House, *State of Frontend 2022*, only 24.6% of developers reported using micro-frontends [8]. In contrast, JetBrains *State of Developer Ecosystem 2022* found that 37% are engaged in microservices [9]. These numbers, together with the trend shown in Figure 1.1, suggest that the adoption rate has stagnated since 2022, and may even have declined.

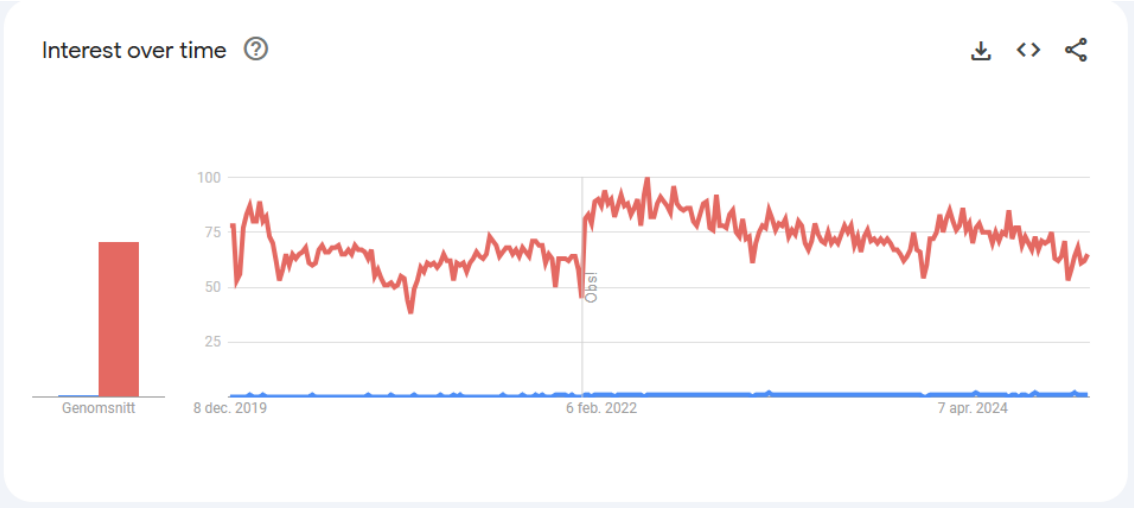

Figure 1.2: Google search interest comparison between "Micro-frontends" (blue) and "Microservices" (red) from December 2019 to late 2024.

In 2024, a Reddit thread raised questions about the current state of micro-frontends [10]. The general consensus among users suggested that the approach is primarily viable for larger companies. Several users remarked that they had never encountered Micro-frontends in practice, while others noted that prominent tech companies had reverted to monolithic architectures.

The slower adoption of micro-frontends may indicate a gap in understanding their value and challenges. Furthermore, developers may also face uncertainties about their implementation, cost-effectiveness, and long-term sustainability. The literature on micro-frontends is sparse, with limited studies evaluating their effectiveness in solving real-world problems.

1.2 Purpose of the Study

The purpose of this study is to investigate the adoption challenges of micro-frontends, explore why they have not achieved the same level of popularity as microservices, and identify the specific problems they aim to solve in modern web development. By examining these aspects, the study seeks to provide a holistic understanding of the trade-offs, benefits, and constraints associated with micro-frontends. The research aims to gather discussions and documented experiences to identify obstacles and assess the problems micro-frontends address.

The study aims to benefit software developers by offering insight into the adoption process, essentially helping them to make informed decisions about implementing micro-frontends, making the architectural approach more accessible and practical to adopt.

Three research questions have been defined to help achieve the purpose of this study:

- **RQ1: What are the key challenges developers face when adopting micro-frontends?** Understanding the challenges developers face is critical to identifying barriers to adoption. These may include technical difficulties, integration complexities, or organizational resistance. We found that developers face both technical and theoretical challenges when adopting micro-frontends. These cover issues related to integration, modularization strategies, migration processes, and infrastructure setup. This research question uncover practical insights to help organizations and teams mitigate these obstacles, both in early and late adoption stages.

- **RQ2: What specific problems in frontend development do micro-frontends aim to solve?**
  By identifying the specific problems they are intended to address, we aim to clarify when, how, and why micro-frontends may be beneficial. Micro-frontends have been proposed as a solution to common frontend issues such as scaling large applications, enabling autonomous teams, and integrating diverse technologies. Our results confirm these motivations to reflect the need to address structural limitations found in traditional monolithic frontend architectures.

- **RQ3: What are the commonly suggested solutions to these challenges?**
  Identifying the solutions to known micro-frontend adoption challenges allows developers to build upon previous work. It can streamline the adoption process and guide future wants and needs from micro-frontend specific frameworks. Our results show that developers often suggest practices such as architectural conventions, shared tooling, clear configuration, and modular design principles to address recurring adoption issues. These solutions emphasize the importance of coordination, planning, and a standardized development environment to successfully implement micro-frontends.

1.3 Significance of the Study

Understanding the adoption challenges of micro-frontends is crucial for organizations implementing this approach to avoid costly mistakes. Companies investing in modern web architectures face critical trade-offs related to development speed, maintainability, performance, and scalability. By identifying the challenges and limitations of micro-frontends, this thesis aims to help organizations to make more informed decisions, ensuring they do not adopt an architecture that may introduce unforeseen complexity or inefficiencies. Furthermore, a clear understanding of challenges can help assess whether this architectural approach aligns with their business and technical needs. Knowing the trade-offs involved can help businesses determine when micro-frontends provide advantages and when a monolithic approach is more suitable. This helps to prevent mistakes where organizations invest in micro-frontends without fully understanding their drawbacks.

Despite the growing adoption of microservices, academic research on micro-frontends remains sparse. Much of the existing knowledge is based on industry articles, blog posts, and company experiences, making it difficult to assess their effectiveness in a structured manner. As a rough indicator, a quick search on Google Scholar conducted in March 2025 for the term *microservices* returned over 64,000 results, compared to *micro-frontends* which yielded fewer than 700 results.

By examining adoption challenges faced by developers, this study contributes to the broader field of frontend architecture and software modularization, aiming to address the research gap.

2

Background

This chapter provides the necessary background to understand the key concepts of this thesis. It introduces micro-frontends as an architectural approach in web development and explains grounded theory as a research methodology.

2.1 Micro-frontends

Micro-frontends extend the principles of microservices to the front-end, including independent development, deployment, and scaling of user interface components [4]. They are intended to address the challenges in scaling large frontend applications and integrating diverse technologies [11]. Traditional monolithic front-end architectures often end up becoming difficult to maintain as applications grow in complexity. Changes to a single feature may require redeploying the entire application, leading to development bottlenecks and increased risks. Micro-frontends aim to solve these issues by decomposing the front-end into smaller, loosely coupled fragments, each owned by an autonomous team and developed with its preferred frameworks and technologies. These teams are typically cross-functional with a distinct area of business on which they primarily focus. Each team builds its features end-to-end, meaning they are responsible for the entire development lifecycle of a feature, from the backend to the frontend.

Micro-frontends follow several architectural principles:

- **Independent Development and Deployment:** Each micro-frontend can be built, tested, and deployed independently without affecting other parts of the application.
- **Technology Agnosticism:** Teams can use different front-end frameworks (e.g., React, Angular, Vue) within the same application, allowing greater flexibility.
- **Isolated State Management:** To prevent global state conflicts, micro-frontends typically maintain their own state management mechanisms.
- **Scalability and Maintainability:** Smaller, modular components are easier to scale and maintain over time, especially in large organizations with multiple

development teams.

2.1.1 System architecture

The current trend in software development is to implement monolithic frontends. In Figure 2.1 there is a visual representation of popular architectures that rely on monolithic frontends. Non-monolithic frontends such as micro-frontends facilitate modularization in frontend development and systems. Through leveraging modularization, new organizational opportunities present themselves such as the application of Domain-Driven Design seen in the vertical architecture in Figure 2.2.

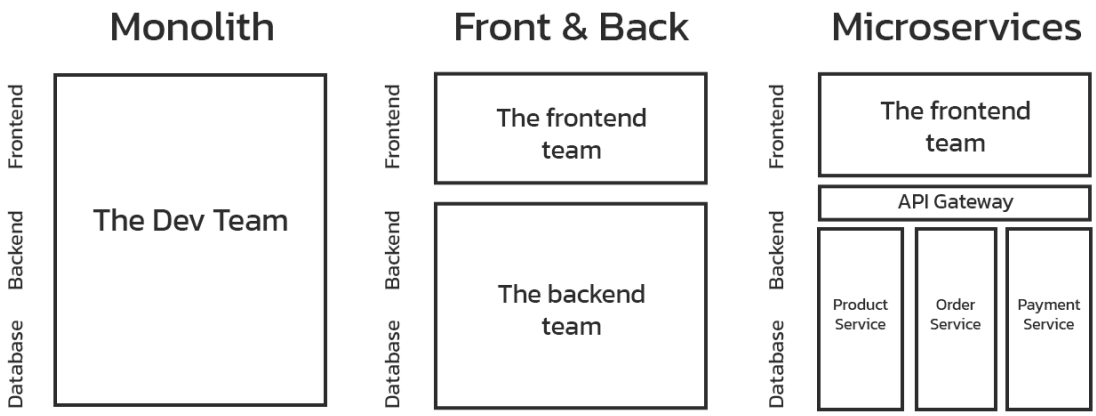

Figure 2.1: Popular architectures with monolithic frontends.

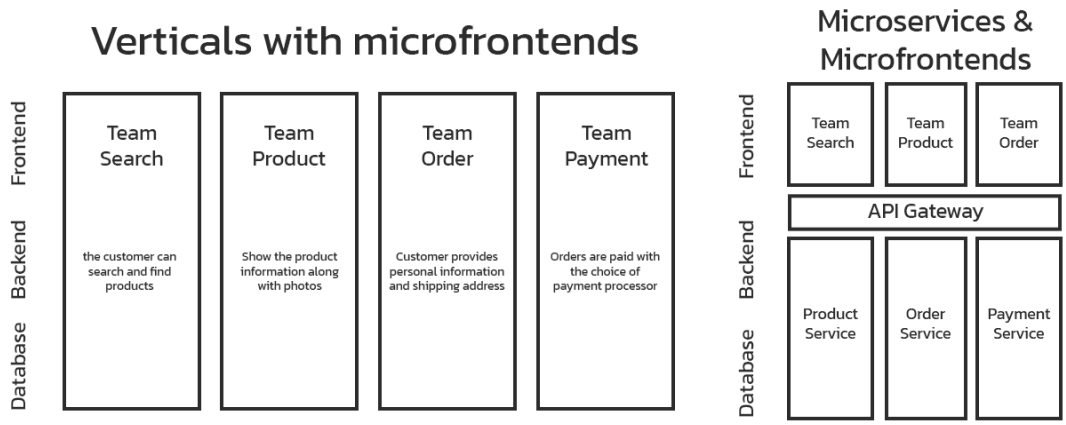

Figure 2.2: Popular architectures with micro-frontends.

2.1.2 Composition patterns

Micro-frontends are typically integrated with either a horizontal or vertical split:

- Horizontal split: Multiple micro-frontends per page.
- Vertical split: One micro-frontend at a time.

The horizontal split is a component-based approach, natively supported by browsers with Custom Elements. Each component is considered a micro-frontend. One team "owns" that micro-frontend and is responsible for that specific element on the page. Meanwhile a vertical split implementation promotes Domain-Driven Design [12], one team is responsible for the functionality of the entire page.

A shell application implemented with horizontal split has additional overhead. First, a team needs to be assigned as responsible for the page layout. Which micro-frontends and where they should be integrated. This puts importance on central governance. The team responsible for the page layout must communicate expectations and establish an API contract of which the other teams should follow. Second, micro-frontend communication, which is even more important with horizontal split as components must display a shared state. Another consideration is how the page will be assembled from micro-frontends, referred to as composition.

Client-side composition integrates micro-frontends directly in the browser from its application shell. Each micro-frontend must therefore declare a single entry point, allowing the shell to dynamically append its DOM elements. Single-SPA is a micro-frontend framework that utilizes client-side composition [13]. It maps micro-frontends to certain routes, then mounts or unmounts the respective application when the user is navigating the site. Single-SPA can be compared to a layout engine as its main responsibility is to display the correct content, the correct micro-frontend, for each route.

Edge-side composition assembles page fragments at the content delivery network (CDN) level to construct a complete HTML page. This approach supports the integration of both static and dynamic content, with pages typically built once or a few times and cached to serve subsequent requests efficiently. While edge-side composition delivers fully rendered HTML pages, it limits personalization at the CDN level due to the static nature of the cached content. However, technologies such as Edge Side Includes (ESI), a markup language designed for page assembly on CDNs, enable granular control over content composition, as illustrated in Listing 1. ESI allows static content to be cached for performance while enabling on-demand retrieval of personalized elements, such as user-specific headers containing login credentials.

```
<esi:include src="/static/header" /> <!-- Cached -->
<esi:include src="/dynamic/user-greeting" /> <!-- Fetch -->
<esi:include src="/static/footer" /> <!-- Cached -->
```

Listing 1: Interleaving static and dynamic content in ESI

Server-side composition constructs a complete HTML page on the server by integrating multiple micro-frontend components. This approach requires generating a unique page for each user request, as runtime integration precludes caching of fully assembled pages. Consequently, creating pages from scratch for every request can significantly impact server performance, affecting both the shell application and

individual micro-frontends. Server-side composition is commonly implemented using server-side includes (SSI), which employ a template-like syntax to incorporate dynamic content into the page structure, as demonstrated in Listing 2. This syntax enables the seamless integration of micro-frontend fragments, facilitating the delivery of personalized content tailored to each user.

```
<my-element data-arg1="hello">
  <!--#include virtual="https://...example.com?q=world" -->
</my-element>
```

Listing 2: Server-side include syntax

Build-time composition assembles micro-frontend fragments during the build process, generating a set of artifacts, including HTML, CSS, and JavaScript files. These artifacts are optimized for deployment on a web server or content delivery network (CDN). Unlike runtime composition approaches, build-time composition requires a new build to be published to reflect changes to micro-frontends in production, limiting the ability to update content dynamically. This static nature enhances performance by enabling efficient caching but necessitates redeployment for modifications, distinguishing it from server-side and edge-side composition techniques.

2.1.3 Definining micro-frontends in this study

In this paper we consider micro-frontends as modularized units of frontend with a shell application and a number of remote applications. This definition encompasses various implementation approaches, including iframes, custom elements, and frameworks such as Single-SPA, all of which are considered micro-frontends irrespective of the tools employed. Furthermore, both scalability and availability are traditionally considered key characteristics of micro-frontends similar to microservices. However, we will not introduce any criteria related to either scalability or availability as those are implementation details that are inaccessible to us. With this wider definition independent of tooling, implementation, and deployment we hope to construct theory that is relevant to micro-frontends as a concept rather than theory around specific frameworks, implementations, or deployment strategies. This inclusive approach allows for investigation of key motivations behind micro-frontends, which is often both organizational as well as technical. Our study shows developers adopting micro-frontends to enable independent development with organizational scalability and flexibility. A rigid definition that excludes architectures with significant organizational benefits would overlook real-world adoption patterns.

2.2 Grounded Theory

Grounded theory is a qualitative research methodology that focuses on developing theories directly from empirical data rather than relying on pre-existing frameworks.

Originally introduced by Glaser and Strauss in 1967 [14], this approach is particularly useful for studying new or under-researched topics, as it allows researchers to construct explanations based on real-world observations rather than theoretical assumptions.

A key characteristic of grounded theory is its iterative process, in which data collection and analysis occur simultaneously. Unlike traditional research methods that begin with a hypothesis, grounded theory enables researchers to refine their understanding as new insights arise. The methodology generally follows three main stages:

- **Data Collection:** Gathering information through interviews, case studies, literature reviews, and industry reports.
- **Coding and Categorization:** Identifying key themes and patterns within the data.
- **Theory Development:** Constructing a theoretical framework that explains the observed phenomena.

Grounded theory is a suitable approach for this study, given the limited academic research on the subject. By applying this method, we aim to provide an analysis of the practical challenges developers face when implementing micro-frontends, ensuring that the findings are grounded in practical experience rather than speculative assumptions.

3

Related Work

This chapter reviews relevant contributions, in foundational works, empirical case studies, and applied research in related areas such as microservices and software engineering practices. By situating our study in relation to these works, we seek to identify knowledge gaps, methodological differences, and recurring insights that our findings aim to complement or challenge.

3.1 Foundational Concepts

Luca Mezzalira's book *Building Micro-Frontends* [11] offers a comprehensive introduction to micro-frontend architecture, providing both conceptual guidance and implementation strategies. Mezzalira describes micro-frontends as a way to extend the benefits of microservices to the frontend layer, enabling modularity, team autonomy, and the integration of diverse technologies. He continues to explain that while micro-frontends solve real scalability and integration problems in frontend development, they also introduce significant challenges, particularly around cross-team coordination, shared infrastructure, and architectural complexity. These challenges are not solely technical but also organizational in nature, requiring certain techniques to avoid inefficiencies. Mezzalira's experience complements our work by offering a holistic view of both the intended goals and the practical complications that teams may encounter. While his work is prescriptive and experience-based, our study aims to examine how these goals and challenges are reflected in real-life developer scenarios.

3.2 Adoption Benefits and Challenges

Peltonen et al. [7] followed a Multivocal Literature Review process analyzing hundreds of sources discussing micro-frontends. A relevant question answered in relation to our work is whether the implementation of micro-frontends introduced any issues. The article leveraged open coding and selective coding, concepts related to grounded theory, to categorize both academic literature and gray literature into categories of technology-related issues and people-related issues. These two categories encompasses additional categories among others, "UX consistency" and "islands of knowledge" respectively. Our work captures adoption challenges, both migration

and maintainability challenges. The categories of issues laid forward by Peltonen et al. are primarily related to maintainability over migration, since the article never mentions issues equivalent to "blockers". Issues that are often seen during the first implementation which could deter potential adopters, such as difficulty in establishing module federation as a result of inadequate or complex tooling.

Antunes et al. [15] builds upon the work of Peltonen et al. by conducting an empirical case study to assess how the challenges and benefits identified in the literature hold up in practice. Their study involved a focus group with developers engaged in a micro-frontend migration project, during which participants rated and discussed a set of 21 statements derived from Peltonen et al.'s findings. The feedback provided both quantitative and qualitative validation of the identified themes. Although many of the expected benefits such as team autonomy, deployment independence, and modularity, were confirmed, the developers also raised trade-off concerns, such as growing complexity, coordination overhead, and long-term maintainability. Notably, their results highlight that even after hands-on experience, developers remained hesitant toward full adoption, stating that the architecture increased cognitive and technical burden. Compared to our work, which draws on a public developer forum to capture real-world concerns during adoption, Antunes et al. provide structured practitioner insight into how those concerns manifest in practice.

Pavlenko et al. [16] conducted a case study in implementing a single-page application, SPA, according to the principles of micro-frontends. The authors found that development time was excessively spent overlooking the architecture and development tools over feature development. As a result they suggested the architecture primarily towards complex large-scale projects with experienced developers where the overhead could be warranted. Furthermore, they remarked that the community lacked ready-to-use solutions to bootstrap projects and handle the newly added overhead at the time of publication. The case study is outdated as the modern frameworks and libraries pertaining to micro-frontends is everchanging. Its result could be misleading depending on the context. Considering that adopting micro-frontends is as much a technical challenge as an organizational challenge, the shifting focus from features to architecture and development tools could be a consequence of organizational resistance, which is not explored in the article. In contrast, our study draws on a large number of discussions from StackOverflow, which allowed us to identify a broad range of real concerns.

3.3 Practitioner Perspectives

Gaur [17] proposes micro-frontends when developing enterprise applications such as a Customer Relationship Management (CRM) system. The paper provides an overview of the development process together with considerations, key features, challenges, and how to address them. Gaur suggests several strategies for tackling challenges faced by practitioners; awareness and discoverability across teams to mitigate duplication, a full team dedicated to governance, CSS governance with scopes to

remedy cross-contamination, and establishing a common layer for communication between micro-frontends. Compared to Gaur's practitioner-oriented study, which aligns closely to our research, we deliberately derive theory solely from published artefacts. By excluding our own biases and prior literature, we aim to produce findings that accurately reflect public discussions traceable to specific StackOverflow posts.

3.4 Lessons from Microservices

In the microservices domain, specifically migration to a microservice architecture or MSA, Söylemez et al. [18] have identified 9 challenges specific to that domain; service discovery, data management and consistency, testing, performance prediction-measurement-optimization, communication and integration, service orchestration, security, MTL, and decomposition. A systematic literature review was used to identify these challenges with academic literature including journal papers, conference papers, workshop papers, and books. Söylemez et al. have also identified suggested solutions to the aforementioned challenges. While both challenges and solutions are insightful, they do not broach topics like tooling and technical maturity that become a prominent factor during implementation. The proposed solutions are rather theoretical concepts over practical help. With our work we hope to bridge this knowledge gap in the micro-frontends domain and provide an overview of solutions used in practice.

3.5 Use of Grounded Theory

Lercher et al. [19] utilized grounded theory to analyze data gathered from 17 semi-structured interviews. The article bears resemblance to our work, although it is only relevant in the domain of microservices over micro-frontends. It lists the strategies and issues developers leverage and face respectively. The issues are further categorized into two broader categories affecting API evolution, "tight organizational coupling" and "consumer lock-in". The authors propose automating change impact analysis together with effective change communication between teams. Our work will follow the same general direction but with a larger emphasis on issues. Strategies, such as frameworks that leverage micro-frontends, are ever-changing, making the issues they share much more valuable. In addition, our work will refrain from comparing grounded theory concepts such as the number of occurrences for a specific code or category. Our data source, StackOverflow, is highly moderated with explicit guidelines to discard duplicates, which would render the comparison unusable.

3.6 StackOverflow in Empirical Software Research

Barua et al. [20] demonstrated that StackOverflow is a valuable data source to understand real-time developer concerns and tool adoption in software engineering. Their study establishes the foundation for using community forums as empirical

data for research. Following on this, both Bandeira et al. [21] and Michael Ayas et al. [22] also leverage StackOverflow as a primary or complementary data source respectively. The data selection described in Bandeira et al. resembles our selection criteria where posts are fetched by the tag "microservices". Posts are filtered through discussion between the authors, based on the posts' relevance to the research. The discussions deemed relevant are then classified into three categories; technical discussions, conceptual discussions, and non-related discussions. This classification is much similar to our categorization of technical and theoretical challenges; however, we classify aggregates of posts in our qualitative analysis; which to an extent, rules out uncertainties such as posts that could belong to either category.

However, in contrast to our approach, Bandeira et al. discards posts without an answer as they are not considered to contain meaningful discussion and therefore deemed irrelevant. Posts that do not exceed a popularity threshold are also discarded. Our approach deliberately includes posts of varying qualities, as long as they are relevant to micro-frontends adoption. We argue that such posts may contain unresolved or emerging challenges and are therefore worth analyzing.

4

Methods

This paper follows the methodological framework outlined in Figure 4.1. The base of our data source was StackOverflow posts. The data was fetched, formatted, downsampled, and then filtered by criteria to capture relevant high-quality posts. Each research question is covered separately in the figure resulting in three artefacts; (i) set of challenge categories, (ii) set of goal clusters, and (iii), a challenge-solution mapping. The results were based on the study of these three artefacts, with each component of the methodology detailed in the next sections.

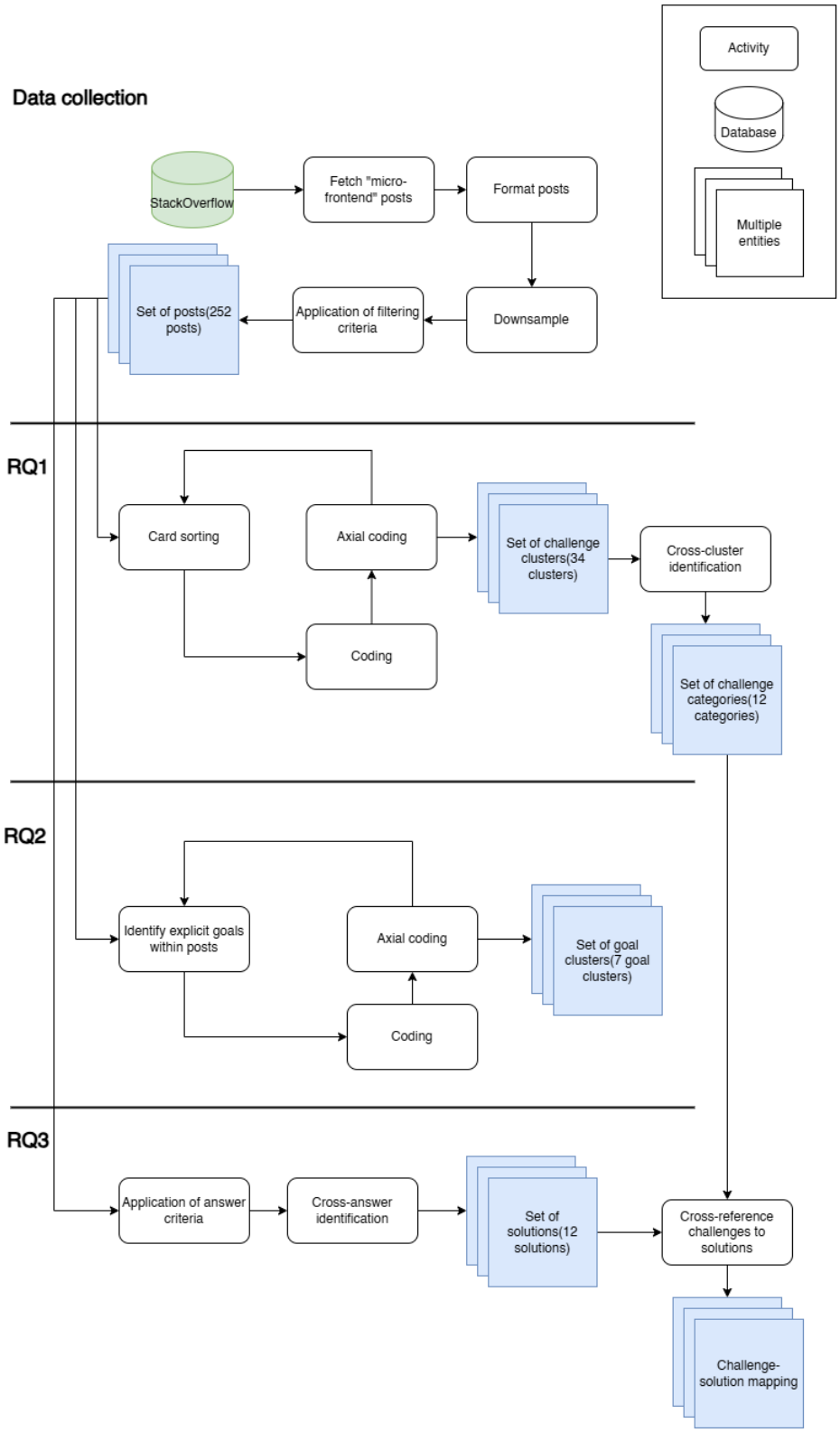

Figure 4.1: Overview of method.

4.1 Data collection

Stack Overflow is a widely used public developer forum and was therefore chosen as our source of data. The forum offers a large volume of questions and answers that reflect real-world development scenarios. These posts are typically created in the context of ongoing implementation, making them valuable for understanding issues in practice. Discussions of interest to us were tagged with "micro-frontend". Both questions and their respective answers were collected. Questions are problem-oriented, laying the foundation in answering RQ1 and RQ2. Meanwhile answers are solution-oriented which contributed to addressing RQ3.

4.1.1 Fetching data

Posts were fetched from StackExchange's **GET /questions** resource. Appropriate parameters were provided for authentication and the "micro-frontend" tag. Start (2016-01-01) and end date (2025-01-15) were also specified to ensure the result was consistent no matter when the data is fetched. Answers to posts were stored separately. They were fetched from **GET /questions/[ids]/answers** then aggregated to a question-answers data structure, as seen in Figure 4.2.

We did not encounter any issues with StackOverflow's API rate-limiting constraints, as we only sent around 14 requests in total, which is well below the limit of 30 requests per second. Further rate limiting was not needed.


```
{⊟
   "tags":[ ⊞ ],
   "owner":{ ⊞ },
   "is_answered":true,
   "view_count":53408,
   "answer_count":3,
   "answers":[ ⊟
      { ⊞ },
      { ⊟
         "owner":{ ⊞ },
         "is_accepted":false,
         "score":0,
         "last_activity_date":1539224945,
         "last_edit_date":1539224945,
         "creation_date":1538793489,
         "answer_id":52675251,
         "question_id":52604189,
         "content_license":"CC BY-SA 4.0",
         "body":"<p>I believe the Angular 6 Elements is ..."
      },
      { ⊞ }
   ],
   "score":39,
   "creation_date":1538467156,
   "question_id":52604189,
   "link":"https://stackoverflow.com/questions/52604189/how-to-embed-an-angular-app-into-another-app",
   "title":"how to embed an angular app into another app?"
}
```


Figure 4.2: This figure shows question data with answers, with some property values shortened and some properties omitted.

While the StackExchange API provided a structured means to collect questions and answers, working with community-generated data introduced complexities - such as

challenges in handling the HTML formatting of questions and answers. We decided to maintain code blocks while stripping non-contextual HTML tags for the purpose of readability while retaining technical details. Initially, we considered removing all code blocks since most questions were described in the text rather than code and the code itself rarely added any valuable context to the problem definition. However, users often used code blocks for other use cases, such as wrapping URLs, ports, and error messages; which may provide important context.

The inclusion of code blocks made the text harder to read in a traditional spreadsheet format due to their length. To address this, we switched from Google Spreadsheets to Notion. Notion offers the ability to display data in a table and a list view, with the latter allowing each row to be displayed as a full-page entry. This enabled us to maintain the full length of the questions and answers while remaining readable.

Data collection through StackExchange's API required authentication. An automated script was used to query the API, fetching both questions and answers. The data was then automatically formatted, from an HTML format to readable text. The outcome was a CSV file containing 681 posts, related answers, and metadata such as tags, creation date, and view count.

4.2 Data preprocessing

Data collection resulted in 681 posts. The data, as fetched, introduced a few issues:

- The text that constitutes a post is written in HTML-like markup.
- Grounded theory excels from a vast and varied dataset, but too much data can be difficult to work with.
- Not all posts were relevant, as posts could be submitted without moderation.

Figure 4.3 is a visual representation of our preprocessing that describes how we mitigated the aforementioned issues. Each step is explained in further detail in the next sections.

Figure 4.3: Overview of preprocessing.

4.2.1 Formatting data

Both the question and the answer body were returned in HTML according to how they were displayed on StackOverflow. Since we read the data in a spreadsheet format over HTML, we applied transformations to achieve readable text. Removing

HTML tags was only done in text that was not within a code block. If a question contained a code block that referenced HTML then the HTML was kept. Escaped HTML characters were also unescaped to improve readability. Formatting can be seen in Figure 4.4.

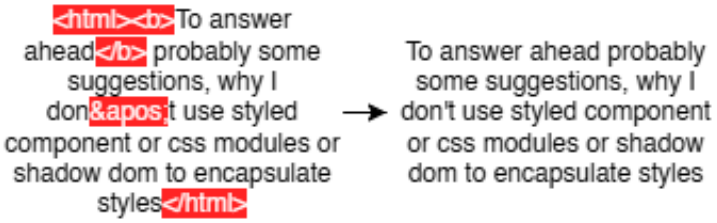

Figure 4.4: Formatting from HTML-like text to readable text.

4.2.2 Establishing relevancy criteria

Relying solely on tags set by users introduced noise in the dataset, as users had mistakenly applied tags due to misunderstanding or misclassification. In other cases, posts did not properly align with our definition of micro-frontends. Community-generated metadata, such as tags, provided accessibility to large datasets for research, however, it introduced noise that requires manual filtering or more sophisticated automated techniques.

Data collection was carried out under the assumption that a post tagged with "micro-frontend" was relevant to micro-frontends and more importantly, relevant to our research questions. We considered posts relevant if they revolved around an aspect of micro-frontends ranging from technical bug-oriented questions to abstract architectural questions. These were specifically useful when answering RQ1, adoption challenges, and RQ2, problems that micro-frontends solve.

Without established criteria regarding relevancy, we started to iteratively label posts "relevant" or "irrelevant" based on perceived relevance according to us. Each of us received 30 of the same posts. We then labeled and compared labels to find where our opinions differed. We considered this a practical iterative exercise where we could compare results, find the differences, establish relevancy criteria, then re-run the exercise with a hopefully greater overlap between our results.

The first batch of 30 posts had inter-rate reliability $k = 0.3647$, calculated from Cohen's Kappa [23]. Before the second batch we established a solid relevancy criteria to boost agreement. The next batch of 30 posts had a satisfactory inter-rater reliability of $k = 0.7000$.

After achieving satisfactory inter-rater reliability we established our criteria for relevance including:

- **Exclusion of Low-Quality Posts:** Posts deemed low-quality, particularly those lacking clarity or requiring additional context to be comprehensible, were excluded from the analysis.
- **Inclusion of Micro-frontend-Like Approaches:** Posts discussing architectures resembling micro-frontends, such as those utilizing iframes or web components, were included. The use of a specific micro-frontend framework was not a prerequisite for inclusion.
- **Inclusion of Diverse Micro-frontend Structures:** Posts addressing both horizontal and vertical split micro-frontend architectures were included to capture a comprehensive range of implementation strategies.

4.2.3 Limiting the dataset

As previously mentioned, grounded theory excels from a vast and varied dataset but too much data could make grounded theory exercises such as constant comparison impractical. There was also value in keeping the dataset within our memory during analysis. It was less time-consuming, there is no need to "rediscover" the dataset. More importantly, we could make purposeful connections between the data as we were familiar with the dataset. In addition, this procedure ensured that our dataset remained manageable.

We considered it reasonable to analyze a dataset comprising 100-300 posts. To reduce the size of our dataset we leveraged unbiased downsampling. Downsampling was performed while ensuring that the sample was representative of the population. The sample size of 247 was determined with a confidence interval of 95% and 5% error margin using the Sample Size Formula for Proportions adjusted for finite population correction [24].

Based on prior experience assessing inter-rater reliability, around 10% to 20% of posts were deemed irrelevant. To address this while ensuring a representative sample, we downsampled to 300 posts, incorporating a margin to account for potential irrelevance.

Random sampling was achieved by adding a new column containing a random generated number between 0.0 and 1.0. The data were then sorted in ascending order based on that column, and posts were omitted until 300 remained. Random sampling removed any potential selection bias, making sure that the downsampled data were not overrepresented by any specific subset of data, such as new or old questions.

4.2.4 Excluding irrelevant posts

Starting from our downsampled dataset of 300 posts, the task of labeling each post was divided evenly. We found that 252 posts of 300 posts were relevant according

to the aforementioned criteria.

Furthermore, the need for defining micro-frontends and manual filtering raises questions about the discrepancies of technical concepts. Our observation suggested that micro-frontends was a concept with varying definitions across different practitioners. This indicated that some discussions relevant to micro-frontend adoption challenges were omitted simply because they were tagged differently or lacked any explicit mention of "micro-frontends".

4.2.5 Data overview

This section provides a concise summary of both datasets' characteristics, as outlined in Table 4.1. These metrics illuminate the datasets' scale, engagement, and differences, a basis for the analysis that follows.

Table 4.1: Comparison of properties between full dataset and downsampled dataset.

Metric	**Full Dataset**	**Downsampled Dataset**	**Notes**
Number of Questions	681	252	Total unique questions
Average Number of Answers	0.78	0.88	Mean answers per question
Average Views	1690.61	1795.67	Mean views per question
Median Question Age (days)	917.5	876	Median days since posting
% Questions with Accepted Answer	16.15%	18.65%	Percentage with accepted answers
Average Question Length (words)	1730.56	1643.48	Mean word count
Top 5 Tags (by frequency)	angular(38.6%), reactjs(33.2%), webpack-module-federation(27%), webpack(19.8%), javascript(16.2%)	angular(41.3%), reactjs(34.5%), webpack-module-federation(31.3%), webpack(21.8%), javascript(17.5%)	Most common tags (excluding "micro-frontend"). Frequency is how many posts of all posts the tag was present
Sample Size Ratio	100%	37%	Proportion of full dataset

4.3 Theory from data

This section details the grounded theory approach applied to analyze the dataset derived from StackOverflow. The quotes were extracted from data where each quote was placed on a separate card. We then leveraged a card sorting exercise which served as open coding, breaking down data into concepts, and axial coding, organizing the concepts into categories. The following sections detail how community discussions were transformed into valuable insights, a foundation in answering our research questions.

4.3.1 Extracting quotes

With a list of relevant posts, we started reading each post in the search for short quotes consisting of around 1-2 sentences that summarized the nature of the post. A post could give rise to multiple quotes. The author might have described various questions and concepts that themselves deserved a separate quote.

Each quote was then extracted and represented by a card in Miro, an online collaborative platform that served as a board or whiteboard [25]. From 252 posts, we found 315 quotes, resulting in 315 cards in our Miro board. In addition to the text containing the quote itself, each card had a link that redirected to the post where the quote was extracted. The link was considered metadata and did not affect the grounded theory process. It was plainly a quick reference in case context was needed for the quote.

4.3.2 Card sorting

Card sorting is an exercise that combines grounded theory's open coding and axial coding. The exercise started by moving cards together that were conceptually similar, overlapping themes or attributes, to form clusters. It was an iterative approach; clusters were meant to be moved, merged, and deleted. Defining clusters was relatively difficult, as a card could not be duplicated or belong to multiple clusters. In addition, a cluster had to be defined in such a way that it consisted of more than one card, but not all cards. The choice of clusters, which required creativity, became more apparent with more iterations.

The level of abstraction also had to be considered. If clusters had too strong a resemblance to individual cards, then clusters were likely to be plentiful but difficult to interpret. While raising the level of abstraction yielded clusters with more cards, labeling them became increasingly difficult. A cluster such as "cross contamination" could encompass cards from many topics such as styling, components, and routing. On the other hand, a cluster specific to styling could be too narrow and therefore irrelevant to our research questions. "Styling is difficult", while potentially true, could be regarded as a trivial statement.

Open coding in card sorting
Working with clusters, especially in a collaborative environment, promotes either

naming clusters or describing them. As clusters started to emerge, conversations naturally ensued. Our informal discussions required us to reference specific clusters, prompting us to assign them names, accurate or not. These informal labels or references established a foundation for the cluster's code, serving as the labeling process of open coding.

Axial coding in card sorting
Axial coding is the process of establishing a relationship between codes. In card sorting, clusters are placed near related clusters. Adjacent clusters could form higher-order clusters.

4.3.3 Categories of challenges

Following card sorting of adoption challenges, we defined higher-order clusters, which consisted of two or more clusters, as categories. Figure 4.5 shows a subset of our categories. The green upper nodes indicate categories, whereas the lower white nodes indicate clusters. The clusters belong to the category above it. In answering our research questions, we refer back to these established categories.

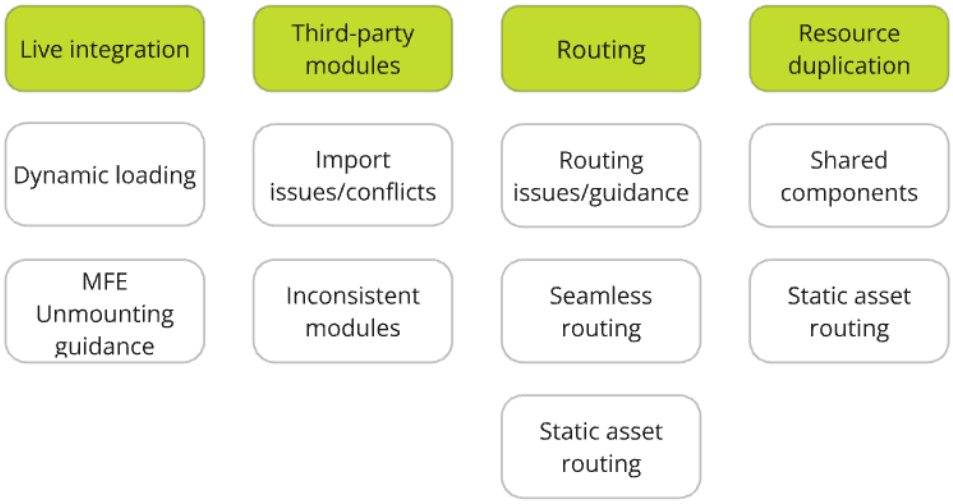

Figure 4.5: A subset of categories. Categories are denoted by a green color and clusters by a white color.

4.3.4 Categories of motivations

To address RQ2: "What specific problems in frontend development do micro-frontends aim to solve?" we identified the developer's motivation to adopting micro-frontends and extracted explicit goal statements from the Stack Overflow posts during the data extraction process. Rather than integrating these with challenges used in the card sorting exercise, we maintained a separate collection of goal-oriented cards. A

smaller quantity of cards meant that this set was sufficiently manageable to analyze without a full iterative card sorting process.

We then manually categorized these quotes based on themes as seen in Figure 4.6. These included technical motivations, such as achieving modular deployment, and strategic motivations, such as improving team autonomy or team independence. This categorization allowed us to map developer intent to broader organizational or architectural goals and provided an empirical foundation for our interpretation of micro-frontend adoption drivers.

Figure 4.6: A subset of card quotes representing adoption motivations. The yellow notes signify higher-level themes or categories derived from the quotes.

4.3.5 Suggested solutions to challenges

Beyond posts our dataset also contained answers that provided possible solutions to the challenges raised by the author. Since these answers were most relevant to the post at hand, including the greater category, another iteration of card sorting was

performed around these established categories. The cards that previously revolved around challenges were rewritten with the intention of highlighting the solution to the same challenge. Since a challenge could have any number of answers, additional cards were created for different possible solutions. Posts without answers were omitted.

Answers were deemed applicable if they had a non-negative rating. Other studies referencing StackOverflow [21, 22], feature additional accepted answer criteria. The reason we chose to omit this was due to the limited number of posts and the quality of many unaccepted answers. We found that a post could have many valuable answers, but because of how StackOverflow was designed, only one could be accepted. In addition, there was no requirement for the author to pick any answer at all as accepted, and therefore, these answers would have been discarded.

The initial dataset consisting of 155 answers, was revised by excluding 8 answers based on the aforementioned criterion of a non-negative rating. Answers did not necessarily present tangible solutions, meaning an answer could contribute with zero or more cards. As a result, 147 solution-oriented cards were created.

Each category had a number of associated solutions where clusters could be defined and named accordingly based on the similarity between solutions. The same properties that linked solutions in one cluster could be seen emerging in other categories belonging to completely different challenges. To capture these widely accepted solutions we reused the same cluster of solutions between categories. A subset of clusters can be seen in Figure 4.7. The result was a list of proposed solutions based on clusters that were related to one or more categories.

Figure 4.7: A subset of solution clusters.

5

Results

This section presents the findings derived from the data collection and analysis process. The results are structured based on the research questions with information gathered from the grounded theory technique.

5.1 RQ1: What are the key challenges developers face when adopting micro-frontends?

A visual overview of the themes related to **RQ1** can be found in Figure 5.1 at the end of this section. Our identified challenges are categorized into four major themes: Integration, Migration, Modularization and Infrastructure. Each of these categories include specific issues that developers encounter during the adoption and implementation of micro-frontends.

Additionally, we differentiate between two types of challenges:

1. **Technical challenges** (represented in light green), which include hands-on coding and implementation issues such as managing dependencies, handling state, and optimizing performance. These require direct intervention, debugging, and potential refactoring.

2. **Theoretical challenges** (represented in dark green), which focuses on the broader adoption concerns, such as the lack of documentation or knowledge, the need for guidance in choosing the right frameworks.

5.1.1 Integration challenges

Integration challenges proved to be one of the biggest obstacles in micro-frontend adoption. While the micro-frontends approach aims to enable independence between teams, they often lead to unexpected conflicts when different micro-frontends are combined within a single application.

5.1.1.1 Integration conflicts

Framework integration
A key issue in integration conflicts is framework integration, where teams attempt to embed Angular, React or Vue micro-frontends within the same application. Since each framework comes with its own rendering mechanisms, dependency management strategies, and state-handling techniques, combining them within a single host application leads to increased complexity. Therefore, developers often struggle with dependency conflicts, duplicate libraries being loaded, and ensuring smooth communication between frameworks.

> *Vite-Plugin-Federation can't import react components with hooks. Host website will get error when the remote components have hooks.*
>
> stackoverflow.com/q/73843683

> *How to setup entire architecture so that angular app can host components developed in react in its environment without loosing the essence of react core features.*
>
> stackoverflow.com/q/77949882

Cross-contamination
A common issue where global styles, JavaScript variables, or event listeners interfere across micro-frontends. Independent micro-frontends may use conflicting dependencies, which often leads to visual inconsistencies, CSS overrides both unintentional and intentional, or unintended behavior when micro-frontends share the same DOM.

> *Why is that happening? How come the encapsulation in one parent element changes the behaviour?*
>
> stackoverflow.com/q/77946622

> *When two applications are mounted simultaneously on the view the one who loads last overrides the i18next instance and thus the translations for the first one are never found as they were not loaded on the latter.*
>
> stackoverflow.com/q/62531428

Standalone vs shell behavior
Some micro-frontends are built to run independently, while others assume a shell application handles routing, shared state, or dependencies. Integrating these differing approaches is difficult and introduces additional problems. This specific category of challenge arises when developers want a micro-frontend that can both be run independently and within a micro-frontend architecture.

> *The micro-frontend app has its router to manage the routing, and it is working as expected as a standalone application. When used in combination with the host application, routing does not work correctly.*
>
> stackoverflow.com/q/71800101

Cross-repo integration
Developers may decide to build micro-frontends in different repositories. When micro-frontends are built in separate repositories, integration challenges become even more pronounced, unlike monorepos, where dependencies, configurations, and shared resources can be managed centrally.

> *I've been able to create a monorepo project with several micro frontends without any issues, but I'm struggling to add a micro frontend from a different repo.*
>
> stackoverflow.com/q/72088711

5.1.1.2 Third-party Modules

Handling third-party modules in a micro-frontends setting introduces unique challenges, particularly when multiple micro-frontends depend on the same library but require different versions. These issues often manifest in two major ways: import conflicts and inconsistent module behavior.

Import conflicts
Import conflicts arise when micro-frontends struggle to correctly import third-party libraries due to mismatched dependencies, missing modules, or incompatibilities between different bundlers like Webpack and Vite. Webpack's feature, Module Federation is often used for sharing dependencies across micro-frontends, which further introduces complexities.

> *With the BrowserAnimationsModule or any platform related module like the BrowserModule or NoopAnimationsModule itself imported in my shared module, the host app will not work anymore*
>
> stackoverflow.com/q/67386392

Other common issues include missing module declarations, import paths that are not properly resolved, and cases where remote modules fail to load dynamically. These errors can be particularly problematic when teams attempt to federate libraries like React, Angular, or Vue across different micro-frontends.

> *Cannot find module 'example/exampleApp' or its corresponding type declarations.*
>
> stackoverflow.com/q/78656326

Inconsistent modules
Even when third-party libraries are successfully imported, they may not behave as expected across micro-frontends:

> *I've been successfully able to federate simple and nested components from the remote app to the host, but the components that are based on Ant Design (antd) do not render on the host.*
>
> stackoverflow.com/q/78285660

In this specific case, the developer had misconfigured the remote and host applications, as the antd (Ant Design) packages did not need to be shared on the host application. However, upon correcting the configuration, additional problems emerged, which had to be resolved by importing additional third-party packages. This demonstrates the complexity of handling third-party modules in a micro-frontend architecture, where resolving one issue may introduce new challenges.

5.1.1.3 Live integration

Developers encountered various types of challenges with live integration, mainly on how to properly load and manage micro-frontends at runtime. We identified issues in two key areas: micro-frontend unmounting guidance and dynamic loading.

Micro-frontend unmounting guidance
Some developers struggled with properly unmounting or unrendering micro-frontends when switching between different remote applications or unrendering the remote's container. Improper unmounting may lead to memory leaks, lingering event listeners, or residual UI elements that interfered with the overall user experience.

> *We cannot find a way to unrender MFE1 when its container is unrendered. I tried to call unmountComponentAtNode inside the useEffect callback of the App container. However since the DOM is already gone at this point so we cannot use this method.*
>
> stackoverflow.com/q/75279848

Dynamic loading
Dynamically loading micro-frontends at runtime proved to be challenge for many developers, as they often lacked the necessary knowledge. The purpose of dynamic loading is to allow micro-frontends to be loaded on demand, rather than bundling them all upfront. However, developers' often unique contexts proved to be the main bottleneck, since they require guidance suited to their specific circumstances, making general solutions insufficient.

> *Is there a way to dynamically load and use an exposed micro frontend component with Native Federation?*
>
> stackoverflow.com/q/79291239

One developer highlighted the need for dynamic loading by aiming to break up their web application to a micro-frontend architecture in order to deploy the UI components which a customer needs or has license for. Since this developer works in a multi-tenant application, this becomes particularly important because it allows each tenant to receive only the UI components relevant to their specific deployment or license. Since not every customer needed all the available microservices, the frontend contained excessive functionality that had to be hidden. However, they found that existing micro-frontend solutions such as Next.js, SystemJS, Piral, and Single-SPA lacked clear guidance on achieving dynamic, multi-tenant UI composition. This specific setup makes general dynamic loading solutions inapplicable and, therefore, requiring tailored guidance.

> *So does anyone know of ways to create a container application which dynamically loads UI components of the deployed microservices backend?*
>
> stackoverflow.com/q/69074425

5.1.1.4 Same-site experience

Lastly, for challenges related to integration, developers found it challenging to ensure a same-site experience, where the combination of different micro-frontends feels like a unified and seamless application rather than disjointed components.

Seamless routing
One of the biggest pain points was seamless routing. Developers wanted navigation between micro-frontends to feel smooth, but some implementations caused full page reloads when switching between modules. This is crucial for the same-site experience, as it disrupts the user experience, making the application feel less cohesive.

> *I was trying to find if there is another framework where navigating between two apps happens without refreshing.*
>
> stackoverflow.com/q/56472071

Seamless loading
Some developers struggled to implement a consistent loading experience across micro-frontends, where existing solutions lacked clear guidance on how to synchronize this behaviour. Lack of solutions for visual consistency, can make micro-frontends feel like isolated fragments rather than parts of a cohesive application.

> *I would be happy to hear some advice on how to achieve one loader animation for microfrontend application including host container app combined with fragments apps. The different applications can include a dedicated loader animation, but when there is a loader for the host container, only one Loader animation should appear.*
>
> stackoverflow.com/q/74521845

5.1.2 Migration challenges

Migration is arguably the most common and crucial aspect of challenges within the adoption of micro-frontends. The challenges we identified in this aspect can be categorized into two main areas: general micro-frontend adoption (including migration from a monolith) and inter-micro-frontend migration. These areas are deeply intertwined, go hand-in-hand and cover many sub-categories that overlap. Therefore, we address them together in this section and present their sub-categories in following sections.

A recurring theme that we found in almost all issues was the lack of structured adoption guidance. Many developers are unsure how to transition from a monolith to micro-frontends in a structured way. While most aimed to adopt established frameworks such as Angular or React, some explored lesser-known frameworks like Blazor.

> *can anyone show me how to use micro-frontends it in a very very basic way ?"*
>
> stackoverflow.com/q/78236488

> *Which is the best approach/framework for micro frontend development?*
>
> stackoverflow.com/q/59659719

Furthermore, migrating from an existing micro-frontend solution to another micro-frontend is a challenge many organizations face as their requirements evolve. Our observation shows that teams adopt a lightweight micro-frontends approach, due to its simplicity and quick implementation. However, soon they find out that this simple solution does not meet their requirements and use-cases, therefore seeking out to adopt a more comprehensive approach.

> *We tried to use the single-spa framework but we have some issues and we are currently found our-self thinking if this is the right approach for us, or should we try a different approach*
>
> stackoverflow.com/q/59451629

5.1.2.1 Lack of documentation

Based on our observations, the lack of documentation is likely the underlying cause of nearly all the issues related to micro-frontends adoption. A recurring patern that emerges across the challenges is insufficient or poorly structured information. Insufficient or poorly structured documentation makes it difficult to understand how to implement micro-frontends properly. Working with micro-frontends is a complex environment on its own, particularly as each presented challenge requires a unique solution due to the many factors involved. Lesser common frameworks are especially affected, due to their limited documentation and fewer resources available.

> *So I want to follow Micro Frontend Architecture for each module. I have searched on internet but did not find anything. So I want to know is it possible to achieve Micro Frontend Architecture for React-Native application. If yes then How ?*
>
> stackoverflow.com/q/72649515

> *I've searched high and low across the Internet and haven't found any useful articles on the issues I'm having. Most of them show a demo of testing a web application that doesn't relate to my scenario.*
>
> stackoverflow.com/q/68705755

5.1.2.2 Choosing the right micro-frontend framework

Developers struggle with choosing the right micro-frontend framework based on their needs. Many are either unaware or overwhelmed of the many available options. The lack of clear documentation on these approaches makes it difficult to decide which solution fits best with their requirements.

> *We have many microfrontend frameworks like single-spa, Piral, Mosaic, and Podium. I need a clear picture of how to implement microfrontends using any framework.*
>
> stackoverflow.com/q/59659719

5.1.2.3 Choosing the right micro-frontend structure

The different approaches in structuring micro-frontends can be overwhelming when attempting to adopt a micro-frontend architecture. A common concern is how to structure micro-frontend repositories and whether each micro-frontend should be fully isolated. For instance, some developers seek to have separate repositories for each micro-frontend module, ideally managed by different teams, while ensuring they can run stand-alone and within a micro-frontend architecture.

> *I want to separate repositories for each micro application. Each application should be managed by a different team and should work as a standalone, but also be part of a larger system. How can I achieve this?*
>
> stackoverflow.com/q/56769253

5.1.3 Modularization challenges

A key goal in micro-frontends is achieving independent development and deployment. However, this independence also introduces complexities in managing state and data, resource duplication, and difficulty in maintaining a smooth routing experience across different modules. While Modularization itself is not a challenge, it

introduces various of challenges developers must address.

5.1.3.1 Data & State management

Managing data and state across micro-frontend modules is one of the major challenges that emerge when modularizing your software. We have split this challenge into two subareas:

Data sharing & communication
Micro-frontends need to properly share data and state with each other when dealing with modular components. A recurring pattern observed in this study, is that each micro-frontend setup is unique and complex, and therefore requires different strategies. There's no universal solution, and our findings indicate that many developers struggle to find the right approach this challenge.

> *What is the best approach to communicate between each other other than using local storage.*
>
> stackoverflow.com/q/76792092

> *i need a common pubsub system (managed by my shell appliaction) to communicate between different micro apps. How to achieve this?*
>
> stackoverflow.com/q/60399944

> *I am tring to share state between MonoRepo and MicroFrontEnd application (hosted on different domain)*
>
> stackoverflow.com/q/78157540

> *How do you share state in a micro-frontend scenario?)*
>
> stackoverflow.com/q/60548251

Downstream communcation Our observation suggests a challenge that often emerges is how to expose components from a shell/host application to child micro-frontends. The aim is to use the parent-component in the child application and allow it to expose it back to the parent.

> *My usecase is I want to expose a component from my host/shell application to my module federated child application*
>
> stackoverflow.com/q/78656326

> *I want to expose a higher order component from parent to child, which can be used in child app and expose back to parent.*
>
> stackoverflow.com/q/78521006

5.1.3.2 Resource duplication

Resource duplication is the result of modularization and is a concern many developers try to address. Since each micro-frontend operates independently, multiple instances of the same libraries, assets, or code may exist.

Shared components
Creating shared components is a common approach to mitigate the duplication of resources, by centralizing reusable UI elements, utilities and other logic. However, despite being common, it appears that developers face difficulties implementing this solution.

> *How to implement a microfrontends architecture that reduces duplication of frontend work but still provides a high quality experience for native clients?*
>
> stackoverflow.com/q/61958407

> *Can i just pack my whole Angular app as Web Component (e.g Angular Elements) and integrate them in my container app?*
>
> stackoverflow.com/q/72062718

Static asset routing
Managing static assets in micro-frontend architectures turns out to be a significant challenge. Developers encounter various problems, such as assets failing to load in one or multiple micro-frontends, incorrect paths and inconsistencies in different environments. If a frontend application is designed to run independently but also in a micro-frontends architecture, then conflicts may arise in how assets are referenced and loaded.

> *When we run the MFE alone, the image is loading when we run the host/shell app, the image is not loading for MFE since MFE is running in a different port(5000) and Shell is running in a different port(4200) and the app is trying to access the image from Shell's asset folder when we run the shell app.*
>
> stackoverflow.com/q/69192229

> *All our MFE are built with webpack and created with CRA. We now want to migrate all of them to CRACO. Everything works, except for the static assets, that take the wrong URL.*
>
> stackoverflow.com/q/74769643

5.1.4 Infrastructure challenges

Managing infrastructure is a challenge even in in traditional monolithic architectures, where tools and workflows are relatively standardized. A micro-frontends

architecture further complicates these tasks, adding additional layers of complexity. Even a simple task in a monolithic approach, may require more sophisticated setup, maintenance and coordination efforts in a micro-frontend approach.

5.1.4.1 Development barriers

The adoption of micro-frontends introduces several challenges that can slow down development, if one is used to the traditional monolithic frontends workflow.

Debugging
Debugging in a micro-frontend setup can be difficult and frustrating. Since different modules run independently, sometimes even in different repositories or frameworks, investigating issues means tracking problems across multiple sources. Inconsistent stack traces, difficulty inspecting shared state, and struggling to debug runtime interactions between micro-frontends are some of the challenges a developer may face. Debugging tools designed for monolithic applications don't always work well in this environment, making it harder to pinpoint issues.

> *How to debug a micro frontend app using module federation dynamic system host pattern in vscode. I am able to debug the host application but not the remote one.*
>
> stackoverflow.com/q/71104719

> *Also, there are no errors or warning on the web browser's developer console.*
>
> stackoverflow.com/q/78285660

Hot module reloading
Hot reloading allows developers to see code changes without restarting the app, and is considered essential when working with modern frontend development. However, in an micro-frontend architecture, setting up hot reloading is not a straightforward process. While in a monolithic application, changes are made seamlessly across a single codebase, micro-frontends consist of multiple independently deployed modules. This setup makes it difficult to ensure updates to a remote module triggers a live reload in the host application.

> *The problem I am facing is that once I change a remote module's code - the app (host that I am looking at) does not live-reload which makes developer experience not as comfortable. Is there a way to let the host know that a module has been changed and the reload should occur?*
>
> stackoverflow.com/q/64919434

> *The issue is that when I change any exposed component, HMR does not work. When I refresh the page I get hydration errors because*

> *the server still has the previous version*
>
> stackoverflow.com/q/79310123

Testing
Testing micro-frontends requires a different approach due to the fragmented nature of the architecture. Like previous ***Development Barrier*** challenges, testing is a challenging area that gets further complicated in a micro-frontend architecture.

> *Is there a way I can mock the auth instance in Cypress? I would like to call the methods the auth instance exposes with mock data.*
>
> stackoverflow.com/q/74112677

> *How can I focus on testing the end result instead of the functionality of the authentication HOC just so I can get the component to render and for my tests to execute?*
>
> stackoverflow.com/q/68705755

5.1.4.2 Independent deployment

Independent deployment is a goal developers have when adopting micro-frontends, allowing teams can deploy updates independently without redeploying the entire application and reduce resource consumption and build times. A common challenge in optimizing the deployment process is avoiding redeployment of the entire application when only a single module has changed. Ideally, deployments should be modular, ensuring that only the modified micro-frontend is rebuilt and deployed.

> *Now for a piece of code change in one module, the whole project needs to be deployed. I want to break my modules into different projects for separate deployment.*
>
> stackoverflow.com/q/74944837

> *I want to optimize the CI/CD pipeline in GitLab so that only the specific application (or its associated library) that has been modified is rebuilt, rather than rebuilding all applications in the workspace.*
>
> stackoverflow.com/q/79193293

Artifact management
Managing build artifacts can be relatively straightforward in a monolithic application, where a single build process generates all assets. However, micro-frontends produce independent artifacts that may need to be properly managed.

> *When a build completes, it generates an artifact that is then sent to the app service. My question is: how do I access those artifacts (from each of my remotes) in the release pipeline?*

> stackoverflow.com/q/69171481

> *Is it possible to somehow deploy to package artifacts somewhere, and then automatically update it in all the projects?*
>
> stackoverflow.com/q/64899764

5.1.4.3 Availability and Resilience

Availability and resilience are critical concerns in a micro-frontends architecture, as failures in individual micro-frontends should ideally not affect the entire system, assuming that's the micro-frontend approach the organization has taken. Developers face challenges in how to ensure high availability in a distributed frontend environment.

> *I think that each modules in module federation should be independent, I tried to make the app2 unavailable as I didn't start it. Therefore I got error when I run the app1. So, can the module federations run independently each other? If not, what's the real difference as normal library dependencies of monolith front end instead of this sophisticated micro frontend, that I assumed it should be able to work independently like microservices?*
>
> stackoverflow.com/q/70796589

Error handling
Another major concern we observed, is implementing error isolation. Achieving this in practice is complex, especially when micro-frontends share resources such as authentication tokens, global state, or UI components.

> *MFE-food calls MFE-cart. When MFE-cart is down, there is an impact on MFE-food, my expectation is MFE should run properly.*
>
> stackoverflow.com/q/72638378

Fallback strategies
Ideally, when a micro-frontend fails, the system should still function, even if with limited capabilities. However, a challenge that arises is how to approach with implementing fallback strategies, such as displaying alternative UI components or providing cached data when a service is down.

> *I wasn't expecting this behavior. I was expecting the host application to render my React application immediatly while Module Federation is trying to load all the remotes and show some kind of managed Spinner from my app.*
>
> stackoverflow.com/q/72638378

5.1.4.4 Versioning

Keeping track of versions in a micro-frontend system is difficult. Unlike monolithic applications, where everything runs on the same version, each micro-frontend can be updated independently, which can lead to version conflicts and unexpected issues. The most common theme we observed in the dataset was confusion or uncertainty about how to manage differing versions across independently developed micro-frontends.

> *In a micro front end architecture can the main applications remain in their own angular versions and consume micro front end apps from higher or lower angular versions?*
>
> stackoverflow.com/q/74217409

Dependency mismatches
Probably one of the most common versioning challenges in micro-frontends arises when different micro-frontends rely on different versions of the same dependency.

Module Federation allows micro-frontends to share dependencies dynamically at runtime. However, it does not always resolve versioning conflicts automatically. If different micro-frontends requires incompatible versions of a shared package, developers must manually resolve the issues, which can be complex and time-consuming.

> *The challenge is to configure Module Federation to handle these different versions without breaking the application or impacting performance.*
>
> stackoverflow.com/q/78780941

Breaking changes
Since micro-frontends are independently developed, one team's changes can accidentally break another team's implementation. This is especially problematic when a shared component or API undergoes a major update.

> *The question is how do we perform an update of Angular in all the SPAs? Application is huge enterprise project and we can't just do 40+ PR's and merge/deploy everything simultaneously for obvious reasons.*
>
> stackoverflow.com/q/77661080

Compatibility management
Another recurring issue is the lack of practical strategies for managing multiple versions of the same micro-frontend or shared component.

> *What is the best approach for Versioning the integrated apps?*
>
> stackoverflow.com/q/72062718

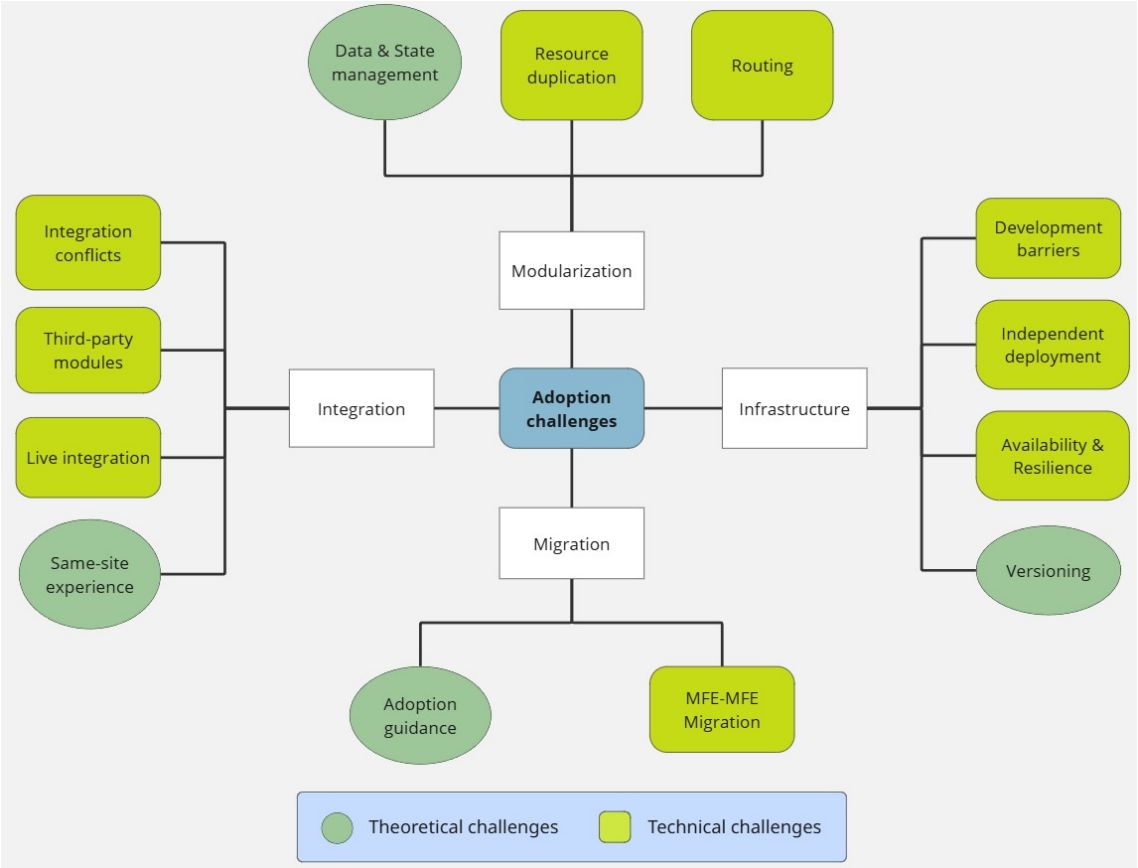

Figure 5.1: Identified adoption challenges of Micro-frontends.

5.2 RQ2: What specific problems in frontend development do micro-frontends aim to solve?

A visual overview of the themes related to **RQ2** can be found in Figure 5.2 at the end of this section. During our investigation of adoption challenges, we also identified the key goals driving the adoption of micro-frontends. These goals were often found within the same discussions as the challenges, including the motivations for adopting this architecture.

Like in the results of **RQ1**, we differentiate between two types of goals:

1. **Technical goals** (represented in light green), which focus on solving practical, hands-on challenges in frontend development. These include issues such as improving modularity, enabling independent deployments, optimizing performance, and handling scalability.

2. **Strategic goals** (represented in dark green), which aims on organizational improvement such as increasing team autonomy and technological agnosticism; goals that encourage better collaboration, management, and long-term sustainability in the development process.

While this simple dissection of goal types helps to highlight the different types of issues that micro-frontends aim to address, in practice, many of them overlap. For example, independent deployment improves technical efficiency while also enabling faster delivery cycles, which could be considered a strategic advantage.

5.2.1 Modularity

Our observation suggests that modularity is the core motivation driving the adoption of micro-frontends. It is the foundational goal that enables all the other benefits. Modularization is practice of dividing up a monolithic application into smaller modules. This practice supports team autonomy by allowing each team to work on a specific part of a larger frontend, without impacting the entire application.

5.2.2 Organizational scalability

As frontend applications grow, managing them becomes increasingly difficult. Micro-frontends provide a way to scale development efforts by dividing a large front-end into smaller manageable micro-frontend. Each micro-frontend can then be developed, maintained and scaled independently, essentially distributing the workload across multiple teams.

> *My angular project is evolving every day, the files are multiplying, the project size is growing. I want to divide the project into several parts. I want to develop the logic of microservice.*
>
> stackoverflow.com/q/68090387

5.2.3 Independent deployment

One of the primary reasons that make micro-frontends a popular adoption choice is its ability to deploy modules independently. Monolithic systems require entire applications to be redeployed on changes. Micro-frontends remove this constraint, allowing teams to update individual modules without redeploying the entire application.

> *Now for a piece of code changes in one module, the whole project needs to be deployed. I want to break my modules in different projects for separate deployment.*
>
> stackoverflow.com/q/74944837

> *I want to optimize the CI/CD pipeline in GitLab so that only the specific application (or its associated library) that has been modified is rebuilt, rather than rebuilding all applications in the workspace.*
>
> stackoverflow.com/q/79193293

5.2.3.1 Performance improvements

Independent deployment can lead to performance improvements. By allowing teams to roll out deployments to specific parts of an application, build and deployment times can be significantly reduced, especially in large-scale systems.

Additionally, micro-frontend support techniques such as lazy/dynamic loading, letting users to only download the parts of the front-ends they need, which theoretically may result in potential performance benefits such as lower memory usage and improved load times.

> *Main aim is to have performant app, and no new deployment in old app should be needed if new app has code update.*
>
> stackoverflow.com/q/61671213

5.2.4 Team independence

Since micro-frontends encourage modularity, organizations can assign independent teams to own specific parts of the application. Each team can then manage their own micro-app with their own repositories, pipelines, and deployment. This autonomy allows teams to make decisions independently and make deployments without being blocked by other teams.

> *We have an application that been managed across multiple teams, we would like to create the flexibility and isolation between the apps by using micro frontend approach, each application will have its own store.*
>
> stackoverflow.com/q/56769253

> *Basically, I want to be able to: Separate Repositories for each micro application. Each application would be managed by a different team.*
>
> stackoverflow.com/q/56769253

5.2.4.1 Tech agnosticism

Since each team has full ownership of a specific part of an application in a micro-frontend architecture, they are free to choose the frameworks or technologies that is best suited for their specific needs.

> *I have more than 5 apps based on angular, react and vue and want to display all 5 apps on a single UI.*
>
> stackoverflow.com/q/78532683

5.2.5 Increased availability & resilience

Failures in monolithic applications can crash the entire system. Micro-frontends, however, is an architecture that promotes fault isolation, meaning that issues in one part of an application do not bring down the whole application.

> *But for lack of planning, this system becomes a single and complicated project, being difficult to install or update. The proposal would then be to modulate it, making parallel development, easy, and effective, since a change in an independent module would not impact the update of the entire system.*
>
> stackoverflow.com/q/74436994

> *Our requirements are the usual requirements for micro-frontend: Each application can be upgraded in production without the downtime and without interfering with the other applications.*
>
> stackoverflow.com/q/59451629

5.2.6 Dynamic loading

Another benefit raised by developers, is the need for lazy-loading or dynamically loading components. Teams want to load only the components or features relevant to specific users or contexts, for instance, based on a condition, such as licensing or access rights.

> *We want to break up the web application, so that we can deploy only those UI components which a customer needs or for which he has a license.*
>
> stackoverflow.com/q/69074425

5.3 RQ3: What are the commonly suggested solutions to these challenges?

In studying adoption challenges it is equally important to mention their proposed solutions. The nature of solutions ranging from trivial to needlessly complex will help with highlighting the maturity of micro-frontends, guiding a potential adopter in making qualitative decisions in regards to micro-frontend migrations. Table 5.1 represents identified solutions in relation to the categories of adoption challenges outlined in RQ1.

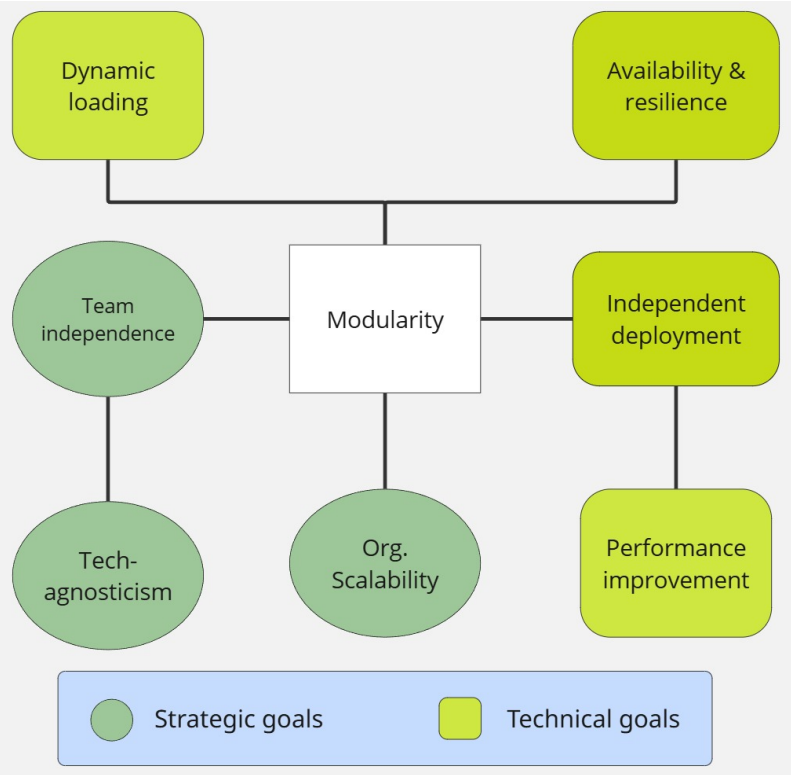

Figure 5.2: Identified adoption goals of Micro-frontends, split up in 2 categories; Strategic and Technical goals.

Table 5.1: The identified challenge-solution mapping.

Solution	**Applicable to (challenge categories)**
Follow configuration specifics	Routing, Development barriers, Third-party modules, Data & State Management
Share instance	Third-party modules, Resource duplication, Integration conflicts, Data & State Management
Prefix names	Third-party modules, Integration conflicts
Separate NPM package	Independent deployment, Resource duplication
Force upgrade	Independent deployment, Data & State Management
Import once	Third-party modules
Promise-based routing	Routing
Runtime variables	Independent deployment
Browser history between micro-frontends	Routing
Same-domain communication	Data & State Management
Unfederated communication	Data & State Management
Catch remote errors in shell	Availability & Resilience

There were a few outliers in our dataset consisting of challenges without viable solutions. The less viable solutions were often accompanied by a disproportionate

amount of effort such as framework migration. Most solutions in our dataset, a subset seen in the challenge-solutions table, could be found in multiple clusters which implies that they were widely suggested and therefore viable solutions. Due to the limited amount of outliers they were classified as noise and a subset is seen in the following:

- **Solutions related to framework pioneering**: Leveraging micro-frontends in frameworks not considered micro-frontend-friendly such as Quasar or Blazor.
- **Solutions related to legacy app communication**: A shell application can be used to wrap legacy applications together with new applications. To provide a cohesive experience apps often need to communicate which necessitates the modernization of legacy applications.

Follow configuration specifics
Much of the micro-frontend integration is delegated to key-value configuration files such as webpack.config.js. These configuration files have a plethora of optional properties to be able to serve both monolith and micro-frontend projects. In addition, input validation and type checking is often an oversight, allowing misconfigured projects to run and exhibit problems in other parts of the application. It is therefore important for developers to familiarize themselves with existing up-to-date documentation and provide configuration without deviating from established patterns.

> *In my case it was requiredVersion property. I had set stiff versions i.e. without '^' which somehow caused a problem.*
>
> stackoverflow.com/q/78656326

> *Fixed my own problem by changing the key version to requiredVersion.*
>
> stackoverflow.com/q/71197822

Share instance
Certain third-party modules hook into the DOM/window/browser, therefore imposing an expectation of "one browser one app". Whether the module is defining browser-wide singletons or registering event handlers, displaying additional apps will overwrite the previous initialization of the module causing unexpected behavior. The suggested solution is to initialize the module once and share the same instance between apps.

> *You'll need to add a field to the "shared" object in the ModuleFederationPlugin*
>
> stackoverflow.com/q/73038460

> *It is better that I18next will be initialized at the shell level with the shell namespaces, and each internal spa will add its namespaces to the shared instance.*
>
> stackoverflow.com/q/62531428

Prefix names

In the DOM and browser there are certain naming conventions such that CSS ids and CSS classes must have unique names. Naming conflicts are not specific to just CSS, HTML elements such as input elements could also be affected among others. These issues are often raised during micro-frontend integration as a single page can contain components from many micro-frontends with overlapping names. An often suggested solution is to prefix all names by prepending the name of the current micro-frontend.

> *One idea is to add a prefix selector to your child micro-frontend (my-unique-child-prefix).*
>
> stackoverflow.com/q/78174513

> *I had the same error and it was caused by name correlation. In my case: the same module name and id attribute name of the html element*
>
> stackoverflow.com/q/76773084

Separate NPM package

There are a few considerations to be made working with a micro-frontend architecture. Between micro-frontends code and assets should not be duplicated, which is a general principle beyond just micro-frontends. More importantly, at least from the user's perspective, is that the presentation is a cohesive experience. Components, color scheme, and design conventions should be consistent across micro-frontends. To maintain independence between teams these overarching artefacts are often suggested to be put in an NPM package that can be updated separately from each app.

> *You can create a theming package for your micro-frontend apps.*
>
> stackoverflow.com/q/75173111

> *you can separate your moduels to pacakge more like a NPM package and you can even reuse it to different projects. Each libraries act and maintained as a separate app*
>
> stackoverflow.com/q/74944837

Force upgrade

Micro-frontends is at its core tech agnostic, meaning many technologies can be integrated to form one final product. But there are still certain limitations, frameworks

might not easily be integrated into a shell application, some techniques useful in managing micro-frontends such as unmounting might be hard to accomplish with the current technology. A newer version of a module could have already solved these challenges prompting the developer to upgrade.

> *Upgrade to React 18 at the end since it is too annoying to handle this in React 17. We do it by eventListener. Since React 18, we can access createdRoot*
>
> stackoverflow.com/q/75279848

> *Webpack 5 is not yet supported. Will be with Nuxt3. As of moduleFederationPlugin, I'm not sure what this is.*
>
> stackoverflow.com/q/68839439

Import once

Some modules are expected to be initialized once in the DOM. Failing to conform to this expectation can cause the application to stop running or exhibit problems that are difficult to debug. The suggested solutions to challenges like these often revolve around importing a module once. However, keeping track of imports breaks independence between teams as teams must be explicitly instructed not to import some module. As with other solutions, trade-offs must be carefully considered.

> *BrowserModule is being imported into too many modules and libraries. It's only necessary for BrowserModule to be imported into the shell application.*
>
> stackoverflow.com/q/69484928

Promise-based routing

Certain routes can have established criteria that must be fulfilled to grant the user access. Routes that belong to some child app can be lazy loaded first when some criteria is fulfilled. The suggested solutions mention putting these routes behind asynchronous functions, also called promises, that must be evaluated for the route to be loaded and registered in the router.

> *You need to use promise-based dynamic remotes.*
>
> stackoverflow.com/q/68123199

> *the function that you put in activeWhen can have complex logic. In this function, you may check whether the user is authenticated*
>
> stackoverflow.com/q/76665872

Runtime variables

Depending on the micro-frontend app it should be reusable in such a way that it can be built once and deployed with different arguments. A perfect example is

deploying an app as a standalone application compared to a child app under a shell application. A solution is to build the app, whether its intended as standalone or child app, and provide environment variables to leverage different contexts.

> *It means, during the build time, if there are no such Environment Variables, the bundle will include the strings'MY_CUSTOM_ENV_VAR'and'MY_CUSTOM_ENV_VAR'. Now, to replace those variables on container starts, we are adding a substitute to the RUN command:*
>
> stackoverflow.com/q/70182774

Browser history between micro-frontends
Routing between micro-frontend apps is notably difficult to implement as navigation must be specified between in-app navigation and cross-app navigation. The suggested solution is to create a reference to the browser's history in the shell app and share the instance between child apps which would navigate relative from the shell instead of the current child app.

> *Then similarly both children apps would import the HistoryRouter and pass the passed history object as a prop so they all reference the same history.*
>
> stackoverflow.com/q/76814429

Same-domain communication
Micro-frontend apps often require some sort of shared state. There are various options, assuming same-domain deployment then cookies and local storage are an option, assuming a federated application then modules and instances can be shared between apps, among others. Since there are many available solutions the selected approach is often up to personal preference.

> *Data sharing management is done using mix of cookies/local storage and redux. Sharing of data between child and parent app is using Post Robot https://github.com/krakenjs/post-robot.*
>
> stackoverflow.com/q/67817670

Unfederated communication
Micro-frontend apps without federation present further challenges in communication. While federated approaches such as module federation and native federation are staple technologies in the space there are still more rudimentary approaches in production such as iframes. Assuming cross-domain deployment, ruling out cookies and local storage, the suggested approaches for communication then ranges from passing URL parameters to browser's postMessage.

> *You could pass the token you've generated in your main application to the url you're using in the iFrame, for example,*

> *"https://yourdomain.com/?token=$token", so you can use this token in your secondary application to sign in the user*
>
> stackoverflow.com/q/75807570

> *Adding HTTP headers to iframe is discussed here. Looks like it's possible, but also might be some limitations...*
>
> stackoverflow.com/q/70384723

> *Sharing of data between child and parent app is using Post Robot https://github.com/krakenjs/post-robot.*
>
> stackoverflow.com/q/67817670

Catch remote errors in shell

Inherent to micro-frontends much like microservices is availability. In production there are countless scenarios that would benefit from availability measures such as overloaded servers, misconfigured integrations, or a child app that is crashing in the browser. A possible solution is to catch potential errors from a remote application in the shell application. To further improve the user experience a placeholder can be rendered.

> *You can load another module if there is an error during remote module load by using the catch block of promise returned by loadRemoteModule.*
>
> stackoverflow.com/q/70848941

6

Discussion

This chapter presents a discussion of the study findings and addresses the research questions, summarizing key insights derived from the analysis and linking them to the relevant literature. We also aim to contextualize our findings within the academic and industry understanding of micro-frontend adoption.

6.1 Research Questions

RQ1: What are the key challenges developers face when adopting micro-frontends?

RQ2: What specific problems in frontend development do micro-frontends aim to solve?

RQ3: What are the commonly suggested solutions to these challenges?

To address **RQ1**, **RQ2**, and **RQ3**, we collected user posts from Stack Overflow using the platform's public API and applied a grounded theory approach following an iterative card sorting process to analyze the data. While iteratively analyzing posts that described adoption challenges (**RQ1**), we also identified statements indicating the motivations behind adopting micro-frontends (**RQ2**), offering insight into both the problems developers face and the problems they hope to solve with this architecture. Following the analysis of adoption challenges we were able to identify relevant solutions suggested by the StackOverflow community (**RQ3**).

Our observations suggest that the challenges and goals span in multiple dimensions, which we grouped into two overarching aspects for structure and clarity. For adoption challenges, we sort them into **technical** and **theoretical** challenges. Within these aspects, we further categorized the challenges primarily into four broad themes: **Integration, Migration, Modularization** and **Infrastructure**. Similarly, for adoption goals, we identify **technical** and **strategic goals**. Although these help distinguish between various types of challenges or goals, it is important to consider that they are multifaceted. Meaning that a technical goal could also be strategic and vice versa. As such, we advise not to enforce strict labels on the goals, but to

acknowledge their multidimensional nature.

Our findings closely align with previous studies and literature. The multivocal literature review by Peltonen et al. [7], brings up issues and benefits with micro-frontends that mirrors the challenges and goals identified in our study. Among the challenges, they highlight issues such as increased payload size, code duplication, shared dependencies, UX consistency and general architectural complexity. On the benefits side, Peltonen et al. outlines several motivations for adopting micro-frontends such as promoting autonomous teams, achieving deployment independence, improving fault isolation, improved performance, and scalability. The overlap between our data and Peltonen et al.'s paper suggests that developers' challenges and motivations in practice are consistent with those in literature.

Peltonen et al. also identified additional challenges that explicitly did not emerge in our data. Most notably, they discuss the concept of *Islands of knowledge* and according to the authors of the study, is a very costly and time consuming issue. Cross-functional teams working on the same product but different code base, may end up with the same implementation over and over again, if they do not communicate their work properly. Similarly, some benefits that Peltonen et al. highlighted, which did not appear in our dataset include improved testability and onboarding speed. One possible reason for this may be the nature of the data: developer forum posts tend to focus on immediate technical and architectural concerns [20], whereas challenges or benefits like onboarding improvements may become more apparent over time or emerge in organizational contexts not often discussed on platforms like Stack Overflow. This also applies to *Islands of knowledge*, which is a type of concern that arises in the more mature phases of implementation.

Building on this, Antunes et al. [15] conducted an empirical case study to explore how the challenges and benefits highlighted by Peltonen et al. holds up in practice. Using a focus group, developers engaged in a micro-frontend migration, where they tested and discussed Peltonen et al.'s findings. While Antunes et al.'s results confirm many of the goals and challenges both identified in our study and Peltonen et al.'s paper, participants highlighted some interesting trade-offs and concerns. For instance, while MFE allows for flexibility of choosing technologies, developers need to acknowledge the maintenance costs associated with utilizing multiple technologies. Moreover, Antunes et al.'s study shows micro-frontends to be a deterrent even after hands-on experience. Despite understanding the benefits, developers hesitated to adopt the architecture due to concerns over added complexity.

The identified solutions, while specific to the aforementioned challenges, could be corroborated with existing research. Gaur [17] introduces an overview covering the implementation of a Customer Relationship Management(CRM) system with micro-frontends together with considerations, key features, challenges, and approaches to address them. Among the proposed approaches they mention CSS governance with scopes as a remedy for cross-contamination, similar to the approach of prefixing names found in our dataset. They also mention runtime conflicts and how a micro-frontend might need to update the version of a dependency to avoid conflicts which is

represented by import once and instance sharing in our dataset. Veeri [26] highlights common pitfalls and their proposed solutions in implementing a micro-frontend architecture in React. Veeri addresses user experience fragmentation, which often manifests as inconsistencies in the user interface, by introducing a shared theme between micro-frontends. The same notion of resource sharing is seen in instance sharing and separated NPM package proposed by us. Kurapati [27] proposes a set of best practices in mitigating increased complexity, coordination overhead, and performance issues. Kurapati addresses state management by proposing tailored solutions such as global event buses and shared state libraries like Redux. These solutions are covered by instance sharing, separate NPM package, same-domain and unfederated communication suggested by us. Our findings overlap both with solutions in practical papers, Gaur and Veeri, in addition to literature reviews such as Kurapati.

6.1.1 Take-aways

Based on our findings, there are a few takeaways that may help practitioners who are considering or currently adopting micro-frontends.

Integration is hard and requires early planning. Integration issues were some of the most common problems found in the study, as developers often struggled to combine micro-frontends using different frameworks or configurations. It is worth spending time early in the process, to plan how modules will communicate, how dependencies will be shared and what tools will support this. Furthermore, teams also report to encounter issues with shared dependencies, such as duplicated libraries, mismatched versions or CSS conflicts; all of which can be mitigated by early planning and establishing conventions. The increase in complexity in integration testing and dependency management can be a deterrent for developers [15].

Lack of guidance. The entire micro-frontends adoption process is complex and intricate. Many developers in our dataset expressed uncertainty about how to begin adopting micro-frontends, and some failed to find any resources on how to deal with many parts of the adoption process. Some pointed out that the documentation around the architecture was unclear or did not cover more advanced use cases. This means that teams should expect to work in a trial-and-error approach and cannot fully rely on existing online resources. The absence of ready-to-use solutions is also highlighted by Pavlenko et al. [16], pointing out that much time is spent figuring out architecture instead of delivering features.

Migration should be gradual. Developers ran into problems trying move their entire monolith into a micro-frontends architecture. Introducing micro-frontends incrementally seems to be a more manageable approach. This is further reinforced by Antunes et al., explaining that teams benefit from gradual migration, enabling them to test architectural changes before committing [15]. They confirm that trying to refactor the entire system at once often results in setbacks and that gradual adoption supports evaluation and rollback when necessary.

Be clear on your motivation. Teams that had a clear goal of why they were adopting micro-frontends were better positioned to deal with the trade-offs. Without a clear idea, it is easy to run into unexpected complexity that suddenly outweighs the benefits. Knowing what problems you want to address helps in the adoption process by guiding decisions and setting realistic expectations for the effort involved.

Micro-frontends is not for everyone. A recurring theme in our data and in our pre-study was the question whether the complexity of micro-frontends is justified. In some cases, developers reverted back to a simpler architecture as maintaining and navigating the intricacy of micro-frontends outweighed the benefits. This suggests that while micro-frontends offer flexibility and autonomy, it may not be the right fit for all teams or projects.

6.2 Threats to validity

This section outlines the potential factors that may affect the validity of our study. Since our data comes from Stack Overflow, our results are influenced by how the platform is used, who uses it, and how quickly frontend technologies change. We divide these concerns into internal and external threats to validity.

6.2.1 Internal validity

The internal validity in our study can be threatened by selection bias and unbalanced representation. Our dataset is exclusively drawn from Stack Overflow posts tagged with ”micro-frontend.” This introduces a potential selection bias, where the data may disproportionately represent problems encountered by individuals in small teams or hobbyist projects. Micro-frontends is arguably more relevant in large-scale organizational environments, and therefore the more complex organizational challenges may be underrepresented.

Another concern is that Stack Overflow may primarily consist of immediate technical problems. This creates a potential bias where higher-level organizational or architectural issues are less likely to be raised.

Lastly, the filtering of irrelevant posts was based on manual labeling, which, despite iterative agreement and inter-rater reliability checks, remains subjective. This introduces the risk of inconsistently including or excluding borderline cases, especially when interpreting vague or ambiguous posts.

6.2.2 External Validity

The results of this study may not fully apply to large-scale systems as these are underrepresented in our data, threatening the external validity in this study. Similarly, our findings may be also less applicable to teams using niche frontend technologies that are not discussed in our dataset.

Moreover, the dataset spans several years, which increases the sample size but introduces the risk of outdated content. Outdated questions could make us perceive technical issues as hurdles inherent to the technology. Since modern web development technology is rapidly evolving, challenges in earlier posts may have since been resolved, while new less common obstacles may have emerged that are not captured in our dataset.

6.2.3 Construct Validity

We treat Stack Overflow posts as examples of real-world problems with micro-frontends. But because anybody can post on StackOverflow, the reliability of our main data source is not guaranteed. The quality of questions might therefore vary greatly, from highly technical to beginner-level questions. Some posts may even be based on misunderstanding and may not reflect actual challenges. However, while this diversity skews the perceived complexity, it reflects real-world usage, which arguably strengthens the validity of the study.

Random downsampling of posts comes with drawbacks, such as the risk of discarding relevant posts while retaining less informative ones. Although our process aimed to filter out irrelevant data, the informative relevance of the content was not considered. Our approach operated under the assumption that all posts related to micro-frontends held equal relevance, regardless of their depth or detail. Although this assumption may overlook posts that provide a more comprehensive discussion, it aligns well with the purpose of this study of identifying overarching adoption challenges, rather than prioritizing the details.

7

Conclusion

This thesis set out to systematically explore the adoption challenges, solutions and motivations behind the use of micro-frontends in modern web development. Using a grounded theory approach applied to Stack Overflow discussions, we were able to analyze the practical difficulties encountered during early and late adoption. In addition, we also analyzed provided solutions, and identified problems developers seek to address through this architectural approach.

Our findings show that the adoption of micro-frontends is hindered by a range of challenges that we categorized into four main themes: Integration, Migration, Modularization, and Infrastructure. Developers struggle with technical hurdles such as managing shared dependencies, coordinating routing, communication between micro-applications, maintaining consistent user experience, and setting up appropriate deployment pipelines. In addition to technical difficulties, we also observed theoretical challenges, such as lack of guidance and best practices, which contributes to the hesitation in adopting micro-frontends.

On the other hand, our results show that micro-frontends offer important benefits, which we categorized into strategic and technical dimensions. These benefits include independent deployment, team autonomy, scalability, and support for using different technologies within the same system, thus giving teams the flexibility to choose tools best suited to their needs and skillsets. We identified these benefits to be the core driving motivations behind developers' adoption of micro-frontends.

In addressing RQ3, we also investigated common solutions that developers propose to overcome adoption challenges. Among the most discussed strategies were the use of configuration practices, sharing common instances and effective communication management between applications. Furthermore, developers also emphasized the importance of establishing strong architectural guidelines early in the migration process to reduce room for interpretation. These solutions suggest that many of the associated problems can be mitigated through thorough planning and organizational coordination.

Moreover, we compared our work and findings to prior work in the field. The challenges and goals we observed closely match those identified in previous studies. By analyzing discussions on Stack Overflow, this thesis adds a new perspective based

on real-world developer experiences rather than controlled case studies or theoretical discussions.

In conclusion, micro-frontends present a promising but complex approach to frontend architecture. They offer clear advantages in certain contexts, but also come with trade-offs that should not be underestimated. Teams considering this architecture must weigh the technical overhead against the benefits of flexibility, autonomy and scalability. As tooling matures and more patterns become standardized, it will be interesting to see whether some of the barriers identified in this study become easier to overcome in the future.

7.1 Future research

Although this study offers a grounded view of the challenges, goals, and solutions to micro-frontend adoption based on real-world developer discussions, many paths remain for future exploration.

First, while our primary data source, StackOverflow, offers valuable real-time insight, it inherently primarily captures short-term and technical concerns. Future work could expand the scope of this study by incorporating additional data sources such as GitHub issues, technical blogs, or internal documentation from industry case studies to better capture organizational-level challenges. Case studies following teams throughout a full migration and even further would complement this area of study by revealing long-term trade-offs such as maintainability and onboarding efficiency.

Second, further studies may analyze how discussions about micro-frontends evolve over time as technologies mature. For instance, examining posts across multiple years could reveal how adoption challenges gradually change as new frameworks and tools evolve; and how priorities change. A temporal analysis would help differentiate between persistent and non-persistent challenges, in order to identify issues that would diminish throughout the years.

Finally, a more automated classification of StackOverflow data could scale this research to cover related frontend topics, such as Module Federation, monorepo vs. polyrepo strategies, or Web Components. Utilizing artifical intelligence such as natural language processing could help identify emerging patterns and trends while revealing valuable insights within the software engineering community.

Bibliography

[1] Raoni Kulesza, Marcelo Fernandes de Sousa, Matheus Lima Moura de Araújo, Claudiomar Pereira de Araújo, and Aguinaldo Macedo Filho. *Evolution of Web Systems Architectures: A Roadmap*, pages 3–21. Springer International Publishing, Cham, 2020.

[2] Phil Powell, Ian Smalley (IBM). What is monolithic architecture? [Online]. Available: `https://www.ibm.com/think/topics/monolithic-architecture`.

[3] Chris Richardson. Microservice architecture. [Online]. Available: `https://microservices.io`.

[4] micro-frontends.org. [Online]. Available: `https://micro-frontends.org/`.

[5] Cam Jackson. Micro frontends. *Martin Fowler*, 2019. [Online]. Available: `https://martinfowler.com/articles/micro-frontends.html`.

[6] Thoughtworks. Micro frontends timeline. [Online]. Available: `https://www.thoughtworks.com/en-us/radar/techniques/micro-frontends`.

[7] Severi Peltonen, Luca Mezzalira, and Davide Taibi. Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. *Information and Software Technology*, 136:106571, 2021.

[8] The Software House. State of frontend 2022, 2022. Accessed: 2025-04-10.

[9] JetBrains. The state of developer ecosystem 2022, 2022. Accessed: 2025-04-10.

[10] Reddit r/webdev. Microfrontend in 2024? [Online]. Available: `https://www.reddit.com/r/webdev/comments/1bs882t/microfrontend_in_2024/`.

[11] Luca Mezzalira. *Building Micro-Frontends*. O'Reilly Media, 2021.

[12] Luca Mezzalira. Micro-frontends in context. *Increment: Frontend*, (13), May 2020. Accessed: 2025-04-29.

[13] Single-SPA. Getting started with single-spa. `https://single-spa.js.org/`

docs/getting-started-overview/, 2023. single-spa.js.org, Accessed: 2025-04-29.

[14] A. Strauss and J. M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Inc., Thousand Oaks, CA, USA, 1990.

[15] Fabio Antunes, Maria Julia Dias Lima, Marco Antônio Pereira Araújo, Davide Taibi, and Marcos Kalinowski. Investigating benefits and limitations of migrating to a micro-frontends architecture, 2024.

[16] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. Micro-frontends: application of microservices to web front-ends. *Journal of Internet Services and Information Security*, 10(2):49–66, May 2020.

[17] Tanmaya Gaur. Applications of micro-frontend application development in a customer support crm. *International Journal of Computer Trends and Technology*, 72(6):15–24, 2024.

[18] Mehmet Söylemez, Bedir Tekinerdogan, and Ayça Kolukısa Tarhan. Challenges and solution directions of microservice architectures: A systematic literature review. *Applied Sciences*, 12(11), 2022.

[19] Alexander Lercher, Johann Glock, Christian Macho, and Martin Pinzger. Microservice api evolution in practice: A study on strategies and challenges. *Journal of Systems and Software*, 215:112110, 2024.

[20] Giuliano Antoniol, Anton Barua, and Ahmed E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. In *Empirical Software Engineering and Measurement (ESEM), 2014 ACM/IEEE International Symposium on*, pages 23–32. IEEE, 2014.

[21] Alan Bandeira, Alberto Carlos, Matheus Medeiros, Paulo Paixao, and Paulo Maia. We need to talk about microservices: an analysis from the discussions on stackoverflow, 05 2019.

[22] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. An empirical study of the systemic and technical migration towards microservices. *Empirical Software Engineering*, 28, 05 2023.

[23] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[24] Leslie Kish. *Survey Sampling*. John Wiley & Sons, New York, 1965.

[25] Miro - the visual workspace for innovation, 2025. Accessed: Feb. 25, 2025.

[26] Veeranjaneyulu Veeri. Micro-frontend architecture with react: A comprehensive guide. *International Journal of Computer Engineering and Technology*,

15(6):130–153, 12 2024.

[27] Lakshmanarao Kurapati. Micro frontend architecture: Benefits, challenges, and best practices. *International Journal for Multidisciplinary Research*, 6(5):1–14, 10 2024. Article ID: IJFMR240528481.

A

Appendix