TU WIEN Informatics

From Backend to Frontend

Case study on adopting Micro Frontends from a Single Page ERP Application monolith

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Ing. Manuel Kroiß, BSc
Matrikelnummer 01526926

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Andrea Morichetta

Wien, 21. April 2021

Manuel Kroiß	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

From Backend to Frontend

Case study on adopting Micro Frontends from a Single Page ERP Application monolith

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Ing. Manuel Kroiß, BSc
Registration Number 01526926

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Assistance: Dr. Andrea Morichetta

Vienna, 21st April, 2021

Manuel Kroiß	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Ing. Manuel Kroiß, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. April 2021

Manuel Kroiß

Danksagung

Ich möchte mich bei meinem Betreuer, Dr. Andrea Morichetta, für die zahlreiche Unterstützung bedanken. Die schriftliche Kommunikation sowie die Online Meetings waren stets sehr aufschlussreich und die Unterstützung bei der Suche nach hilfreicher Literatur hat mir sehr geholfen. Ich konnte die Arbeit nach meinen eigenen Vorstellungen und Gedanken verfassen und bin dankbar für seine Denkanstöße in die richtige Richtung.

Weiters möchte ich mich bei meiner Freundin Katharina bedanken, die mich vor allem in den für mich schwierigen Anfangs- und Endphasen dieser Arbeit oft aufbauen konnte und mir viele hilfreiche Anmerkungen zum struturellen Aufbau meiner Arbeit gegeben hat.

Acknowledgements

I would first like to thank my thesis advisor Dr. Andrea Morichetta for the support. We had constructive written communication and very revealing online meetings. His help in looking for relevant literature was beneficial. He allowed me to include my own thoughts and ideas into the work, but steered me in the right direction when ever he thought it was necessary.

Additionally, I want to thank my girlfriend, Katharina. She often steered me up, especially in the beginning and ending phase of this thesis, and gave me various hints on structuring my thesis.

Kurzfassung

Viele Softwareunternehmen verwenden bereits eine Microservice (MS) Architektur, um deren Software sowie die Entwicklungsteams dieser Software skalieren zu können. Diese Architektur verbessert die Projektwartbarkeit und erlaubt es den Entwickler*innen, schnell auf geänderte Anforderungen zu reagieren oder neu entwickelte Technologien einzusetzen.

Dennoch verwenden viele Firmen diesen Architekturtyp nur für Services, welche im Hintergrund laufen, und entwickeln eine einzige, monolithische Frontend Applikation, welche mit den verschiedenen Services im Hintergrund kommuniziert.

Existierende Micro Frontend (MF) Entwicklungsansätze werden meistens direkt von Softwaregiganten wie Amazon oder SAP entwickelt, welche sich hauptsächlich auf serverseitige Renderingansätze spezialisieren. Es gibt nur wenig Literatur, welche die Verwendung des MS Architekturtyps für clientseitig gerenderte Frontends geprüft hat. Keine dieser Arbeiten behandelt eine MF Performanceanalyse oder benutzt verschiedene clientseitige Frontend Technologien in einer MF Architektur.

Diese Arbeit liefert eine Analyse von existierenden MS Arbeiten und zeigt, wie diese Prinzipien auf MFs übertragen werden können. Die in dieser Arbeit durchgeführte Fallstudie inkludiert eine Implementierung der gleichen Applikation als monolithisches Frontend, eine MF Architektur aufbauend auf dem single-spa MF Framework und eine MF Architektur, die mit Webpack Module Federation entwickelt wurde.

Der letzte Teil der Arbeit beschreibt eine Performanceanalyse, welche diese drei Fallstudien in der gleichen Umgebung bereitstellt und überprüft. Diese Analyse zeigt auf, dass es möglich ist, ein monolitisches Frontend durch eine MF Architektur zu ersetzen, ohne Performanceverluste hinnehmen zu müssen. Stattdessen profitiert man von dieser Ersetzung und kann auf die Vorteile, die bereits jetzt in MSs genutzt werden, auch im Frontend zurückgreifen.

Schlüsselwörter: *Micro Frontend, Application Shell, Micro Frontend Performanceanalyse.*

Abstract

Many software engineering companies already use Microservice (MS) architectures to support scaling their software and the development teams. This architecture improves the project's maintainability and allows developers to react to changing requirements or use newly developed technology.

Nevertheless, most of those companies utilize this architectural style only for the backend services and build one extensive monolithic frontend application that communicates with the different MSs.

Existing Micro Frontend (MF) development approaches are mostly directly developed by big software engineering enterprises such as Amazon or SAP and mainly focus on server-side rendering. Only a few research papers analyzed using the MS architectural style for client-side rendered frontend solutions. None of those papers covers MF performance analysis or uses different client-side frontend technologies in one MF architecture.

This paper provides an analysis of existing MS literature and shows how one can adopt those principles to MFs. A case study performed in this work exposes an implementation of the same application as a monolithic frontend, an MF architecture using the single-spa MF framework, and an MF architecture built with Webpack Module Federation.

The last part of the thesis presents a performance analysis that investigated those three cases in the same deployment environment and reveals that one can replace an existing monolithic frontend over an MF architecture without losing runtime performance but profit from the advantages that MSs already exhibit for the backend.

Keywords: *Micro Frontend, Application Shell, Micro Frontend Performance Analysis.*

Contents

CHAPTER 1

Introduction

In 2018, Camunda Inc. [2] conducted a survey of 354 software engineering companies, revealing that already 63 percent use Microservices (MSs) as an integral part of their system architectures. The main reasons reported by the survey attendants were improved scalability (64%), faster time to market (60%), and supporting digital transformation as well as generating autonomy for development teams (both 64%). While performing research for this thesis, only a few companies (less than 20) using distributed frontends could be found. Most software engineering companies still use a frontend monolith.

Tim Berners-Lee [8] first introduced the Hypertext Markup Language (HTML) in 1991, which was the basis of the World Wide Web (WWW) we know today. After that, many companies and researchers worked on pioneering projects based on the initial HTML version or later introduced standards with extended functionality.

Since then, the number of online services grew, and people have tried to make everything accessible to their users via the internet. These services got complex over time, and businesses and researchers realized the need for a modern and agile approach for software development and distributed computing. In 2004, Service-oriented architecture (SOA) was first announced. [30] A few years later, people started to focus on Cloud Computing [16] to support essential features like auto-scaling, Continuous Delivery (CD), hot deployments, and high availability. [86] Soon, new research areas like Infrastructure as a Service (IaaS) [88], Platform as a Service (PaaS) [7], or Software as a Service (SaaS) [3, 74] emerged, which led to greater efficiency in the operation by allowing to scale just some parts of a system on-demand to reach better resource usage.

This Everything as a Service (XaaS) [27] trend needed decoupled solutions to develop and deploy independently. MSs soon started to gain interest in both the research community and IT businesses. Lewis and Fowler [51] state that the Microservice Architecture (MSA) "is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP

resource API." Dragoni et al. [26] describe an MS as "a cohesive, independent process interacting via messages" and the resulting MSA as "a distributed application where all its modules are microservices."

Many companies already focus on using an MSA, with their initial goal to have vertical silos that can deal with everything from storing the data in a database to providing a User Interface (UI) for manipulating this data. Current MSAs only deal with splitting the backend into multiple services and provide a horizontal UI layer.

With the growing number of MSs, implementing horizontal UIs requires more frontend developers in one team, which raises many concerns. The more developers are hired to work on the project, the more the monolithic frontend's codebase increases. Those monolithic applications often exhibit concepts of poor isolation with adverse effects reflected in different aspects:

- The Developer Experience worsens because the focus transfers from coding to addressing the developed solution's maintainability. Developers need to spend additional time in meetings to discuss the dependencies between the different parts of the system instead of coding.
- On the business side, the company faces problems with deploying new features in an appropriate amount of time to fulfill customers' needs.
- The company has difficulties in getting new hires. One reason is that they have to stick to their old technology because updating or replacing their frameworks and maintaining the codebase is very costly and time-consuming.
- Each deployment is a very crucial step for companies since it needs a large number of tests for maintaining the old features' quality constraints. Those tests can become even more complicated than the system itself.

Some companies like Amazon, Facebook, Zalando, SAP, Dazn, AirBNB, HelloFresh, Allegra, Klarna, and others have already found and implemented unique solutions tackling their company-specific frontend problems. Most of them use a Server-Side Rendering (SSR) approach that requires numerous servers and frameworks to manage components' composition on the server-side. Those techniques are also known as Micro Frontend architecture (MFA).

Betts et al. [9] present software architecture and design trends (see figure 1.1) and mention that Micro Frontends (MFs) are intended to bring the MS benefits to the UI. Smaller companies can not invest such a considerable amount of time implementing a complicated and distributed frontend infrastructure. Numerous companies use single page application frameworks like Angular [1], Vue.js [2], React.js [3], or others for their web development.

[1] https://angular.io/
[2] https://vuejs.org/
[3] https://reactjs.org/

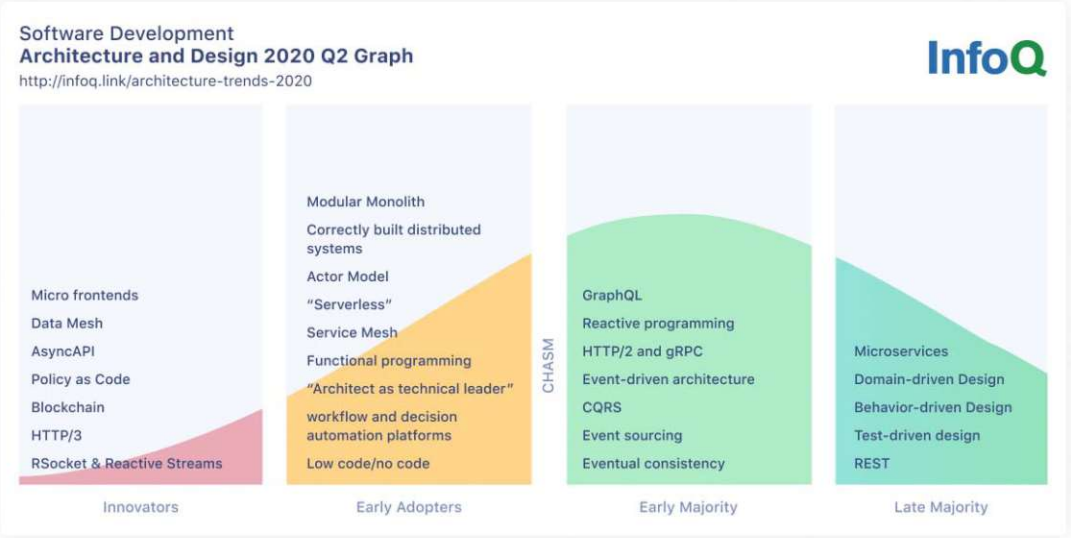

Figure 1.1: Software architecture and design trends 2020 [9]

In particular, the example application in the Enterprise Resource Planning (ERP) field implemented in this thesis is a very descriptive example of why it is necessary to convert a big monolithic architecture in an MFA. There is a significant rise in industry 4.0. Nearly all machines used in modern manufacturing halls offer a Human Machine Interface (HMI) that provides data necessary for the company.

More and more companies want to be able to manage all of their processes in one single tool. Further, one can see a considerable rise in adapting the work in a company from static processes to processes that focus on using mobile devices. [25, 39, 49, 56, 60] These processes require to have different implementations of a UI to provide a perfect User Experience (UX) for back-office employees (working with a monitor, a keyboard, and a mouse), and for manufacturers in the production hall, who often just have a small touch display and maybe a barcode scanner like described in the patent by Hicks et al. [44].

However, newly deployed MF solutions can represent a possible way out for companies. This thesis answers the following research questions to fill the gap of developing independent frontends.

- Can an MFA replace a monolith ERP Single Page Application (SPA) with a client-side rendering solution without losing performance and, therefore, UX?
- Can the MF approach lower the risks of updating or adding a new feature in a complex web application?

The thesis shows that the following hypothesis can be confirmed.

- An MFA reduces upgrading/adding feature risks and improves or at least maintains page load speed compared to a monolith SPA.

The thesis covers the following parts to evaluate the MF topic.

In the first part, there will be a state-of-the-art analysis of existing solutions for MSs. With this analysis's findings, the paper shows how one can adopt the requirements and specifications of MSs to MFs in the second part.

Then, the thesis presents a case study on a migration from a monolithic Angular application to an MF-based implementation in the context of an ERP application. The thesis analyses existing MFAs and frameworks. The focus of this research will be on frameworks that support client-side rendering, especially single-page application frameworks, but the paper will also briefly mention other available solutions like SSR, including Edge Side Includes (ESI) or Server Side Includes (SSI). Each of the implemented applications will use different SPA frameworks like Angular, React.js, Vue.js, or others to fill the literature gap where only one specific framework was used to implement the MFA. [67, 92]

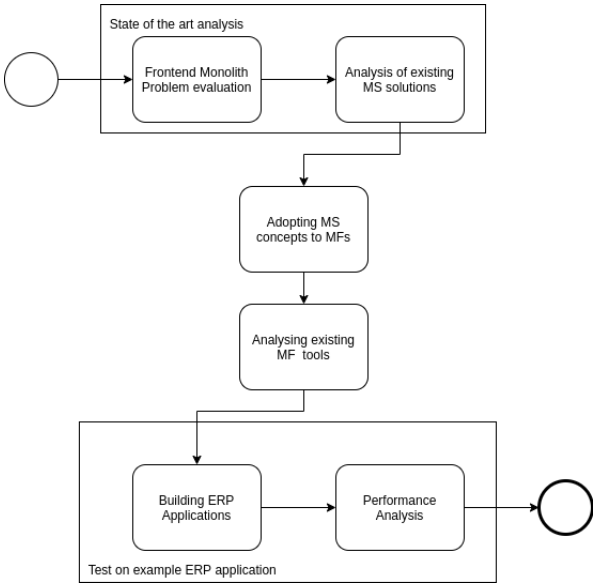

Figure 1.2: Thesis flow chart

Lastly, the thesis compares the implementation of the same system with the two different MFAs. In the end, the thesis shows quality metrics such as bundle size, Time to interact (TTI), Time To First Byte (TTFB), and others to evaluate the results.

Since using Continuous Integration (CI)/Continuous Delivery (CD) is an essential feature required by MFs, the thesis provides a working CI/CD pipeline configuration. With that in place, the thesis shows how one can quickly deploy a bug fix or a new feature in an MFA context.

Another interesting question of MFs is how to provide a consistent UI to all the different MF applications. Since this is a highly complex topic on its own, which is related to UX/UI design, this thesis will not cover that part of implementing an MFA.

The following chapters are structured as follows (Figure 1.2 visually represents the steps performed in this thesis). Chapter 2 shows the current state-of-the-art of MSs and MFs approaches, and introduces related works that handle monolith to MF migrations. In chapter 3, the thesis analysis literature concerning MF principles and challenges and adopts the MS concepts to MFs. The case study is presented in chapter 4, followed by an evaluation in chapter 5. Chapter 6 summarizes the results and presents future work.

CHAPTER 2

State-of-the-Art

This chapter describes the state-of-the-art of the different research fields that the thesis uses to answer the research questions. A literature review using GoogleScholar[1] and IEEE XPlore[2] was performed to find information regarding MSs, MFs and MF-related papers. The chapter contains three sections, presenting the summary of the literature analysis. The first section (see 2.1) describes the current state-of-the-art concerning MSs. In the second section (see 2.2), the thesis summarizes all currently available concepts and methods available for implementing MFs. Section 2.3 shows existing papers that deal with implementing MFA related systems.

2.1 Concepts and requirements of Microservices

This section presents MS concepts and requirements, which form the basis for the adoption of MS concepts in the frontend discussed in Chapter 3.

Before one can analyze the MSs architectural style [34], it is essential to define what a microservice is. Lewis [50] first introduced the term MS in 2011. About the same time, Geroge [37] gave a talk about the MSA at the YOW! conference in Melbourne.

This architectural style already existed even though it had different names. For example, Netflix started to migrate its monolithic architecture towards an MSA and named the new architecture fine-grained SOA. [89]

In 2014, MSs started to attract the interest of large organizations. Di Francesco et al. [23] show in their research that there was a significant rise in the number of published research papers regarding MSs (see table 2.1).

[1] https://scholar.google.at/
[2] https://ieeexplore.ieee.org/Xplore/home.jsp

Year	2010	2011	2012	2013	2014	2015	2016
Published research papers	1	2	1	0	3	23	41

Table 2.1: Research distribution of MSs [23]

Since then, there is a considerable discussion whether the MSA is a new architectural style or it belongs to SOA. Newman [62] describes MSs as "one way of doing SOA (right)," and Dragoni et al. [26] state that MSs are the second iteration of SOA and service-oriented computing (SOC). SOA, in this regard, describes one possible approach to realize the SOC concepts. [66]

MSs focus on simple services that implement a single functionality and leave away unnecessary levels of complexity. [26] James Lewis, in an interview with Thones [84], state that "a microservice, [...], is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility", and that "it does only one thing and one thing alone and can be easily understood." James Lewis [84] mentions that the "key thing is to make the stack lightweight." Further, Hasselbring and Steinacker [41] argue that MSs typically have no centralized control.

Zimmermann [94] states that there are two contrary positions of microservices. The first one is the position by Lewis and Fowler [51], who describe the MSA as follows:

> "The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." [51]

Newman [62] defines MSs via seven principles:

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

	Characteristics described by Lewis and Fowler [51]	Relationship	Newman's principles [62]
1.	Componentization via services (running in own process and communicating with lightweight mechanisms)	(Similar to)	Hide Internal Implementation Details
2.	Organized around business capabilities	(Matches)	Model Around Business Concepts
3.	Products not projects	(No pendant)	
4.	Smart endpoints and dumb pipes	(Included in)	
5.	Decentralized governance (enabling polyglot programming)	(Subset of)	Decentralize All the Things
6.	Decentralized data management (and polyglot persistence)	(Subset of)	
7.	Infrastructure automation (and decentralized management)	(Subset of)	Adopt a Culture of Automation
	(attribute in definition, but not elaborated upon in dedicated section of article)	(Matches)	Independently Deployable
8.	Design for failure	(Superset of)	Isolate Failure
9.	Evolutionary design	(No pendant)	
10.		(No pendant)	Highly Observable

Table 2.2: MS definition comparison [94]

Both definitions mix process, architecture, and development concerns, and one could ask if the definition should include process-related and organizational aspects. [94] Presumably, this inclusion is because of Conway's law, which the thesis briefly describes in Section 2.1.2. Zimmermann [94] provides a comparison of those two definitions that one can find in table 2.2.

The thesis structures the following sections as follows. Section 2.1.1 describes nine reasons why one could benefit from using MSs. The next section (see 2.1.2) concentrates on the principles that MSs exhibit. Lastly, Section 2.1.3 deals with common challenges a company faces when developing an MSA.

2.1.1 Microservice characteristics

Performance and maintainability are the most investigated quality attributes as mentioned by Di Francesco [21] when it comes to migrating monolithic applications to an MSA. Linthicum [52] states that if one turns a monolith into a complex, distributed system, one must require that the new system is more productive, agile, and cost-effective.

Both researchers [21, 52] explain why it makes sense to migrate a monolithic application into an MSA. The following section summarizes MS characteristics that literature often mentions as arguments for using MSs.

Nine reasons to use Microservices

The following listing orientates itself on a survey of 21 practitioners who moved to an MSA, which was carried out by Taibi et al. [83], and combines it with the arguments of Di Francesco [21] and Linthicum [52].

Further, the listing includes answers from a survey performed with 25 experts involved in projects related to an MSA by Ghofrani and Lübke [38]. Newman [62] provides reasons to split a monolith. The listing includes the initial thoughts by Lewis and Fowler [51] and

integrates migration reasons from Balalaie et al. [5], who define 15 migration patterns one can work off while migrating from a monolith to an MSA.

Di Francesco et al. [23] provide characteristics from a systematic mapping study of 71 studies. Dragoni et al. [26] researched on MSs covering the history, the state-of-the-art in 2016, and a future outlook. The authors of [95] describe characteristics found when researching the possibility of an incremental integration of MSs in cloud applications. Another work by Balalaie et al. [4] provides MS characteristics from a project on migrating from a monolithic application to an MSA.

In the following paragraphs describe those nine main characteristics in detail.

Scalability Scalability [4, 5, 23, 26, 38, 51, 83, 95], which is a crucial factor to reach the goal of availability [4] is the attribute that the research mentions most frequently. For large software systems, one should scale the system according to users' needs without losing performance. [21, 23] It is important to note that with the use of MSs, the scaling does not imply duplication of all parts of the system, but it is possible to scale each MS independently. [26]

Maintainability Maintainability [23,26,38,83], which is closely related to small services and automated deployment, primarily means to reduce the complexity [51, 83] of the system so that it remains easy to fix bugs. Newman [62] states that maintainability can increase the pace of change. Separate, autonomous units loosely coupled into small and isolated services [91] can be changed faster than one big monolith. [62] Each microservice is independent of other services and has a very high cohesion [26]. This independence leads to a lower amount of Lines of Code (LOC) and can therefore increase the code understandability. [83] The small codebase [26] and the limited amount of functionalities of MSs [26] make the code easy to understand [4] and therefore help to limit the scope of bugs. [26] Maintainable systems are easy to extend [38], and easily re-used for other systems [5].

Delegation of Team Responsibilities Delegation of Team Responsibilities is a major reason why it is useful to migrate from a monolithic architecture to an MSA. [83] With the use of MSs, it is possible to split large teams into more efficient small teams, developing their belonging service independently. [83] These smaller teams lead to a reduced need for coordination and therefore reduced communication overhead. [83] Ghofrani and Lübke [38] and Di Francesco et al. [22] as well as Balalaie et al. [4] and Zúñiga-Prieto et al. [95] mention that agility is a very relevant aspect when migrating towards an MSA. Smaller development teams help to achieve the required agility and make it possible to grow the development teams in a controlled way. [51]

DevOps Support An MS enables and requires DevOps [46] support because each team needs to be able to easily develop, test, and deploy their MSs independent of other teams. [83, 86] This independent deployment [4, 62, 95] allows gradual transitions

to new versions, often supported by continuous integration [33], which allows to test and investigate changes of a module in isolation without the requirement for a complete reboot of the system. [26] The small size of each service makes it easier to automate the deployment [5] and enables short re-deployment times. [26]. One can easily containerize and execute each service [83], which leads to the ability that services can be auto-scaled separately. [52] Separate auto-scaling means that it is possible to expand and de-expand resources depending on each service's resource needs without the need to scale services with lower resource consumption. [52]

Reliability Ghofrani and Lübke [38], Taibi et al. [83] and Di Francesco et al. [23] mention that an MSA can greatly improve the reliability and fault tolerance of a software system. In a monolithic architecture, a failure of one component affects the whole system, whereas a failure in one service in an MSA impacts this one service only, and the rest of the system can still work as expected. [83] Further, a faulty MS can be quickly restarted compared to a large monolithic application, or it is even possible to replace the faulty MS with an old, working version that one can run in parallel for reliability reasons. [83]

Independent Technology The ability to use different technology in each MS is an immense advantage compared to a monolithic architecture. Each team can decide to write their service in the most appropriate programming language for the task. [4, 62, 83] MSs allow teams to use a different internal structure that fits their needs. [83] As an example, it can be necessary to use different programming languages if one service needs to communicate to a database, one service makes use of a logic programming library, and another service implements a machine learning model. [4] Di Francesco et al. [23] describe this aspect of MSs with the term functional suitability. This flexibility supports all necessary modifications to a system without the danger of a vendor lock-in. [4, 26]

Responsibility Separation Separation of Software Responsibilities means that each MS is responsible for one task with well-defined boundaries that is self-contained in a closed system. [83] Taibi et al. [83] mention that splitting a software system into smaller parts can significantly simplify the development. Balalaie et al. [5] state that this separation of responsibilities can be of particular interest when one wants to persist data. With the possibility to split a big database into small parts, MSs help to decentralize data governance. [5] The separation forces an independent communication scheme via language-agnostic APIs, for example, via Representational State Transfer (REST) [87], which can further increase the complex software system's understandability. [52]

Security In general, security is a complex topic that becomes even more complicated when it comes to distributed systems. [10, 23, 36] Newman [62] mentions that MSs do not only increase the effort one has to invest into security, but an MSA can also apply additional protection mechanisms like easier monitoring or special data protection for data belonging to services with the requirement for high security.

Innovation Lewis and Fowler [51] write that with the use of MSs, companies can innovate quickly because of the aforementioned reasons. The ability to be flexible in using the new technology can avoid market loss in the future as it could be possible if a company makes the wrong technology choice for a monolithic application. [83] Each service should be built around business capabilities. [95] This flexibility enables companies to get new functionalities out fast [95] and reduces time-to-market [4]. Additionally, the small and independent services can make use of the elasticity and better pricing model of cloud environments compared to an on-premise installation. [4]

2.1.2 Microservice principles

According to the characteristics of MSs (see Section 2.1.1), one can derive principles that an organization should follow when implementing an MSA. The following sections describe those MS principles that the literature mentions. The first section describes the aspect that choreography is a better choice for an MSA than orchestration. The next section underlines what is important to consider when one thinks about communication between MSs, followed by a fragment that mentions the importance of CI/CD [45, 72]. Service Responsibility talks about the right MS size and independence, and the MS teams section describes what culture a company should have when using MSs. The last section explains some common patterns that are useful when dealing with an MSA.

Choreography over orchestration

Dragoni et al. [26] and Newman [62] state that with many cooperating MSs, it becomes essential to prefer and use choreography [65] over orchestration [55]. Orchestration would require a central manager that would conflict with the share-nothing philosophy [38] and the need for no centralized control. [62] Dragoni et al. [26] mention that using orchestration would lead to "coupling and uneven distribution of responsibilities," whereas when applying choreography, one could use events or publish/subscribe [23] mechanisms to establish a collaboration between services inherited from SOA. [26]

Communication

Decentralize all the Things is one of the seven microservice principles that Newman uses in his definition of MSs. [62] He mentions that for an MSA, it is crucial to use dumb middleware and smart endpoints. [62] One can extend this with the concepts of smart endpoints and dumb pipes, which Zúñiga-Prieto et al. [95] recommend to enable inter-service communication via simple messaging. One can confirm those communication strategies that Newman [62] and Zúñiga-Prieto et al. [95] provide with the findings by Dragoni et al. [26] who state that one needs simple components with clean, published interfaces for message passing to achieve high reliability.

The communication over the network can imply network issues that negatively impact the communication between the services. [26] Therefore, one has to find and define well-bounded contexts [31] for the MSs to lower the message rate over the network [26]

because it is evident that in-memory calls are much faster and less error-prone than network communication. [26]

Continuous Integration / Continuous Delivery

Zúñiga-Prieto et al. [95] refer to infrastructure automation, Dragoni et al. mention that one should use MSs with CI/CD and Newman [62] and Humble and Farley [45] state that the use of MSs requires to adopt a culture of automation and to practice CD and DevOps. Besides, Balalaie et al. [4] indicate that if the number of services rises, it becomes more and more interesting to use an automated delivery process, otherwise at some point, one would spend more time on deployment tasks than on developing. At Netflix [53], they also transformed their monolith to an MSA. The major reason was that they wanted to deploy in seconds to satisfy the fast-changing business requirements. This example of Netflix coincides with the statement of Zúñiga-Prieto et al. [95] that one has to introduce CI/CD to enable changes in the software system at the pace of business change. Di Francesco et al. [23] confirm this tight coupling between MSs and DevOps.

One should "separate the source code, the configuration, and the environment specification." [4] CI/CD allows automated configuration for different environments like a test system and a production system.

Service Responsibility

One key question is how big a service should be. On the one hand, Hatton [42] and Compton and Withrow [19] already found that small-size software can have a very high fault density in the early days of software development. El Emam et al. [29], on the other hand, state that if a software increases in size, one can also determine an increased fault proneness. There are no general rules like the amount of LOC or number of files an MS should have. Newman [62] states that currently, there is no "commonly-accepted definition of the desired size of a microservice." In literature, almost all researchers agree that services should be modeled around business concepts [62, 95] and that a service should have a bounded context and only include functions related to that context. [26] Domain-Driven Design (DDD) [31] is a well-known approach to model service responsibility from a business context. If a developer feels that an MS becomes too large, one should split it up to preserve the granularity and focus on a single responsibility. [26] This "componentization of services" helps to fulfill the goal of MSs that they should be independently replaceable and upgradeable. [95]

Shared or decentralized data governance is a central principle of microservices. [62] Each MS team needs the opportunity to use the right technology (tool, language, data storage) for the job, and each MS needs to be the one and only owner of the data it is processing. [95] Technology agnostic APIs help to hide those implementation details from the service consumers. [62]

Since each service has the aforementioned freedom, it has to ensure that it can provide the offered functionalities with correctness. The small size of the services assists developers in

reaching this goal. The service isolation enables independent tests, improves testability, and limits tests' scope if changes occur. [26] Dragoni et al. [26] confirm this fact when they write that "discovering a bug and or adding a minor improvement do [sic] not have any impact on other services and on their release schedule," with the condition that the service is backward compatibly and the interface remains unchanged.

Microservice Teams

A company that decides to use the MS pattern [71] as their primary software architecture has to change how they organize their development teams. Conway's Law [43] describes how the company structure affects the output that teams produce in their daily work. Mezzalira [57] mentions the Inverse Conway Maneuver and states that "teams and organizations" should "be structured according to" the "desired architecture and not vice versa."

Zimmermann [94] uses the term "cross-fertilization" to describe this aspect and states that there is a relationship between the architectural style of software and the engineering process and culture, and also Villamizar et al. [86] mention that an MSA requires an according company structure. It is, therefore, crucial to gain a general agreement on how the business organizes teams. Zúñiga-Prieto et al. [95] introduce the important aspect "Products, not Projects." Each development team should be responsible for a complete product and not work on a project until it is finished and quickly forget about it.

In 2016, Amazon Chief Technology Officer (CTO) Werner Vogels introduced the principle "you build, you run it." [64] Newman [62] states that one should structure each microservice team so that the team is responsible for taking full ownership of a software system, from the database to the UI. If a change or a new feature is requested for a specific MS, only the owning team should be responsible for changes to the MS. [62] Teams should be responsible for the complete software stack and take full responsibility for the software in production. [95] Newman [62] widens this by saying that an MS team should cover all aspects of a software development lifecycle, "from sourcing requirements to building, deploying and maintaining the application."

If a company applies this culture, each team will have "an incentive to create services that are easy to deploy," which will lead to increased autonomy and delivery speed. [62]

Independent development can, of course, lead to a large number of duplicated LOC where teams could agree that it would make sense to share those parts of their systems across all teams. This independence can lead to unwanted coupling between the MSs. To overcome this issue, Newman [62] recommends treating the shared code as "Internal Open Source." Only one MS team should be responsible for a shared library, and if another team needs some change in that code, they can ask the owning team to perform the change, or they do it by themselves and submit a pull request. [62] Dealing with shared code as an open-source library allows sharing code between teams while the owning team still controls it. [62] Hasselbring et al. [41] provide a reason why one should favor internal open-source over shared private libraries:

> "Apparently, open sourcing the code instead of sharing common private libraries seems to be almost the same. However, the open-source approach has some psychological effects: Developers show a tendency to apply higher quality standards if they know that the code will be publicly available." [41]

Microservice Patterns

Some explicit works [48,71] are available in the literature dealing with MS patterns. Most of the time, authors do not explicitly define recommendations as patterns, and one could also see those patterns as recommended infrastructure services each MSA should contain. The following five paragraphs describe patterns that literature analyzed as being useful when dealing with MSs.

Business Microservice A business MS is one of many MSs that implement the main logic for the system. Pavlenko et al. [67] mention that a business MS should handle only one business problem, and different MSs "should not introduce indirect communication through a database."

Adapter Microservice An adapter MS embeds an external system into the MS environment and maps requests from different protocols that the underlying external services use to a protocol that the MSA can understand. [67]

API Gateway The API Gateway [5, 23, 62, 69, 86] is the pattern that the literature mentions most frequently. The goal is to have one single entry point, sometimes also called edge server [69], responsible for receiving and forwarding traffic from outside (e.g., a frontend). [5] Newman [62] states that the API Gateway becomes "one giant layout for all our services," which is the goal of using this pattern, but it introduces some new problems. With the use of an API Gateway, one can lose the isolation of MSs and limit independent releases because a new service will probably also lead to a change of the API Gateway. [62]

Backend for Frontend Instead, or additional to, an API Gateway, Newman [62] introduced the Backend for Frontend (BFF) pattern. He mentions that, using the BFF pattern, the UI teams also handle server-side components, namely the BFF, which acts as an API Gateway but can also include some logic to call and collect data from multiple APIs and transform the received data into a format that the UI can understand. Pavlenko et al. [67] describe the BFF as an orchestration layer that calls and combines responses from multiple MSs. In addition to the message conversation, the BFF can filter the received data and remove information that the client does not need to decrease the message size. [67] Newman [62] already mentions that this approach has the danger that the BFF could also include business logic. Instead, the BFF "should only contain behavior specific to delivering a particular user experience." [62]

Circuit Breaker A Circuit Breaker [23, 62, 63] is a piece of software that one usually integrates into an API Gateway or a BFF to isolate failure from underlying MSs. Possible implementations could, for example, cache the last result from an API call and return it in case of an error or a timeout of the underlying service or cache pages for listed entities that the client may request in the future. [67]

Service Discovery / Load Balancer Linthicum et al. [52] define the location independence pattern, which explains the ability to discover services at runtime and activate them with late binding. The primary technical solution to satisfy the requirements of this pattern is service discovery. [5] MSs register themselves at the service discovery, also called service registry, and , e.g., a BFF that wants to call a service can ask the Service Discovery where to find it dynamically. [67] Additionally, to locate one service, it is possible to integrate a Load Balancer [5, 23] that can take care of forwarding requests to multiple, scaled services of the same type to balance the load across those instances.

Other Infrastructure Services Additionally to the aforementioned patterns, there are other recommendations for infrastructure services. Di Francesco et al. [23] mention monitoring services that deal with logging and profiling as well as a health management service as a system-level management service in addition to an autoscaling and load balancing service. Balalaie et al. [5] argue that an MSA could benefit from a configuration service that allows administrators to change the service configurations without redeploying the code. Pavlenko et al. [67] describe the Log Aggregator pattern, where a service can handle all logging messages from the different MSs. The Correlation identifier (ID) pattern assigns a unique ID to each request from the frontend. [67] Each service should use this ID in their log messages to trace a frontend call through all involved services.

The patterns API Gateway and BFF need caution. Whereas they offer some simplification in the communication, they can create a single big bottleneck because they see an MSA as an architecture with multiple backend services and one big frontend. This aggregation of services is not the initially intended meaning of MSs that should take responsibility from the database to the UI. The API Gateway pattern especially could be problematic because it introduces a layer that combines all APIs from the different MS teams into one big layer that exposes those APIs like a monolithic application would do.

2.1.3 Microservice challenges

From the previous discussion, it is clear that developing MSAs is not an easy task. This section explores the main challenges. Taibi et al. [83], as mentioned in Section 2.1.1, ran an empirical study with 21 practitioners who migrated from a monolith to an MSA. Their interview partners reported several issues that arose during the migration, which they categorized into three types of issues: (i) technical issues, (ii) economic issues, and (iii) psychological issues. [83] This output coincides with the findings from Di Francesco et al. [22], who reported technical challenges (e.g., infrastructure automation, distributed

debugging) and organizational challenges (e.g., creation of cross-functional teams, long time to release new features, low productivity of developers).

The following three sections make use of this categorization into the three types of challenges regarding MSs and combine the results from the studies by Taibi et al. [83] and Di Francesco et al. [22] with other literature. At the end of this section, the thesis presents in two paragraphs lessons learned while migrating towards an MSA as well as reasons when it is not helpful or recommended to use an MSA.

Technical Challenges

Di Francesco et al. [22] report that the most challenging part when implementing an MSA is the initial infrastructure setup. One has to deal with numerous new components like a Service Discovery or monitoring and managing services. [5] Additionally, the DevOps infrastructure setup effort is much higher than when one opts for a monolithic architecture. [83] As an example, one can consider how a company should maintain its source code. This task can already take a substantial amount of working hours since one has to decide between different solutions like using a mono-repository, a multi-repository, or a hybrid approach like Brousse discusses in [13]. Mezzalira [57] also investigates the pros and cons of multi-(poly-) and mono-repositories. Pavlenko et al. [67] also describe those repository organization possibilities and add information about the git-repo tool[3] and compare the different methods (see 2.1).

	Mono-repository	**Multi-repository**	**Multi-repository with git-repo tool**
VCS	Any	Any	Git
Additional tools required	No	No	git-repo
Package registry required	No	Yes[1]	No
Partial codebase downloading	Impossible	Possible	Possible
Setup complexity	Simple	Complex	Complex
Checkout speed	Slow	Fast	Fast

Figure 2.1: Repository organization possibilities [67]

Testing can also be a tricky part of a microservice environment. With an increased number of services, the system will become fault-prone on the integration level. [83] This integration needs integration tests, which check the working connections and communication between the components to ensure correctness because some "anomalies can emerge from collaboration of a number of services." [26]

[3] https://gerrit.googlesource.com/git-repo

In Section 2.1.1, the thesis mentioned the possibility of using different technologies for different MSs as an MS benefit. This characteristic can also be a disadvantage. Dragoni et al. [26] mention that it can require additional effort to define service composition contracts in different languages. Taibi et al. [83] state that different technologies can require extra effort for library conversions because one can not simply reuse a library written in one language in an MS implemented in another programming language. Lewis and Fowler [51] recommend defining technology guidelines that contain a subset of all available technologies that should be mainly used in the company.

Particular challenges do arise when one tries to migrate a monolith system into a system with an MSA. Balalaie et al. [5] and Di Francesco et al. [22] express that it can be tough to decompose a system into small units because of the often high level of coupling. Many companies use an incremental migration process because it would be difficult and dangerous to perform the migration with a big-bang approach. Di Francesco et al. [22] mention that companies often have difficulties identifying the boundaries of services. Taibi et al. [83] state that decoupling parts of a monolithic system can be challenging, especially in database migration and data splitting.

Organizational Challenges

Dragonie et al. [26] mention that distribution is the greatest weakness of MSs and state that "programming distributed systems is inherently harder than monoliths." Zimmermann [94] describes this fact as follows: "Microservices usage promises to be more dynamic and flexible, requiring more runtime and configuration effort than coding." Further, Zimmermann [94] states that "in general, syntactic and semantic contracts always exist, either implicitly or explicitly (as machine- and human-readable contracts)." The main concern about this is that currently, MSs have only informal documentation, mostly in a natural language only, which is very error-prone because of the potential ambiguities. [26]

"Effort Estimation and Overhead" is another big challenge when one uses MSs. [83] The estimation accuracy is generally lower for MSs as for a monolith, and one has to deal with a lot of effort overhead when implementing MSs, which is often compensated much later in the maintaining phase of a system. [83] Villamizar et al. [86] state that MSs should be a long-term strategy and not a project where it is clear that additional effort is required and one has to develop an MSA incrementally. The following quote by Taibi et al. [83] illustrates this issue related to the Return on Invest (ROI).

> "Microservices-based systems are less expensive than monolithic systems in the long run, allowing for good ROI. Our interviewees reported that ROI is achieved during maintenance of the system, as maintenance costs are lower than in monolithic systems." [83]

Psychological Challenges

Taibi et al. [83] mention a challenge related to people's minds. It happens that older developers do not believe in the new technologies and often do not "want to accept a big change to 'their' system." [83]

Lessons Learned Balalaie et al. [4] present their lessons learned when migrating a monolithic system to an MSA. The following list summarizes their learnings:

- Deployment in the development environment is difficult

  MSs bring an additional burden on developers because they need to deploy dependent services locally.

- Service contracts are double important

  Balalaie et al. [4] recommend avoiding service versioning. Their advice is to use techniques like Tolerant Readers, which "make every attempt to continue with message processing when potential violations are detected." [24]

- Distributed system development needs skilled developers

- Creating service development templates is important

  The use of different programming languages can be a benefit for an MSA, but they could also result in chaos and maybe make the system unmaintainable. [4] Balalaie et al. [4] advise defining standards and provide templates for developers allowing to start developing quickly.

- Microservices are not a silver bullet

  The "challenges should be considered before the adoption of microservices." [4]

Reasons against Microservices As Balalaie et al. [4] already mentioned in their lessons learned, the implementation and use of an MSA is not a silver bullet. Zimmermann [94] mentions that MSs introduce an inherent complexity and that MSs are highly flexible due to their fine-grained dynamic nature, but there are numerous subtleties one has to tackle.

Taibi et al. [83] recommend not to use this architectural type for quick prototypes or small projects because the overhead that MSs introduce can not be compensated. Newman [62] states that at first, one has to understand the domain and, in the case of a migration, the monolith. If that is not the case, one should first invest time to learn what the system does and look at the module boundaries before splitting it into MSs. [62] For greenfield projects, Newman [62] recommends starting with a monolith and splitting it up if it gets too big.

I had the chance to define and implement an MSA. The most challenging part was splitting the data into independent databases, and we did not come up with a solution

that uses distributed databases. Write access can often be granted to one single service, but many services often require read access, and we thought that we could not perform a network request for simply reading a single row in a table. For big companies or projects where many developers are involved, it can make sense to strictly encapsulate the services. Smaller companies or smaller projects with only two or three teams working on the project, in my opinion, need to find a trade-off between the power of a "real" MSA and a system consisting of functional services.

2.2 Micro Frontend approaches

MFs were first introduced at the ThoughtWorks Technology Radar at the end of 2016. [1] Some big companies like Amazon[4], SAP[5], IKEA[6], Zalando[7], Thalia[8], Spotify[9], or DAZN[10] already use MFs to create their web applications.

Geers [58] mentions that with the increase of the project scope and team size, knowledge silos arise when one uses a monolithic frontend the same way as when using monolithic backends. Thus, one could start to split the application into smaller parts. Multiple teams own one or more distinct MFs that are then combined to form the final page. There is currently no general definition of MFs. People often speak of the term MF but mean the frontend code only. Geers [58] describes an MFA as an architectural style where each MF includes everything from the database to the UI with one team responsible for all parts without the need for a central UI team. Peltonen et al. [68] mention that "Micro-Frontends extends the Microservice architecture idea and many principles from Microservices apply to Micro-Frontends."

One can implement web frontends in many ways. Schulte-Coerne [73] describes different options of frontend integration. One can use a frontend monolith SPA that offers strict separation between the frontend and the backend and divide this monolith into modules that can already solve some maintainability problems. [73] Another approach is the flat SPA, where one app acts as a dashboard that provides links to different standalone pages. [73] When it comes to MFs, Schulte-Coerne [73] mentions three general approaches. (i) Use links on the page to route between different pages, or (ii) use client-side transclusion that loads parts of the system dynamically from some Content Delivery Network (CDN) and inject those parts dynamically into the Document Object Model (DOM), or (iii) use iframes to include different pages into one page.

Mezallira [57] and Peltonen et al. [68] state that there are two ways to implement an MFA.

[4] https://www.amazon.com/
[5] https://luigi-project.io
[6] https://www.infoq.com/news/2018/08/experiences-micro-frontends/
[7] https://www.mosaic9.org/
[8] https://tech.thalia.de/another-one-bites-the-dust-wie-ein-monolith-kontrolliert-gesprengt-wird-teil-i/
[9] https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1/
[10] https://www.youtube.com/watch?v=BuRB3djraeM

(i) The vertical split shows only one MF per time. (ii) The horizontal split composes one view out of many different MFs.

The following sections explain the current state-of-the-art MF concepts. One can split a webpage vertically, which means to separate the application into different pages, and horizontally, which describes composing the view out of many small fragments. Section 2.2.1 explains how one can implement a vertical split, and Section 2.2.2 explains horizontal composition techniques. Both, vertical and horizontal splitting, require communication models described in Section 2.2.3. One special approach of horizontal splitting is the application shell architectural style. The case study (see Chapter 4) will make use of this approach, and Section 2.2.4 explains the concepts.

2.2.1 Vertical split: Page transition via links

The most straightforward approach to implement an MFA is to create an infrastructure where each MF is a standalone application. The distribution of the MFs works via a web server that delivers a complete page via a specific route. [47] The different MFs reference each other through hyperlinks, and one can provide context information via Uniform Resource Locator (URL) or query parameters. [58, 79] To provide a better user experience for pages that require authentication, Steyer [79] recommends using Single Sign-On (SSO) so that the user does not have to perform the authentication on every route change.

The following paragraphs describe the advantages and disadvantages of this approach, provide information on how one can deal with contracts between teams, and show how a reverse proxy can improve the UX.

Advantages The most significant advantage of the hyperlink approach is that it is straightforward. [80] Each application is a standalone application that leads to loose coupling between the different MFs and, therefore, high robustness. [58] Besides, it is possible to optimize the bundles for each domain separately. [80]

Disadvantages Exchanging one MF through another using a hyperlink requires a full page reload each time the user navigates between different pages. [47, 92] If one uses frameworks to build the individual MFs, each page transition leads to a reload and re-initialization of the framework code. [78] Both the application before and after the route change lose the application state and the page's control. [78, 92] Geers [58] mentions that it is impossible to combine two MFs into one view with the hyperlink option. Since each team needs to implement the whole page, technical redundancy (e.g., using the same code for the page header) can occur. [58]

Contract between teams Each MF team should be independent and isolated and should not know how other teams deploy their MFs and where to find other teams' MFs. One can define one well-known machine-readable location where teams provide

their internal routes (e.g., via a JavaScript Object Notation (JSON) file on a server), and other teams can access these routes via pre-defined team namespaces. [58] Uniform Resource Identifier (URI) templates[11], JSON-home[12] or the Swagger OpenAPI[13] can help to achieve this.

Server-side routing via Reverse Proxy Route changes in the browsers' address bar can be annoying for users. It is possible to use a reverse proxy that can load different applications from multiple HTTP servers based on some well-defined re-routing rules that match parts of the URL to overcome this issue and improve the UX. [58, 90, 92] Geers [58] provides two strategies for defining re-routing rules:

- Team routes only

  Assign an URL prefix (e.g., the teams' name) to each team and route each request matching this prefix to the teams' web server.

  Example: www.my-cool-site.com/**team-1**/welcome-page

- Dynamic route configuration

  Define a re-routing rule for each page of the teams.

  Example: www.my-cool-site.com/**article**/5

It is crucial to have in mind that using a reverse proxy will introduce a single point of failure into the system, and one needs people that operate this reverse proxy, which might become a bottleneck for all teams. [58]

2.2.2 Horizontal split via Composition

Another approach that one can use in combination to split the app into vertical slices is to build one page out of many components horizontally. Jackson [47] mentions that one can use a build-time integration approach to compose one big application out of many individual applications with the disadvantage that one needs to recompile the whole system if a change in one MF occurs.

Another option is to implement a transclusion mechanism via "client-side includes." [68] With transclusion, one can append or change MFs to the DOM dynamically. [68]

Big companies (e.g., Amazon, IKEA, Zalando) use a server-side composition technique where the server renders the complete page and delivers the result to the client. [68] Geers [58] and Pavlenko et al. [67] mention that it is possible to include client-side integration techniques into the server-generated markup for deferred loading so that it is

[11] https://tools.ietf.org/html/rfc6570
[12] https://mnot.github.io/I-D/json-home
[13] https://swagger.io/specification

still possible to inject dynamic parts (e.g., those fragments that are not initially visible in the user's viewport) at the client.

With server-side integration techniques like the well-proven SSI or ESI methods, one can reach an excellent first page load performance and good Search Engine Optimization (SEO) support. [58] Nevertheless, those integration methods are not always easy to configure and provide no technical isolation between the MFs. [58] Further, Peltonen et al. [68] and Geers [58] state that for an app-like UI that should instantly react to user input, a server-side integration is probably not the best choice because one has to send many requests to the server if each page is personalized for the user.

The following paragraphs describe the possible server-side and client-side integration options in detail.

Server-side: Server Side Includes SSI, which dates back to the 1990s, is an approach where one can get better load times for fetching fragments from multiple servers because of the faster internal network traffic. [58] One defines specific SSI HTML tags with an URL that a web server can identify and replace through the requested markup.

Example: `<!--#include virtual="/my/url/to/include" -->`

It is possible to include a timeout and error fallback that the server renders in case of a failure. [58] There exist implementations (e.g., NGINX[14]) that can load the markup for multiple SSI tags in parallel.

Server-side: Edge Side Includes A similar approach to SSI is using ESI [76, 85]. The Varnish Reverse Proxy[15], for example, implements ESI.

Example: `<esi:include src="https://my.example/fragment" />`

The ESI standard supports error fallbacks by default but has no timeout support. [58]

Server-side: Other options "The response time for the complete markup, also called time to first byte (TTFB), is defined by the time it takes to generate the page markup and the time of the slowest fragment." [58] A server implementing the SSI standard does not start sending the composed markup until it resolved every SSI tag request. The same behavior applies to ESI. Frameworks like Tailor or Podium can help to overcome this issue because they support streaming, allowing to start sending available DOM fragments as soon as they arrive. [58]

[14] https://www.nginx.com/
[15] www.varnish-cache.org

Zalando Tailor[16] is a server-side composition library supporting fallbacks for slow or faulty fragments out of the box and has built-in streaming support. The Tailor framework further has some extended functionality like asset handling via HTTP headers.

Podium[17] is another server-side composition library that improves the Tailor concepts by using a manifest where one can define where to find the content, the fallback DOM, or the assets.

Client-side: iframe embedding The presumably easiest approach to include markup from different MF teams at runtime is the integration via iframe embedding. [58, 68, 75, 79, 90, 92] Each iframe gets a standalone hosting environment from the browser that is fully encapsulated from the rest of the page. [92] Communication between iframes is possible via the `window.postMessage`[18] API. [92]

Steyer [79] recommends using the iframe integration approach if one needs high isolation of the embedded fragment or embedding a legacy system or systems from other vendors.

The integration of fragments via iframe is probably the fastest way to build a client-side integrated MSA. [58] With the iframe approach, one can independently develop each module and use different technology in each MF. [90, 92] The iframe tag is a web standard and works in every browser. [58] The most important aspect of using iframes is that one gets strong and technical isolation that has a huge benefit on security because it is not possible that scripts inside the iframe leak vulnerable code. [47, 58, 80, 90, 92]

Wang et al. [90], Yang et al. [92], and Jackson [47] mention that using iframes leads to large bundles because it is not possible to share dependencies between the iframes due to the strong isolation, which leads to higher download times. Besides, Geers [58] mentions that iframes bring a considerable performance overhead because of the isolation that needs extra memory and Central Processing Unit (CPU) power. SEO is difficult when using iframes because the crawler would index each iframe embedded fragment as a separate page. [58] These separate pages also hurt the page accessibility. [58] Assistive technologies like screen readers have a hard time processing a page with iframes because each iframe is a separate page with its own `<head>` tag. [58] Wang et al. [90] mention that nested iframes are problematic. Geers [58] reports that iframe usage leads to probably unwanted layout constraints and coupling between MFs. The outer document that embeds a fragment via iframe needs to know the iframe's exact height to maintain the intended fragment style and size (except one uses additional libraries like the iframe-resizer[19] for auto-resizing iframes). [58]

Client-side: AJAX Jackson [47], Geers [58] and Hasselbring et al. [41] mention the runtime integration of features using client-side Asynchronous JavaScript and XML

[16] https://github.com/zalando/tailor
[17] https://podium-lib.io/
[18] https://html.spec.whatwg.org/multipage/web-messaging.html
[19] https://github.com/davidjbradshaw/iframe-resizer

(AJAX). One can create a separate bundle for each MF that the client can load and mount when it is needed. [47]

One advantage of this approach is that one can pre-load shared dependencies to reduce the bundle size of the separate feature bundles. [47] Geers [58] states that the AJAX approach results in a natural document flow where all the content is available in one DOM, enabling excellent SEO and accessibility support. Using client-side JavaScript (JS), one can implement progressive enhancement and flexible error-handling, such as providing fallbacks to the requested standalone MF or the possibility of showing alternative static content instead of the dynamically loaded fragment. [58]

The asynchronous loading of the different MFs has the negative effect that the client renders the fragments with some delay that can lead to "site jumps and wiggles." [58] Besides, each user interaction that triggers an AJAX call resides in a server request that can lead to a bad UX, especially on poor network conditions. [58]

With the client-side integration via JS, one does not have any technical guarantee for isolation. [58] Geers [58] recommends namespacing everything that belongs to one team, including "cookies, storage, events, or unavoidable global variables," as well as styles and to wrap scripts in Immediately Invoked Function Expressions (IIFEs)[20]. One possible option to lower boilerplate code for AJAX call is the use of wrapper libraries like h-include[21]. [58] A project setup with automated namespacing rules can help to avoid collisions at runtime. [58]

Geers [58] states that one should use the AJAX approach when a system already provides server-generated markup, but one must keep in mind that those JS calls introduce lots of interactivity. Working with the local state can be tricky, and a high number of reloads can annoy users using the site with high network latency.

Client-Side: Web Components The term Web Component is an umbrella term that combines CustomElements, ShadowDOM, and HTML templates. [58, 92] Web Components behave like standard HTML elements and can be defined via

`window.customElements.define`.

Using Web Components allows developers to declare domain functionality with business logic encapsulated in the component. [58]

Each Web Component offers a technology-neutral interface with the lifecycle methods

`constructor`, `connectedCallback`,`disconnectedCallback`, and `attributeChangedCallback` that developers can use to run custom code when those events occur. [58] An example of Web Components is AngularElements[22].

[20] https://en.wikipedia.org/wiki/Immediately_invoked_function_expression
[21] https://github.com/gustafnk/h-include
[22] https://angular.io/guide/elements

The ShadowDOM API is a central part of web components. One has to enable the ShadowDOM with the `attachShadow()` function that initializes a `shadowRoot` property, which holds a reference to the opened ShadowDOM. [58] Using ShadowDOM enables "to isolate a subtree of the DOM from the rest of the page," which leads to the fact that "no prefixing or explicit scoping is required." [58]

This isolation via the ShadowDOM makes the app robust since it enables an iframe-like technical isolation that prevents global styles from leaking into MFs. Web Components are a widely implemented web standard. [58] They allow developers to create reusable custom elements with encapsulated functionality. [92]

A current drawback of Web Components is that not all browsers provide full support. [92] Specifically, the ShadowDOM API is problematic, and existing polyfills are heavy and rely on heuristics. [58] Besides, Web Components require JS to work, and there is no possibility to declare the ShadowDOM via HTML. [58] Web Components are similar to client-side AJAX composition because each "JavaScript bundle has to load first and register the components in order for the DOM to load." [92]

Geers [58] recommends using Web Components when one wants to build an app-like application with lots of interactivity.

2.2.3 Communication

MFs need to communicate with each other, although Geers [58] mentions that lots of UI communication can show bad project boundaries. Steyer [78] states that one possible way to communicate between MFs is via the backend. Each MF can send requested changes to the backend that persists those changes, and other MFs can request this data again. It is not always possible due to performance reasons to communicate via the backend but to communicate on the client-side. The following paragraphs show different communication strategies between MFs. The described options do not uniquely apply to those strategies but can also be useful for the other shown strategies.

Page-to-page communication The simplest form of communication is vertical communication between pages. One can use simple links with URL or query parameters to exchange data between sites. [58] This approach is easy to implement, allows deep linking, URL sharing, and the MF load order is irrelevant. [78]

Parent-to-fragment communication An MF that includes another MF can communicate with its child via simple HTML attributes. One can achieve unidirectional data-flow by simply changing the HTML attributes of the child element. [58] The child fragment needs to subscribe to the `attributeChangeCallback` in order to get notified about attribute changes. [58]

Fragment-to-parent communication Communication from a child fragment to a parent MF is possible via the CustomEvents API. [58, 78, 92] There, the child fragment emits an event on which the parent component can react.

Steyer [78] and Geers [58] describe sending events using an event-bus approach via built-in browser events. Other fragments can listen to those events dispatched by the MF. An MF can dispatch an event the other fragments can listen on. Geers [58] recommends dispatching the events via `element.dispatchEvent` instead of `window.dispatchEvent` and let the event bubble through the DOM. Dispatching events on the element has the benefits that the event origin (`event.target`) is maintained, and that a receiver of an event can stop further bubbling through the DOM via `event.stopPropagation`. [58]

Fragment-to-fragment The most straightforward way to communicate between two MFs on the same hierarchical level is direct communication. Direct communication means that one could search the DOM for the element to which a message should be sent and invoke a function that this element provides directly. Geers [58] recommends not using this option because it will create very high coupling between those fragments.

Another possibility to achieve fragment-to-fragment communication is the orchestration via the parent. [58] This communication style combines fragment-to-parent and parent-to-fragment approaches. The disadvantage of using parent orchestrated communication is that every change regarding the communication interface requires the change of MFs owned by two teams. [58]

The Broadcast Channel API [23] is another communication option where an MF can publish a broadcast channel, and other MFs can subscribe to this channel to receive messages. [58] The Broadcast Channel API enables an application to exchange messages across browser windows, tabs, and iframes. [58] This approach is very powerful, but one must be careful with exchanging complex data because it can introduce coupling. [58] Geers [58] recommends to keep the payload to a minimum and use the Broadcast Channels for sending notification events only without transferring data.

Other communication mechanisms Sometimes it can be necessary to have some globally available context information about the currently active user like GPS coordinates, the preferred currency, or the logged-in state. [58] One can provide this information via HTTP headers or cookies that a backend service or proxy can set on each request. [58] Other options are to use a global JS API or browser storage to share this global information. [58]

2.2.4 Application Shell

Pavlenko et al. [67] state that with "the growth of usage of mobile devices and poor coverage of high-speed mobile internet, any reload of the page should be avoided." One

[23] https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API

can avoid page reloads by replacing or loading the content dynamically into the DOM on the client-side. One can implement this dynamic page reload with the previously explained concepts of client-side horizontal composition techniques. Another prominent approach is using SPAs. Mikowski and Powell [59] explained that SPAs are written in JS, where one browser environment displays dynamic content to the user without reloading the page during use.

A SPA is a standalone, monolithic application usually delivered as a single bundle. [67] Existing lazy loading techniques allow splitting a SPA into multiple modules (chunk splitting) that one can deploy as separate files on one CDN that a browser can fetch when needed. [67] This split into multiple chunks happens at compile time from one monolithic codebase.

SPA frameworks like Angular, Vue.js, or React.js currently do not provide options to separate parts of the monolithic application into standalone applications that one can deploy to different CDNs from which a browser can request the chunks.

The application shell is the proposed solution that solves deploying multiple standalone MFs and loading them into one application at the client-side. [58, 68, 75, 78, 80] Those MFs can internally use any framework. Geers [58] defines the name Unified SPA that describes an application shell for SPA MFs. He recommends using a Unified SPA when users of an application often need to switch between multiple parts of an application (MFs owned by different teams) and where productivity is more relevant than the initial page load time.

The following two paragraphs explain what an application shell is and how to implement the routing between the different MFs.

What is an Application Shell? The application shell is a standalone application that operates as the parent application for all included MFs. [58] This shell application receives all routing requests from the browser and bootstraps the currently required MF into one `index.html` file by dynamically creating and inserting the MFs root elements into the DOM. [78]

The shell has four essential parts described by Geers. [58] (i) Provide a shared HTML document that will be the root document for all MFs. (ii) Handle client-side routing by mapping URLs to team pages. (iii) Create and Render the matching MF into the root DOM. (iv) (De-) initialize the previous/next page on navigation.

Geers [58] states that since the application shell will be the main application that handles every other MFs, it should be as simple as possible. The shell can handle some additional tasks like collecting and providing context information (e.g., preferred language, users country, authentication state) or common requirements like, e.g., performance monitoring or metadata handling. The MF teams only need to publish their URL mappings that the application shell needs to map to the different MFs.

Using the application shell as an integration layer for multiple MFs provides a great UX. [78] This integrated solution does not require page reloads and preserves the application state during application usage. [80] An MF change or adding a new MF only requires to update the shell. [58]

The drawbacks of the application shell approach are that one needs an extra meta-framework. [78] This meta-framework provides the additional infrastructure code that one needs to load and switch between MFs. [80] Further, using SPA frameworks to implement the MFs can lead to an increased download size of the overall app because each MF has to load its framework code. [80] Nevertheless, there are ways to share the framework code between multiple MFs (e.g., using Webpack 5 Module Federation [81]), introducing coupling between MFs that need to agree on shared dependencies. Developers need to keep in mind that an application shell will become a single point of failure, where an error in one MF can affect the whole system. [58]

Routing inside the Application Shell Geers [58] describes two navigation types. (i) Hard navigation, where the client loads the required HTML from the server on page transition, and (ii) Soft navigation, where the client performs the page transition via client-side rendering. [58] Using an application shell enables two types of client-side routing. [58] (i) The application shell can use hard navigation between MF teams, and each MF internally uses soft navigation. [58] (ii) A Unified SPA shell can dynamically load the MFs on demand, which leads to soft navigation between the MFs.

One can implement this routing via the HTML5 history library. [67] Each team should have a prefix, and the application shell can determine the required MF based on that prefix. [78] This routing solution enables multi-level routing where the application shell only handles the first part (team prefix) of the URL, and the currently active MF handles its internal routing. [58]

2.3 Related Work

The MF topic is a very new research field, and only a few papers exist that investigate that topic. This section describes works related to an MFA that already present solutions for migrating from a monolithic frontend to an MFA.

Pavlenko et al. [67] provide a report of a case study on the implementation of an Education Hub System, a software that "aggregates online courses from different online course providers to serve as a single entry-point and search engine for users." They use the BFF pattern (see 2.1.2) to provide a single entry point for the frontend. For the frontend implementation, Pavlenko et al. [67] used the single-spa[24] framework for building their MFA and ReactJS[25] as the SPA framework for all their MFs. This paper acts as a basis for implementing the second case study in Chapter 4. Nevertheless, Pavlenko et al. did

[24] https://single-spa.js.org/
[25] https://reactjs.org/

not perform any performance measurements, and used only one SPA framework for all MFs. This thesis includes and analyzes those missing parts.

Mena et al. [56] present a Progressive Web App (PWA) with an algorithm that can select components based on the user's context. This context is, e.g., whether the user is using the app on a desktop or a mobile device or the user's location. [56] The frontend sends this context information to a backend service that uses a "Component Selection Algorithm" to select appropriate components for the requesting user. [56] The component service sends those components to the frontend, which then composes a view of those components. With their solution, they achieve showing the same data from the backend in two different variations on the frontend (e.g., show data on a map or as a list), based on the user's device screen size. In their paper, Mena et al. [56] do not describe any technical aspect of their solution on how they implement, deliver, and compose the components.

Yang et al. [92] built an MF-based Content Management System (CMS) using the Mooa[26] framework. The Mooa framework is an MF framework based on the single-spa framework and Angular, optimized for iframes. They use a server for storing the application configuration. This configuration file contains the locations for the separately deployed MFs. When deploying a new version of an MF, they only have to point the `apps.json` file of the main application to the latest configuration file to inform the main application about the new MF deployment. [92]

Wang et al. [90] provide research on an Educational Management Information System. They use iframes to include legacy code and WebComponents for new features. [90] The current version of each MF is stored in a configuration file that the parent application loads when composing the system.

Both works ([92] and [90]) use a configuration file for storing the MF deployment information. This thesis builds upon and expands this configuration file idea by deploying it to a separate server and changing it on MF updates (see Section 4.5).

Hasselbring and Steinacker [41] report on migrating a monolithic architecture at Otto.de towards an MSA. They use so-called verticals (fragments) to compose the shop's pages. [41] Client-side AJAX acts as the leading technology for integrating verticals that are not primary content or initially invisible fragments. [41] (e.g., the shopping cart preview "that is included on almost every page") Server-side ESIs (see 2.2.2) are used to integrate primary content at the server-side. [41] This integration is well-suited for content-rich applications. This thesis does concentrate on highly interactive applications that would not profit from such a technique.

None of those works evaluated the performance of their MFA. This thesis provides an MF performance analysis and compares MFs with a monolithic system. This performance analysis also includes an evaluation of an MFA implementation that uses multiple different SPA framework in the MFs. Furthermore, Section 5.4.2 evaluates whether and how an MFA can help to speed up the development and deployment. The thesis also shows how

[26] https://phodal.github.io/mooa/

one can expand the configuration file approach by Yang et al. [92] and Wang et al. [90] by using import maps and an import map deployment tool.[27]

[27] https://github.com/single-spa/import-map-deployer

CHAPTER 3

Adoption of Microservice Concepts and Principles to Micro Frontends

MFs are closely related to MSs. Mezzalira [57] already showed how one could adopt the seven MS characteristics from Newman [62] to MFs. However, there are many more principles and characteristics than those seven described by Newman. Section 2.1 shows a summary of MS features.

Currently, there are no papers available (except for the book preview by Mezzalira [57]) that looked at how one can adopt the MS knowledge to MFs. In this chapter, the thesis tries to fill this gap. The thesis uses the findings from the literature review in chapter 2 and combines these with information from other scientific sources and grey literature (e.g., blog posts, magazine articles). The information about MSs and MFs was then compared and set against each other to gain knowledge for MFAs.

The following Section 3.1 explains MF characteristics found in the (grey) literature, followed by Section 3.2 that presents a table on how one can transfer the MS features to MFs.

3.1 Micro Frontend Characteristics

In 2020, only very few research papers concerning MFs were available. Yang et al. [92] stated in 2019 that the concept of MFs is still in an exploration stage and not mature enough to use them in productive systems. Peltonen [68] performed a literature review in July 2020 and found only one peer-reviewed conference paper and 41 sources from grey literature (e.g., articles, blog posts, videos, books, and podcasts). In the following

section, the thesis analyzes and reports the findings from both, peer-reviewed conference papers as well as grey literature.

Geers [58] wrote the book Micro Frontends in Action, which covers many aspects of MFs. Mezzalira [57] currently works on a book called Building Micro Frontends, which was only available as a preview version when writing this thesis. Not the whole content of the book was available. Therefore, the thesis includes just parts of the book.

Springer [75] wrote a short article on how one can adopt the benefits of MSs to MFs. Harms et al. [40] present guidelines on applying frontend architectures and patterns to an MS-based system.

Furthermore, it is essential to look at the problems of frontend implementations that use the monolithic architecture pattern. [70] Villamizar et al. [86] and Balalaie et al. [4] evaluated the monolithic and the microservice architecture patterns focusing on deploying web applications in cloud environments.

The following sections are structured as follows:
Section 3.1.1 shows problems that arise when one uses a monolithic approach for building big frontends. Section 3.1.2 shows the general principles of MFs. Available MF approaches are shown in Section 2.2. Section 3.1.3 describes the challenges that arise when using MFs. Section 3.1.4 describes when one should use an MSA, and Section 3.1.5 describes when using MFs is not recommended.

3.1.1 Monolith problems

Dragoni et al. [26] state that "a monolith is a software application whose modules cannot be executed independently." This monolithic architecture [70] is the leading architecture that companies use when they develop applications. However, at some point, with increased project size, a monolithic architecture is no longer sufficient. The following paragraphs show common problems that researchers found when analyzing monolithic architecture.

Maintainability The maintainability [20,67,90,92] of the software system is the biggest problem when the system gets more and more features. The complexity increases [67, 86] because the code base gets more extensive [20], and it becomes difficult to understand the code. [4] Pavlenko et al. [67] report that those many LOC often lead to a "dependency hell" where it is tough to understand how the systems interact with each other.

Deployment Monolithic systems bring many issues concerning the deployment. [67,92] Villamizar et al. [86] mention that with monoliths, a release requires a restart of the whole system, which leads to a bad UX. Wang et al. [90] confirm this issue and state that each change requires a re-deployment of the whole system, leading to high downtimes during the system reboot and, therefore, an increased deployment time, even for small updates on some insignificant modules, because it is necessary to build and deploy the entire codebase on every change. [4, 57, 67] Mezzalira [57] reports that changes in one

part of the system can introduce new bugs or even break some interfaces that affect the whole deployment. It becomes more complicated to guarantee that all parts of the system work as expected when one adds features or implements a simple change of existing modules. [86] A monolith is always a single point of failure. [86]

Scalability Many research papers [4,20,67,90,92] report scalability issues on monolithic systems. In the frontend, different from the scalability of MSs, scaling does not mean to run more instances in parallel but instead refers to an increased compile time due to the large file size, which has adverse effects on the download time. [20]

Innovation For companies, it is crucial to stay up to date and be innovative to meet user requirements. Monolithic systems have a significant impact on this innovation. Companies cannot easily switch the technology used in a legacy system. [67, 86] Balalaie et al. [4] state that one makes a long-term commitment to the technology stack with the decision to use a monolith.

3.1.2 Micro Frontend principles

MFs try to solve the problems of frontend monoliths (see Section 3.1.1). The following paragraphs show the main principles of MFs that current research tries to solve with different solutions.

Maintainability Steyer [78] mentions that the primary aspect for the use of MFs is maintainability. Geers [58] and Pavlenko et al. [67] and Hasselbring and Steinacker [41] coincide when they describe that the loose coupling and the small codebase make an MF easier to understand. An MFA should have a shared-nothing architecture, where one accepts "redundancy in favor of more autonomy and higher iteration speeds." [58]

Scalability Geers [58] mentions that technically, scaling does not apply to the frontend code. MFs scaling refers to scaling software regarding its complexity and therefore scaling of the development teams, which enables faster feature development. [41]

Adopted to business needs An MF should be modeled around business domains. [68] It should represent the business domain in order to make sure that the team members feel responsible for the features they build. [57] The agile approach reduces the need for slow and formal inter-team communication and helps developers to deliver business value as soon as possible without waiting for other teams. [58, 78]

Fault Tolerance Fault tolerance and failure isolation are key aspects of an MFA. [58,68] An MF should be resilient in a way that each "feature should be useful even in the case if JavaScript failed or did not run." [58]

Automation Hasselbring et al. [41] and Mezzalira [57] state that automation is the key to DevOps success. Peltonen et al. [68] and Yang et al. [92] also mention a significant need for a culture of automation with CI/CD pipelines. "Solid automation pipelines will allow our micro-frontends projects to be successful, creating a reliable solution for developers to experiment, build, and deploy." [57] This automation is especially critical when one uses horizontal composition (see Section 2.2.2) because it can lead to many (tens or hundreds) of artifacts. [57]

Mezzalira [57] recommends using the Infrastructure as Code (IaC) principle for new pipelines. This IaC principle allows each MF team to create the pipeline with the companies' best practices easily. Therefore, one can keep feedback loops as fast as possible, which leads to lots of iterations and constant feedback in, at most, minutes. [57] The pipelines should be visualized so that every team member can see their pipelines constantly (e.g., on a big screen in the office) so that teams can review and adapt their pipelines as soon as they fail or begin to take too long.

Each team should own separate pipelines that enable a fast and independent release of updates. [79] These separate releases reduce the update's scope [79] and reduce the downtime because of faster pipelines. [67]

Autonomy and Independence When one looks at existing papers and tries to find principles that one should keep in mind when developing an MFA, nearly every author mentions independent development and especially independent deployment as a key attribute of an MF. [57, 58, 68, 75, 78, 90, 92] Autonomous or autarkic teams decoupled from other teams enable this independence. [58, 78] Each team owns one MF that is self-contained [58] and allows them to make decentralized decisions. [68] This decentralization of decisions lets each team decide on the technology stack and "micro architecture" they want to use. [58, 78, 79, 92] With the possibility to freely choose a different technology in each architecture, one enables companies to support new technologies and avoids the need for a rewrite of the complete frontend if one wants to use a different framework in some parts of the system. [58, 90] The autonomous codebase [57] enables teams to test individually [92] and hide implementation details. [68] Teams need to define contracts for their MFs that other teams can use to interact with hidden implementations. [68] To avoid conflicts when one combines the different MFs into the complete system, Geers [58] and Yang et al. [92] recommend to isolate the team's code and use a prefix for each team to avoid runtime conflicts and collisions with shared runtimes, global variables or Cascading Style Sheet (CSS) class names. This prefixing further allows identifying the ownership of parts of the code. [58]

3.1.3 Micro Frontend Challenges

MFs are not the only solution that developers will use in the future. They bring lots of challenges one has to tackle. The following paragraphs briefly explain common problems developers will face when implementing an MFA.

Micro Frontend Teams Steyer [77] and Springer [75] state that introducing different teams that probably use different frameworks to build an application adds additional complexity to the development process and the resulting system. This system complexity leads to a steeper learning curve for new developers, even if they already know the used frontend framework. [79]

Although good team boundaries will reduce inter-team communication, teams must create contracts for their application to share with other teams. [93]

Geers [58] and Pavlenko et al. [67] state that the initial setup is challenging and leads to higher organizational complexity. Furthermore, in case of a bug or negative web page performance, it can become challenging to find the team responsible for a bug or lousy performance. [58]

Other aspects are cross-cutting concerns like analytics, monitoring, error tracking, or running an internal npm registry. [58] One has to define which team is responsible for those shared topics or even decide to have specialized component teams like the Spotify Infrastructure Squads[1]. [58]

Nevertheless, Geers [58] states that "when done right, the boost in productivity and motivation should be more significant than the added organizational complexity."

To ease the initial setup, Mezzalira [57] recommends providing an MF blueprint (e.g., a Command Line Interface (CLI) tool) to scaffold a new MF project that already includes common libraries (e.g., a logging library) or a sample pipeline configuration.

Isolation and Independence Isolation of MFs is not always as easy as running different MS in separate processes. One has to think of solutions on how to avoid collisions in the shared browser environment. [93] MFs can bring considerable heterogeneity into a system by allowing each development team to freely choose their preferred technology. This freedom can become challenging for developers, especially if there is a need that a developer switches between teams. Geers [58] recommends discussing the level of freedom an MF team has and possibly limiting the allowed technologies to a subset of all available technologies. [58] It could become a nightmare if each MF setup is different from others without the possibility to provide templates for new projects. Not sharing but duplicating code can also lead to problems because one can not easily fix a bug of a function implemented by different teams. [58]

Fault Tolerance and Error Handling Geers [58] mentions that it can become tricky to find the error origin in an MFA. Finding memory leaks due to inadequate cleanup can also become a challenge when the browser dynamically loads several MFs into one big HTML document.

[1] https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1/

Performance and Dependency Management Dynamic resource loading requires more frontend code than a monolithic UI implementation, and it is essential to keep an eye on web performance. [58] Geers [58] recommends using the performance budget concept where one can, e.g., define a maximum bundle size for the individual MFs.

Springer [75] mentions that loading multiple MFs can lead to higher transfer rates because each MF includes the required libraries. Therefore, it is crucial to think of dependency management and options "how to avoid one library to be loaded more than once." [93]

Compiling One can deploy an MS as a single service, e.g., as a container. [75] In an MFA, all MFs have the same client-side browser as a runtime environment. [75] Therefore, one needs a strategy on how to compile and deploy an MF. Using server-side integration techniques allows to inline script and style tags into the MF markup, leading to redundant script tags that the browser executes multiple times, which can have unforeseen side-effects and increases the CPU load. [58] One needs to decide on bundle granularity, from an all-in-one bundle over team bundles to page and fragment bundles. [58] Geers [58] and Steyer [78] recommend using just one bundle for one MF. Loading multiple bundles from one server was a performance problem with the HTTP protocol, but with HTTP/2, this is not an issue anymore. [58] The HTTP/2 "protocol reduced the overhead cost of loading multiple resources from the same domain. Its built-in multiplexing and server push features removed the need to manually inline assets into the page, which reduces complexity in the application and is also great for cacheability." [58]

Asset handling A possible option to decrease the MF's download size is to reduce or reuse vendor libraries. [58] One can share a library as a global script tag that makes this library available in all MFs. [58] Other possible solutions for sharing libraries are:

- Asset referencing via RequireJS[2]
- Asset referencing via CommonJS[3]
- The proposed new web standard Import maps[4], which maps a unique name to an absolute URL. [58]
- Webpack externals
- ES Modules with Rollup.js[5], a web standard without the need for extra libraries that allows dynamic loading of required vendor bundles [58]
- "ngx-build-plus" Tool [78], which produces a UMD bundle that allows splitting an MF into library code and custom code

[2]https://requirejs.org
[3]http://www.commonjs.org
[4]https://github.com/WICG/import-maps
[5]https://rollupjs.org/

- Webpack DLL plugin[6]

These options have the problem that teams need to agree on a global version or accept that the same framework code in different versions needs to be downloaded, which increases the download size.

Another problem is caching libraries or MF code. Each time a bundle changes, one needs to make sure that the client fetches the library's correct version. Geers [58] recommends the use of cache busting, where each filename includes a hash. This naming strategy requires to update all references to the new bundle name. One can reference the bundle via the non-hashed name and use a server redirect that redirects a request to the hashed version. Using a server-side rendering approach, one could fetch the latest bundle from the teams' server via SSI or ESI.

Another challenge can be to keep the markup and available bundle versions in sync. [58] Using a load balancer, it is possible that on an update, a request arrives at an already updated version of a container and an "old" container receives the fetch request of the required dependency, which does not yet know the new bundle version. [58] Possible solutions are sticky sessions in the load balancer that send all requests from one origin to the same container or a global CDN that one needs to update before the actual markup so that it serves old and new assets. [58]

Style Wang et al. [90] state that using an MFA should not harm the UX. That includes the UI's consistency, and one has to think of "how to manage common styles and make sure that UX is consistent." [93]

Jackson [47], Wang et al. [90], Steyer [80], and Geers [58] recommend using a shared UI component library that defines the basic look and feel of a system and provides a consistent look for the whole application.

Testing An MFA consists of many separate parts that need to interact with each other. One can test each MF individually to control the internal correctness. Nevertheless, there is a high need for a solid testing strategy with End-to-End (E2E) tests to ensure functional integration of the different MFs. [57] When using an application shell, Mezallira [57] recommends making the shell team responsible for the E2E tests because it can access all MFs.

3.1.4 When to use Micro Frontends

The web frontend is the prime interaction surface for users. [58] However, using MFs is "not about the software. It's about the people designing and building it." [58] In the best case, users do not recognize any adverse effect when one migrates from a monolithic architecture to an MSA.

[6] https://webpack.js.org/plugins/dll-plugin

An MFA is best for the web. [58] A native app is monolithic by design, with the possibility of splitting it into vertical REST APIs and a horizontal monolithic UI layer. Thus, the MF approach does not fit into the native app world.

Geers [58] states that the most significant difference to a monolith is the team structure. Like MSs, one would move towards an MFA with an increasing project scope. At some point, large groups lead to a lot of communication overhead and, therefore, complicated decision-making. [58] MFs try to solve this problem.

The team size should be between five and ten due to the Two-Pizza Team rule suggested by Amazon Chief Executive Officer (CEO) Jeff Bezos. [17]

3.1.5 Reasons against Micro Frontends

Springer [75] states that one "might not need Micro Frontends" and provides some preconditions to fulfill when thinking of migrating towards an MSA. The application needs to have extended functionality, which means to be involved and big enough. [75] Otherwise, the overhead that one has to invest in MFs is not worth the time. [75] When thinking of implementing an MFA, one should consider that it needs separated development teams with a high degree of autonomy, e.g., taking the right tool for the job. [75] Each team should own a database, the backend, and the frontend; and be responsible for the implementation, the release, and the maintenance. [75]

Steyer [78] also mentions that it is essential to make sure that one needs MFs and that they solve occurring problems. On a survey with five companies about MFs, one company decided not to use an MFA because they have just one agile team without needing a separate deployment. [78] Therefore, an MFA would introduce too much additional complexity for development and deployment. [78]

Harms et al. [40] state that the "use of a UI monolith should be considered when UI changes occur frequently across service boundaries or if the application's domains cannot be unambiguously assigned to the microservices."

Geers [58] coincides when he says that one should not use MFs with few developers where communication is not an issue or if one has an unclear domain or overlapping features.

3.2 Adoption of Micro Service principles in Micro Frontends

In this chapter, the thesis shows how one can adopt the MS principles and characteristics to MFs. To do that, it was required to analyze and group the findings from Section 2.1 and map the to the principles and challenges from Section 3.1. Table 3.1 shows those adoptions. The table has the following structure.

In the first column, *Features*, one can see a categorization into different features closely related to the headings in Section 2.1 and 2.2. The second column, *Microservices*, contains

the detailed principles or characteristics of the feature, which belongs to MS concepts and requirements. The *Relationship* column defines the characteristics' relationship from MSs to MFs. There are three possible characters in this column.

- **+** The *"+"* sign means that the given MS principle does thoroughly apply to MFs
- **-** A *"-"* sign shows that one can not adopt the MSs principle in this row to MFs
- ***o*** The *"o"* character highlights that one can partially apply the given MS principle to MFs.

Column 4, *Difference in Micro Frontends*, shows a brief explanation of why one can not apply the MS characteristic to MFs or briefly explains the difference to MFs.

The following paragraphs describe the principles from table 3.1 that have a - or *o* sign in the *Relationship* column. One can find the row's explanation by matching the paragraph name and the letter in the table's *Ref* column.

a Availability concerning MSs means that one can scale the number of services according to the system's current load. MFs do not require such a property because the client-side will perform most of the work.

b Scaling services independently is a significant benefit of an MSA, which helps to scale only those parts of the system with a higher load. An MFAs, as described in paragraph a, does not have the requirement to scale services. Scaling can also mean to be able to scale the number of development teams. This team scaling applies to MSs and MFs.

c Preventing performance loss by scaling the system depending on the load is only partially related to MFs. As mentioned in paragraph (a), most of the code runs in the client's browser. When one uses server-side rendering, scaling with regards to prevent performance loss can probably be a required MF characteristic if the server has to execute lots of logic to construct the pages.

d Auto-scaling MSs is a crucial feature, which, as described in paragraphs (a) and (c) does not apply to MFs.

e In an MSAs, it is possible to run multiple instances of the same service, possibly with different versions, to quickly route the incoming traffic to another service if one instance fails or a newly deployed version contains a bug. An MF is only a chunk of code that one can request in order to use this MF. Therefore, restarting an MF on the server is impossible, but one can deploy multiple versions of the same MF and provide an older MF version as a fallback for newer versions.

Feature	Microservices	Relationship	Difference in Micro Frontends	Ref
Scalability	availability	-	does not apply to MFs	a
	independent scaling	o	scaling refers to scaling of teams, not to components	b
	prevent performance loss (scale depending on load)	o	Server-side: maybe scaling required Client-side: code executed in the browser	c
Maintainability	reduce complexity	+		
	increase pace of change	+		
	loose coupling	+		
	high cohesion	+		
	increase code understandability	+		
	small codebase	+		
	limit bug scope	+		
	extendable	+		
	reusable	+		
	Choreography over orchestration	+		
	no-centralized control	+		
	share-nothing philosophy	+		
DevOps Support	independent development/test/deployment	+		
	auto-scaling	-	scaling does not apply to MFs	d
CI/CD	test and change parts of the system automatically	+		
	short re-deployment times	+		
	culture of automation	+		
	needed because of high number of services	+		
	separate test and production system	+		
Reliability	improved fault tolerance	+		
	reduce impact area of failure	+		
	quick restart/replace of faulty part	o	replace faulty part possible	e
Independent Technology	support for different technologies	+		
	different internal structure	+		
	avoid vendor lock-in	+		
	Possibly hard sharing of libraries	+		
	Requires technology guidelines	+		
Responsibility Separation	well-defined boundaries	+		
	self-contained	+		
	closed system	+		
	hide implementation details behind API	o	hide implementation details behind fragment	f
	no commonly accepted size	+		
	modelled around business concepts	+		
	bounded context	+		
	split if too large	+		
	Split if not dealing with single responsibility	+		
	decentralized data governance	o	decentralized API governance	g
	technology agnostic APIs	-	need to decide on technology (e.g. Web Components)	h
Security	increased effort because of distributed system	+		
	additional protection (e.g., for services with critical data)	-	protection does not apply to frontend	i
Innovation	limit threat of wrong technology choice	+		
	develop and deploy new features fast	+		
	reduce time-to-market	+		
	use elasticity and pricing model of cloud environments	o	not as important as in MSs	j
Communication	dumb middleware	o	applies to an application shell Maybe difficult when using ESI or SSI	k
	dumb pipes	+		
	smart endpoints	o	smart fragments/pages	l
	simple components with clean interface	+		
	simple messaging	+		
Organization	Conway's Law/Inverse Conway Maneuver: structure teams and organizations according to architecture	+		
	team responsible for product not working on project	+		
	team responsible for complete stack (including pipeline, ...)	+		
	cover complete software lifecycle	+		
	efficient small and independent teams	+		
	reduce coordination and communication overhead	+		
	Architecture as a long-term strategy (not project)	+		
Code Sharing	Internal Open Source	+		
Patterns	API Gateway	-	usually not used, MF should have its own service	m
	Adapter MS / BFF	+		
	Circuit Breaker	o	fallbacks if fragment/page can not be loaded	n
	Service Discovery	o	find MF bundle	o
	Load Balancer	-	not necessary	p
	Monitoring	+		
	Logging (Log Aggregator)	-	logging already only in browser	q
	Profiling	-	not required in frontend	r
	Health Management Service	-	not necessary	s
	Configuration Service	+		
	Correlation ID Pattern	+		
Development Templates	define standards, provide templates for project start	+		
Challenges	Initial Setup	+		
	Migration of legacy system	+		
	Finding service boundaries	o	decide on horizontal or vertical split (including boundaries)	t
	Communication between services over network	-	communication of fragments/pages in same environment	u
	additional configuration effort	+		
	More complex effort estimation	+		
	Need for skilled developers	+		
		-	Shared Runtime	v

Table 3.1: Adoption of MS Concepts and Principles to MFs

f An MS commonly exposes its features via an API and hides the implementation details behind it. When looking at MFs, one has to differentiate between vertical and horizontal composition. Vertical composition, which divides an application into multiple pages owned by different teams, exposes a complete page with hidden implementation details and possibly configuration options (e.g., via query parameters). A horizontal MFs, also called fragment, is self-contained and exposes, e.g., HTML attributes for configuring it and emits events on which the parent component can listen (see 2.2.2).

g In an MSA, each service should own a separate database. An MF does not directly communicate with a database to get information. Instead, an MF requests the required data from an MS. It is recommended that each MF only communicates with its assigned backend MS.

h MSs are entirely technology-agnostic. An MSA only requires that each MS exposes its API via a commonly accepted, technology-agnostic interface like a REST API. An MFA can require to deploy each MF for a specific technology (e.g., Web Components, single-spa parcels). Nevertheless, the technology behind this "wrapper" technology can be different for each MF.

i An MS can implement special protection mechanisms if it deals with critical data. This protection does not apply to the frontend and is therefore not relevant to MFs.

j A browser loads the MF code and performs possible computations on the client-side. Therefore, cloud pricing models are not relevant to MFs because they only require some simple cloud space for hosting them.

k, l In an MSA, service implementations should be smart, and messaging as well as possible middleware should be dumb. Those requirements also apply to MFs. An application shell can act as middleware, and a page or a fragment is the pendant to a service. Problems can arise when one combines client-side and server-side composition techniques. There, it is relevant to keep the middleware (e.g., a CDN that uses ESI or SSI) as simple as possible.

m As previously described in paragraph *g*, each MF should own its backend service. One should not use the API Gateway pattern in an MFA because it would introduce coupling between MFs.

n A Circuit Breaker deals with faulty or unreachable MSs. One can apply this pattern's principles to an MF parent application (e.g., an application shell) in a way that the MF parent implements fallback mechanisms in case of a faulty or not available MF.

o Service Discovery can locate MSs locations at runtime. An MFA can adopt this pattern by introducing a runtime discovery mechanism for MFs.

p Since most of the computation power required by an MF will be requested from the client machine, an MFA does not require a load balancing mechanism.

q MSs are distributed across multiple server instances. This fact does not hold for MFs because an MFA combines all the different fragments into one computation environment and, therefore, already one central log destination. It can be valuable for MFs to use a shared logging library that provides a consistent log message format.

r, s Profiling and Health management are not required for MFs since there are no running instances of MFs.

t Finding service boundaries is even more involved in an MFA compared to an MSA. One has to find service boundaries for a vertical split that will probably have the same boundaries as the MS. Additionally, one has to think of extracting horizontal parts that can the different vertical MFs can reuse.

u, v Network communication does not occur between MFs because they already live in the same environment and can use other communication mechanisms. This shared runtime introduces additional challenges that are not present in an MSA. One can find those challenges, e.g., namespacing rules, in Section 3.1.3.

CHAPTER 4

Case study

Section 2.2 describes the different approaches that are currently known for building an MFA. One can generally divide those implementation techniques into client-side and server-side rendering techniques.

Server-side rendering is perfectly suitable for content-rich web application (e.g., a Webshop), because server-side rendering is faster than client-side rendering, mostly because of caching abilities. This thesis concentrates on architectures that operate on the client-side, especially on architectures that deal with implementing an MFA where each MF uses an SPA. As one can see in Section 2.2, one can best implement an MFA with SPAs as its main building blocks using the application shell approach.

The focus of this thesis is to evaluate whether one can use MFs for ERP application, as mentioned in the introduction (see 1). ERP applications do not deal with huge content. Instead, the main goal is to allow users managing the company's processes. Users of such applications spend a lot of their working time manipulating data via the ERP software. Most of the newly developed applications use SPAs frameworks for implementing the frontend. Therefore, this case study also concentrates on different methods to implement an application shell-based MFA for SPAs.

Various MF frameworks exist. When performing research on MFs, no evaluation of the existing MF frameworks could be found. Therefore, the thesis provides this framework analysis.

Chapter 4.1 performs an analysis of existing client-side MF frameworks. In the following sections, the thesis describes the use cases this thesis implemented (Section 4.3), introduces the example application (Section 4.4), and describes the use case's implementation details (Section 4.5).

4.1 Micro Frontend framework analysis

Currently, no literature exists that analyzed available MF frameworks. Table 4.1 shows a framework analysis of 26 MF frameworks to fill this gap in the literature. Each row in the table represents one MF framework. The columns specify the examined attributes which one can look up in Section 4.2.

Table 4.1 colors the cells in red, yellow, and green, which acts as a heatmap where one can quickly check whether a framework exhibits specific attributes or not. A red-colored cell shows that the framework represented in this row does have the attribute, or it was not possible to analyze this feature due to a lack of documentation. A yellow-colored cell either shows that the framework has only pieces of documentation or that the framework only works for server-side rendering, which is not the topic of this thesis. A green-colored cell shows support for the given attribute.

When looking at the table 4.1 heatmap, one can filter out two frameworks, which support nearly every attribute. Those two frameworks are Web Components and single-spa. Web Components are a new web standard that, at the time of writing this thesis, every well-known web browser supports. Canopy[1] initially developed the single-spa framework. This framework looked most mature when comparing all the different client-side frameworks listed in table 4.1.

4.2 Comparison attributes' definition

This section explains in the following paragraphs the attributes one can find in the column headings of Table 4.1.

Documentation The Documentation column shows the framework documentation quality. Possible options are no, partly, and good. No shows that the framework does not provide any documentation at all. Partly means that there is, e.g., only a short readme file with a simple how-to-start guide and probably some boilerplate code that one needs to start using the framework but no further documentation about the framework. The value good in the documentation column shows that there is online documentation available on the web. Most frameworks with good documentation often provide working examples one can clone or download. Especially in such a new field, this helps a lot when trying out a new framework.

Free tooling choice Free tooling choice refers to the option for a developer to freely choose a build tool like rollup.js[2] or webpack[3]. When a free tooling choice is not possible, the framework either provides its own tools or requires to stick to the one specific build tool.

[1] https://www.getcanopy.com/
[2] https://rollupjs.org/guide/en/
[3] https://webpack.js.org/

Technique This column specifies the technique that the frameworks try to support. Frameworks analyzed in this thesis either work client-side, server-side, or a combination of both. Additionally, the table shows frameworks built for bundle-loading, which is required load the different MFs at the client.

Type The type column specifies the framework type. The Mosaic9 framework is a collection of multiple different frameworks and services that one has to use together to implement a server-side MFA. Server-side composition means composing the view out of many files requested from different CDNs similar to ESI or SSI. A component library is just a collection of pre-defined components one can use to build a frontend. Those components are sometimes loaded at runtime or already included at compile time. A framework with the type application shell is described in Section 2.2.4.

Boilerplate / Code Scaffolding / CLI This column shows whether the framework provides a CLI tool or another tool that allows framework users to generate boilerplate code. This generation, also called code scaffolding, is an essential feature for MFs, as mentioned in Section 3.1.3 to allow MF teams to start with new projects quickly.

Multi-framework support Some of the analyzed frameworks allow splitting a monolith into different MFs. Not all of them can handle different SPA frameworks but require to stick to one framework pre-defined by the selected MF framework. Other MF frameworks support the use of a different SPA framework for each MF. This multi-framework support fulfills the requirements to a good MFA that each team should be able to select its individual technology stack, independent of other MF teams.

Base framework Some frameworks are built on top of another MF framework. The base framework column shows this framework that is used as a basis for the given MF framework.

License It is interesting to note that 25 out of 26 MF frameworks are open source. Only Bit, a component library, provides a paid solution to users. The other frameworks are either and web standard or use the MIT or Apache-2.0 open source license.

Framework	Documentation	Free tooling choice	Technique	Type	Boilerplate generator / Code Scaffolding / CLI	Multi-framework support	Base framework	License	Link
Mosaic9	partly	no	Server-side	service collection	yes	no		MIT license	https://www.mosaic9.org/
PuzzleJS	no	yes	Server-side	Server-side composition	no	no		MIT license	https://github.com/puzzle-js/puzzle-js
Podium	good	?	Server-side	Server-side composition	no	no		MIT license	https://podium-lib.io/
BigPipe	partly	no	Server-side	Server-side composition	no	?		MIT license	https://github.com/bigpipe/bigpipe
Bit	good	no	Server-side	component library	yes	yes		paid	https://bit.dev/
Luigi	good	no	Server-side	application shell	no	no		Apache-2.0 License	https://luigi-project.io/
Web Components	good	yes	Client-side	HTML standard	no	yes	Polymer	Web standard	?
Single-spa	good	yes	Client-side	application shell	yes	yes		MIT license	https://single-spa.js.org/
Quiankun	partly	?	Client-side	application shell	yes	yes	Single-spa	MIT license	https://github.com/umijs/qiankun
Piral	good	yes	Client-side	application shell	yes	no	React	MIT license	https://piral.io/
FrintJS	partly	no	Client-side	application shell	yes	yes		MIT license	https://github.com/frintjs/frint
Mooa	partly	?	Client-side	application shell	no	yes	Single-spa	MIT license	https://github.com/phodal/mooa
NgxPlanet	partly	no	Client-side	application shell	no	no		MIT license	https://github.com/worktile/ngx-planet
icestark	no	?	Client-side	application shell	no	yes		MIT license	https://github.com/ice-lab/icestark
Isomorphic Layout Composer	partly	?	client and server-side	Server-side composition + frontend rounting	no	yes	Single-spa/TailorX	Apache-2.0 License	https://github.com/namecheap/ilc
OpenComponents	partly	no	client and server-side	component library	yes	yes		MIT license	https://github.com/opencomponents/oc
ARA	partly	?	client and server-side	application shell	yes	yes	Airbnb Hypernova	MIT license	https://ara-framework.github.io/website/
Webpack 5 Module Federation	good	no	Bundle-loading	?	no	yes		Web standard	https://webpack.js.org/concepts/module-federation/
Systemjs	partly	no	Bundle-loading	?	no	yes		MIT license	https://github.com/systemjs/systemjs
nuz	partly	?	Bundle-loading	?	yes	?		MIT license	https://docs.nuz.app/introduction/overview/
One-app	no	no	?	component library	no	no		Apache-2.0 License	https://github.com/americanexpress/one-app
Nut	no	?	?	?	?	?		MIT license	https://github.com/nut-project/nut
Cellular JS	no	yes	?	?	?	?		MIT license	https://github.com/afsec/cellular-js
Misk	partly	?	?	?	yes	no	React	Apache-2.0 License	https://cashapp.github.io/misk-web/
Scalecube-js	no	?	?	?	?	?		Apache-2.0 License	https://github.com/scalecube/scalecube-js
Berial	no	?	?	?	?	?		MIT license	https://github.com/berialjs/berial

Table 4.1: Micro Frontend framework comparison

4.3 Case selection

The thesis goal is to answer the research questions defined in Chapter 1, which are repeated here:

- Can an MFA replace a monolith SPA with a client-side rendering solution without losing performance and, therefore, UX?
- Can the MF approach lower the risks of updating or adding a new feature in a complex web application?

To evaluate these questions, it is required to implement two general things. (i) One needs a traditional SPA implementation that acts as a reference point for analyzing the implemented SPA MSAs. (ii) One needs one or more implementations of a SPA MSA.

The monolithic application represents applications implemented by most companies who use an SPA framework. This monolith is the basis for the evaluation of the MFA implementations, and also aligns with the research question that tries to answer whether one can use an MFA instead of a monolith without loosing performance. The exact chosen case itself is not relevant to the evaluation. It is required that the case study contains different pages which refer to the vertical split defined in Section 2.2.1. Furthermore, horizontal composition (see Section 2.2.2) is a major part of such complex web applications to avoid code duplication. All of these standalone pages and fragments need some type of communication between themselves. The case studies show one possible approach for this communication, as it is described in Section 2.2.3.

This thesis uses three cases to evaluate the aforementioned research questions that the following three paragraphs describe.

Monolithic Angular Single Page Application The Angular monolith SPA is the reference implementation implemented using the Angular[4] SPA framework. This implementation will use the lazy-loading technique that allows the application to ship just the main application initially and dynamically reload more modules as users will request them. (e.g., by clicking on a navigation item)

Multi-framework Single Page Application based on a Micro Frontend Framework One major factor for using MFs is to allow the developers to choose the required technology stack freely. This feature can be enabled by using the single-spa[5] MF framework, which allows to implement separate MF applications and load them dynamically in one application shell. The decision to use the single-spa framework was made because the framework evaluation (see 4.1) revealed that this framework offers the most features. The single-spa framework especially provides excellent documentation, including sample

[4] https://angular.io/
[5] https://single-spa.js.org/

applications, which allows building such a complex architecture quickly. It is important to note that one should not expect the same performance as the Angular monolith since each framework will require to load its own framework code. This case primarily focuses on showing the possibility to fulfill the MF requirements one can find in chapters 3.1.2 and 3.

Angular Micro Frontend using Module Federation At the time of writing this thesis, a lot was going on in blogs and social media entries concerning MFs and the new Webpack 5 feature Module Federation[6], which was released in October 2020. The most exciting blog entry concerning this topic is from Manfred Steyer [82], who shows a short case study on using the new Webpack 5 ModuleFederation Plugin to implement an MF using Angular. Therefore, the third case study of this thesis will implement and evaluate this new feature, which will probably be the new "gold standard" for implementing an MFA.

4.4 Introduction to the example application

As already mentioned in the introduction (Chapter 1), using an MFs architecture is very relevant for ERP applications. An ERP application is mostly a huge application where users log in to the system once and then use the system for their daily working tasks. Therefore, a server-side rendering approach that is especially suitable for content-rich applications like webshops is not the best approach for the ERP scope. Users need to interact with the system and often change between different pages that load data dynamically, which shows the need for a SPA that was designed for such tasks since they do not require a page reload on navigation or data managing tasks.

Therefore, the decision fell on implementing a simple ERP application as the example app. A Welcome page (see figure 4.1a) shows just a simple welcome message, a navigation panel, and a button that navigates the user to a login page. There is a simulation of different teams where each team is responsible for specific parts of the system to simulate a big ERP system development company. One can identify the parts of each team's system by looking at the colored borders in figures 4.1, 4.2, and 4.3.

The following sections describe the different pages that separate teams would implement in a real-world application.

4.4.1 Navigation Panel

The team navigation implements a navigation panel (see figures 4.1, 4.2, 4.3) that displays Links to different other teams' modules. An anonymous user only sees a login button, whereas an authenticated user can see further links to the available pages. It is interesting to note that the login button is encapsulated into a so-called user panel (see figure 4.1b) that either displays the login button only if no user is logged in to the system

[6] https://webpack.js.org/blog/2020-10-10-webpack-5-release/

or displays the current user's name and a logout button to simulate inter-fragment communication between different MF fragments. Team authentication provides this user panel as a separate MF fragment, and team navigation uses this fragment from the other development team.

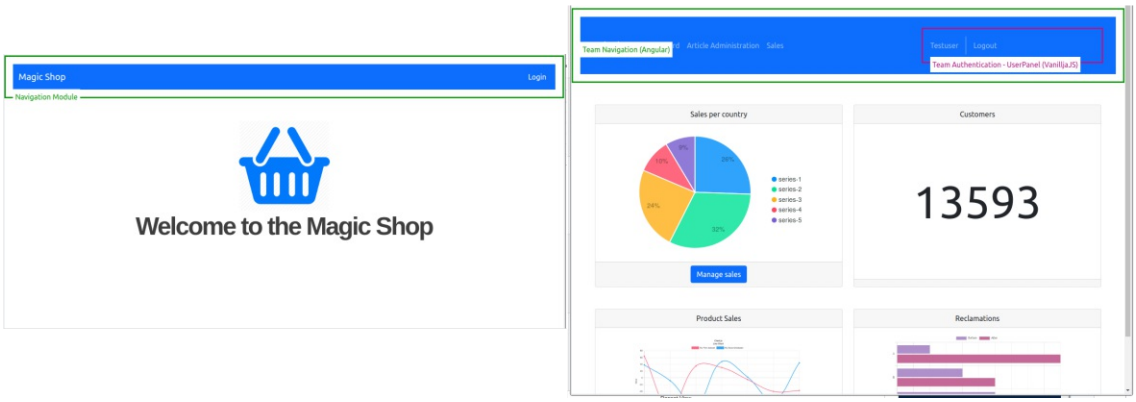

Figure 4.1: Case Study Application: (a) Welcome Page and, (b) User Panel

4.4.2 Login

The login page (see figure 4.2a) provides a simple username/password login form. The Angular monolith and the Angular Module Federation case studies implement this page as part of a login module implemented in Angular. The single-spa case study implements this case study as a single-spa React[7] application.

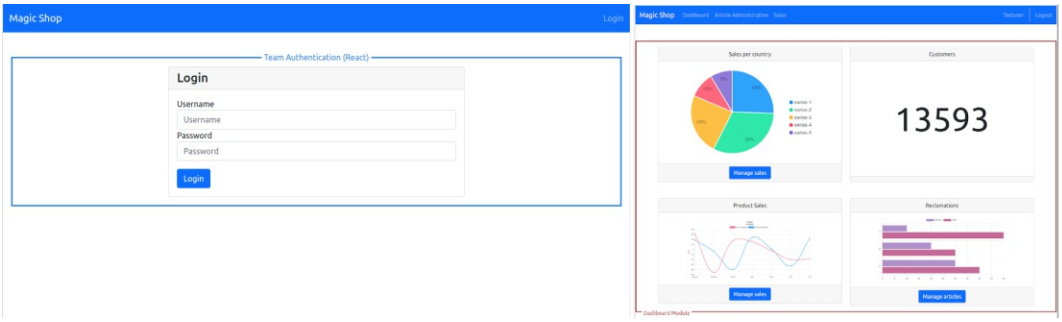

Figure 4.2: Case Study Application: (a) Login and (b) Dashboard

4.4.3 Dashboard

As soon as a user correctly logs in to the system via the login page, he or she will be redirected to the dashboard page (see figure 4.2b) that the team dashboard owns. The

[7] https://reactjs.org/

dashboard page only shows some placeholder images in the example application that would display statistics or other vital information in a real-world application. The team dashboard implements this application as an angular SPA in all three study cases.

4.4.4 Article Administration / Sales

The team article provides an overview (see figure 4.3a) of articles that the example company produces to simulate an ERP application. Team Sales provides an overview for a salesperson to manage article orders (see figure 4.3b). The singles-spa case study implements those pages a Vue.js[8] applications.

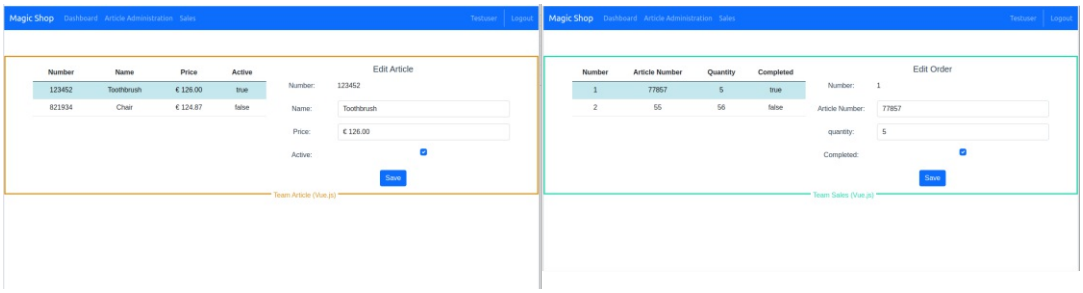

Figure 4.3: Case Study Application: (a) Article Administration and (b) Sales Administration

4.5 Implementation details

This section covers the case studies' implementation details and explains some specifics of each approach. Section 4.5.1 explains how those MFs were implemented. Section 4.5.2 explains how and where the applications were deployed.

4.5.1 Case Study implementations

Implementing a new technology can often be very challenging, especially if there is only little documentation available. For the architectures implemented in the case studies for this thesis, one had to research various web pages, example videos, and blog posts. Furthermore, especially for using the single-spa MF framework, it was necessary to contribute to the open-source project by notifying the maintainers about small but still existing bugs. Thankfully, the developers from single-spa reacted very quickly so that those problems could be solved very fast.

The backend for those applications is just an example REST API that would, in a real-world architecture, be an MSA (see figures 4.4, 4.5, 4.6). It does simply expose some

[8] https://vuejs.org/

example GET endpoints that provide some dummy data. The thesis used Node.js[9] with the Express.js[10] framework for implementing these REST endpoints.

The following paragraphs briefly explain the most important implementation details and the system architecture.

Angular Monolith The Angular monolithic application does not use any new technology. It was created to have a reference point for the performance measurements (see Chapter 5). To have an MF-ready application, the application was already divided into different modules that represent the single MFs. The application uses the Angular Lazy-loading technique to shrink the download size of the initial app download. This behavior is already similar to loading MFs dynamically, but in a traditional Angular application, the complete code, including all modules, has to be compiled together. Nevertheless, the lazy loading technique requires that the builder compiles the application into multiple files that the browser can request when needed. Figure 4.4 shows this architecture. The application and it's modules are bundled at once. Each module gets a separate bundle that the main application can load on demand.

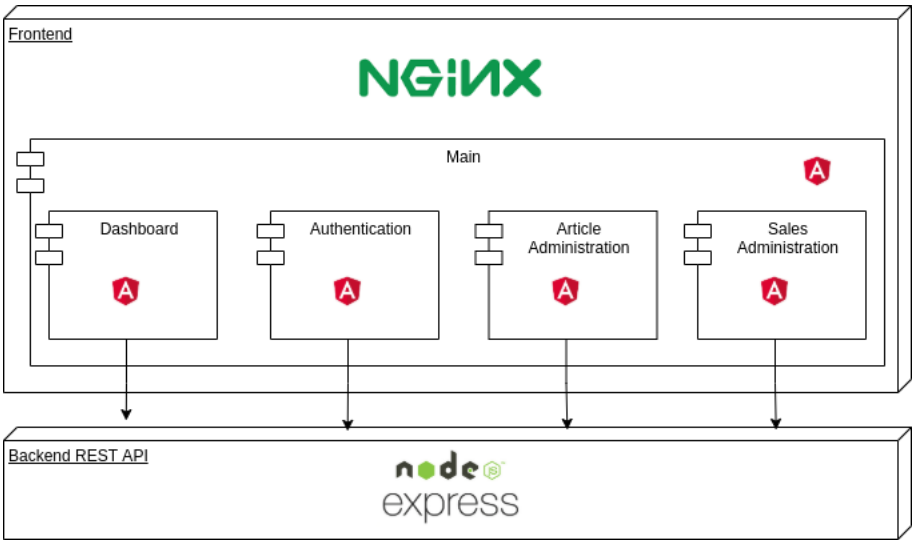

Figure 4.4: Case Study Application: Monolith Architecture

Single-spa Micro Frontend To create a single-spa application, the single-spa team provides the create-single-spa[11] CLI interface to create boilerplate code. This CLI can create three types of a single-spa application where this case study uses two of them. The main building block of a single-spa architecture is a so-called root-config. This root config

[9] https://nodejs.org/en/
[10] https://expressjs.com/de/
[11] https://single-spa.js.org/docs/create-single-spa/

is a shell application responsible for loading the registered MFs based on the current route. It uses the browser's history API to do that.

Each single-spa MF is called a single-spa application and can use a different SPA framework. To allow the single-spa root-config to load an application, it is required that the application exports three lifecycle properties (`bootstrap`, `mount`, and `unmount`). The single-spa root-config calls those lifecycle properties accordingly. The `bootstrap` function is called exactly once before the application gets mounted for the first time. The `mount` function gets called when the application is currently not mounted, and the single-spa root-config determined a route change that matches this application's route. This `mount` function should then create, e.g., DOM elements, to render its content. The `unmount` function gets called when an application is currently active but is not requested anymore. The task of the application is then to clean up itself (e.g., remove DOM elements, free allocated memory).

At the single-spa root-config, one can then use `registerApplication` to register a single-spa application that provides those lifecycle methods. Each single-spa MF application should be deployed separately. (for more details, see the following Section 4.5.2) To determine where to find the applications, it is recommended to use an importmap[12], which is an object of key-value pairs that the browser queries for a specific key (a unique value for an MF) and gets the corresponding value (an URL that points to the location of the MF to load). Currently, not all browsers support the importmap standard. Therefore, single-spa root-config uses SystemJS[13], a JS library that provides the same functionality as a native importmap.

As one can see in Section 4.4, the team navigation has to include a fragment from the team authentication, namely the user panel. The team authentication implemented the login page as a single-spa application and a separate fragment, the user panel, as a Web Component to test and show how one can use WebComponents (see Section 2.2.2) combined with single-spa. To enable communication between applications and the user panel fragment, the architecture uses the Broadcast Channel API (see Section 2.2.3). The fragment code, including CSS, is encapsulated into the ShadowDOM to prevent style or script leakage to other components outside the user panel.

One can find an architecture diagram in figure 4.5.

Module Federation Micro Frontend As described in the first paragraph of this section, Angular Monolith, it was not possible to build an application with lazy-loaded modules where one or more modules are separate projects until the Release of Webpack 5 in October 2020. Webpack 5 now supports Module Federation. With this approach, one can tell Webpack that parts of the application do not exist locally but can be loaded into the application at runtime.

[12] https://wicg.github.io/import-maps/
[13] https://github.com/systemjs/systemjs/blob/master/docs/import-maps.md

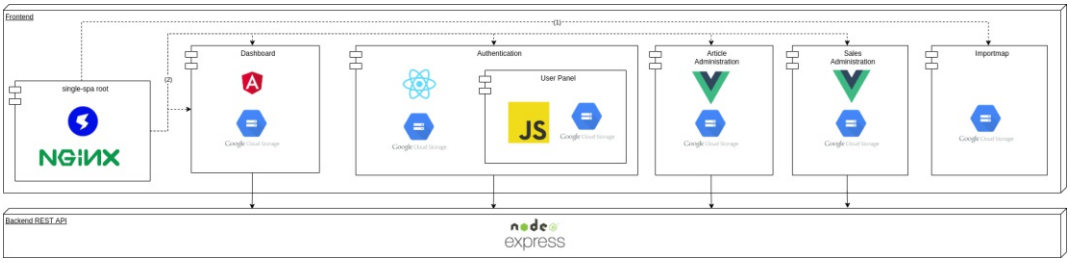

Figure 4.5: Case Study Application: single-spa Architecture

Module Federation generally provides two different operation modes that one can specify via the Webpack 5 Module Federation Plugin. The first operation mode can be used for applications built separately and later loaded into another application. One can tell the module federation plugin one or more modules that the application exports at this operation mode. Each exported application needs to have a unique name so that it the consuming application can identify it. One can use the second operation mode in applications that want to consume exported modules from other projects. It is necessary to define the exported modules in the consuming applications 'remotes' property of the Module Federation plugin.

This requirement to specify remotes with the MFs locations upfront introduces coupling between the exported applications and the consuming application. But there is some other way to consume an exported module without knowing the remote applications' URLs at compile time.

The Module Federation case study uses this approach in the shell application similar to the single-spa root-config. With the help of the function `loadRemoteModule` from the `@angular-architects/module-federation` package[14], one can dynamically register remote applications for specific routes and lazy-load them dynamically when they are requested. This example application loads the importmap, as described in the previous paragraph, via a `LookupService` to obtain existing remotes and their location. Each application defined in the importmap gets a navigation item assigned at the navigation bar so that users can navigate to the different MFs. Figure 4.6 shows this architecture.

4.5.2 Continuous Integration / Continuous Deployment

A working and reliable CI/CD setup is essential when dealing with an MFA. Each part of the system should be independent of other systems to avoid unwanted dependencies between the individual parts. Thus, this thesis implemented the case studies by following all the guidelines and requirements reported in Section 3.1.2 and Chapter 3. As already mentioned in those previous sections, there are multiple ways to organize the source code (e.g., single repository per project, monorepo). The two MF case study implementations

[14] https://github.com/angular-architects/module-federation-plugin

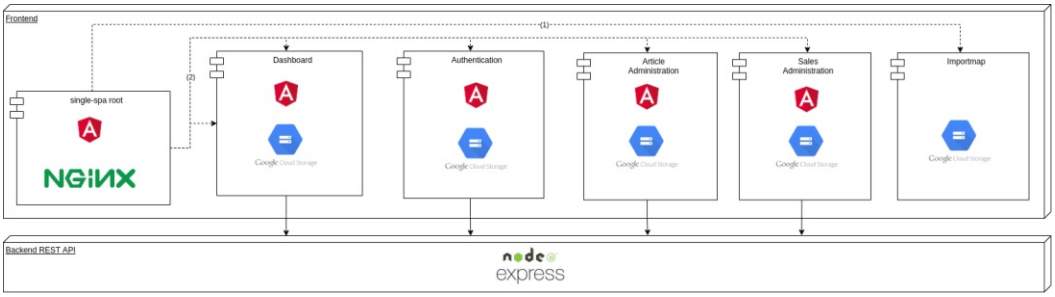

Figure 4.6: Case Study Application: Module Federation Architecture

use the single repository per project approach, supporting the independence principle. One can find all the repositories online at public Gitlab[15] repositories. The URLs to those repositories are provided in the appendix (see Chapter 6.1).

Google Cloud Platform Google provides a free tier for new users with a 300 USD credit that one can spend on the Google Cloud Platform (GCP)[16], which offers various cloud services like Virtual Machine (VM) instances, Cloud Storage or Kubernetes[17] clusters. This thesis makes use of this offer for deploying the case study implementations.

The Angular monolith application is containerized using an Nginx[18] container image that is configured to serve the static Angular deployment files.

The single-spa MFA also uses a containerized Nginx instance for serving the root-config. The individual MFs are packaged into single JS files, which are then served on a publicly exposed Google Cloud Storage Bucket.

The same procedure is used for serving the Angular Module Federation case study. The shell application lives inside a containerized Nginx instance, and the packaged MF files are statically served via the Google Cloud Storage.

CI/CD pipeline Gitlab, which stores the source code of the applications, also provides a free and easy to use CI/CD pipeline infrastructure with publicly available agents.

Each repository has its pipeline for building, testing, and publishing each MF individually. Each pipeline automatically runs on a push to the repository.

For the Angular Monolith, the single-spa root-config, and the Module Federation shell, the pipelines operate in three steps. Step one (build) builds the project to ensure that the current commit's code is still buildable. In the second step (package), the project is containerized using Docker[19], and the resulting container is published to the GCP

[15] https://gitlab.com/
[16] https://console.cloud.google.com/
[17] https://kubernetes.io/
[18] https://www.nginx.com/
[19] https://www.docker.com/

container registry. Step three (deploy) calls the Kubernetes cluster running inside the GCP to update the currently running containers with the newly deployed container image.

The other MFs (e.g., the dashboard MF) are built in a first step and then pushed to the Google Cloud Storage Bucket. Each deployment file gets assigned a unique hash value in its name to ensure that the browser always downloads the newest MF version.

When a pipeline pushes a new version of an MF to the Google Cloud Storage, it is necessary to make the shell (root-config) applications aware of those new versions. As explained in the previous paragraph, an importmap is used to accomplish this task. This importmap (a JSON file consisting of key-value pair that contains mappings of unique MF names to URLs) is stored next to the MF deployment files on the Google Cloud Storage. When a new commit is pushed to an MF repository, the pipeline's last step is to update the importmap so that the belonging MF name (the key of the importmap's) points to the new filename of the currently deployed MF. When this importmap file gets updated by two or more pipelines synchronously (e.g., team navigation and team authentication publish a new version of their MF simultaneously), it is possible that race conditions apply and one of those two updates are lost. To resolve this issue, the single-spa team created a tool called import-map-deployer[20]. This thesis implementations use this import map deployer, which is again deployed as a Docker container to the GCP Kubernetes cluster and has access to the Google Cloud Storage. The import-map-deployer exposes API endpoints that allow a developer to update, insert or delete an importmap entry. The import-map-deployer takes care of possible synchronous updates to the same import map and ensures that no changes are lost during the update process.

[20] https://github.com/single-spa/import-map-deployer

CHAPTER 5

Evaluation

The case studies described in Chapter 4 describe the different approaches with their used technology and the implementation details. To evaluate the thesis research questions (see Chapter 1), it is essential to evaluate the deployed architectures. For evaluating the first research question dealing with performance, it was necessary to perform website performance tests. The second research question, whether an MF approach can lower the risks when changing a frontend software system, can be answered by conducting the architecture metrics defined in Section 5.1.2.

A correct and reproducible evaluation ensures comprehensible results. This thesis deployed the case study applications on the GCP. Deploying all artifacts in the same region and zone is crucial to avoid negative side-effect like network latency issues. Figure 5.1 shows the evaluation architecture. All services required for running the tests live inside the `eu-west-3a` region/zone provided by the GCP. A private WebPageTest[1] instance was also deployed in the same zone. Section 5.3.1 describes the exact evaluation details.

This chapter is structured as follows: Various quality metrics for measuring website quality exist in literature that Section 5.1 describes. Not all of these metrics are suitable for evaluating this thesis' case study, and Section 5.2 will briefly mention the metrics this thesis conducts for the evaluation. Section 5.3 explains how the measurements for those metrics were performed, and Section 5.4 presents and describes the results.

5.1 Quality metrics

To evaluate the implemented MFs compared to a monolithic system, one must research existing quality metrics for web pages. There are two general ways in research for analyzing how users think of UIs. UX research deals with analyzing the "acceptance, experience,

[1] https://github.com/WPO-Foundation/webpagetest

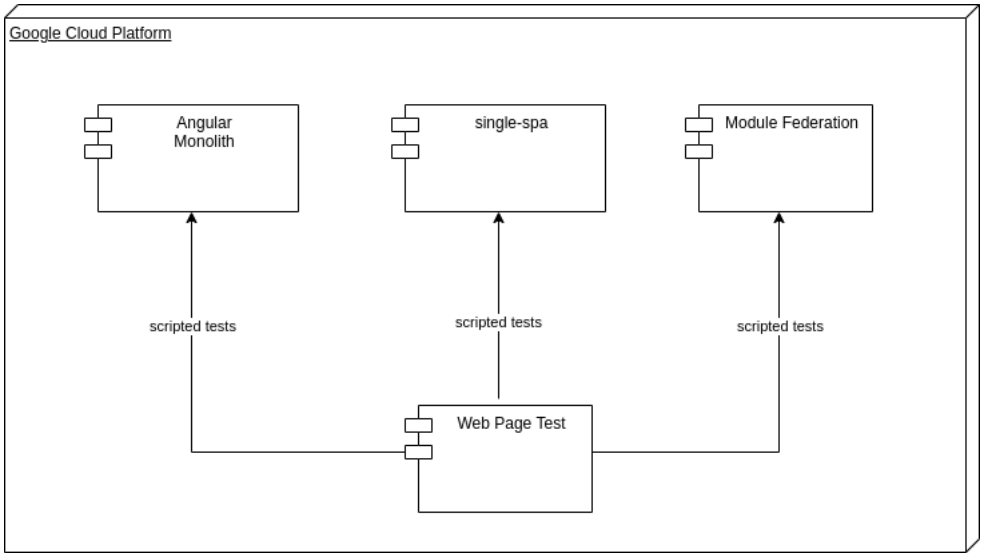

Figure 5.1: Case Study Application: Evaluation Architecture

and crucial design factors," while Quality of Experience (QoE) primarily focuses on "technical aspects and subjective perception of waiting times." [6] The following Section 5.1.1 only focuses on QoE aspects relevant for analyzing MFs load times compared to monolithic systems. Furthermore, it is relevant to validate that a distributed architecture like an MFA helps improving the maintainability of a software system compared to a monolith. Cojocaru et al. [18] presented a survey that shows metrics to analyze the quality of MSs automatically decomposed from monolithic applications. Section 5.1.2 analyzes those metrics for measuring the complexity of a distributed software system a software system and presents the metrics that one can use to measure the MFA quality.

5.1.1 Quality of Experience

Bouch [12] and Galetta et al. [35] already analyzed how page load times impact user satisfaction. Bocchi et al. [11] state that companies like Amazon, Google, or Shopzilla report up to 12% increased revenue due to load time reductions by just a few seconds. Butkiewicz et al. [15] report that more than 50% of randomly selected web pages from the Quantcast[2] top-20000 have a page load time of more than 2 seconds. "User studies and industry surveys show that users are likely to be frustrated beyond this two second threshold." [61] The higher a web page's delay, the lower the QoE is, resulting in worse UX and, therefore, a higher likelihood of user disengagement leading to more considerable economic losses. [11]

Barakovic et al. [6] describe QoE as the user's "degree of delight or annoyance." They performed a QoE survey and found that many studies (11) report that the "waiting

[2]http://www.quantcast.com/top-sites

time is the key influencing factor when it comes to end-user Web QoE." [6] Egger et al. [28] state that "QoE tries to link performance as closely as possible to the subjective perception of the end user." They formulated the WQL hypothesis, which claims that the "relationship between **W**aiting time and its **Q**oE [...] is **L**ogarithmic." Figure 5.2 shows this logarithmic correlation between page load times and the MOS on typical loading or search tasks.

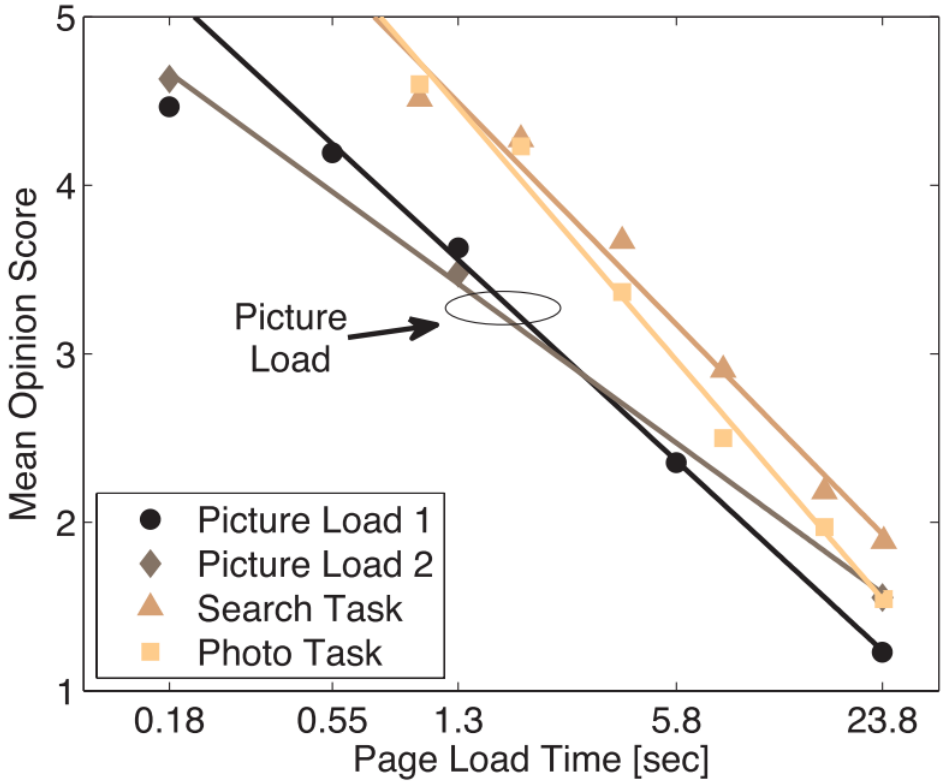

Figure 5.2: "User satisfaction for various constant page load times" [28]

The following section shows metrics for analyzing the QoE of web pages. Bocchi et al. [11] provide a categorization into four categories. (i) Time-Instant metrics, (ii) Time-Integral metrics, (iii) Compound scores, and the (iv) Mean Opinion Score. The Mean Opinion Score is a benchmark for other metrics that calculates the average users' subjective ratings and is not further analyzed due to the challenging task of collecting MOS points. [11] The following paragraphs list and briefly describe metrics of the first three categories that Bocchi et al. [11] provide and extend them by metrics from other literature.

Time-Instant metrics

Time-Instant metrics are easy to measure but can not express complicated dependencies between rendering tasks and the perceived UX. The following paragraphs briefly explain Time-Instant metrics.

onLoad The onLoad (also called Page Load Time) measures the time taken to load all page objects and is generally excepted as the leading performance indicator. [11]

Time To First Byte The TTFB metric expresses the page reactivity. [11]

DOM event The DOM event occurs when the browser downloaded the DOM, and the rendering can start. [11]

Time To First Paint The Time To First Paint (TTFP) metric shows when the first object is rendered. [11]

Above-the-Fold Above-the-Fold (ATF) [14] defines the time at which a browser shows all content visible in the user's viewport. Calculating this metric is very complicated since it requires taking screenshots during the rendering phase and pre-process those for calculating the ATF metric. [11]

Other Time-Instant metrics

- The total number of objects loaded [15]
- The number of these loaded objects that are JSs [15]
- The total web-page size [15]
- The number of servers contacted to load a page completely [15]
- The number of origins contacted in loading objects on the page [15]
- the TTI that defines the period before a user can interact with UI [40]
- The latency for the initial web page call [40]
- The load time for a page change [40]

Time-Integral metrics

The most prominent metric in this category is Google's SpeedIndex. The Speed Index considers the whole process leading to the visual completion of the web page. [11] To calculate the Speed Index, one needs to take snapshots of the web browsing session (default: 10fps) and compose a filmstrip from those snapshots using it to infer the visual page completion. [11] Bocchi et al. [11] state that the Speed Index metric has limited use due to computational complexity. It is interesting to note that Time-Integral metrics have a lower bound given by the TTFB because rendering can not start before the first byte of payload arrived at the client. [11] Additionally, Bocchi et al. [11] claim that besides time-integral metrics are more fine-grained, "any time-instant metric can be considered as the upper bound of the time-integral metric."

Compound Scores

Compound Scores like Yahoo's YSlow [3], Google's Page Speed Insights[4], or dynaTrace[5] are usually expressed as a set of heuristics combined with weights. [11] Those heuristics (e.g., image compression rate) are often "unrelated to event timing" and, therefore, hard to map to QoE.

Besides those metrics, Barakovic et al. [6] mention that since the extensive use of HTML/5 and AJAX, not only the page load times are interesting to calculate but also how users perceive element refreshes triggered by dynamically load fragments. They recommend addressing this issue in future research. Furthermore, they describe that it can be of particular interest to examine the web QoE on real interactive page views that a given task requires instead of just gathering metrics for different sites separately. [6]

5.1.2 Micro Frontend Architecture metrics

Decomposing a monolithic architecture into individual parts must not negatively influence the UX or even improve it. Companies decide to implement an MFA because they want to improve, e.g., system maintainability, deployment speed, and others, as this thesis describes in Section 2.1.1. When developing such a system, it is crucial to evaluate whether the requirements can be fulfilled or not. Cojocaru et al. [18] presented metrics that one can classify into two types. (i) Static analysis, and (ii) dynamic analysis. The following sections and paragraphs present those metrics defined by Cojocaru et al. [18], adopted to MFs, which one can use to measure the migration effects when migrating from a monolithic frontend application to an MFA.

Static analysis

The static analysis deals with analyzing a software system without the need to run a system. [18]

Granularity This metric measures the system size. One can analyze the LOC count or the number of exposed interfaces as a workload for this metric.

LOC and Open Interfaces Cojocaru et al. [18] define the LOC and Open Interfaces metrics, which are already included in the aforementioned granularity metric. They mention that this LOC metric is "rather informative than reliable" and that it is more useful to use this metric for comparing the relative size between components instead of conducting the plain LOC count to identify possible "big components." [18]

[3] http://yslow.org/ruleset-matrix/
[4] https://developers.google.com/speed/docs/insights/v5/get-started
[5] https://www.dynatrace.com/

Cohesion Cohesion measures whether the service has only single responsibility or not. One can calculate this metric by determining the ratio of "operations per client" and the "operations per client times the available operations" of the complete system. A low cohesion value can reveal possible candidates that could be further decomposed into standalone parts.

Coupling Coupling measures the dependencies between MFs. If this metric shows circular references between the individual parts, those can be seen as merge candidates. One can calculate this metric by analyzing the ratio between the number of exposed attributes and events and the number of attributes and events consuming from other MFs.

Technology Heterogeneity This metric counts the number of involved technologies in an MF.

Dynamic analysis

Dynamic analysis operates on running software and can be used for validating static analysis metrics or "identifying new flaws." [18]

Execution cost The execution cost is the monetary value required for hosting an MF. Since most of the computing power will reside on the client-side and not on the server, this metric will not be very informative, but MFs can introduce some additional running service costs (e.g., an import-map-deployer as described in Section 4.5), which one must have in mind.

Usage frequency The usage frequency in an MSA describes the ratio of requests to an MS and the overall requests to all services of an MSA. One can adopt this metric vy calculating the ratio of the number of MFs that consume another MF and the overall number of MFs.

Reusability The usage frequency metric is related to the reusability metric and can be used to analyze the reusability of MFs.

Maintainability The maintainability metric is a combination of other metrics (granularity, technology heterogeneity, coupling, and cohesion). A good maintainable MFA should exhibit high cohesion, low coupling, and a homogenous technology stack.

Deployment and Agility To raise the maintainability of a software system, it is crucial to shortening deployment cycles. A good and fast CI/CD setup helps to achieve this goal by automating manual deployment tasks. One can measure the deployment times in an MFA by measuring the pipeline runtimes of the individual MFs.

Health Management The Health Management metric is a boolean indicator that describes whether an MF can deal with failures. In an MF case, this metric shows whether the MF team provides fallbacks to static content if the MF is not available.

Organizational Alignment This metric is related to Conway's Law (described in 3.1.2) and shows how good or bad a company structure is related to the software architecture. This metric is rather intuitive, and Cojocaru et al. [18] did not define any calculations for it.

5.2 Performance metric selection based on literature analysis

This section explains the metrics this thesis uses for evaluating the case studies. The following Section 5.2.1 describe the webpage performance metrics and Section 5.2.2 lists the used architecture metrics.

5.2.1 Performance metrics

Section 5.1.1 describes the available QoE metrics. The thesis uses the following metrics for the evaluation of the case studies:

- onLoad
- DOM event
- Total web-page size

As described in Section 5.1, those time-instant metrics are an upper bound for time-integral metrics. Thus, it is not necessary to evaluate any time-integral metrics for answering the research questions.

5.2.2 Architecture metrics

In addition to those performance metrics, architecture metrics described in Section 5.2.2 help to formally analyze whether architectural requirements to an MFA can be fulfilled with the technologies used in the case studies.

This thesis will evaluate the following architecture metrics:

- LOC
- Technology Heterogeneity
- Deployment and Agility

5.3 Performance metric measurement on the monolith and the Micro Frontend implementation

As described in Section 4.5, the thesis used GCP for deploying the different MFA parts in a cloud environment. All those services were deployed in the `eu-west-3a` region/zone, a Google Datacenter in Frankfurt in Germany. To decrease the risk of measurement errors due to network latency, it is essential to run the tests from a virtual machine that lives in the same zone as the services.

The following Section 5.3.1 describes how the performance metrics were gathered, and Section 5.3.2 describes the architecture metrics data origins.

5.3.1 Performance metric gathering - WebPageTest

SPAs have a significant advantage compared to other technologies like SSR in that they operate entirely on the client-side and, therefore, do not require a page reload on navigation or data loading operations. Thus, it is impossible to perform "normal" webpage tests that send a request to each of the different pages and measure the data required for calculating the performance metrics. To solve this problem, the open-source software WebPageTest and a WebPageTest Agent were deployed as a private instance on the GCP Kubernetes cluster in the same zone as the case study deployments (see figure 5.1). The WebPageTest software offers a frontend to interact with the test system manually and offers an API that one can use to send test requests to the system. This thesis used the API approach for the evaluation. Furthermore, WebPageTest allows performing scripted tests. Scripted tests provide a browser instance that is observed regarding performance while one can use JS to perform actions on the page. One can define multiple events that perform any number of JS commands. Each of those steps is analyzed by the WebPageTest system.

This thesis uses a test scenario of five steps that one can find in the WebPageTest Script repository (see appendix 6.1):

1. Navigate to the Home page: Instruction that loads the system's home page
2. Navigate to the Login page: Script that clicks on the login button in the top right corner
3. Login: Fills the login form with test user data and clicks the login button to simulate a user login
4. Navigate to Article Administration
5. Navigate to Sales Administration

For the analysis, the script was executed on the deployed case study implementations. Each test script was executed with three runs, which means that each test case was

executed three times on clean web browser instance and the results show the average values of those runs. A private WebPageTest instance allows publishing test results to the publicly available WebPageTest instance. This thesis's test results were published to the public instance. (One can find the links to those test results in the appendix 6.1)

5.3.2 Architecture Metrics

Most of the data required for calculating the architecture metrics come from the actual case study implementations. For counting the LOCs, the thesis used the CLOC[6] program. The pipeline runtimes come from the pipelines on Gitlab and are show the mean value of all succeeded pipelines for each project.

5.4 Comparison and evaluation

As described in Section 5.3, the case study implementations were analyzed using the open-source software WebPageTest. Each test case was executed three times to lower the risk of anomalies in testing. Additionally, in each of those three runs, the test system was conducted to test the given webpage using a first-view and a repeat-view test. A first-view test always runs on a clean browser instance with a clear cache, whereas a repeat-view test runs on the same browser instance as the first-view test. The browser cache is not cleared between the first-view and the repeat-view test. The repeat-view test is crucial for testing SPA applications since they mostly act as a native app-like software system. One can consider the first-view test as an installation step and the repeat-view test as the "normal" operation test. One can strengthen this argument by stating that such applications are often PWAs that offer a real "installation" step, which is also known as "add to home screen." The following two sections present the test results. Section 5.4.1 shows the performance metric evaluation, and Section 5.4.2 presents the evaluation of the architecture metrics.

5.4.1 Performance metric evaluation

The following paragraphs explain the WebPageTest results from testing the three case studies. As mentioned above, it is essential to differentiate between the first-view and the repeat-view.

Total web-page size Figure 5.3 shows the different page's download sizes of the three case studies. The Angular monolith loads most of the framework and application code already on the main page. The 0 value of the login page is explainable because the Authentication Module is already loaded with the Main page that uses the UserPanel component, which is part of the Authentication module. The dashboard has a slightly higher value because it includes static files. When looking at the single-spa bars, one can recognize that this approach uses different MF implementation frameworks. The login

[6] https://github.com/AlDanial/cloc

page uses React.js, the dashboard uses Angular, and the article and sales page use Vue.js. The Angular Module Federation MF benefits from using only Angular. Nevertheless, one can identify a slightly higher download size than the Angular monolith.

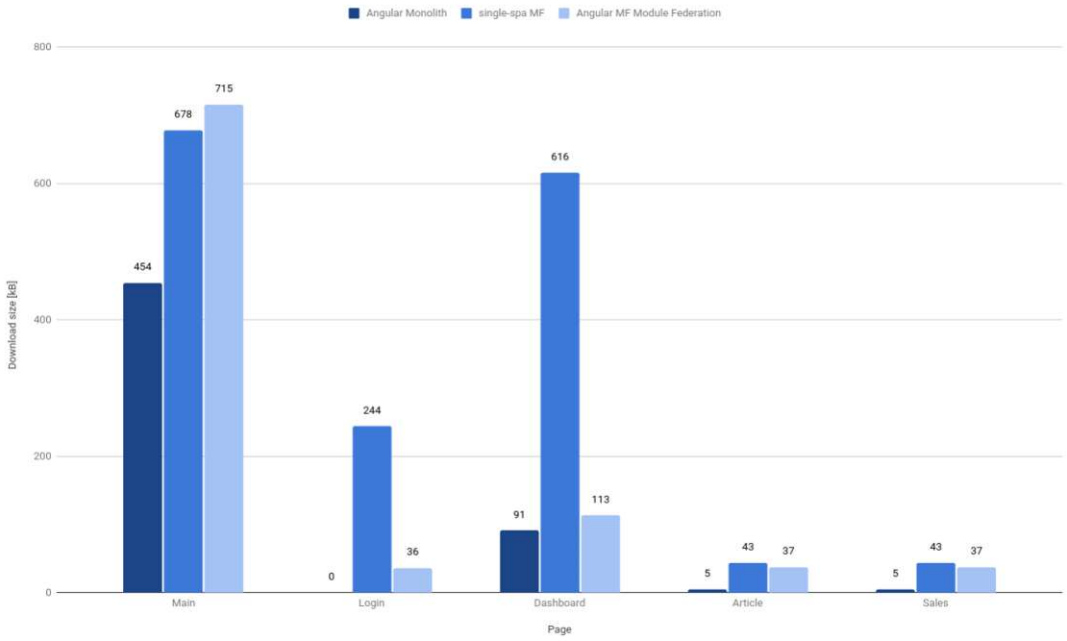

Figure 5.3: Performance Evaluation: Total Webpage size

DOM event The DOM event is fired when all DOM objects are available, and the browser can start the rendering process. Loading a monolith SPA framework requires the most upfront loading time (see figure 5.4). The MF approaches are more lightweight in the beginning. They have to perform more work afterward when running the page scripts.

onLoad The onLoad event (see figure 5.5) occurs when the browser finished loading all page objects. When comparing the onLoad event and the overall page load time (the time at which the page is completely loaded, including the execution of all scripts) shown in figure 5.6, one can see that the single-spa MF architecture has to perform extra work in the browser to initialize the MFs. The first-view always requires more loading time for the MF implementations, which is expected because more code is needed to initialize and mount the MF. Additionally, both MF implementations require a lookup in the importmap, requiring one more additional request to the server than the monolith. It is interesting to see from the measurement results that the repeat-view has nearly the same performance on all three architectures.

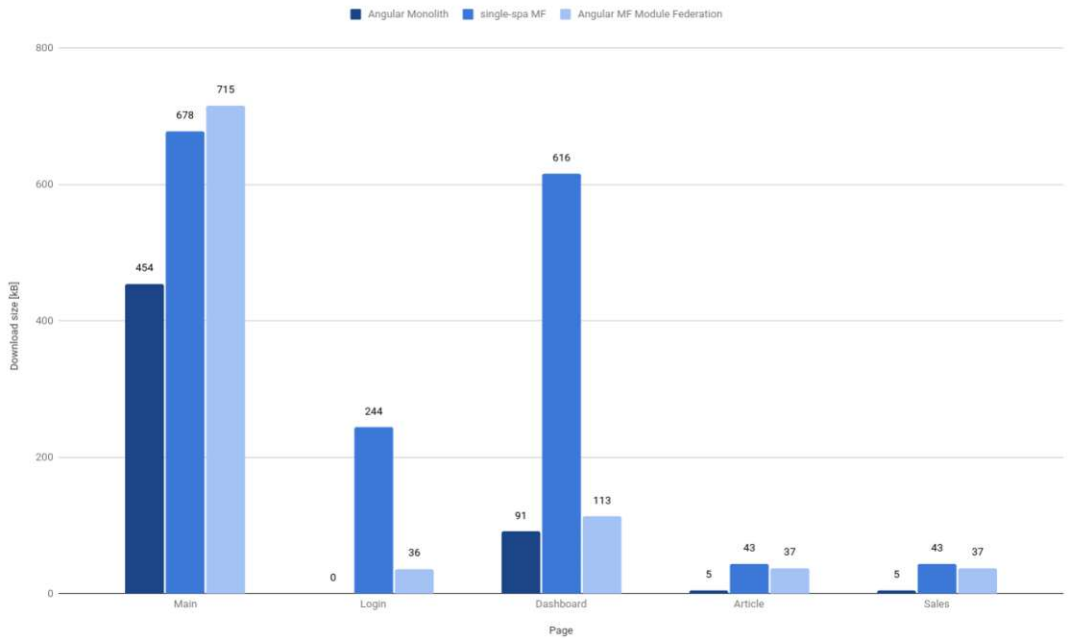

Figure 5.4: Performance Evaluation: DOM event

Overall page load time The overall page load times (see figures 5.7 and 5.8) coincide with the webpage size. The higher values for the login and the dashboard page at the first-view come from using different SPA technologies in the MFs.

5.4.2 Architecture metric evaluation

The following paragraphs explain the architecture metrics evaluation to show that using an MF approach can help a company to generate maintainable code that multiple, separates teams can develop and quickly deploy separately.

LOC Figure 5.9 shows the LOC count for all project repositories (see appendix 6.1). The figure visualizes the HTML, CSS, and JS LOC counts as a stacked bar for each source code repository. When implementing an MFA, the primary goal is to have a more flexible and maintainable system. Splitting a monolith project into many smaller projects (MFs) helps to achieve this goal as figure 5.9 illustrates.

Technology Heterogeneity Technology heterogeneity improves the project's maintainability because a low number of different technologies lets developers exchange their knowledge among themselves. The Angular monolith and the Module federation case studies use only Angular and therefore provide excellent technology heterogeneity. On the other hand, an MFA should allow companies to avoid a vendor or technology lock-in.

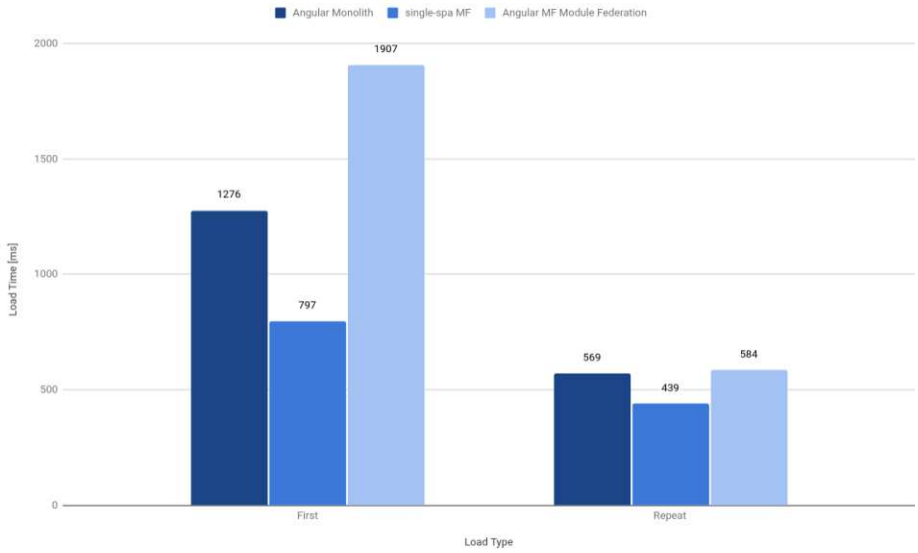

Figure 5.5: Performance Evaluation: onLoad event

The single-spa case study shows that using a different SPA framework in each MF is possible. The technology heterogeneity is, therefore, lower. Each company has to decide when and how many different technologies they allow.

Deployment and Agility It is crucial for an MFA that developers do not have to spend lots of time performing recurring tasks like running unit tests or deploying the system. Furthermore, there is a need that companies can deliver new features or bug fixes to customers as fast as possible. It is not up-to-date anymore to have long release-cycles. CI/CD pipelines help achieve this goal, and splitting a monolithic frontend into small MFs can help speed up those deployment times further. Figure 5.10 shows the project's pipeline runtimes on Gitlab. One can see that deploying a monolithic application requires more time than deploying the individual MFs. The high deployment time of the module federation shell comes from the fact that this pipeline did only run once and could not make use of the pipeline's caching options (especially for the `node_modules` folder).

To summarize this chapter, this paragraph briefly explains the steps performed to gather and evaluate the results. For the data gathering, a WebPageTest instance at the same geographical location as the servers serving the web applications was deployed. Scripted tests were sent to this WebPageTest instance that performed the tests in a first- and repeat-view test. Each test was executed multiple times, and the mean values for the performance times were used to decrease measurement errors. Furthermore, architecture metrics were calculated to analyze the Developer Experience improvements and performance gains for company processes by decreasing deployment times.

In figure 5.11, one can see a summary of the performance evaluations. The left part of

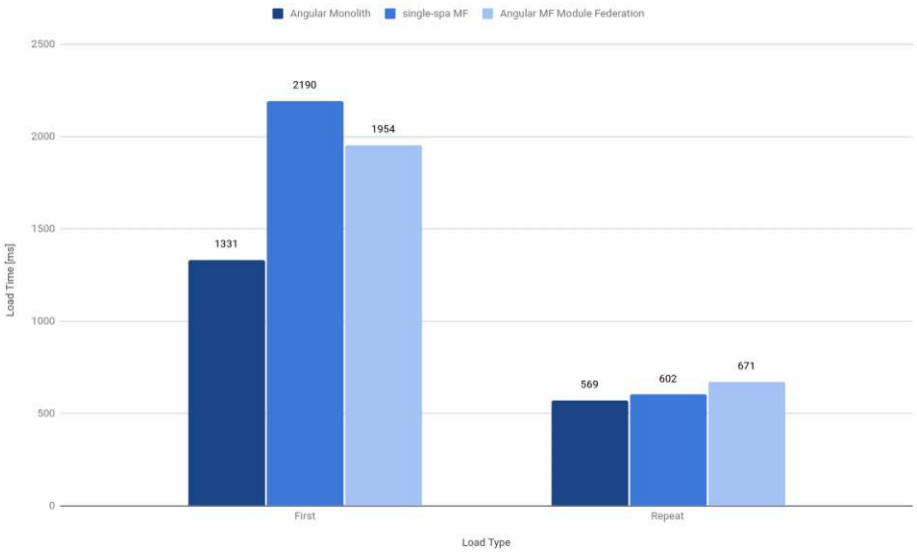

Figure 5.6: Performance Evaluation: Page fully loaded

the figure shows repeat-view values, and the right part shows first-view values. This figure clearly shows that once the web application has passed the "installation" step, it has nearly the same or even better performance when using an MF approach compared to a monolithic application. All architectures (monolith and MFA) require more work on the first load and afterward profit from caching, which one can see as the "normal" operation for an app-like webpage. A monolithic application needs the most time to load everything at the beginning is fast afterward. An MF using the single-spa framework differs from a monolith and requires more time when an MF with a "new" (not already used and loaded by another MF) SPA framework has to be loaded.

Furthermore, the LOC count is reduced for each MF compared to a monolith, but summed up requires more LOCs because of the boilerplate code needed in each MF. CD/CD runtimes are significantly reduced, which helps to deploy faster, and developers get possible failure response quickly.

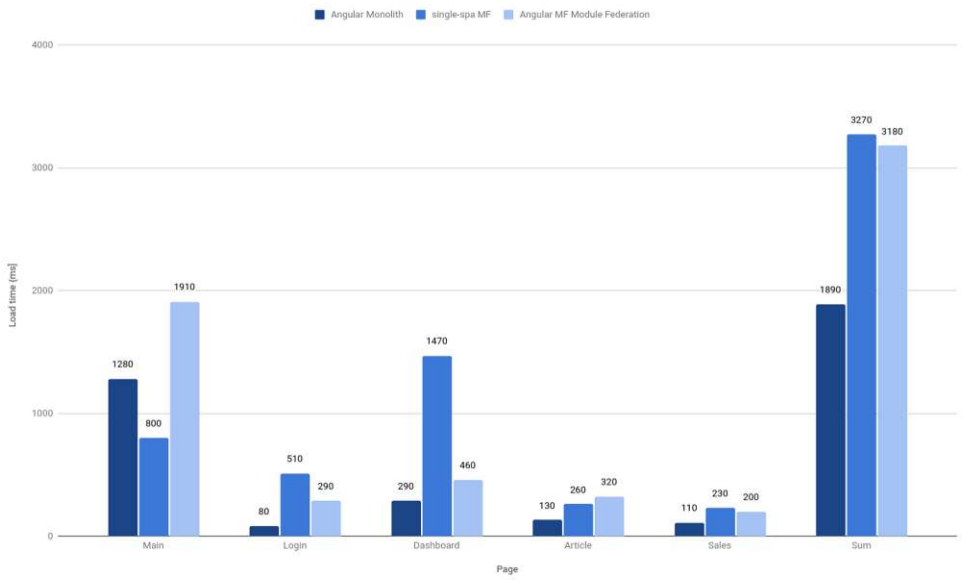

Figure 5.7: Performance Evaluation: Load time page-first

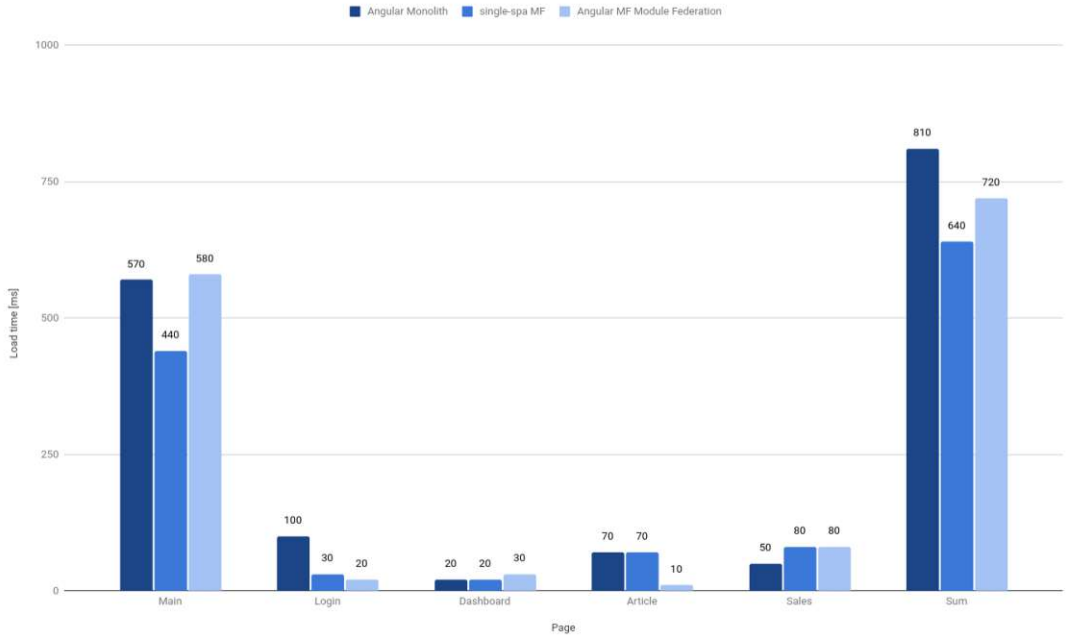

Figure 5.8: Performance Evaluation: Load time page-repeat

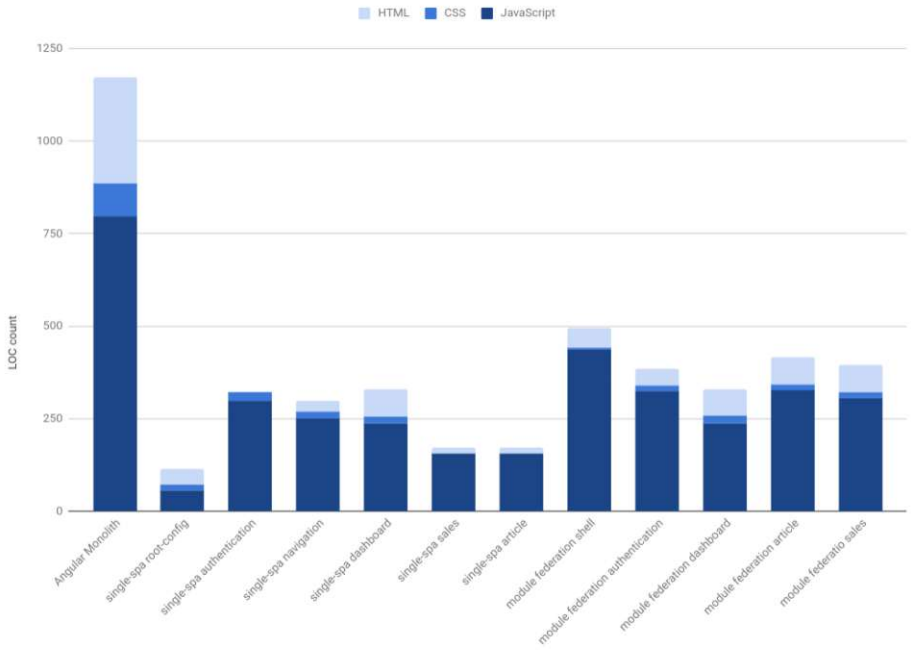

Figure 5.9: Performance Evaluation: LOC count (HTML, CSS, and JS)

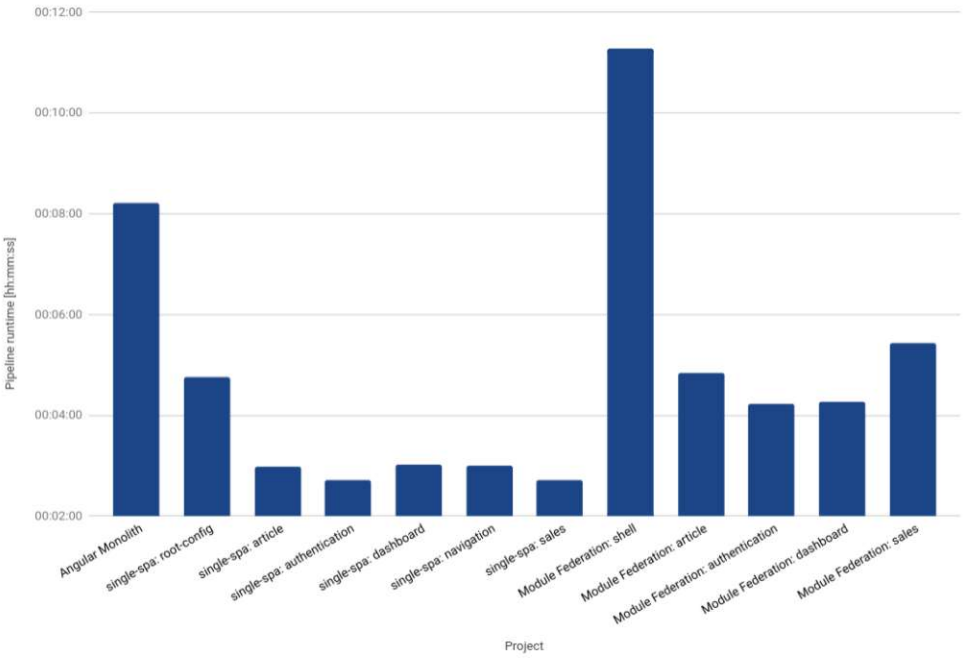

Figure 5.10: Performance Evaluation: CI/CD runtimes

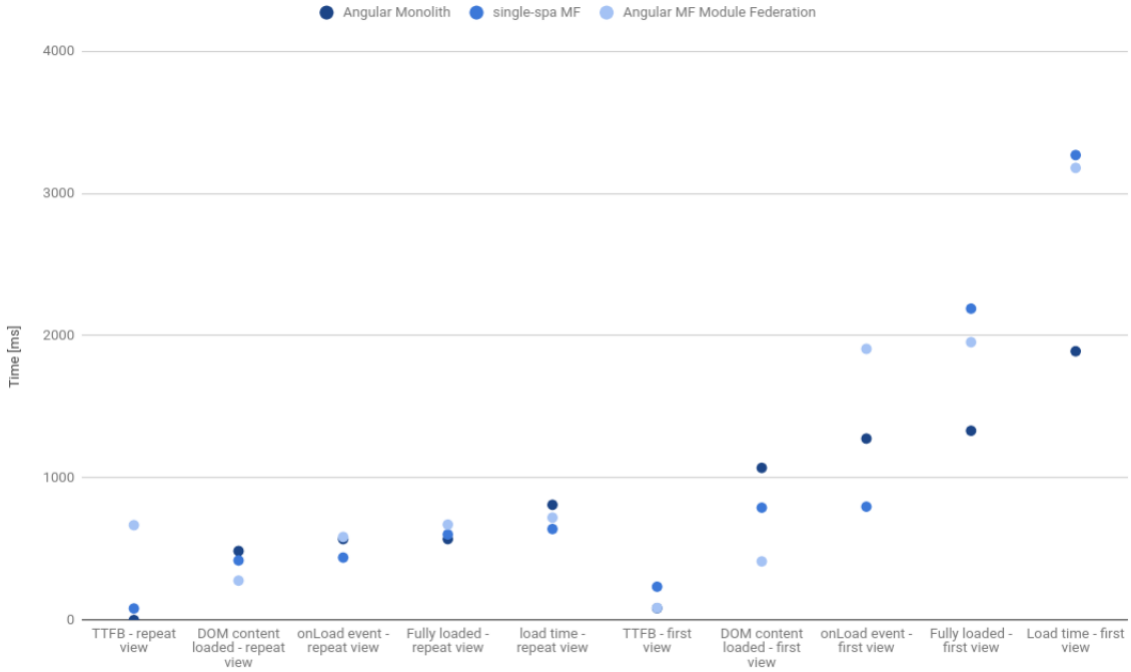

Figure 5.11: Performance Evaluation Summary

CHAPTER 6

Conclusion

This thesis covered an in-depth study on the maturity of MFs, investigating their principles, the state-of-the-art of this architectural approach, and testing it in the case of an ERP application, providing performance analyses. In the first part, the work inspected the current limitations and discussed how the MS approach can serve this scenario.

The initial MSA definition by Lewis and Fowler [51] already intended that an MS is a small part of a bigger system that can handle everything, from storing data in a database to providing a UI for users to deal with the software. The current state-of-the-art of MSAs mostly adopted those initial thoughts and focused on all parts of the intended architecture except handling the UI as an essential part of an MS.

MS literature provides many interesting insights and findings into MSAs that where studied in this thesis to gain knowledge for implementing an MFA. The most important reasons for using an MSA are increasing team scalability, implementing maintainable code, and getting fast and reliable DevOps support to deliver features faster than using a monolith. Furthermore, an MSA offers the possibility of using independent technology to avoid vendor lock-in and stimulate innovation.

MS patterns like the API gateway show that developers could implement solutions that fill the gap between a backend MSA and a monolithic frontend. These solutions also show that there is a need for frontend architectures comparable to a backend MSA. Nevertheless, concepts and solutions that deal with MFAs exist. One can classify those solutions into two types - server-side and client-side. Both types already have multiple implementation options, which are currently not well researched, and most of the literature sources are grey literature like books or blog posts.

Big companies like Amazon use a server-side MF approach because their pages heavily rely on rendering large content. This work did not deal with those technologies but focused on client-side rendering using SPAs. Those applications are primarily useful for

native app-like software systems that help users in their daily work, which was previously mostly a program directly installed on the Operating System (OS).

The thesis showed how to adopt existing, well researched MS principles to MFs and found that most of the findings for MSs are directly applicable to MFs. The framework analysis revealed that many MF frameworks exist with little to no documentation, and most of them appear as they are a Proof of Concept (PoC). Furthermore, no generally accepted MF solutions could be found, and it seems that every company that tries to implement an MFA develops its own framework. Completely new approaches like the Webpack 5 Module Federation are currently not researched at all.

In the practical part of this thesis, three case studies were implemented. Those case studies are based on (i) an Angular monolith, (i) a single-spa MFA, and (iii) a Module Federation MFA implementation with Angular. There are just a few implementations of similar architectures available in the literature. The Module federation approach and the single-spa approach using different SPA frameworks were never researched until now. Besides, when performing research for this thesis, no information on how to set up working CI/CD pipelines was found. This work closes this gap.

This thesis's evaluation part shows that a client-side MFA using SPAs has less performance than a monolithic application on a first-page load (without cached resources) but can provide the same performance as a monolithic frontend application as soon as cached sources are available. Such a distributed architecture requires more effort when implementing the system, but in the end results in a more maintainable system that allows faster deployment cycles due to independent releases. Furthermore, it helps a company to be able to scale the development teams individually and allow teams to be more productive since there is less communication overhead in smaller teams.

6.1 Future work

One exciting research question would be whether or how it is possible to use the MF architectural style for native applications since all existing works focus on web engineering.

This thesis used case studies to answer the research questions. It would be fascinating to research on migrating a big, existing monolithic frontend application to an MFA. Especially for evaluating metrics defined by Mazlami et al. [54], who performed a study on migrating a monolithic backend into an MSA using an automated migration tool. Those metrics are the team-size-reduction-ratio metric based on contributions (commits) to the source code and the domain-redundancy metric that analyzes how well the split into the different services worked.

List of Figures

List of Tables

Glossary

API "An application programming interface (API) is a computing interface that defines interactions between multiple software intermediaries."[1]. 8, 11, 13, 15, 16, 24, 26, 27, 40, 42, 43, 52, 54, 57, 66, 75

Developer Experience Developer Experience (DX) describes the experience that developers make in their everyday life when designing and programming software systems. Mezzalira [57] states that it is crucial to study and explore tools that developers use in der daily life, with the main goal to simplify the development process from setting up the system until deployment.. 2, 70

DevOps "DevOps is an approach to culture, automation, and platform design intended to deliver increased business value and responsiveness through rapid, high-quality service delivery. This is all made possible through fast-paced, iterative IT service delivery. DevOps means linking legacy apps with newer cloud-native apps and infrastructure."[2]. 10, 13, 17, 36, 75

HTTP "The Hypertext Transfer Protocol (HTTP) is an application layer protocol for distributed, collaborative, hypermedia information systems."[3] [32]. 8, 22, 24, 27, 38

MOS The MOS (Mean Opinion Score) is the arithmetic mean over all individual values (resulting in a range of 1 to 5), which represents the overall quality of a stimulus or system.[4]. 61

[1] https://en.wikipedia.org/wiki/API
[2] https://www.redhat.com/en/topics/devops
[3] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[4] https://en.wikipedia.org/wiki/Mean_opinion_score

Acronyms

AJAX Asynchronous JavaScript and XML. 24–26, 30, 63

ATF Above-the-Fold. 62

BFF Backend for Frontend. 15, 16, 29, 42

CD Continuous Delivery. 1, 4, 12, 13, 36, 55, 56, 64, 70, 71, 73, 76, 77

CDN Content Delivery Network. 20, 28, 39, 43, 47

CEO Chief Executive Officer. 40

CI Continuous Integration. 4, 12, 13, 36, 55, 56, 64, 70, 73, 76, 77

CLI Command Line Interface. 37, 47, 53

CMS Content Management System. 30

CPU Central Processing Unit. 24, 38

CSS Cascading Style Sheet. 36, 54, 69

CTO Chief Technology Officer. 14

DDD Domain-Driven Design. 13

DOM Document Object Model. 20, 22–25, 27, 28, 54, 62, 65, 68, 69, 77

E2E End-to-End. 39

ERP Enterprise Resource Planning. 3, 4, 45, 50, 52, 75

ESI Edge Side Includes. 4, 23, 30, 39, 42, 43, 47

GCP Google Cloud Platform. 56, 57, 59, 66

HMI Human Machine Interface. 3

HTML Hypertext Markup Language. 1, 23, 25, 26, 28, 29, 37, 43, 63, 69

IaaS Infrastructure as a Service. 1

IaC Infrastructure as Code. 36

ID identifier. 16

IIFE Immediately Invoked Function Expression. 25

JS JavaScript. 25–28, 54, 56, 62, 66, 69

JSON JavaScript Object Notation. 22, 57

LOC Lines of Code. 10, 13, 14, 34, 63, 65, 67, 69, 71, 73, 77

MF Micro Frontend. 2–5, 7, 20–30, 33–47, 49–57, 59, 60, 63–65, 67–71, 75, 76, 79

MFA Micro Frontend architecture. 2–5, 7, 20, 21, 29, 30, 33, 35–41, 43–45, 47, 49, 50, 55, 56, 60, 63–66, 69–71, 75, 76

MS Microservice. 1, 2, 4, 5, 7–19, 33–35, 37, 38, 40–44, 60, 64, 75, 76, 79

MSA Microservice Architecture. 1, 2, 7–20, 24, 30, 34, 39–41, 43, 44, 49, 52, 64, 75, 76

OS Operating System. 76

PaaS Platform as a Service. 1

PoC Proof of Concept. 76

PWA Progressive Web App. 30, 67

QoE Quality of Experience. 60, 61, 63, 65

REST Representational State Transfer. 11, 40, 43, 52, 53

ROI Return on Invest. 18

SaaS Software as a Service. 1

SEO Search Engine Optimization. 23–25

SOA Service-oriented architecture. 1, 7, 8, 12

SOC service-oriented computing. 8

SPA Single Page Application. 3, 4, 20, 28–30, 45, 47, 49, 50, 52, 54, 66–71, 75, 76

SSI Server Side Includes. 4, 23, 39, 42, 43, 47

SSO Single Sign-On. 21

SSR Server-Side Rendering. 2, 4, 66

TTFB Time To First Byte. 4, 62

TTFP Time To First Paint. 62

TTI Time to interact. 4, 62

UI User Interface. 2, 3, 5, 14–16, 20, 23, 26, 38–40, 59, 62, 75

URI Uniform Resource Identifier. 22

URL Uniform Resource Locator. 21–23, 26, 28, 29, 38, 54–57

UX User Experience. 3, 5, 21, 22, 25, 29, 34, 39, 49, 59–61, 63

VM Virtual Machine. 56

WWW World Wide Web. 1

XaaS Everything as a Service. 1

Bibliography

[1] Micro Frontends (2016), https://www.thoughtworks.com/radar/techniques/micro-frontends, (last visited: 2020-12-12)

[2] Usage study on Microservices (2018), https://www.globenewswire.com/news-release/2018/09/20/1573625/0/en/New-Research-Shows-63-Percent-of-Enterprises-Are-Adopting-Microservices-Architectures-Yet-50-Percent-Are-Unaware-of-the-Impact-on-Revenue-Generating-Business-Processes.html{#}:{~}:text=Sixty-t, (last visited: 2021-03-23)

[3] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Others: A view of cloud computing. Communications of the ACM **53**(4), 50–58 (2010)

[4] Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to Cloud-Native architectures using microservices: An experience report. Communications in Computer and Information Science **567**, 201–215 (2016)

[5] Balalaie, A., Heydarnoori, A., Jamshidi, P., Balalaie, C.A., Heydarnoori, A., Jamshidi, P.: Microservices Migration Patterns. Software: Practice and Experience **48**(11) (2015)

[6] Baraković, S., Skorin-Kapov, L.: Survey of research on Quality of Experience modelling for web browsing. Quality and User Experience **2**(1) (2017). https://doi.org/10.1007/s41233-017-0009-2

[7] Beimborn, D., Miletzki, T., Wenzel, S.: Platform as a service (PaaS). Business & Information Systems Engineering **3**(6), 381–384 (2011)

[8] Berners-Lee, T.: www-talk from september to october 1991 (1991), https://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html, (last visited: 2020-12-12)

[9] Betts, T., Humble, C., Bryant, D., Stenberg, J.: Software Architecture and Design InfoQ Trends Report (2020), https://www.infoq.com/articles/architecture-trends-2020/, (last visited: 2020-12-01)

[10] Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The role of trust management in distributed systems security. In: Secure Internet Programming, pp. 185–210. Springer (1999)

[11] Bocchi, E., De Cicco, L., Rossi, D.: Measuring the quality of experience of web users. Internet-QoE 2016 - Proceedings of the 2016 ACM SIGCOMM Workshop on QoE-Based Analysis and Management of Data Communication Networks, Part of SIGCOMM 2016 **46**(4), 37–42 (2016). https://doi.org/10.1145/2940136.2940138

[12] Bouch, A.: Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service (2000). https://doi.org/10.1145/332040.332447

[13] Brousse, N.: The issue of monorepo and polyrepo in large enterprises. In: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming. pp. 1–4 (2019)

[14] Brutlag, J., Abrams, Z., Meenan, P.: Above the fold time: Measuring web page performance visually. In: Velocity: Web Performance and Operations Conference (2011)

[15] Butkiewicz, M., Madhyastha, H.V., Sekar, V.: Understanding website complexity: Measurements, metrics, and implications. Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC pp. 313–328 (2011). https://doi.org/10.1145/2068816.2068846

[16] Buyya, R.: Cloud computing: The next revolution in information technology. In: 2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010). pp. 2–3. IEEE (2010)

[17] Choi, J.: Why Jeff Bezos' Two-Pizza Team Rule Still Holds True in 2018 (2018), http://blog.idonethis.com/two-pizza-team/, (last visited: 2020-12-09)

[18] Cojocaru, M.D., Oprescu, A., Uta, A.: Attributes assessing the quality of microservices automatically decomposed from monolithic applications. Proceedings - 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC 2019 (1), 84–93 (2019). https://doi.org/10.1109/ISPDC.2019.00021

[19] Compton, B., Withrow, C.: Prediction and control of ADA software defects. Journal of Systems and Software **12**(3), 199–207 (1990). https://doi.org/https://doi.org/10.1016/0164-1212(90)90040-S, http://www.sciencedirect.com/science/article/pii/016412129090040S

[20] Darling, J.M., Nation, D., Jibins, J.: Systems and methods for developing a web application using micro frontends (2020)

[21] Di Francesco, P.: Architecting microservices. Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings pp. 224–229 (2017). https://doi.org/10.1109/ICSAW.2017.65

[22] Di Francesco, P., Lago, P., Malavolta, I.: Migrating Towards Microservice Architectures: An Industrial Survey. Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018 (Section VII), 29–38 (2018). https://doi.org/10.1109/ICSA.2018.00012

[23] Di Francesco, P., Malavolta, I., Lago, P.: Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017 pp. 21–30 (2017). https://doi.org/10.1109/ICSA.2017.24

[24] Doigneau, R.: Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services, vol. 5. Pearson Education, Boston (2012)

[25] Dospinescu, O., Fotache, D., Munteanu, B.A., Hurbean, L.: Mobile enterprise resource planning: New technology horizons. Innovation and Knowledge Management in Business Globalization: Theory and Practice - Proceedings of the 10th International Business Information Management Association Conference **1-2**, 73–79 (2008)

[26] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. Present and Ulterior Software Engineering pp. 195–216 (jun 2016). https://doi.org/10.1007/978-3-319-67425-4_12, http://arxiv.org/abs/1606.04036

[27] Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N.C., Hu, B.: Everything as a service (XaaS) on the cloud: origins, current and future trends. In: 2015 IEEE 8th International Conference on Cloud Computing. pp. 621–628. IEEE (2015)

[28] Egger, S., Reichl, P., Hosfeld, T., Schatz, R.: 'Time is bandwidth'? Narrowing the gap between subjective time perception and Quality of Experience. IEEE International Conference on Communications pp. 1325–1330 (2012). https://doi.org/10.1109/ICC.2012.6363769

[29] El Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., Rai, S.N.: The optimal class size for object-oriented software. IEEE Transactions on Software Engineering **28**(5), 494–509 (may 2002). https://doi.org/10.1109/TSE.2002.1000452

[30] Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., Luo, M., Newling, T.: Patterns: service-oriented architecture and web services. IBM Corporation, International Technical Support Organization New York, NY (2004)

[31] Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software: Amazon.de: Eric J. Evans: Fremdsprachige Bücher **7873**(415), 529 (2003)

[32] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol–HTTP/1.1 (1999)

[33] Fowler, M., Foemmel, M.: Continuous Integration (2006), https://www.thoughtworks.com/continuous-integration, (last visited: 2020-11-17)

[34] Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., Self, F.M., Microservices, R., Ábrahám, E., Bonsangue, M., Broch, E., Theory, J.: Self-Reconfiguring Microservices To cite this version : HAL Id : hal-01336688 Self-Reconfiguring Microservices. Theory and Practice of Formal Methods (2016)

[35] Galletta, D., Henry, R., McCoy, S., Polak, P.: Web Site Delays: How Tolerant are Users? J. AIS **5**, 0– (2004). https://doi.org/10.17705/1jais.00044

[36] Gasser, M., Goldstein, A., Kaufman, C., Lampson, B.: The Digital distributed system security architecture. In: Proceedings of the 12th National Computer Security Conference. pp. 305–319 (1989)

[37] George, F.: Micro service Architecture (2012), https://slides.yowconference.com/yow2014/George-ImplementingMicroserviceArchitectures.pdf?feature=oembed, (last visited: 2020-12-12)

[38] Ghofrani, J., Lübke, D.: Challenges of Microservices Architecture: A Survey on the State of the Practice. Zeus (April), 9–16 (2018)

[39] Gronau, N., Fohrholz, C., Plygun, A.: Mobile Prozesse im ERP. HMD Praxis der Wirtschaftsinformatik **49**(4), 23–31 (2012). https://doi.org/10.1007/bf03340715

[40] Harms, H., Rogowski, C., Lo Iacono, L.: Guidelines for adopting frontend architectures and patterns in microservices-based systems. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering **Part F1301**, 902–907 (2017). https://doi.org/10.1145/3106237.3117775

[41] Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce. Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings pp. 243–246 (2017). https://doi.org/10.1109/ICSAW.2017.11

[42] Hatton, L.: Reexamining the fault density component size connection. IEEE Software **14**(2), 89–97 (mar 1997). https://doi.org/10.1109/52.582978

[43] Herbsleb, J.D., Grinter, R.E.: Architectures, coordination, and distance: Conway's law and beyond. IEEE software **16**(5), 63–70 (1999)

[44] Hicks, B.J., McWhirter, B.K., McArthuer, D., Williams, B.: Mobile Barcode Scanner Gun System with mobile Tablet Device having amMobile Pos and Enterprise Resource Planning application for customer checkout/order fulfillment and real time in store inventory management for retail establishment (2016)

[45] Humble, J., Farley, D.: Continuous delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education, Inc. All, Boston (2011)

[46] Hüttermann, M.: DevOps for developers (expert's voice in Web development). Apress, Berkeley, CA, USA (2012)

[47] Jackson, C.: Micro Frontends (2019), https://martinfowler.com/articles/micro-frontends.html, (last visited: 2020-12-03)

[48] Krause, L.: Microservices: Patterns and Applications: Designing fine-grained services by applying patterns (2015), http://www.amazon.com/Microservices-Patterns-Applications-Designing-fine-grained-ebook/dp/B00VJ3NP4A)

[49] Kurbel, K., Dabkowski, A., Jankowska, A.M.: A Multi-tier Architecture for Mobile Enterprise Resource Planning. Wirtschaftsinformatik 2003/Band I (January 2003) (2003). https://doi.org/10.1007/978-3-642-57444-3

[50] Lewis, J.: Micro services - java, the unix way (2012), http://2012.33degree.org/talk/show/67, (last visited: 2020-12-03)

[51] Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term (2014), http://martinfowler.com/articles/microservices.html, (last visited: 2020-11-17)

[52] Linthicum, D.S.: Practical Use of Microservices in Moving Workloads to the Cloud. IEEE Cloud Computing **3**(5), 6–9 (2016). https://doi.org/10.1109/MCC.2016.114

[53] Mauro, T.: Adopting microservices at netflix: Lessons for team and process design. (2015), https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/, (last visited: 2020-11-17)

[54] Mazlami, G., Cito, J., Leitner, P.: Extraction of Microservices from Monolithic Software Architectures. Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017 pp. 524–531 (2017). https://doi.org/10.1109/ICWS.2017.61

[55] Mazzara, M., Govoni, S.: A case study of web services orchestration. In: International Conference on Coordination Languages and Models. pp. 1–16. Springer (2005)

[56] Mena, M., Corral, A., Iribarne, L., Criado, J.: A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things. IEEE Access **7**, 104577–104590 (2019). https://doi.org/10.1109/access.2019.2932196

[57] Mezzalira, L.: Building Micro Frontends. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2020)

[58] Michael Geers: Micro Frontends in Action (2020)

[59] Mikowski, M., Powell, J.: Single Page Web Applications: JavaScript End-to-End. Manning Publications Co., USA, 1st edn. (2013)

[60] Møller, C.: ERP II - Next-generation Extended Enterprise Resource Planning. Supply Chain Management An International Journal **18**, 6 (2002), http://pure.au.dk/portal/files/32334597/0003167.pdf

[61] Nah, F.F.H.: A study on tolerable waiting time: how long are web users willing to wait? Behaviour & Information Technology **23**(3), 153–163 (2004)

[62] Newman, S.: Building Microservices. O'Reilly Media, Inc., 1st edn. (2015)

[63] Nygard, M.T.: Release it!: design and deploy production-ready software. Pragmatic Bookshelf (2018)

[64] O'Hanlon, C.: A conversation with Werner Vogels. Queue **4**(4), 14–22 (2006)

[65] Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14). pp. 305–319. {USENIX} Association, Philadelphia, PA (2014), https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[66] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. Computer **40**(11), 38–45 (2007)

[67] Pavlenko, A., Askarbekuly, N., Megha, S., Mazzara, M.: Micro-frontends: Application of microservices to web front-ends. Journal of Internet Services and Information Security **10**(2), 49–66 (2020). https://doi.org/10.22667/JISIS.2020.05.31.049

[68] Peltonen, S., Mezzalira, L., Taibi, D.: Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review (2020), http://arxiv.org/abs/2007.00293

[69] Richardson, C.: Pattern: API Gateway / Backends for Frontends (2017), https://microservices.io/patterns/apigateway.html, (last visited: 2020-11-14)

[70] Richardson, C.: Pattern: Monolithic Architecture (2017), https://microservices.io/patterns/monolithic.html, (last visited: 2020-11-14)

[71] Richardson, C.: Microservices patterns. Manning Publications Company, (2018), http://microservices.io/patterns/microservices.html

[72] Rossberg, J., Olausson, M.: Continuous Delivery. In: Pro Application Lifecycle Management with Visual Studio 2012, pp. 425–432. Springer (2012)

[73] Schulte-Coerne, T.: Optionen der Frontend-Integration. Entwickler Spezial **22**, 76–83 (oct 2019)

[74] Schütz, S.W., Kude, T., Popp, K.M.: The impact of software-as-a-service on software ecosystems. In: International Conference of Software Business. pp. 130–140. Springer (2013)

[75] Springer, S.: Vom Backend ins Frontend. Entwickler Spezial **22**, 84–90 (2019)

[76] Steinacker, G.: On Monoliths and Microservices (2015), https://dev.otto.de/2015/09/30/on-monoliths-and-microservices/, (last visited: 2020-12-04)

[77] Steyer, M.: A Software Architect's Approach Towards Using Angular (And SPAs In General) For Microservices Aka Microfrontends (2018), https://www.angulararchitects.io/aktuelles/a-software-architects-approach-towards/, (last visited: 2020-12-09)

[78] Steyer, M.: 6 Steps to your Angular-based Microfrontend Shell (2019), https://www.angulararchitects.io/aktuelles/6-steps-to-your-angular-based-microfrontend-shell/, (last visited: 2020-12-02)

[79] Steyer, M.: Architektur für agile Teams. windows .developer **6**, 40–46 (2020)

[80] Steyer, M.: Enterprise Angular: DDD, Nx Monorepos and Micro Frontends. Leanpub, Online (2020), https://www.angulararchitects.io/book

[81] Steyer, M.: The Microfrontend Revolution: Module Federation in Webpack 5 (2020), https://www.angulararchitects.io/aktuelles/the-microfrontend-revolution-module-federation-in-webpack-5/, (last visited: 2020-12-09)

[82] Steyer, M.: The Microfrontend Revolution – Part 2: Module Federation with Angular (2020), (last visited: 2021-01-27)

[83] Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing **4**(5), 22–32 (2017). https://doi.org/10.1109/MCC.2017.4250931

[84] Thönes, J.: Microservices. IEEE Software **32**(1) (2015). https://doi.org/10.1109/MS.2015.11

[85] Tsimelzon, M., Weihl, B., Chung, J., Frantz, D., Basso, J., Newton, C., Hale, M., Jacobs, L., O'Connel, C.: ESI Language Specification 1.0 (2001), (last visited: 2020-12-04)

[86] Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 2015 10th Colombian Computing Conference, 10CCC 2015 (October), 583–590 (2015). https://doi.org/10.1109/ColumbianCC.2015.7333476

[87] Vinoski, S.: REST Eye for the SOA Guy. IEEE Internet Computing **11**(1), 82–84 (2007)

[88] Vosshall, P.: Web scale computing: The power of infrastructure as a service. In: International Conference on Service-Oriented Computing. p. 1. Springer (2008)

[89] Wang, A., Sudhir, T.: Announcing Ribbon: Tying the Netflix Mid-Tier Services Together (2013), https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62, (last visited: 2020-11-17)

[90] Wang, D., Yang, D., Zhou, H., Wang, Y., Hong, D., Dong, Q., Song, S.: A Novel Application of Educational Management Information System based on Micro Frontends. Procedia Computer Science **176**, 1567–1576 (2020). https://doi.org/10.1016/j.procs.2020.09.168, https://doi.org/10.1016/j.procs.2020.09.168

[91] Wolff, E.: Microservices: flexible software architecture. Addison-Wesley Professional (2016)

[92] Yang, C., Liu, C., Su, Z.: Research and Application of Micro Frontends. IOP Conference Series: Materials Science and Engineering **490**(6) (2019). https://doi.org/10.1088/1757-899X/490/6/062082

[93] Zafer, Ö.: Understanding Micro Frontends (2020), https://hackernoon.com/understanding-micro-frontends-b1c11585a297, (last visited: 2020-12-09)

[94] Zimmermann, O.: Microservices tenets: Agile approach to service development and deployment. Computer Science - Research and Development **32**(3-4), 301–310 (2017). https://doi.org/10.1007/s00450-016-0337-0

[95] Zúñiga-Prieto, M., Insfran, E., Abrahao, S., Cano-Genoves, C.: Incremental integration of microservices in cloud applications. 25th International Conference on Information Systems Development, ISD 2016 (July), 93–105 (2016)

Appendix

Gitlab repositories

- Angular Monolith

  `https://gitlab.com/01526926/magicshop-angular-monolith`

- single-spa

  `https://gitlab.com/01526926/single-spa-root-config`

  `https://gitlab.com/01526926/single-spa-authentication`

  `https://gitlab.com/01526926/single-spa-navigation`

  `https://gitlab.com/01526926/single-spa-dashboard`

  `https://gitlab.com/01526926/single-spa-sales`

  `https://gitlab.com/01526926/single-spa-article`

- single-spa Helper

  `https://gitlab.com/01526926/single-spa-helper`

- Import Map Deployer

  `https://gitlab.com/01526926/import-map-deployer` (copy of `https://github.com/single-spa/import-map-deployer`)

- Module federation

  `https://gitlab.com/01526926/module-federation-shell`

  `https://gitlab.com/01526926/module-federation-authentication`

  `https://gitlab.com/01526926/module-federation-dashboard`

  `https://gitlab.com/01526926/module-federation-article`

  `https://gitlab.com/01526926/module-federation-sales`

- WebPageTest script

  `https://gitlab.com/01526926/webpagetest-script`

WebPageTest Results

- Angular monolith

  https://www.webpagetest.org/results.php?test=210128_Di3P_92a9d083d527067bf5521791fe7f2cda

- single-spa

  https://www.webpagetest.org/results.php?test=210128_Di68_aeba988d13ef793991b055e63ebe0864

- Module Federation

  https://www.webpagetest.org/results.php?test=210128_DiBW_45af0f4a463d2b1d9956de7532a1be8c