HAW HAMBURG

MASTER THESIS
Kübra Tokuc

Suitability of Micro-Frontends for an AI as a Service Platform

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Kübra Tokuc

Suitability of Micro-Frontends for an AI as a Service Platform

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Ulrike Steffens
Supervisor: Prof. Dr. Lars Hamann

Submitted on: 22.12.2023

Kübra Tokuc

Thema der Arbeit

Eignung von Micro-Frontends für eine AI as a Service Plattform

Stichworte

Micro-Frontends, Frontend Architekturen, Module Federation, AI as a Service Platform, Synthetische Datengenerierung

Kurzzusammenfassung

Die Microservices-Architektur fördert die Entwicklung von wartbaren, erweiterbaren und skalierbaren Softwarelösungen in cloudbasierten, verteilten Umgebungen. Die Plattform des Forschungsprojekts DaFne, ein "Artificial Intelligence as a Service" (AIaaS), nutzt diese Ansätze bereits im Backend. Dadurch stellt es eine Plattform bereit, die erweiterbar ist und es ermöglicht, neue AI Software Services für die Generierung von synthetischen Daten hinzuzufügen. Während das Backend bereits erweiterbar ist, zeigt der ursprüngliche Architekturentwurf einen UI-Monolithen als einzigen Frontend-Dienst. Diese Arbeit erforscht die Anwendung von Micro-Frontends unter Nutzung der Library Module Federation, um auch im Frontend eine Erweiterbarkeit zu ermöglichen. Die Plattform wird dabei in einzelne Frontend-Module aufgeteilt, die aus verschiedenen Domänen stammen und unterschiedliche Anforderungen an den Tech Stack haben. Unter Berücksichtigung der Prinzipien und Herausforderungen von Micro-Frontends und der Entwicklung einer Event-basierten Kommunikationsstrategie wird der erstellte Systementwurf in Form eines Prototyps umgesetzt, der das Ergebnis dieser Forschung darstellt. Als Validierung wird ein extern mit Vue.js entwickelter AI Software Service in die React-basierte Host-App integriert. Die Vorteile zeigen sich vor allem in der Flexibilität der Systemgestaltung und im Maintenance-Bereich, allerdings eignet sich die Architektur vornehmlich für größere Projekte mit mehreren Entwicklerteams aufgrund der komplexen Herausforderungen und des hohen Implementierungsaufwands.

Kübra Tokuc

Title of Thesis

Suitability of Micro-Frontends for an AI as a Service Platform

Keywords

Micro-Frontends, Frontend Architectures, Module Federation, AI as a Service Platform, Synthetic Data Generation

Abstract

The microservices architecture promotes the development of maintainable, extensible, and scalable software solutions in cloud-based, distributed environments. The DaFne research project's platform, characterized as "Artificial Intelligence as a Service" (AIaaS), already leverages these approaches in the backend. Consequently, it provides a platform that is extensible and allows for the addition of new AI software services for synthetic data generation. While the backend is already extensible, the original architectural design featured a UI monolith as the sole frontend service. This work explores the application of micro-frontends using the Module Federation library to enable extensibility in the frontend as well. The platform is divided into individual frontend modules originating from different domains and having varying requirements for the tech stack. Taking into account the principles and challenges of micro-frontends and the development of an event-based communication strategy, the designed system architecture is implemented as a prototype, representing the result of this research. As validation, an externally developed AI software service using Vue.js is integrated into the React-based host app. The advantages primarily manifest in the flexibility of system design and maintenance. However, this architecture is best suited for larger projects with multiple development teams due to the complex challenges and implementation effort involved.

Contents

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

2 Background **3**

2.1 Synthetic Data Generation Platform 3

2.1.1 DaFne: Platform Data Fusion Generator for Artificial Intelligence . 3

2.1.2 Artificial Intelligence as a Service 4

2.1.3 Quality Attribute: Extensibility 6

2.2 Architectural Styles and Patterns 9

2.2.1 Microservices Architecture 9

2.2.2 Event-Driven Architecture (EDA) 10

2.2.3 Domain Driven Design (DDD) 11

2.2.4 API Gateway and Backend for Frontend (BFF) 12

2.3 The UI Monolith 12

2.4 Micro-Frontends 15

2.4.1 Benefits of Adapting Micro-Frontends 16

2.4.2 Challenges and Decisions 17

2.4.3 Available Approaches for Implementation 20

3 Context and Requirements Analysis **23**

3.1 Problem Scenario 23

3.2 Existing System: Module Federation and React 25

3.3 Domain Analysis 30

3.4 Requirements 34

4 System Design **37**

4.1 Identification of Micro-Frontends 37

4.2 Updated Technology Stack . . . 38
4.2.1 Vue.js . . . 40
4.2.2 Next.js . . . 40
4.3 Communication between Micro-Frontends . . . 40
4.4 Deployment and Maintenance . . . 42

5 Implementation **46**
5.1 Configuration of Apps . . . 46
5.1.1 Bootstrapping and Loading Micro-Frontends . . . 46
5.1.2 Routing . . . 48
5.1.3 Performance Considerations . . . 50
5.1.4 Fault Resilience . . . 51
5.2 UX/UI Consistency . . . 53
5.3 Communication and State management . . . 54
5.3.1 API Gateway and State Management . . . 55
5.3.2 Authentication and Security . . . 55
5.4 SEO with Next.js for Marketing App . . . 57
5.5 Integration of Vue.js Application Neighborhood Generation . . . 59
5.6 Deployment and Maintenance . . . 60
5.6.1 CI/CD . . . 61
5.6.2 Medusa Client . . . 63

6 Evaluation **64**
6.1 Constraint: Implementation of Microfrontend Architecture for extensibility 64
6.2 Consistency in User Experience . . . 66
6.3 Performance . . . 67
6.4 Communication and State Management . . . 68
6.5 Security and Authentication . . . 69
6.6 Search Engine Optimization . . . 70
6.7 Deployment and Maintenance . . . 71

7 Conclusion **72**

Bibliography **75**

A Appendix **82**
Declaration of Autorship . . . 85

List of Figures

2.1 Generic Platform Functionalities [KKZS22] 4
2.2 AIaaS stack [LPT+21, p. 443] 5
2.3 AI Software Services [LPT+21, p. 444] 7
2.4 (a) API Gateway and (b) Backend for Frontend [HRLI17, p. 904] 12
2.5 From Microservices Micro-Frontends [PPS21] 16
2.6 Horizontal vs. Vertical Split [Mez21, p. 177] 18

3.1 Vertical Split into three applications 26
3.2 Navigation Content of the Core Domain 31

4.1 High-Level System Design as Solution Strategy 39
4.2 Communication Design based on API Gateway and Event-Based Architecture 42
4.3 Using AWS Cloudfront CDN and S3 Buckets to serve Build Files to the Browser 44
4.4 Automation Pipeline 45

5.1 File structure of Shell Application container 49
5.2 Routing behavior example between Container, Marketing and Auth . . . 51
5.3 Implemented Security and Authentication Communication Flow 56

6.1 Neighborhood App Integrated into Dafne App 65
6.2 UML Diagram in Medusa Client showing the Federated Landscape 66
6.3 Switching between different Versions of the Design System 67
6.4 Lighthouse Analysis Output of Isolated Next.js App 70
6.5 Lighthouse Analysis Output of Integrated Next.js App inside the Container App 71

List of Tables

3.1 Domains of the DaFne Platform according to Domain Driven Design (DDD) 32

4.1 Identified Micro-Frontends inside the Domains 39

5.1 State Management and Data Fetching Technologies of each Micro-Frontend 55

6.1 Performance Comparison 68

1 Introduction

Microservices and service-oriented architecture in general have established themselves as a common practice in modern agile software development. Advantages such as improved scalability, maintainability and extensibility are just some of the reasons why many companies opt for this architecture. However, microservice architecture is not limited to back-end development. There is also a paradigm shift from monolithic applications to micro-frontend architectures in the frontend area. With micro-frontends, the frontend code is also divided into smaller, independent units. These units can be developed as independent applications by different teams and can be deployed independently. The term micro-frontends was first introduced by ThoughtWorks in 2016 [Tho16]. Since then, the architecture has become increasingly popular in practice and is being adapted by large companies such as DAZN, Zalando or Ikea. In the scientific literature, but also in comments from the software developer community on social platforms, it is often emphasized that the architecture tends to benefit larger software projects. The higher implementation effort pays off in the long run through easier and safer management of the software project [TM22, PMT21, PPS21, PAMM20]. In this work, the micro-frontend approach is applied to the Data Fusion Generator (DaFne) platform. The DaFne research project aims to develop a platform for synthetic data generation, which makes it an Artificial Intelligence as a Service (AIaaS) [LPT+21]. Users are offered a graphical user interface through which they can access various machine learning algorithms for synthetic data generation. The backend architecture of the platform was presented by Kunert et al. (2022) [KKZS22]. In the microservice-based architecture, the user interface was initially listed as a single service. The non-functional platform requirement extensibility aims to ensure that the platform can be extended with different algorithms and ML models for data generation without much effort.

Although methods for synthetic data generation are already widely researched and already accessible to computer science-savvy individuals, synthetic data may also be of interest to domain experts in other disciplines, e.g., smart cities. Therefore, the platform offers domain- and use-case-specific methods (e.g. Neighborhood Generation) for data

generation in addition to generic in-house methods. When a new service is added, a front-end for that service must also exist. The extensibility of the platform is therefore also dependent on the extensibility of the user interface.

The objective of this work is to investigate how micro-frontend architecture facilitates the extension of AI services on a software platform. The central research question is:

> *How can the micro-frontend architecture support the extensibility of an AI as a Software Service platform at the front-end level?*

To investigate this research question, the Design Science Research (DSR) methodology is applied to create a practical prototype, aligning the research with a structured, problem-solving approach. DSR involves iterative cycles of design and evaluation, with a focus on developing innovative solutions to address real-world problems (Dresch et al., 2015). This methodology forms the foundation of the thesis structure:

Chapter 2 provides a knowledge base, explaining prior platform characterizations, general software architectural concepts followed by an introduction to monolith-based frontend architectures and lastly micro-frontends. This chapter explores the necessity of micro-frontends and their role in resolving frontend modularization challenges.

Chapter 3 analyzes the environment and organizational aspects, defines a problem scenario, and conducts a Domain-Driven Design (DDD) analysis to set prototype requirements. It also references an existing system using Module Federation and React, developed as a preliminary experiment without comprehensive context and domain analysis.

Chapter 4 designs a system based on requirements and domain analysis from Chapter 3, which is implemented in Chapter 5. Chapter 6 evaluates this implementation against the requirements and the intended system design. The thesis concludes in Chapter 7 with a summary and outlook.

2 Background

This chapter establishes the knowledge base required for the Design Science Research methodology applied in this thesis. It starts by presenting general knowledge about the Synthetic Data Generation Platform and its associated characteristics. The chapter then gradually introduces the concept of micro-frontends, initially exploring relevant architectural styles and patterns primarily known from backend development. This sets the stage for a deeper examination of frontend applications as typical monoliths, thereby providing a solid foundation for the introduction and understanding of micro-frontends.

2.1 Synthetic Data Generation Platform

This chapter broadly outlines the project context of this thesis, describing previous work and research conducted within the DaFne research project. It details the DaFne platform and its general functionalities, subsequently positioning it within the realms of AI as a Service. Finally, the chapter scrutinizes the quality requirement of extensibility, which is central to this thesis.

2.1.1 DaFne: Platform Data Fusion Generator for Artificial Intelligence

DaFne is a multidisciplinary research project in cooperation with universities and industry partners that targets the domains of artificial intelligence and smart cities. It aims to integrate different approaches for data generation on a digital platform in the form of services [KKZS22]. In addition to the infrastructural provision of services, one goal is to improve the usability of these methods through a well-designed user interface. The functionalities can be divided into generic methods and use case specific functions. Figure 2.1 shows an overview of the generic functionalities, which are divided into data-related, generation-related and evaluation-related services.

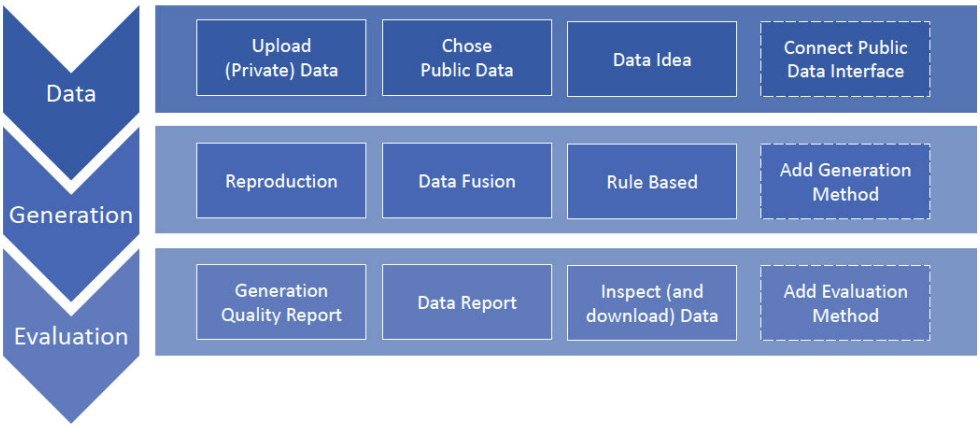

Figure 2.1: Generic Platform Functionalities [KKZS22]

First, the platform provides a data interface through which users can either upload their own data or access (smart city) Open Data through an interface managed by the platform. The variety of available data sets is also extendable by connecting new interfaces. The data generation then can be done by 3 different methods. With the *reproduction* method, the user can generate new rows of an existing dataset for either increasing the data volume for ML tasks or to circumvent privacy restrictions that would prohibit the use of the data. The *rule-based* method enables the user to compose a custom data set based on defined rules for the columns. *Data fusion* allows multiple data sets, such as private and public data, to be combined to increase the information value of a data set. In contrast to generic methods, use case specific approaches provide algorithms that are designed to solve a specific urban problem. One example is a neighborhood generation service that can generate a livability index-optimized urban map for a selected geographic area. The platform can be extended by multiple use case based algorithms. All the generated datasets are lastly evaluated by the evaluation service so the user can have an insight on the quality, usability and reliability of the data.

2.1.2 Artificial Intelligence as a Service

Cloud computing is defined as a "model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resource" with originally three service models - Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [MG+11]. With the paradigm shift in software development

and delivery, cloud computing has since spawned a variety of different service models. The trend of *Everything as a Service* simply means that almost any IT artifact can be provided as a service in the cloud [DFZ+15] - so can AI. Lins et al. (2021) [LPT+21] introduce the definition of the emerging model *Artificial Intelligence as a Service (AIaaS)* as "cloud-based systems providing on-demand services to organizations and individuals to deploy, develop, train and manage AI models". It increases the accessibility and affordability of AI regardless of the organization's technology background by guiding the user through the process of developing, deploying or using ML models. This allows users to only focus on training or configuring the models. An attempt to conceptualize the term dividea AIaaS into the three layers of the conventional cloud service stack:

1. **AI Software Services** relating to SaaS, are ready-to-use applications that can be divided into inference as a service (accessing pre-trained models) and machine learning as a service (MLaaS, creation and customization of ML models).
2. **AI Developer Services** relating to PaaS, provide tools for coding AI capabilities.
3. **AI Infrastructure Services** relating to IaaS, offer computational power for building and training AI algorithms, network and data storage, sharing capacities.

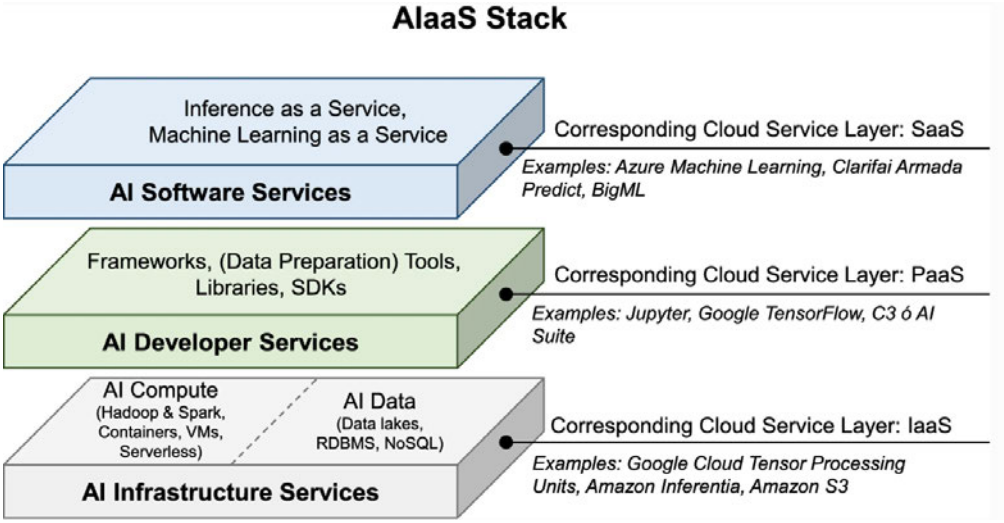

Figure 2.2: AIaaS stack [LPT+21, p. 443]

By enabling the usage of generative models, the DaFne platform primarily forms an **AI Software Service**. The generic methods can be regarded as **machine learning as a service (MLaaS)** as they offer the training and the customization of ML models. Using it requires a basic knowledge of the user about terms of ML, while the software

service is only a guidance and assistance. The user is helped by the fact that he only has to focus on training and parameterization instead of carrying out infrastructural installation configurations. Typically reaching the final prediction results would include the whole machine learning pipeline beginning with assistance in data pre-processing, feature selection and parameterization up to model validation and data evaluation (see fig. 2.3). Associated terms are deep learning as a service [BDEM+18], neural network as a service [HSM14] and training as a service [ZFZ+17].

In the example of the DaFne platform, the reproduction service enables the user to train the models Conditional Tabular Generative Adversarial Networks (CTGAN) and Tabular Variational Autoencoders (TVAE) [XSCIV19] on their own data set with possibility for custom parameterization.

In contrast, use-case-specific models like the neighborhood generation or pedestrian path generation simply provide an interface to query pre-trained models, making them available as **inference as a service**. This type of AI software service represents more of a black-box model that can be used by users whose core competencies are not related to AI. Any other tasks in the pipeline, such as training data provision, data storage, or classification, are handled by the vendors. It is simply a matter of providing predictions that benefit different areas of society.

At the same time, DaFne has the characteristics of a Platform as a Service (PaaS) with the contribution feature [MG+11]. While the internal research team is also developing generative models, the platform architecture is designed to allow contributions from external actors to enable knowledge sharing. It is important to distinguish between use-case-specific and generic approaches in this situation as well. With the hosting of urban use cases, the project aims to create an ecosystem that can leverage advancements in smart cities. Besides providing insights with synthetic data, a product feature called *Use Case Explorer* could inspire urban planners with new data ideas.

2.1.3 Quality Attribute: Extensibility

The success of software and the underlying platform primarily derive from its functional value, such as the generation and evaluation of synthetic data in the case of DaFne. However, the long-term viability of a software system depends on qualitative criteria, often referred to as “ilities” [Ing18]. The general non-functional quality criteria for the DaFne

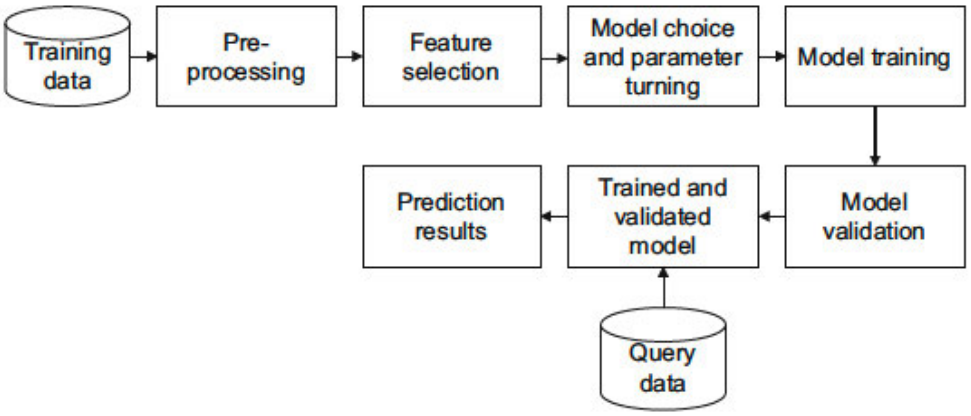

Figure 2.3: AI Software Services [LPT+21, p. 444]

platform have been defined and described as scalability, expandibility, flexibility, monitoring, and security [KKZS22]. Expandability is continued in this work as extensibility. While scalability refers to computatation power, extensibility refers to the platform's ability to accommodate changes, enhancements, and customizations to the functionality. It involves providing the means for architects and third-party developers to extend the platform's functionality seamlessly. In software architecture literature [Ing18, FRSD21], *maintainability* is often used as a generic category for the quality attributes *modifiability, extensibility and flexibility.*

Maintainability refers to the degree of ease with which a software system can be effectively taken care of and modified. It encompasses the ability to make changes, apply patches, upgrade frameworks, and accommodate evolving requirements efficiently [FRSD21]. It is crucial because as software operates, it inevitably undergoes changes, which are essential to retain the software's value over time. A significant portion of a software project's costs is attributed to maintenance rather than development, making it essential to build software that is easy to maintain. Code that is straightforward to maintain facilitates quicker maintenance tasks and helps minimize the overall lifetime costs of the software [Ing18].

Modifiability is another facet of maintainability and signifies the software's capacity to undergo changes without introducing errors or diminishing its quality. It is a vital quality attribute because software often requires alterations due to evolving needs or agile development methodologies, and enhancing modifiability not only benefits maintenance but also the entire software development process.

Extensibility and flexibility pertain to a software system's ability to accommodate

future growth and adapt to unforeseen changes. Extensibility revolves around the ease of adding new functionalities and features, while flexibility deals with altering capabilities for unanticipated uses. These qualities significantly influence the ease of performing perfective maintenance, ensuring that a software system remains adaptable and open to enhancements.

While modular architectures can contribute to maintainability, they cannot be universally applied, as monolithic architectures can also provide maintainability. Therefore, it is advisable to introduce granularity only when needed for extensibility, such as when additional contexts need to be incorporated into the application through new features [FRSD21].

Designing maintainable systems can be supported by considering some metrics and and principles:

- Reducing component coupling: Coupling refers to the degree of interdependence between different modules or components within a system. It is essential to minimize coupling and strive for loose coupling to ensure that changes to one module do not significantly impact others.
- Increasing component cohesion: Cohesion measures the extent to which elements within a module or component are closely related and functionally connected. Design efforts should aim to enhance cohesion by avoiding the inclusion of disparate elements within a module. High cohesion often aligns with loose coupling.
- Reducing component size: Large modules are often more complex and challenging to modify, emphasizing the importance of reducing module size through strategies such as splitting them into smaller, more manageable components to enhance maintainability and flexibility.
- Reducing Cyclomatic complexity: Cyclomatic complexity is a quantitative software metric to measure the complexity of a software module. It assesses the number of linearly independent paths within a module or design element, with higher values indicating increased complexity.

2.2 Architectural Styles and Patterns

Architecture patterns are used to organize the high-level structure of a software. They provide reusable solutions to common architectural problems related to relationships, structures and behavior of components. The application of the patterns typically addresses non-functional requirements such as performance, scalability, extensibility, security or maintainability in addition to functional requirements [HA10]. While in the literature the difference between styles and patterns is not always entirely clear [GS93, KMLS18], it can be said that styles tend to provide a framework and a general vocabulary for designing software (e.g. Layered, Event-Driven, Object-Oriented), while patterns provide solutions to specific problems within architectural styles [GS93]. Some of the relevant approaches are explained below.

2.2.1 Microservices Architecture

Evolved from architectural style of SOA, microservices have become a common approach to modularize the backend monolith [HRLI17]. First detailed explanation is provided 2014 by Martin Fowler [Fow14]. They describe it as a style for developing an application as a collection of multiple small and loosely coupled services, which each have their own processes and communicate via lightweight mechanisms such as HTTP (REST) API [MGM+18] or messaging services like ActiveMQ (Apache), OMG DDS (omgwiki), AWS SQS (Amazon) [ACCT21]. Each service follows the Single Responsibility Principle (SRC) and has a clear purpose organized around business capabilities which form the explicit boundary of the service.
Their service independence enables the application to be developed, deployed and scaled independently with a diverse tech stack. The microservice architecture is particularly suitable for large and complex applications that require high scalability and flexibility. Under monolithic circumstances, this applications would suffer under the ”dependency hell” that impedes the maintainability. In addition to other benefits such as modularity, agility, fault isolation and resilience, the style can also introduce some challenges [PAMM20].
The cooperation between different microservices and the inter-process communication for fluent business processes is one of the main challenges that has to be addressed with either orchestration or choreography [MGM+18].

When identifying services or decomposing a monolith, it is important to determine the degree of modularity and to make certain trade-offs in relation to available resources. Strategies for the decomposition process are widely discussed in literature [TS20]. One suggestion [Ric23] is to decompose by identifying subdomains according to Domain Driven Design, by business capabilities, by use cases (verbs) or by resources (nouns).
Another advantage of microservices is the ability to run services with heterogeneous data sources and structures. At the same time, however, this poses a challenge in terms of establishing data consistency [Fow14].
In addition, the complexity of the system increases with the number of services, which can lead to a loss of control and overview and operational overhead. It is important to implement a deployment automation including continous integration and continous deployment [PAMM20].

2.2.2 Event-Driven Architecture (EDA)

The idea behind the event-based architecture style is the implicit invocation of procedures and functions. It enables the components to be self-adaptive to changes that effect the system [GS93]. Examples of application areas are real-time systems with high-volume IoT-events [KBC+22] or in complex business processes [Tay09, Mic06] or in combination with microservice architectures [MGM+18]. In these systems, components communicate through the production, detection, and consumption of events. Events are extremely loosely coupled and asynchronous. This means that the component producing the event has knowledge only about its own states and none about which component is interested in it and none about the subsequent processing of the event [Mic06]. With the resulting responsive and flexible interaction, a scalable, extensible, resilient and maintainable overall system is enabled [Bel20].
The publish/subscribe pattern (also called the observer pattern) is a frequently applied way to handle microservice communication [ACCT21]. It defines a one-to-many relationship where consumer components can subscribe to events of interested producer components. When the state of the publisher changes, the subsriber will be notified. Techniques such as event emitter, custom events or reactive streams implement the pattern [Mez21].

2.2.3 Domain Driven Design (DDD)

While the microservice architecture represents design at a higher level of abstraction, design is more an activity for the creation of a solution space [EK03]. Domain Driven Design (DDD) introduced by Evans (2004) [Eva04], is a resource-oriented approach to building microservice-based web applications. It provides a set of principles, patterns and practices for identifying services by defining logical boundaries in the system called *bounded context* exposing an API. Understanding and identifying the problem domain, defining domain models, and capturing the business logic are the basis for this [SGHA17]. In the following, the key concepts of DDD are briefly explained:

- The *core domain* represents the most important part of a system and deals with the general problem space that a company occupies and for which it provides solutions. The domain includes everything related to the problem space: rules, processes, ideas, and terminologies. The domain exists independently of the existence of an application [Bel20].
- *Core subdomains* are the application's raison d'être and provide the unique business value as well as a competetive advantage to the organization. They are subordinate to the main domain and contain the most complex business logic [Mez21].
- *Supporting subdomains* are also subordinate to the main domain but only provide complementary value like discovery services [SGHA17, Mez21].
- *Generic subdomains* are not necessarily needed for the core business, but are used to complete the system. Therefore, they can be implemented with third-party services, which means that they would be suitable for other domains as well [Mez21].
- The *domain model* represents a structual abstraction for the domain. It is a collection of concepts and the relationships between them. The model is the basis for communication between domain experts and developers and states a part of the solution space [Bel20].
- *Bounded contexts* are logical boundaries that define the scope of a domain model. With inputs, outputs, events, requirements, processes, and data models, they are relevant to the subdomain. They are used to separate the domain model into smaller, more manageable parts. Each bounded context has its own domain model and is responsible for a specific part of the domain. The context is the basis for the microservice architecture. Since they should have a strong focus, they should have

strong internal cohesion and should not cross boundaries in the communication. In order to minimize the impact of changes in one context on a neighboring context, they should be loosely coupled. Communication between the contexts should only happen based on a contract represented by APIs [Bel20, Mez21].

2.2.4 API Gateway and Backend for Frontend (BFF)

For the communication of the frontend application with the microservices in the backend, the requests and responses can be exchanged via a single intermediate server. This server is responsible for aggregating the requests and directing them to the appropriate microservice, which also includes request shaping, caching and authentication. An API Gateway provides a single point of entry to a backend that can be used by multiple clients with different types. In order to distribute the logic in the gateway for different types of requests depending on the type of clients and end devices, the pattern backend for frontend can be used as shown in figure 1. Here, a separate gateway is provided for each client type or UI service as a central point for requests. In addition to performance, the advantage here is that each front-end team can develop and handle its own interface independently.

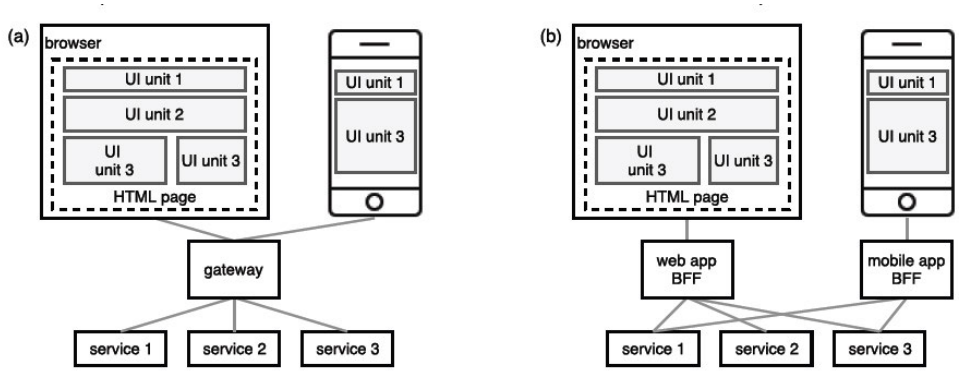

Figure 2.4: (a) API Gateway and (b) Backend for Frontend [HRLI17, p. 904]

2.3 The UI Monolith

The microservice architecture attempts to break through the backend monolith by dividing the application logic by individual services for one business function each. The

individual microservices can be developed, deployed and scaled independently of each other. When the application is divided into separate independent components in design processes such as DDD, the user interface is barely considered or just considered as a single service [SGHA17]. Web applications typically follow the client-server model, where the server, known as the provider, owns and provides access to the resources for the client, which is the service user [PAMM20]. However, even when developing the UI monolith, different architectural approaches to system development are available depending on the requirements and type of application.

Static Web Pages Static Web pages are the beginning of the Web, where HTML pages are generated in advance during the build process and sent to the client. No server-side processing is required at runtime for each user request, since the pages are already populated with the required information. While efficient, this approach is more suitable for simple non-dynamic applications such as blogs, documentation pages, or landing pages [Mez21].

Single Page Applications Single Page Applications (SPA) are particularly well suited for interactive and dynamic web applications. This is based on client-side rendering, where the client (e.g. the browser) preloads the entire code, like the HTML and JavaScript files, and manipulates the Document Object Model (DOM) on runtime with JavaScript. When the user first accesses the application, the server or the Content Delivery Network (CDN) only serves an empty root element `<div id="root">` and some JavaScript code to the client. Then, the root element is updated by the Single Page Applications (SPA) with the necessary files such as JavaScript or CSS according to the user's need. Besides compiling the JSX files, the client is responsible for loading the data through the API calls, handling events and promises. It uses only a "Thin Server" which acts only as a data API and passes the JavaScript code to the client [Tha20]. When the user clicks on a link, the page is not refreshed, but only the content is exchanged. This gives users the feel of a native desktop application such as Google Docs [PAMM20]. Frameworks and libraries such as React [Reab], Angular [Ang] and Vue.js [Vue] support the development of SPAs. These frameworks have standardized application lifecycle methods such as `componentDidMount()` and `componentWillUnmount()`. With calling these methods, the developer can specify the application behaviour when the component is inserted into the DOM and when the component is removed from the DOM. Because

all code is downloaded at the beginning of the lifecycle, a resource-efficient application design must exist to prevent long initial load times [PMT21].

Server Side Rendering (SSR) Server Side Rendering (SSR) is a technique for rendering web pages on the server, which is particularly beneficial for Search Engine Optimization (SEO) and performance. Search engines, such as Google, rely on the HTML content of web pages to index and rank them. Since SSR provides pre-rendered HTML content, it ensures that search engines can efficiently crawl and index the content of web pages, potentially leading to improved search engine rankings and discoverability. The client receives the HTML, CSS and JavaScript files generated from the server and displays them in the browser. Here the server also executes the JavaScript code responsible for the application logic and delivers a ready-to-render HTML page that is already populated with the requested data [Tha20, Gee23]. To implement SSR, web developers often use server-side frameworks and libraries, such as Next.js for React applications or Nuxt.js for Vue.js applications. These frameworks simplify the process of server-side rendering by offering built-in functionality and routing capabilities tailored to SSR. Moreover, using these frameworks can be particularly useful for web applications with dynamic content and interactive features, as it allows for a balance between server-generated content and client-side interactivity. The disadvantage is that there is a higher load on the server, especially when handling a large number of concurrent requests. Generating HTML content on the server for each request can be resource-intensive, potentially requiring more server resources to maintain optimal performance. Besides, the implementation often involves writing complex server-side code to handle data fetching, rendering, and routing. Developers need to manage both client-side and server-side logic, which can increase the overall code complexity of the application. [Gor18].

Isomorphic Applications To overcome the disadvantages of SSR and SPAs a hybrid approach called universal or isomorphic applications emerged. The application code is shared between the client and the server and can be executed on both sides. The server is responsible for data retrieval, compilation and rendering the initial HTML, in order to pass the HTML file to the client. Then, the content only needs to get inserted into the HTML template by the client. This approach benefits SEO, Time-To-Interaction and A/B Testing by SSR [PMT21]. In addition, client side rendering allows the use of interactive and complex UI elements such as modals, as well as browser functions such

as local storage or history [Gor18]. Besides the benefits, two-sided rendering can create scalability issues when there are millions of user requests to the server [PMT21].

JAMStack architecture JAMStack stands for JavaScript, APIS and Markup and is a modern web development architecture with growing popularity [Mez21] that breaks the typical client-server model [PAMM20]. The goal is to quickly develop secure pages and dynamic applications without serving files from a web server and thus reducing download times. This is made possible by deploying pre-rendered static sites directly to the CDN without managing, scaling or patching servers. Popular frameworks that support the implementation are are Next.js, Gatsby.js and Nuxt.js [Mez21].

2.4 Micro-Frontends

While microservices have established themselves as a common practice in modern agile software development, there is also a paradigm shift in the front-end to break the previously described UI monolith. ThoughtWorks [Tho16] introduced the term micro-frontends in 2016 as an architectural style that extends the concept of microservices on the frontend side (see Fig. 2.5). It should be underlined that this approach is not about the reusable components, where the reusable frontend code is also assembled into a monolith at the end [Lan21]. A typical use case is migrating a legacy UI monolith into a micro-frontend architecture [Mez21].

The idea behind micro-frontends is to treat a web application as a combination of functions and business subdomains with the goal to enable independent development, deployment and testing of individual parts of the application [Gee23]. They have much in common with self-contained systems, where each system is an autonomous web application that fulfills a use case without depending on other services within well-defined technical boundaries [HRLI17]. Technical boundaries are also one of the main ideas in micro-frontends, as individual applications should be able to be developed technology agonostically within different teams. This isolates the team code from those of the other features and domains, so there is a smaller codebase for each team with no dependency on shared state and global configurations. Instead of relying on customized interfaces for communication between services, more is being done with native browser features such as events and local storage [Gee23].

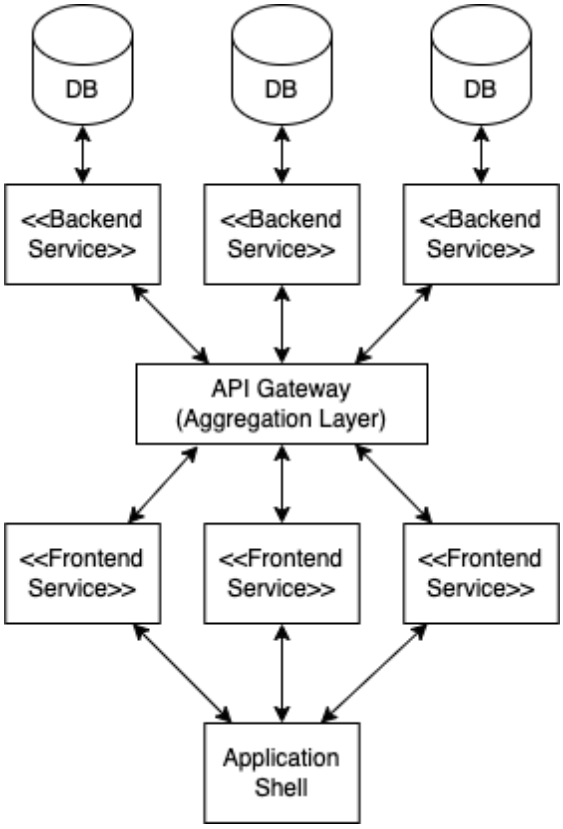

Figure 2.5: From Microservices Micro-Frontends [PPS21]

2.4.1 Benefits of Adapting Micro-Frontends

Many larger corporations and complex projects, such as Zalando, DAZN, Ikea, or SAP, that implement the micro-frontend architecture justify its use based on organizational, market-related, or technical advantages. In the context of the DevOps culture within agile software development, microservices have enabled independence and flexibility within teams. They, in fact, share the benefits of microservices as they are both modeled around business domains and hide implementation details between them [PMT21]. As a result, they simplify organizational processes, offer technical decision-making autonomy within teams, and boost work motivation. Additionally, decentralizing governance enables the creation of more specialized domain-specific teams while reducing potential conflicts in the development. [Sch, PPS21]. From a technical perspective, allowing teams to develop features independently can significantly reduce time-to-market by accelerating feature releases. This approach also facilitates faster Continuous Integration/Continu-

ous Deployment (CI/CD) pipelines. Moreover, observability benefits from better-isolated monitoring and logging, as well as improved fault isolation and detection. With this architecture, if one service encounters an issue, it doesn't cause the entire application to break. Additionally, having a smaller surface for testing can lead to quicker build times [PPS21, Mez21].

2.4.2 Challenges and Decisions

While the micro-frontend architecture promises improvements and developments in the frontend landscape, it also comes with some challenges. One drawback is that the implementation can introduce a lot of redundant code and increase code size. This redundancy can lead to performance issues as the application may load duplicate code for different micro-frontends, consuming more resources and potentially slowing down the user experience. Additionally, there is a risk that the code can diverge significantly as teams work independently, potentially affecting code quality [PPS21].

To address the drawbacks and harness the advantages, as of 2023, there is no universally accepted standardization for micro-frontend development. However, within the literature, authors seem to agree on the general concepts [TM22, PAMM20, PPS21, Mez21]. Different principles and challenges must be considered when deciding on a system design. Finding a strategy for setting up a micro-frontend architecture means finding a balance between specifications that are too loose and those that are too detailed [Mez21, chapter 9].

Luca Mezzalira, one of the dominating authors in the field of micro-frontends, structures the most important principles according to the following aspects [PMT21, Mez21, TM22].

Horizontal or Vertical Splitting

Just like the modularization of the backend into individual microservices, there is also a decision to be made at the frontend level as to where and how granularly an overall system should be split to individual applications.

- *Vertical Splitting* is the division of the application by business domains according to the principles of Domain Driven Design (DDD) and requires the setting of a bounded context. In this approach, each micro-frontend represents an SPA.
- *Horizontal Splitting* is the division into several micro-frontends within a view. This approach is well suited for the reusability of a subdomain when it is used within multiple views. Moreover, the approach is chosen when search engine optimization has a high priority.

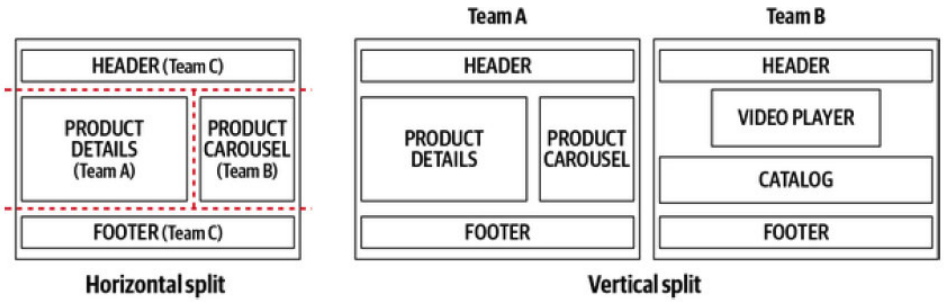

Figure 2.6: Horizontal vs. Vertical Split [Mez21, p. 177]

Composition and orchestration

The next thing to decide is how and from where the applications will load the target view when they are needed. Here it also depends on how the application is split.

1. *Client Side Composition:* An application shell loads multiple micro-frontends on runtime directly from the CDN (more common for vertical split). The entry point of the application is typically an HTML or JavaScript file, where the shell either appends the HTML file to the DOM or initializes the JavaScript file. This approach is especially, as with SPAs, good for creating a native-like dynamic app feel, since the HTML markup is updated directly in the browser [Gee23].
2. *Server Side Composition:* Microfrontends are composed on the server and sent to the client as one HTML document (more common for horizontal splits).
3. *Edge Side Composition:* Composition is done at the CDN level and is sent to the client pre-rendered.

Routing

Another crucial decision is how and where to determine which micro-frontend should be loaded for a given view. One significant challenge is that individual frontends form autonomous applications that can have an internal routing system, which must be integrated with the routing system of the overarching application shell during composition. How the routing is handled depends on the composition approach.

- *Client Side Routing:* Macro routing takes place in the application shell, while the micro-frontend handles internal routing.
- *Edge Side Routing:* Routing is based on the requested URL and handled at the CDN level.
- *Server Side Routing:* Routing based on the requested URL and handled at the server level.

Communication

The communication between micro-frontends poses a dilemma in itself because it contradicts the principle that micro-frontends should not be aware of each other's existence [PPS21]. Therefore, it is essential to minimize communication while ensuring cross-application communication. Both vertical and horizontal composition require a means to exchange data and states. Given that individual applications may be built on different technological stacks and should maintain loose coupling, developers often rely on asynchronous messaging mechanisms, such as publish/subscribe systems like event emitters or custom events. For exchanging data, native browser functionalities such as session or local storage are commonly used to ensure technology agnosticism. While there are also solutions to design and implement a remote shared state, it forms an antipattern to the micro-frontends principle because it would increase the coupling [Mez21].

UX/UI Consistency

The nature of micro-frontends, with its diversity of technologies, frameworks, and design approaches, can lead to inconsistencies in UX/UI elements. Despite the use of different applications, the merging of the applications into one user interface should provide a

uniform overall impression. The key challenges here include creating a common design system or at least shared design guidelines. Design systems are the representation of a websites visual language including a collection of design elements such as colors or fonts [God16]. Additionally, the use of reusable components, such as web components, can help facilitate the maintenance of UI consistency [Mez21].

2.4.3 Available Approaches for Implementation

Even though there is no widely accepted standard for micro-frontend development, there are several approaches and frameworks available to implement the architecture. The following list provides an overview of the most common approaches and their characteristics.

- **IFrames** offer a unique solution for loading Micro-Frontends within a sandboxed environment. They particularly support horizontal splitting, isolating each micro-frontend to prevent issues like conflicting libraries and memory leaks. They are particularly effective for desktop and B2B applications where the user's environment can be controlled. With efficient memory management that disposes of elements upon URL changes, they streamline dependency management as well. This makes them particularly well-suited for applications with limited interactivity requirements, providing a straightforward means to present various micro-frontends in a cohesive manner [TS20]. SAP has developed an open source micro-frontend framework called Luigi based on the IFrames technology [SAP].

- **Web Components** represent a standardized approach for developing framework-agnostic components. Leveraging Custom Elements, Micro-Frontends can be encapsulated within Custom Elements, enabling client-side composition of web pages. This method effectively conceals the implementation details of each Micro-Frontend through the use of web standards. However, a limitation of this approach is its compatibility, as Web Components are only supported in the latest browser versions. To address this, polyfills are available to ensure compatibility with older browser versions. This approach is well-suited for projects with extensive knowledge of frontend technologies, especially when adopting a horizontal splitting strategy [TS20, Mez21].

- **single-spa** framework is a JavaScript solution for frontend microservices, inspired by modern frameworks and their component life cycles. It emerged from the need to

use React and React Routing instead of AngularJS and UI Routers. single-spa functions as a top-level router, dynamically loading micro-frontends using SystemJS, which simplifies handling technologies like Web Components. It offers helper libraries for popular SPA frameworks, enabling the dynamic loading of these frameworks as ES-Modules. Micro-frontends are registered in the root-config, providing their names, code-loading functions, and status indicators. Each micro-frontend manages its bootstrapping, DOM mounting, and unmounting. This approach allows diverse applications to run concurrently without interference, listening for specific routing events when active and remaining absent from the DOM when inactive. single-spa accommodates the simultaneous use of Microfrontends developed with diverse technologies, such as Angular and Vue.js, ensuring seamless coexistence [ssa]. It supports the separation of Microfrontends into individual repositories with versioning, global imports, and routing configuration between Micro-Frontends. For sharing and provisioning UI code, the source code is packaged into a runtime package which is called parcels in the single-spa ecosystem and forms an advanced feature of the frameworks. These parcels can be compared with components within a framework and can vary in size, encompassing entire applications or individual features [HJ20, ssb]. single-spa can fullfill both vertical split and horizontal split use cases.

- **Module Federation and Webpack 5** Module Federation is primarily not a Micro-Frontend framework but rather a plugin of the JavaScript library Webpack from version 5 onwards. It enables the sharing of (JavaScript) code with other applications. This capability of sharing various types of code, including business logic and state management code, paves the way for loading UI code within micro-frontend frameworks [HJ20]. Still, the main advantage of Module Federation is that it can work with any file types that Webpack is able to process.

  > "If you can `require` it, it can be federated" [HJ20, p. 11]

  Webpack is an open source build tool for JavaScript applications, which is used for bundling frontend resources like JavaScript, CSS and images. During the Webpack build, a dependency graph is created that represents the relationships between the different modules in the application. This gives Webpack an understanding of which modules depend on each other, whether through imports or other links and ensures a correct execution order of the modules [Webb]. Loaders are another essential feature of Webpack, enabling the processing of various file types beyond JSON

and JavaScript. With loaders, developers can transpile code (babel-loader, ts-loader, etc.), handle styling (css-loader, sass-loader, etc.), and incorporate different frameworks (vue-loader, angular2-template-loader, etc.) [Webc]. The open-source community contributes numerous loaders, further extending Webpack's capabilities. All these loaded files are converted into valid modules and added to the dependency graph [Weba].

Zack Jackson [Zac], one of the co-creators of Module Federation, along with Jack Herrington [Jac], appear to be the primary contributors who are actively developing various use cases of Module Federation within the frontend landscape. Their experimentation includes exploring its application in server-side rendering (SSR), different JavaScript framework or various build tools. Due to the openness of this ecosystem, the use of Module Federation appears not to limit the use cases for Micro-Frontend applications. It can be employed both on the client-side and server-side, supporting both horizontal and vertical splits [Exa].

3 Context and Requirements Analysis

This chapter provides a context and environment analysis for the Design Science Research, along with defining a problem space, examining the existing system, and conducting a domain analysis. From these investigations, requirements for the final artifact, the prototype, are derived.

3.1 Problem Scenario

The context and environment of the DaFne project primarily define the scope of the Design Science research. This interdisciplinary project involves multiple teams with diverse areas of expertise from various organizations, all united by a common overarching objective: to develop methods for generating synthetic data, particularly for training AI models. The ultimate deliverable of this research project is intended to be a prototype for a cohesive platform.

The areas of expertise within the organizations can be divided into the following groups:

- **Group A / Institution A** comprises software developers and architects responsible for integrating methods developed by data science and smart city experts into a cohesive platform as AI software services. This group includes task forces with backend and frontend developers, who are responsible for designing and developing the main functionality as well as the supporting services for platform usage, like authentication, user management, monitoring or job management. The backend is constructed on a Kubernetes cluster, while the frontend operates with greater technological freedom, constrained only by the need to prototype a micro-frontend application architecture. Both frontend and backend focus on creating an architecture that facilitates the contribution feature, enabling an open platform for the development and access of synthetic data methods.

- **Group B / Institution A** includes a data science team responsible for developing generic synthesis methods in the form of ML as a service, such as Reproduction. They have flexibility in choosing their technological stack, provided they can offer valid interfaces for backend integration.
- **Group C / Institution B** consists of a data science team tasked with developing evaluation methods for synthetic data. Close alignment with groups B and D is necessary.
- **Group D / Institution C** comprises smart city experts responsible for developing smart city use-case-specific synthesis methods in the form of inference as a service, such as Pedestrian Path Simulation. Similar to group B, they have flexibility in technological choices, as long as valid interfaces are provided.
- **Group E / Institution D** consists of a data science team that develops the Neighborhood Generation feature as an inference-as-a-service, with the flexibility to choose the preferred technology stack.

As can be derived from the organizational structure, the prototype requires a flexible way to integrate both backend and frontend services. For the validation of the goal, here with a focus on frontend integration, the integration of the Neighborhood Generation feature is focused on. Some assumptions and constraints are set for this:

1. The developers assigned to the Neighborhood Generation feature have basic frontend development skills and therefore limited knowledge of advanced frontend architectures.
2. They are given the freedom to choose any frontend framework that suits their preference and the needs of the feature.
3. Minimal, feature-specific guidelines are in place for the developer to ensure straightforward integration.
4. The developers do not need insight into the main code of the platform, but can obtain it as it is an open source project.
5. A backend service providing the functionality for Neighborhood Generation is already available, with the developers handling the full-stack application.

6. The developers have the option to deploy their application independently but prefer to integrate it into the existing platform to benefit from its community value and supporting services.

7. The developers face time constraints that limit their ability to collaborate with Group A on integrating the application.

3.2 Existing System: Module Federation and React

Based on the constraint that the front-end application of the DaFne platform must implement a micro-frontend architecture, a first experiment with the structure of such an architecture has already been carried out. In prior work, we collaboratively designed the user flow for the generic reproduction algorithm, incorporating valuable user insights. These designs have since been implemented using React 17 and TypeScript as well as Module Federation as an approach for composing micro-frontends. The selection of the frontend stack was primarily influenced by the expertise of the developer. Additionally, Module Federation was selected, aligning with the initial literature research [Mez21], which emphasized the advantages of this approach. The UI designs have been transformed into three distinct vertically splitted micro-frontends, which are all composed on the client side with a central shell element loading the applications based on the requested routes (see Fig. 3.1). The applications serve following distinct purposes:

1. **marketing:** Dedicated to informing users about the platform's features and encouraging their engagement.

2. **auth:** Designed for user registration and authentication, ensuring secure access to the platform.

3. **dafne:** Provides all platform functionalities for generating synthetic data.

4. **container:** Is the central shell element that loads the micro-frontends. The container application is suitable for providing a base layout and navigation for the platform, but in this case, it does not contain any UI elements.

These React applications are set up in a mono repository using the Webpack build tool, i.e. three different applications are housed in a single GitHub repository. A

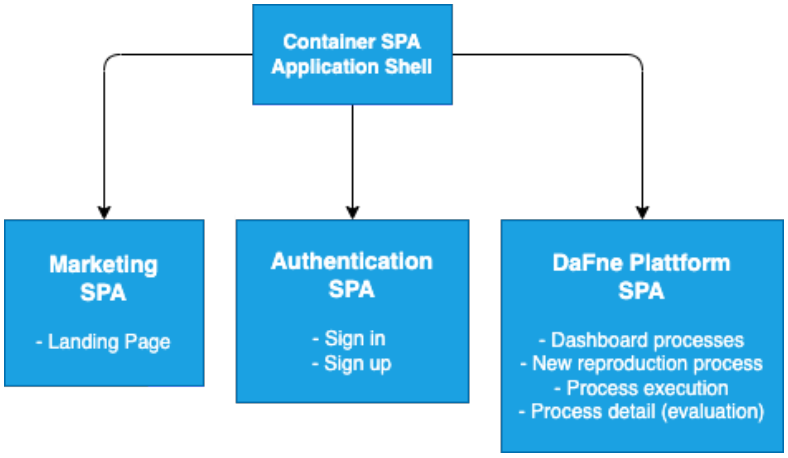

Figure 3.1: Vertical Split into three applications

monorepo centralizes the codebase for multiple projects and allows for simpler development processes and dependency management, especially in the context of this experiment where one developer is working on all three applications. Conversely, a poly-repository (polyrepo) approach, which allocates each application to its own repository, appears excessive for the scale of this project. The monorepo strategy thus offers a more pragmatic and cohesive development environment for smaller-scale projects [Mez21].

React, renowned for its efficient development of user interfaces for SPAs, adopts a declarative, component-based approach. Each component in React encapsulates a part of the UI, encompassing its presentation, logic, and state. A significant feature of React is the Virtual DOM, which acts as a lightweight abstraction layer, optimizing UI updates. When a component's state changes, React only updates the necessary parts of the actual DOM, minimizing performance costs [Reab]. React Router complements this by providing client-side routing, allowing for effective management of different views within the SPA [reaa].

The setup of each individual application follows the same principle as the webpack configuration of the *marketing* app, therefore the structure is explained in more detail below (see listing 1):

- **Module Rules**: Within the module object, the rules array specifies how to process files. It instructs the system to process files with extensions .m.js or .js (JavaScript files) using Babel. To integrate Babel into the Webpack build process, the babel-loader is used. Babel serves as a JavaScript compiler, transforming

```
const devConfig = {
 module: {
     rules: [
         {
             test: /\.m?js$/,
             exclude: /node_modules/,
             use: {
                 loader: 'babel-loader',
                 options: {
                     presets: ['@babel/preset-react', '@babel/preset-env'],
                     plugins: ['@babel/plugin-transform-runtime']
                 }
             }
         }
     ]
 }
 mode: 'development',
 entry: './src/index.js',
 devServer: {
     port: 8081,
 },
 plugins: [
     new ModuleFederationPlugin({
         name: 'marketing',
         filename: 'remoteEntry.js',
         exposes: {
             './MarketingApp': './src/bootstrap'
         },
     }),
     new HtmlWebpackPlugin({
         template: './public/index.html'
     })
 ]
}
```

Listing 1: Webpack configuration of the marketing application

ECMAScript 2015+ code into a version compatible with older browsers. The preset @babel/preset-react allows Babel to compile JSX into JavaScripts and optimizes React-specific features. @babel/preset-env ensures the JavaScript code is compatible with different browser environments. [bab]

- **Mode:** The mode is set to `development`, which means Webpack will optimize the build for development, enabling features like enhanced debugging. In this stage of the experiment, no production build was made yet.

- **Entry Point:** The entry field specifies the initial file to be processed by Webpack, which is `./src/index.js` in this case.
- **Development Server:** The devServer configuration sets up a development server for the application. It's configured to run on port 8081.
- **ModuleFederationPlugin:** This is crucial for setting up Module Federation. It's configured with:
  - **name:** Identifies the remote as `marketing`.
  - **filename:** Specifies `remoteEntry.js` as the file to be generated by Webpack. This file will handle the exposure of modules from this application to others.
  - **exposes:** Defines what module(s) this application exposes. Here, `./MarketingApp` is exposed, which points to `./src/bootstrap`. The bootstrap file (see Listing 2) serves as an isolated entry point for the micro-frontend, encapsulating the mounting logic for the application. It provides a method to render the app both in development and when integrated into a container, allowing for a seamless transition between these environments. By pointing to ./src/bootstrap, the Module Federation setup exposes the module in a way that's decoupled from the app's internal logic, making it easier for other micro-frontends or a container to dynamically import and initialize the MarketingApp.
- **HtmlWebpackPlugin:** This plugin generates an index.html file for the application, using the template provided at `./public/index.html`. This is useful for injecting scripts or linking to external resources.

The core concept of Module Federation lies in dividing modules between the host and remotes. The host module serves as the entry point to the application, while the remotes are independently developed micro-frontends (see listing 3). Module Federation complements React by enabling the host for dynamic loading of these remotes as modules by specifying their location in the Webpack configuration. As illustrated in listing 4, the React shell application employs lazy loading to dynamically load the mounting point of each micro-frontend at runtime, depending on the requested route. This approach ensures that the initial rendering of the container application does not immediately load

```
const mount = (el) => {
    ReactDOM.render(
        <App></App>,
        el
    )
}
// for isolation mode
if (process.env.NODE_ENV == 'development') {
    const devRoot = document.querySelector('#_marketing-dev-root');

    if (devRoot) {
        mount(devRoot);
    }
}
// else: export the mount function if running through container
export { mount };
```

Listing 2: Initialization and mounting logic of marketing app in isolated and composed environments

```
  plugins: [
       new HtmlWebpackPlugin({
           template: './public/index.html'
       }),
       new ModuleFederationPlugin({
           name: 'container',
           remotes: {
               marketing: 'marketing@http://localhost:8081/remoteEntry.js',
               auth: 'auth@http://localhost:8082/remoteEntry.js',
               dafne: 'dafne@http://localhost:8083/remoteEntry.js'
           }
       })
   ]
```

Listing 3: Module Federation Plugin configuration of the application shell

the entire codebase, thereby preserving the app's performance. Besides, it allows different teams to independently develop and deploy distinct features or sections of the SPA, which are dynamically assembled into a cohesive application.

In this scenario, a hierarchical relationship exists between the host and the remote, where code sharing follows a unidirectional data flow since the host doesn't expose any modules. This unidirectional approach aids in ensuring that micro-frontends are self-contained and isolated. Each micro-frontend independently manages its own logic, data, and presentation, without depending on direct inputs or outputs from others. Although

there is no communication implemented at this stage, unidirectionality could potentially be compromised when the applications need to interact, such as for sharing authentication states.

However, an application is not limited to being only a host or a remote. Each application can specify a list of remotes to which it has access. Additionally, it can release its own modules to the outside world by adding them to the `exposes` property, allowing other applications to consume them.

```
const MarketingLazy = lazy(() => import('marketing/MarketingApp'))
const DaFneLazy = lazy(() => import('dafne/DafneApp'))
const AuthLazy = lazy(() => import('auth/AuthApp'))
const renderMFE = (MFE) => {
    return (
        <React.Suspense fallback="Loading...">
            <MFE />
        </React.Suspense>
        )
}
return (
          <Routes>
              <Route path="/" element={<Layout />}>
                  <Route index element={<Navigate to={"/marketing"} />} />
                  <Route path="/marketing/*" element={renderMFE(MarketingLazy)} />
                  <Route path="/auth/*" element={renderMFE(AuthLazy)}/>
                  <Route path="/dafne" element={renderMFE(DaFneLazy)} />
              </Route>
          </Routes>
  )
```

Listing 4: Lazy Loading of micro-frontends (MFE) in the application shell

3.3 Domain Analysis

The previously described system, initially partitioned into micro-frontends based on broad intuition for experimental purposes, requires an analysis to target the benefits of a micro-frontend architecture. To determine the appropriateness of this division into micro-frontends, domain-driven design (DDD) is employed as a methodology. This approach focuses on defining the modularization and granularity of the application. It's crucial to develop a strategy for segregating micro-frontends that strikes a balance between being too loose and too detailed [Mez21].

The initial design of the system already includes a representation of the core domain of the platform by translating the business logic - obtained through the preceding user and process analysis - into a user interface design. While the system first was designed around the users' needs, this design also provides the software architect with relevant vocabulary as well as an information architecture that helps to obtain clear boundaries for the code structure. The content and navigation of the *dafne* app, as shown in figure 3.2, partially show this information structure.

Although using DDD for the front-end landscape is an unusual use case for the method, applying the main concepts provides support for identifying logical boundaries of a system. By considering the organizational structure described in section 3.1 and the navigation structure (Fig. 3.2), it is possible to identify bounded contexts and subdomains, which can be regarded as groups of related business processes [Gee23]. In the following, the platform is broken down and described according to the individual domains.

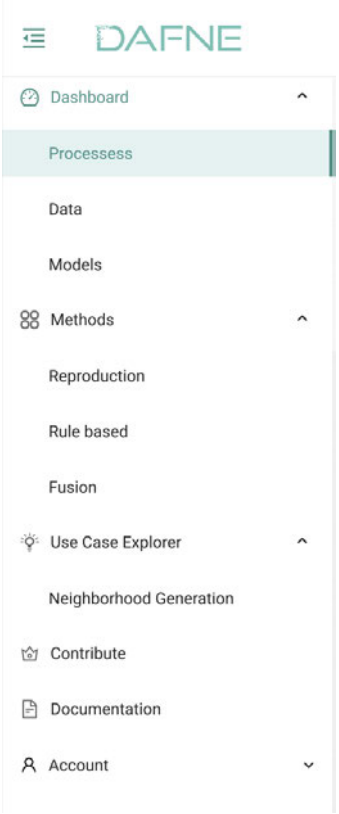

Figure 3.2: Navigation Content of the Core Domain

Table 3.1: Domains of the DaFne Platform according to Domain Driven Design (DDD)

ID	Name	Type
CD	Synthetic data generation platform	Core domain
GSD-1	Value proposition	Generic subdomain
GSD-2	Authentication and user management	Generic subdomain
CSD-1	Generic data synthesis methods	Core subdomain
CSD-2	Use case explorer	Core subdomain
SSD-1	Job, data and model management	Supporting subdomain
SSD-2	User support	Supporting subdomain

Core domain: Synthetic data generation platform The main value of the platform is to enable the user to generate synthetic data without the need for prior knowledge of data science. Although the focus is that the synthetic data can be used in machine learning use cases in the field of smart cities, the generic methods are also intended to support domain-independent use cases. Project management, product development and design as well as integration are the responsibility of Group A and B.

Generic subdomain: Value proposition In this domain, the main objective is to effectively visualize all essential information related to the usage and features of the platform, persuading the user to engage with the core subdomains and calling to action for authentication. This domain is in the responsibility of the UX/UI designer and the frontend developer of Group A.

Generic Subdomain: User authentication and management Authentication and user administration are combined in this subdomain, as a single team member from group A deals specifically with the logic of user and access administration. Authentication is the necessary condition for accessing the core subdomains.

Core subdomain: Generic data synthesis methods This core subdomain of the platform, focusing on generic data synthesis methods, involves collaborative efforts from Group A and B at the same institution. Group B is dedicated to developing generic

generative algorithms, such as tabular data reproduction (MLaaS), and exploring data fusion and rule-based generation features, as illustrated in figure 2.1. They also integrate evaluation methods from Group C and support group A with data science domain knowledge. Group A is in charge of the platform's design and software development, ensuring the seamless integration of features into an AI Software Services Platform. This encompasses architecting and developing a backend infrastructure for various ML models and frontend development tasks. Consequently, the development of core functionalities, integration, and deployment processes are closely interconnected in this domain, enabling effective communication among team members across various research areas.

Core subdomain: Use case explorer This subdomain forms another main value of the platform, along with the generic approaches to data synthesis. It focuses on cataloging, detailing, and executing data use cases characterized as inference as a service. Presently, these use cases are being developed by project members from various institutions (Group D and E) who work independently and communicate irregularly. Additionally, the subdomain is designed to accommodate new use cases from external community contributors, thereby extending and enhancing the use case explorer. In this domain, each use case is treated as a distinct function and possesses its own unique business logic. Within this domain, each added use case represents an isolated feature and has its own business logic.

- **Core subsubdomain: Neighborhood Generation** The Neighborhood Generation, an internal project developed by a partner in Group E, is an 'inference as a service' tool. It employs a Graph Neural Network (GNN) to analyze urban patterns, learning the relationships between various types of Points of Interest (POIs) within a city. Utilizing this model, the GNN can predict the function of a building based on its neighboring structures. This functionality is particularly beneficial for supplementing incomplete urban land-use data [Yal]. From the user's perspective, the process is straightforward and interactive. The user selects a specific polygon area or a city on the map and then clicks the "Generate" button. This action triggers the pre-trained GNN model, which then processes the selected area and returns an optimized urban plan for that region. This user scenario emphasizes ease of use and the practical application of complex neural network analysis for urban planning.

Supporting subdomain: Job, data and model management This subdomain is for managing and monitoring running or past jobs, data sets or models. The user can view the details of the triggered jobs and datasets here. Both core domains, generic synthesis methods and use-case specific methods, are supported by this domain.

Supporting subdomain: User support This subdomain includes services that help users use the platform and provide additional information about the platform's features when needed. Given the variety of user types, who have different skill sets and user roles, this area is designed to serve them all. This includes users who contribute use cases and need guidance on how to improve the platform.

3.4 Requirements

The identified and described domains serve as a basis for splitting contexts and gathering requirements. This process allows for the identification of problems within each domain, thereby establishing system requirements. However, this work will predominantly focus on non-functional aspects related to the overall architecture, particularly considering the quality attribute *extensibility*. While internal functional requirements are recognized, they are secondary to the architectural considerations. The functional design of each application, which is inside the area of responsibility of respective development teams, falls outside the scope of this analysis.

The foundation for all requirements is determined by the constraint that the platform should implement a micro-frontend architecture, with the assumption that a UI monolith can pose challenges when extending software with new features, especially in situations where features are developed by independent teams. This approach is aimed at ensuring modularization, maintainability, and independent feature development. Consequently, the requirements arising from the core domain are related to general concepts of micro-frontend architecture, with the simultaneous goal of integrating AI software services into the platform.

1. Constraint: Implementation of Microfrontend Architecture for extensibility
    a) Extending the platform with new features should be possible without affecting the existing features.

   b) Each micro-frontend should encapsulate either a single domain or a bounded context of related domains using an ubiquitous language.

   c) The micro-frontends should be developed without prior knowledge of how they will be deployed.

   d) The architecture should allow the integration of applications developed with different frontend frameworks, requiring the architecture to be technology-agnostic.

   e) A failure in one application should not affect the availability of another one.

2. Consistency in User Experience

   a) Within the different domains, the user must be offered a coherent user interface and user experience.

   b) Develop and adhere to a shared style guide, theme or UI component library.

3. Performance

   a) Packages used more than once have to be shared in the context of the whole application.

4. Communication and State Management

   a) Microfrontends should not rely on global variables and shared states.

   b) Micro-frontends should never be directly linked for data transfer.

   c) Global state storage should utilize native browser functionalities like local storage or cookies

5. Security and Authentication

   a) The access to the core subdomains CSD-1 and CSD-2 should be restricted to authenticated users.

   b) The authentication domain should inform concerned subdomains upon successful authentication.

6. SEO: Enhancing the platform's visibility and accessibility through search engines.

a) Ensure that the value proposition domain is optimized for search engine crawling and indexing.

7. Deployment and Maintenance

   a) Implement automated CI/CD pipelines to ensure seamless deployment and updates for each micro-frontend.

   b) The different versions of the micro-frontends should be observable and accessible through a central management interface.

   c) The integration and interaction of the applications must be testable.

4 System Design

Based on the problem scenario, the existing initial system, and the domain analysis, this chapter introduces a system design intended to address the identified requirements. It begins by determining the granularity of modularization, identifying the micro-frontends to be separated. Subsequently, the technology choices are updated, and the communication between the micro-frontends is described. Finally, the infrastructure required for the deployment and maintenance of the application is outlined.

4.1 Identification of Micro-Frontends

Following the principle that each identified domain can represent a micro-frontend, the division could be based on the table referenced in 3.1. If the degree of modularization aligns with the overall scope of the project, which is a Minimum Viable Product (MVP), and if this modularization is intended to add value rather than create additional effort, then some domains can be combined if they can be put into one bounded context if share a common vocabulary or ubiquitous language and data logic.

Furthermore, considering that the development of the Reproduction feature, its software integration, and supporting services like monitoring on a dashboard are being conducted by groups at the same institution, it is impractical to split the frontend services. This decision is reinforced by the aim to maintain low coupling between features while ensuring high cohesion within the *dafne* app's functionalities. As a result, in the new system design, the *dafne* will continue to be treated as a micro-frontend. It will implement all generic data synthesis methods and supporting services, aligning with the core competencies of Institution A. Additionally, it will serve as a host for integrating additional applications that provide use-case-specific features, as shown in Fig. 4.1. This arrangement ensures a uni-directional data flow, aligning with React's design principles and maintaining a hierarchical structure in the application's architecture. It's worth noting

that only those applications that are relevant for the MVP and are highlighted in color will be included in the system design. The most significant changes from the previous system are introduced as follows:

- **Splitting:** Transition from vertical split to hybrid split as the *dafne* now also serves as a host rather than only being a remote of the container App.
- **Routing:** Implementing client-side routing in the high-level `BrowserRouter` of the container app with React Router 6, which protects restricted routes and adapts to route changes within the internal `MemoryRouter` of the remotes.
- **Communication:** Implementing an API Gateway pattern to manage backend communication and Custom Events for communication between micro-frontends in order to react to authentication states, route changes, or notifications.
- **container:** Introduction of an `AuthProvider` to restrict access to applications of core domains, namely the `DafneApp` and consequently the `NeighborhoodApp`.
- **theme:** Adding a remote Material UI theme and a remote color palette, consumable by all applications.
- **marketing:** Switching from React to Next.js for static site generation, enhancing SEO. This approach transitions from a typical SPA to an isomorphic application, merging server-side and client-side rendering. Aiming the server to initially deliver the SEO-friendly HTML, then the app to shift to dynamic client-side content.
- **dafne:** Changing to a horizontal split in order to enable the integration of use-case-specific applications. Additionally, the application code will be written in TypeScript in order to increase the reliability of the core subdomain.
- **neighborhood:** Adding a Vue.js application for the Neighborhood Generation to the React application *dafne*.

4.2 Updated Technology Stack

The system design update introduces two new frontend frameworks to leverage the benefits of the micro-frontend architecture. While detailed explanations are beyond the scope of this section, key features and benefits of each framework are highlighted.

Table 4.1: Identified Micro-Frontends inside the Domains

Domain	Micro-frontend
CD	Container / Application Shell
GD-1	Marketing App / Landing Page
GD-2	Authentication / Sign in / Sign up
CSD-1	Tabular data reproduction
CSD-1	Tabular data fusion
CSD-1	Rule based tabular data generation
CSD-2	Neighborhood Generation
SSD-1	Dashboard
SSD-2	My account
SSD-4	Documentation and help
CD	Theme

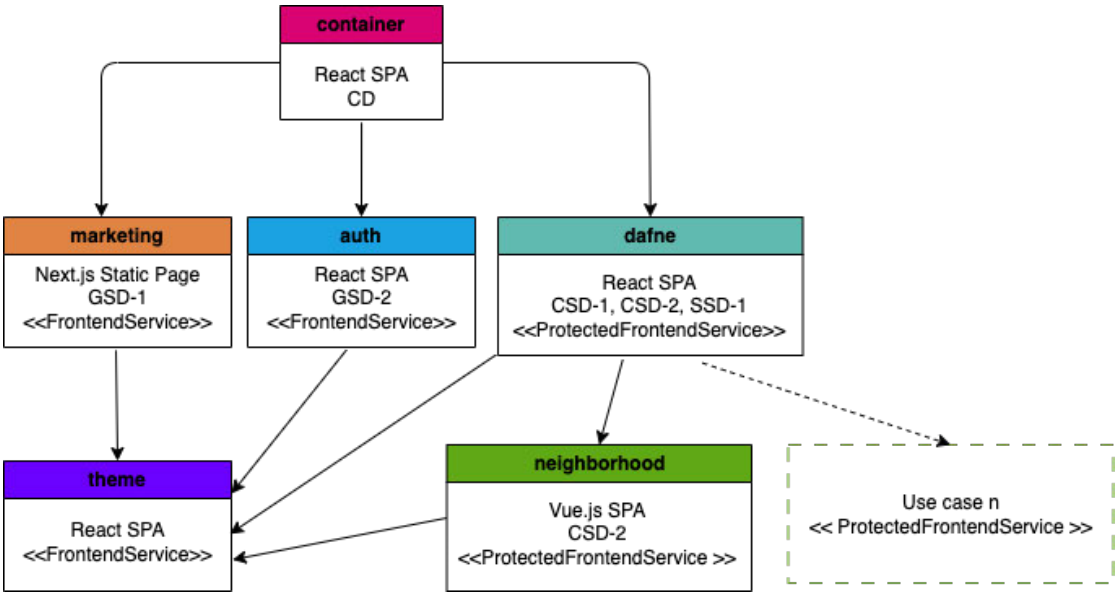

Figure 4.1: High-Level System Design as Solution Strategy

4.2.1 Vue.js

The integration of the Neighborhood Generation feature will be facilitated by incorporating the Vue.js framework into the system design. Vue.js specializes in building user interfaces and Single-Page Applications (SPAs), with a focus on the view layer. It offers reactive data binding and composable view components, simplifying the integration with other libraries or existing projects. Vue.js is renowned for its simplicity and approachability, especially for those with basic HTML and JavaScript knowledge, making it accessible even to developers without extensive frontend experience [Vue].

4.2.2 Next.js

Next.js [Nexa] is a framework that utilizes the React library but offers capabilities beyond traditional SPAs, such as server-side rendering and static site generation. These features significantly enhance the performance and Search Engine Optimization (SEO) of web applications [Nexb]. Contrary to client-side rendered React applications, which rely on client-side routing, Next.js manages routing based on the file system. It can pre-render pages on the server and deliver fully rendered HTML to the client, whereas traditional React applications typically deliver only a root div initially. This approach is particularly beneficial for applications requiring quick initial load times [TS20], such as accessing a platform's landing page in the Marketing application.

4.3 Communication between Micro-Frontends

In adherence to the core principles of micro-frontends — wherein components are loosely coupled, capable of functioning independently, and without direct links to one another, communication between micro-frontends is designed to be event-driven. This autonomy is facilitated by utilizing Custom Events, which leverage native browser functionalities to eliminate the need for external libraries, thereby reducing dependencies and ensuring compatibility across diverse JavaScript environments.

Custom Events Figure 4.2 depicts an architecture where each application is responsible for executing its state management and logic, with the liberty to utilize its own

state management technologies, such as Redux, within its boundaries. The only connection between applications occurs when they dispatch events to the browser environment, which can be consumed by interested applications executing their own event handlers. The data and information relevant to the event are shared via the event details. It is by design that only publish/subscribe relationships are established between the host (Container) and the remotes, preventing child applications from directly subscribing to each other's events. Focusing on extending AI Software Service functionalities, each remote application added to the *dafne* host, such as the Neighborhood Generation feature, communicates its internal status (e.g., job status) through events to the platform. This method of communication enables the integration of activities like the initiated AI service into the dashboard and also facilitates providing feedback to users, such as notifications, based on the status of these services.

Routing A crucial application of custom events is routing. While individual applications can operate autonomously using a browser router, a memory router is required when micro-frontends are composed within the shell application. The browser router takes advantage of the browser's History API, enabling navigation that reflects in the URL bar without server requests. Typically, there can only be one Browser Router that directly manipulates the browser's URL and history. Thus, when micro-frontends are nested within a shell application, a memory router is introduced. Unlike the browser router, the memory router retains the navigation state internally within the application's memory. Ensuring that only one application controls the browser's history stack is crucial for avoiding route collisions and unpredictable navigation behavior. The container app synchronizes the browser's URL with the user's navigation within the application by listening to custom events that contain the current micro-frontend route in the event detail, ensuring seamless navigation coordination.

API Gateway Pattern Another significant design decision is the implementation of the API Gateway pattern, which centralizes backend communication in a microservice architecture. It routes requests to the appropriate microservices and returns the response through the same channel. This strategy is particularly advantageous due to the close collaboration within Group A, where the backend team can provide an architecture that supports this approach.

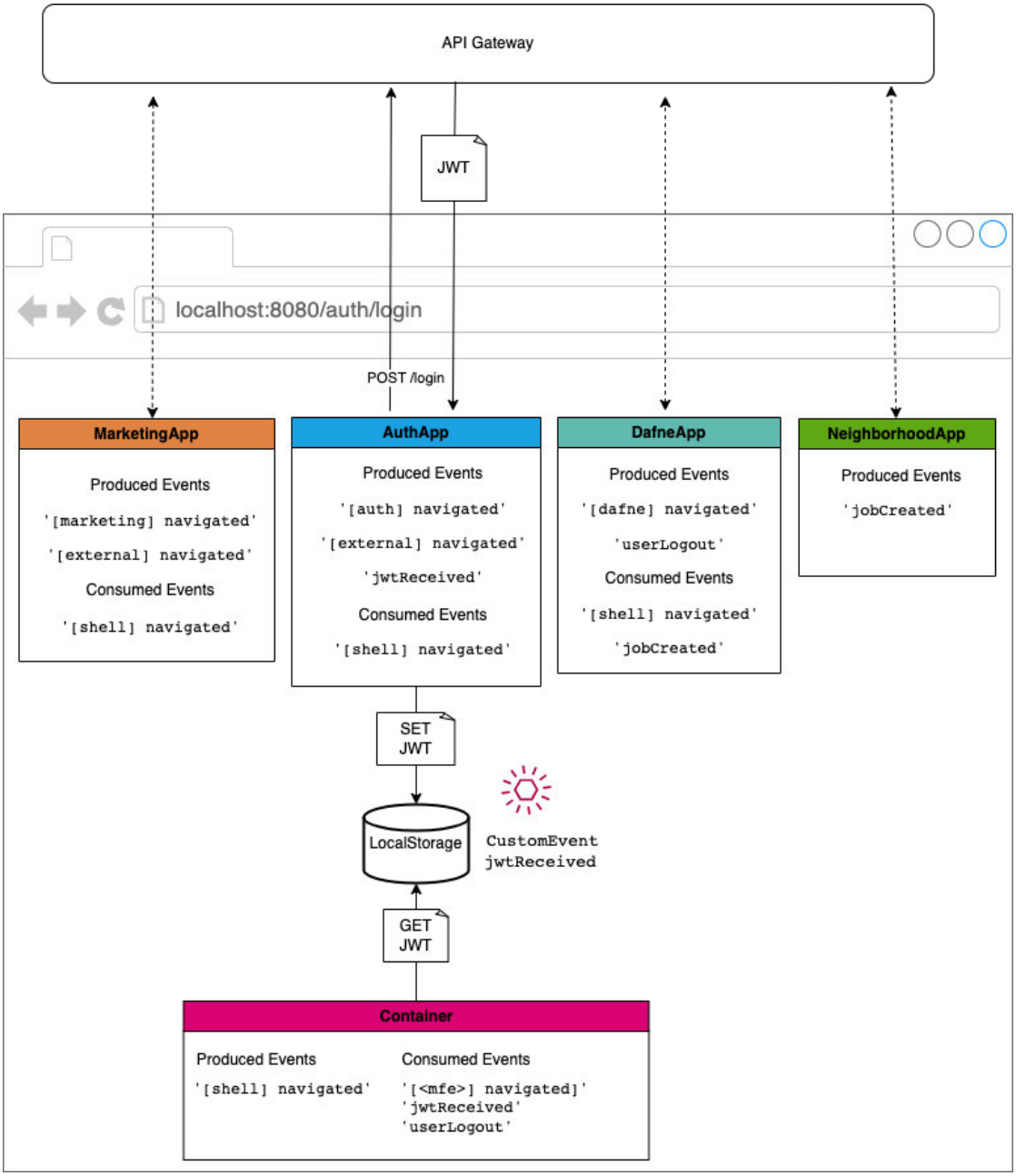

Figure 4.2: Communication Design based on API Gateway and Event-Based Architecture

4.4 Deployment and Maintenance

Transitioning from a development environment to production will highlight the advantages of the micro-frontend architecture. This chapter outlines a robust strategy for build-

ing and deploying micro-frontend applications, emphasizing the creation of production-ready builds, the selection of suitable hosting solutions, and the implementation of continuous integration and deployment (CI/CD) processes. Each micro-frontend (MFE) will be independently deployed to ensure modularity and facilitate seamless updates.

Webpack Build The first step in preparing for deployment is creating a production build using Webpack. This involves compiling the application source files along with the necessary configuration files. A critical detail is ensuring that the main.js file includes the correct references to where the remoteEntry.js files for each MFE are located. This information is essential during build time, as the location of the remoteEntry.js files dictates how MFEs will load in production. A specific production configuration will be created where the target domain for production is specified, replacing any development or staging references.

AWS S3 Buckets and Cloudfront CDN For deployment, Amazon Simple Storage Service (Amazon S3) buckets are chosen as the hosting solution. The S3 service will house the built versions of all subprojects, effectively managing and storing the build files. The advantage of S3 is its scalability, durability, and integration with AWS services like IAM for access management [S3A]. Amazon CloudFront, a content delivery network (CDN), will be used in conjunction with S3 to distribute the static files efficiently. CloudFront's edge locations will cache the content, reducing latency by delivering the content from the nearest edge location to the user, which is particularly beneficial for a distributed architecture like micro-frontends [Clo, S3C18]. In this context, it's essential to visualize the flow (see Fig. 4.3): When the container starts with index.html, it will load main.js, which contains a reference to the remote entry of the *marketing* application. Subsequently, the remote entry will be loaded, prompting the loading of main.js for *marketing*, and so on. Cloudfront will ensure that the correct files are made available in the user's browser based on the specific requests.

Version Control and Automation GitHub is used as the version and version control platform of choice for managing the code base. Within GitHub, a monorepo structure is maintained, providing a unified repository for all micro-frontend projects. This approach is particularly suitable for this experiment as it is carried out by a single developer. This setup lends itself to the straightforward creation of a continuous integration and

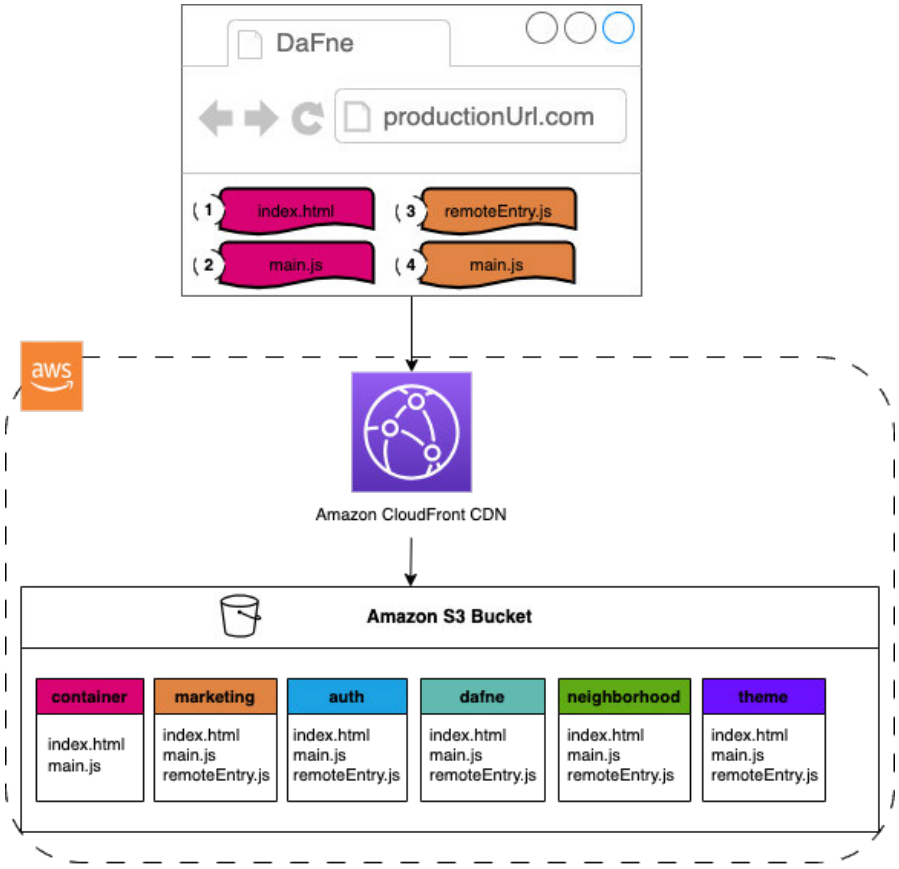

Figure 4.3: Using AWS Cloudfront CDN and S3 Buckets to serve Build Files to the Browser

continuous delivery (CI/CD) pipeline using GitHub Actions as an automation platform. GitHub Actions [git] facilitates the automation of various tasks and workflows within GitHub repositories, including building, testing and deploying code, as well as automating tasks such as code formatting, dependency management and issue tracking. This automated pipeline (see Fig. 4.4) will trigger a build for each micro front-end application when changes are pushed to the main branch of the respective project folder. After the build, each application will be uploaded independently to an S3 bucket.

The container application, being the only application that depends on other applications, is prioritized in the testing strategy. In order to develop a prototype test strategy, an initial end-to-end test is planned that focuses on the container application. With the help of Cypress [cyp], the entire application flow will be tested from start to finish, checking

whether the correct routes are being loaded after certain user interactions. Only when the test is passed and the correct integration of the micro-frontends is confirmed can the container application be deployed.

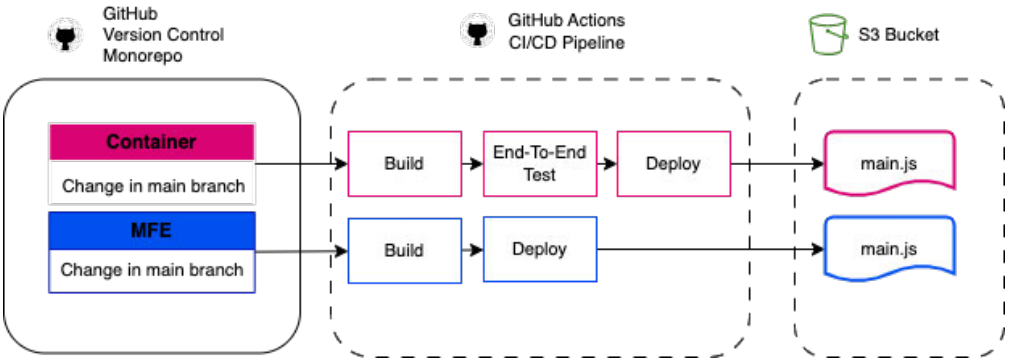

Figure 4.4: Automation Pipeline

5 Implementation

This chapter outlines the practical implementation of the designed system. It starts with configuring the remote applications and then details specific implementation aspects of certain application flows highlighting the communication strategy, using sequence diagrams for enhanced understanding. Additionally, it demonstrates an attempt to transform the app into an isomorphic one by integrating static site generation with Next.js. The incorporation of a Vue.js application extends the React-based DaFne app, showcasing the platform's extensibility. Finally, the implementation of the CI/CD pipeline and the Module Federation Dashboard Plugin is presented.

5.1 Configuration of Apps

This section details with how general app-related tasks like mounting, loading, routing, securing or performance optimizations are implemented in the designed federated micro-frontend architecture.

5.1.1 Bootstrapping and Loading Micro-Frontends

With bootstrapping, every micro-frontend is initialized, ensuring it operates effectively both in isolation and when integrated into a larger application. This setup is essential for maintaining autonomy of individual components while allowing them to function cohesively within a unified system.

Central to the bootstrapping process is the implementation of a mount function. This function is not just about rendering the Micro-Frontend onto the DOM, but also about configuring critical aspects like the routing strategy. The routing strategy is particularly important in a micro-frontend setup, as it manages navigation and URL handling in a way that prevents conflicts between multiple coexisting applications. It ensures that each

Micro-Frontend has its own navigation context, avoiding disruptions in the overall user experience.

As the other micro-frontends are set up following a similar approach, the bootstrap file of the *dafne* app should showcase how each application is initialized (see Listing 5). This includes setting up the mount function with parameters such as mountPoint for defining where the micro-frontend will be rendered, initialPathname for setting the initial route, and routingStrategy to determine how routing is managed (browser-based/memory-based). Additionally, it involves integrating state management and wrapping the app with necessary providers, like in this case the Redux Provider.

This mount function not only renders the app but also returns a cleanup function to unmount the micro-frontend when it's no longer needed. This aspect is crucial for resource management and preventing memory leaks.

The exported mount function can now be utilized by the container app. Instead of directly loading the remote as in the initial design, the host now creates components (such as DafneApp, AuthApp, or MarketingApp, see Figure 5.1) for each application to be integrated, each with very similar logic. These Components handle navigation and location changes (see next section 5.1.2), mounts the micro-frontend when needed, and ensures it's properly unmounted when the component is no longer needed. This setup is typical in Module Federation scenarios where different applications or components are dynamically loaded into a shared shell or host.

Every loading logic will work in a similar logic as depicted in Listing 6:

- An effect hook manages the mounting process. It calls the mount function with parameters like the mount point, initial pathname, and routing strategy, ensuring proper initialization and placement in the DOM. The hook is executed once, controlled by the isFirstRunRef flag, to prevent multiple mountings of the micro-frontend.
- A cleanup effect hook handles the unmounting process when the component is removed from the DOM. This function unmounts the micro-frontend, releasing resources and preventing memory leaks.
- The component renders a div element, referenced by wrapperRef, which acts as the mount point for the micro-frontend. This allows precise control over the micro-frontend's display location.

```
   const mount = ({
    mountPoint,
    initialPathname,
    routingStrategy,
}: {
    mountPoint: HTMLElement;
    initialPathname?: string;
    routingStrategy?: RoutingStrategy;
}) => {

    const router = createRouter({
        strategy: routingStrategy || 'browser',
        initialPathname: initialPathname || '/',
    })

    ReactDOM.render(
        <Provider store={store}>
            <RouterProvider router={router} />
        </Provider>
        ,
        mountPoint
    )
    return () => queueMicrotask(() => ReactDOM.unmountComponentAtNode(mountPoint));
};
if (process.env.NODE_ENV == 'development') {
    const devRoot = document.querySelector('#_dafne-dev-root') as HTMLElement;
    if (devRoot) {
        mount({ mountPoint: devRoot, routingStrategy: 'browser' });
    }
}
export { mount };
```

Listing 5: bootstrap.tsx of Dafne application

5.1.2 Routing

As seen in Listing 6 and 5, the routing strategy is determined based on the runtime environment. For implementing the routing solution presented in Section 4.3, each frontend initializes a router with React Router 6, depending on the environment. In the mode where the apps are composed through the container, the container uses a `BrowserRouter`, and the remote employs a `MemoryRouter`. When a remote application runs in isolation, as shown in the mount function in Listing 5, it is initialized with a `BrowserRouter` and does not require route synchronization.

The sequence diagram in Figure 5.2 exemplifies the routing behavior during user's journey from the Landing Page (MarketingApp) to the Sign Up Page (AuthApp). It is worth

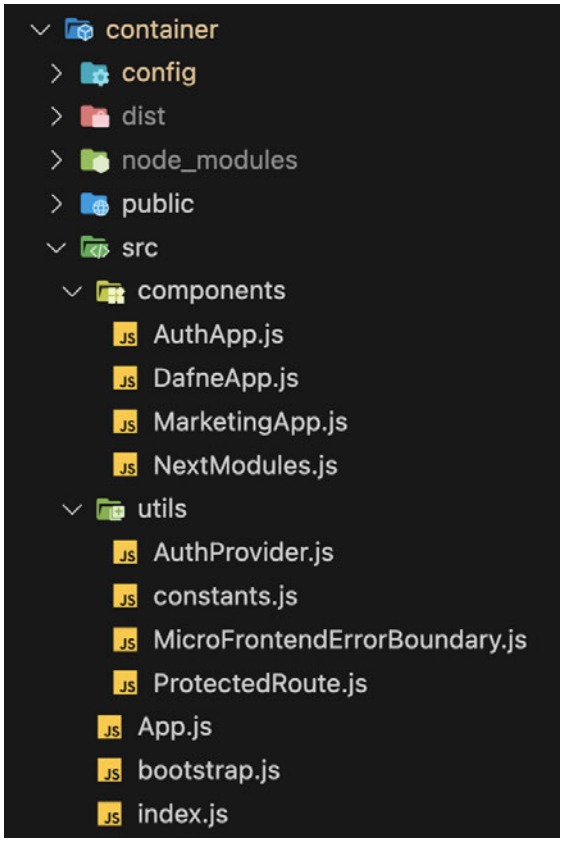

Figure 5.1: File structure of Shell Application container

highlighting that there is never direct communication between the applications; instead, they communicate solely through events, where the handling logic is executed by each app independently.

Each remote includes a `NavigationManager` (see Listing 7), which listens for navigation events from the container ('`[shell] navigated`') and navigates within the remote application using the Memory Router. Concurrently, an event ('`[auth] navigated`') is dispatched by the remote app, which is subscribed by the container so that it can update the browser path with the current location. Synchronization in the container app occurs in the custom navigation event handler set up for each initialized remote component, as seen in Listing 8. Depending on the requested path, the container ultimately mounts and unmounts the remote components via its own routes configuration.

```
const isFirstRunRef = useRef(true);
const unmountRef = useRef(() => { });
useEffect(() => {
  if (!isFirstRunRef.current) {
      return;
  }
  unmountRef.current = mount({
      mountPoint: wrapperRef.current,
      initialPathname: location.pathname.replace(dafneRoutingPrefix, ""),
      routingStrategy: "memory",
  });
  isFirstRunRef.current = false;
}, [location]);

useEffect(() => unmountRef.current, []);
return (
    <div ref={wrapperRef} id="dafne-mfe" />
)
```

Listing 6: DafneApp.js: Loading and unloading dafne MFE in container app

5.1.3 Performance Considerations

Given that each individual application can operate autonomously, they must be able to load all necessary dependencies. However, it is crucial to ensure that redundant code is not loaded during a composed execution through a host if it is already being used by another application. The `shared` property in Module Federation plays a crucial role in this context (see Listing 9). It allows for the specification of individual modules or the entire dependency list to be shared across different micro-frontends. This mechanism helps in preventing the duplication of common dependencies, reducing overall bundle size and load times. The shared configuration also facilitates version management, ensuring that different parts of the application use compatible versions of shared libraries.

When configuring shared modules, the host determines which version of the shared module to use, while the remote declares what it can provide. This distinction ensures that the correct versions are used and avoiding version conflicts. The specification of Singletons becomes particularly relevant when using libraries like React or ReactDOM, which have an internal state. Singletons ensure that only a single instance of these libraries is used at any given time in the application flow, maintaining consistency and preventing potential state conflicts.

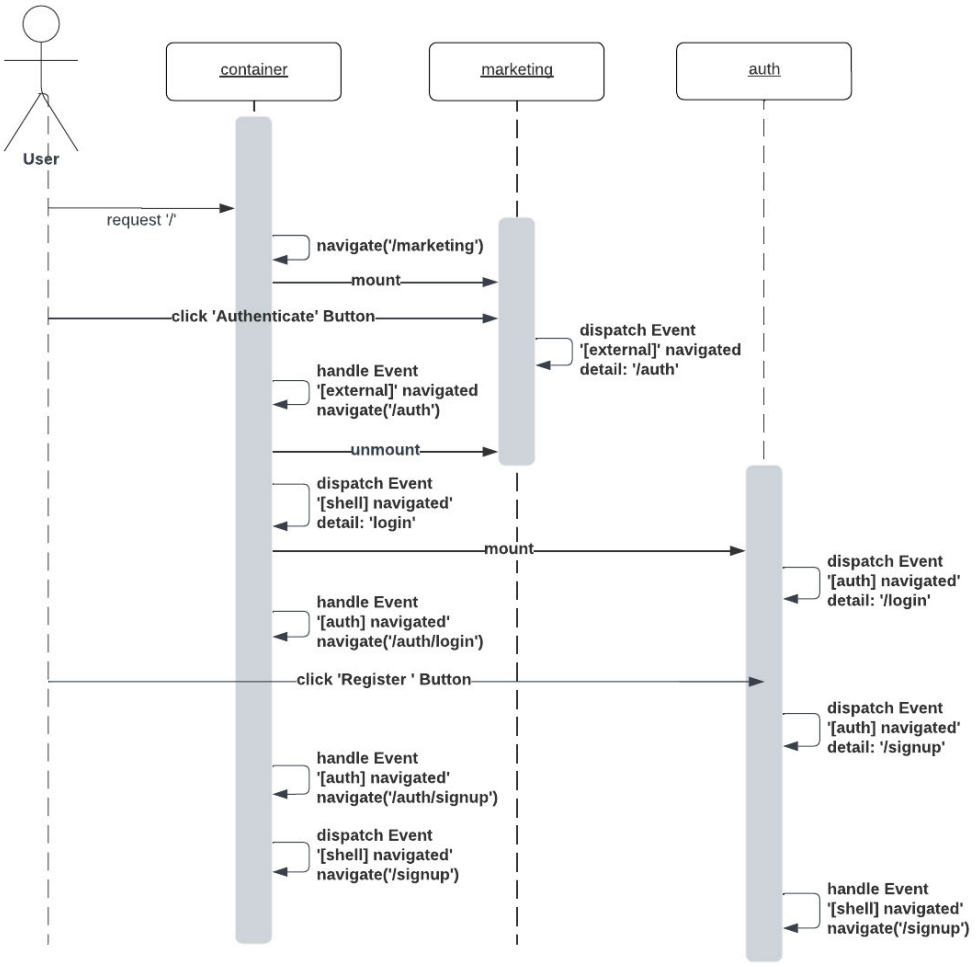

Figure 5.2: Routing behavior example between Container, Marketing and Auth

5.1.4 Fault Resilience

While applications are already loaded lazily and bootstrapped, which can help delay the display of errors in specific micro-frontends until they are loaded, additional measures are necessary to prevent the entire application from breaking in case of an error. To address this, error boundaries are wrapped around the micro-frontends. rror boundaries are crucial to ensuring that a fault in one part of the application does not lead to a complete application failure because they isolate the errors. The implementation of this strategy is referenced in Listing 10, where a `MicroFrontendErrorBoundary` component is used to wrap around the micro-frontend, providing a safeguard against potential failures.

```
    export function NavigationManager({ children }) {
    const location = useLocation();
    const navigate = useNavigate();
    useEffect(() => {
        function shellNavigationHandler(event) {
            const pathname = event.detail;
            if (location.pathname === pathname || !matchRoutes(routes, { pathname })) {
                return;
            }
            navigate(pathname);
        }
        window.addEventListener("[shell] navigated", shellNavigationHandler);
        return () => {
            window.removeEventListener("[shell] navigated", shellNavigationHandler);
        };
    }, [location]);

    useEffect(() => {
        window.dispatchEvent(
            new CustomEvent("[auth] navigated", { detail: location.pathname })
        );
    }, [location]);

    return children;
}
```

Listing 7: Internal Navigation Manager of Auth Micro-Frontend with React Router 6

```
 useEffect(() => {
  const authAppNavigationHandler = (event) => {
      const pathname = event.detail;
      const newPathname = `${authBaseName}${pathname}`
      if (newPathname === location.pathname) {
          return;
      }
      navigate(newPathname);
  }
  window.addEventListener("[auth] navigated", authAppNavigationHandler);
  return () => { window.removeEventListener(
  "[auth] navigated",
          authAppNavigationHandler
      )
  }
  }, [location]);
```

Listing 8: Container handling integrated AuthApp's Navigation

```
shared: {
      ...packageJson.dependencies,
      react: {
          singleton: true,
          requiredVersion: packageJson.dependencies.react,
      },
      "react-dom": {
          singleton: true,
          requiredVersion: packageJson.dependencies["react-dom"],
      },
}
```

Listing 9: Container app Webpack Configuration: Sharing packages with other modules

```
const renderMFE = (MFE) => {
      return (
          <MicroFrontendErrorBoundary>
              <React.Suspense fallback="Loading...">
                  <MFE />
              </React.Suspense>
          </MicroFrontendErrorBoundary>
      )
 }
```

Listing 10: Container app's Micro-Frontend loading with Error Boundary

5.2 UX/UI Consistency

Among methods to ensure a cohesive design across micro-frontends, this work has focused on employing the Material-UI design system. Material-UI is a renowned React-based framework that provides an extensive library of pre-defined design elements and styles [MUI]. To facilitate this, a new remote application theme has been added, which includes a color palette integrating the DaFne project's color concept with primary or secondary colors, as well as other design and layout-specific elements like typography (refer to Listing 11). Developers of individual Micro-Frontends are given the flexibility to establish their own design system, but when the application is executed through the container, it has to access the remote theme object. While the remote theme module serves as a React application for testing the theme in a web application, it only exposes the finalized theme object and a palette object that contains the color values (Listing 12). This palette object is especially critical for micro-frontends that do not implement React, given that

MUI is a design system tailored specifically for React. Thus, this theme or palette are added to the remote list of all micro-frontend applications except the container.

```
  import { createTheme } from '@mui/material';
  import palette from './shared-palette';
   const customTheme = createTheme({
    typography: {
        fontSize: 13,
        .... },
    components: {
        MuiCssBaseline: { ... },
        MuiTypography: {... },
        MuiButton: { ... },
        MuiAppBar: { ... },
    },
    palette: palette,
    layout: {
        drawerWidth: 240,
        appBarHeight: 20,
    },
});
export default customTheme;
```

Listing 11: Shared MUI Theme

```
    new ModuleFederationPlugin({
            name: 'theme',
            filename: 'remoteEntry.js',
            exposes: {
                './theme': './src/shared-theme',
                './palette': './src/shared-palette',
            },
            ...
    })
```

Listing 12: Theme Config

5.3 Communication and State management

The communication solution depicted in Figure 4.2 shows the static structure of the event-based communication landscape, including the API gateway. While the implementation of routing-based events was described in section 5.1.2, this section takes a closer look at how state management and API requests work together to enable communication

between micro-frontends. This is further explored through a user flow that requires authentication, as shown in the sequence diagram (Figure 5.3). The flow begins with a user attempting to access the protected 'dafne/dashboard' route but gets redirected to the login page due to lack of authentication. Successful login then leads the user to the dashboard, followed by an immediate logout. It should be noted that, compared to the diagram 5.2, this sequence does not take into account the logic for routing, mounting and unmouting in order to focus on state management and API communication.

5.3.1 API Gateway and State Management

As shown in Figure 5.3, the authentication communication process involves multiple micro-frontends, none of which directly communicate with each other or share a common, global state management system. Each application independently chooses its state management and data fetching methods. All remote applications can issue requests to the API Gateway with their individual technology of choice, given that they are capable of making HTTP requests. The table 5.1 below outlines the specific technologies used by each application. Unlike other applications, the container app does not perform API requests, focusing instead on administrative tasks like routing, security, and loading of the micro-frontends. In addition, the remote application theme does not need to set up state management or fetch data, as it only provides static modules. The browser's local storage serves as the sole global data and state storage solution, primarily used for storing and accessing the JSON Web Token (JWT) for authentication purposes.

Table 5.1: State Management and Data Fetching Technologies of each Micro-Frontend

Micro-Frontend	**State Management**	**Data Fetching**
container	React Context	-
marketing	React useState	Fetch API
auth	React useState	Fetch API
dafne	Redux	RTK Query
neighborhood	reactive (Vue)	Fetch API
theme	-	-

5.3.2 Authentication and Security

When a user attempts to access a restricted route, such as any subroute within the Dafne application, the container app employs the `ProtectedRoute` component as a

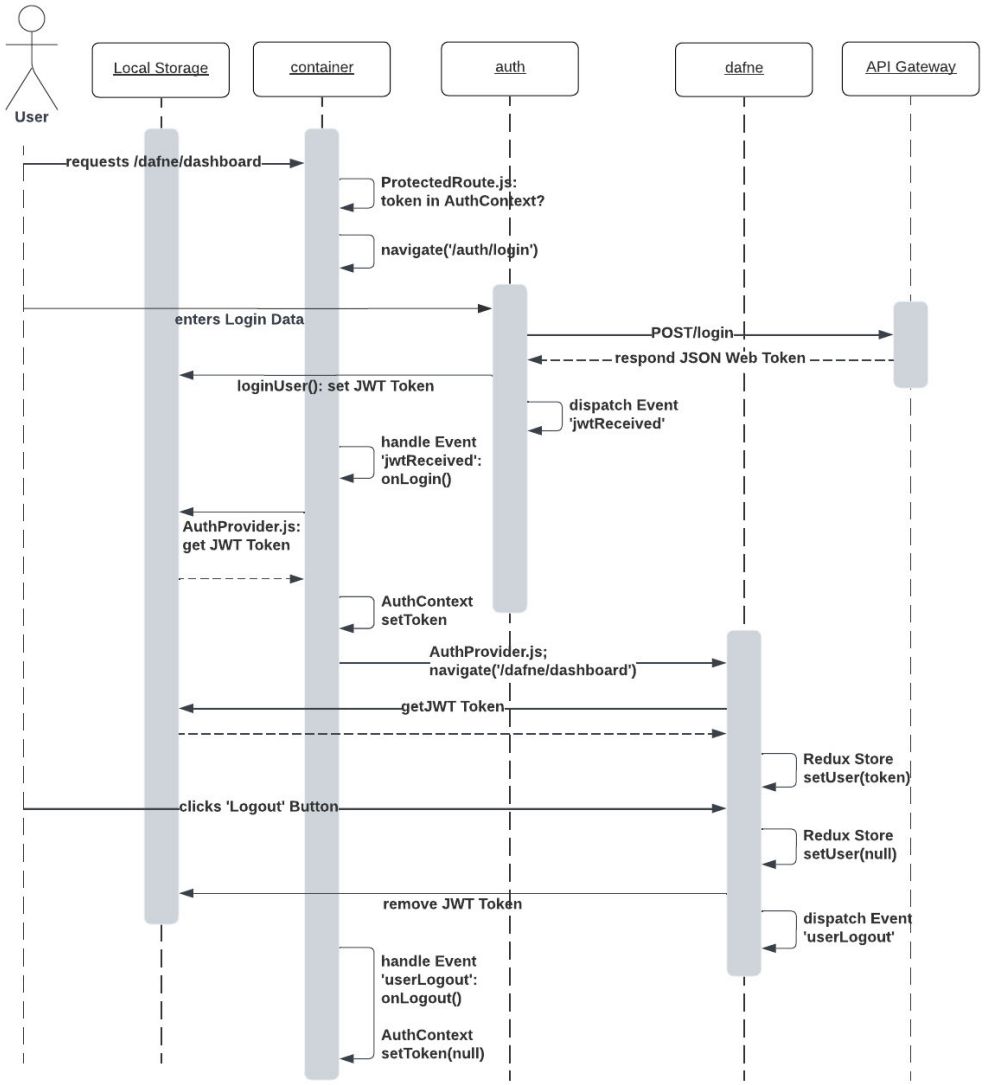

Figure 5.3: Implemented Security and Authentication Communication Flow

route guard. This component, which encapsulates the rendering function of the Dafne app, verifies the presence of a token in the `AuthContext`, the internal state manager for authentication in the container app. If no token is found, the user is redirected to the Auth app's login page to enter their credentials. Notably, the login request is executed by the Auth app, not by the container. Upon successful login, the Auth app stores the JWT token in the browser's local storage and simultaneously dispatches an event. This event prompts the container's `AuthProvider` to retrieve the token from local storage, updating the AuthContext with the user's authentication status and directing the user to their requested route. Given that there is no direct connection between the micro-frontends and each app manages its own state, the Dafne app independently retrieves the token from local storage. It parses the token information and updates the user object in its Redux store. With a validated user in its global state, the Dafne app can then perform requests to the API gateway for additional data using RTK Query.

5.4 SEO with Next.js for Marketing App

The plan to leverage the micro-frontend architecture for the diverse needs of a comprehensive system initiates the experiment of combining SPAs with static sites. Through Static Site Generation and Server-Side Rendering, instead of only a root div for the Shadow DOM of SPAs, the entire HTML content of the page is pre-rendered. This approach aims to integrate a Next.js application for the marketing content of the Landing Page into the container application. The Module Federation Community continuously expands the capabilities of Module Federation, and with the `NextFederationPlugin`, even server-side rendered micro-frontend applications are enabled [Nex23]. As shown in Listing 13, static chunks are now created for the remote entry. Since the Marketing App content has not yet been designed and fully implemented, simple HTML tags were added to the document for testing. When the Next.js application is run in isolation mode, the HTML content of the page is displayed in the source code (see Appendix 20). However, when the Next.js application is integrated into the container application, the HTML content of the page is displayed in the browser, it is not visible in the source code (see Listing 14), making this experiment obsolete in this setup.

```
const NextFederationPlugin = require('@module-federation/nextjs-mf');
module.exports = {
  webpack(config, options) {
    const { isServer } = options;
    config.plugins.push(
      new NextFederationPlugin({
        name: 'landing',
        filename: 'static/chunks/remoteEntry.js',
        exposes: {
          './NextApp': './pages/index.js',
        },
        remotes: {
          theme: 'theme@http://localhost:8085/remoteEntry.js',
        },
        shared: {
          '@mui/material': {
            singleton: true,
          },
        },
        extraOptions: {
          exposesPages: true
        },

      })
    );
    return config;
  },
};
```

Listing 13: NextFederationPlugin: Module Federation for Next.js

```
<!DOCTYPE html>
<html>
  <head> <script defer src="/main.js"></script></head>
  <noscript id="__next_css__DO_NOT_USE__"></noscript>
  <body style="margin: 0">
    <div id="root"></div>
  </body>
</html>
```

Listing 14: Next.js source code after integration into SPA container

5.5 Integration of Vue.js Application Neighborhood Generation

The focus now shifts to implementing the extensibility of the AI as a Software Service platform. To achieve this, a Vue application responsible for Neighborhood Generation is being integrated as a remote into the Dafne app. The developer is instructed to build using Webpack, utilize Module Federation, and adhere to the platform's design system, with access to examples in the Dafne repository for guidance. The loading of this Vue app is managed within the internal routes of the Dafne app, similar to how the container app incorporates React applications as remotes (refer to Listing 15). The primary distinction in this integration is the necessity to include the VueLoaderPlugin and Vue-specific style loaders in the Webpack configuration to process the template-like structure of .vue files. Another difference to the React applications is that the shared Material UI theme is designed for React applications and therefore cannot be imported directly into the Vue application. To maintain design consistency, the Vue application imports the ES module of the color palette from the remote theme application (see listing 17) Sass operates as a CSS preprocessor during the compilation phase, before the CSS is served to the browser [Sas]. In contrast, JavaScript functions within the browser, manipulating the Document Object Model (DOM) and interacting with the rendered HTML and CSS. Consequently, the Sass theme of PrimeVue [Pri] cannot be directly altered using JavaScript, necessitating a workaround. This workaround involves appending CSS variables to the DOM natively to align the Vue application's styling with the overall platform design.

Regarding the communication strategy of the neighborhood app, the same principle applied to other applications was followed. While the remote app can function autonomously, the overarching *dafne* app is interested in the app's status. Consequently, events are dispatched when the neighborhood app makes an API request. The response from the request is included in the event details, enabling the *dafne* app to decide how to handle this response. Typically, the response is added to the Redux notifications state within the Dafne app, ensuring that the user is informed about the status or outcome of the request (see Listing 16). This approach facilitates seamless interaction between the remote neighborhood app and the *dafne* host application.

```
{
            path: 'use-case/neighborhood',
            element:
                <MicroFrontendErrorBoundary>
                    <React.Suspense fallback={<CircularProgress />}>
                        <NeighborhoodLazy />
                    </React.Suspense>
                </MicroFrontendErrorBoundary>
 }
```

Listing 15: Routes configuration of the dafne app

```
useEffect(() => {
  const handleJobCreated = (event: Event) => {
      const customEvent = event as CustomEvent;
      if (customEvent.detail) {
          displayNotification(
              {
                  type: customEvent.detail.type,
                  header: customEvent.detail.header,
                  message: customEvent.detail.message,
                  timeout: 10000,
              }
          )
      }
  };
  window.addEventListener('jobCreated', handleJobCreated);
  return () => {
      window.removeEventListener('jobCreated', handleJobCreated);
  }
}, []);
```

Listing 16: DaFne app handling events of neighborhood app

5.6 Deployment and Maintenance

Following the implementation of the applications, the next step is to transition them from the development environment to production. This involves executing the deployment strategy outlined in Section 4.4.

```
  async function fetchPalette() {
    const palette = await import('theme/palette');
    return palette.default;
  }

  async function createPaletteVariables() {
  const paletteData = await fetchPalette();
  for (const category in paletteData) {
    if (paletteData.hasOwnProperty(category)) {
      const categoryData = paletteData[category];
      for (const color in categoryData) {
        if (categoryData.hasOwnProperty(color)) {
          const variableName = `--${category}-${color}`;
          const colorValue = categoryData[color];
          console.log('variableName', variableName);
          document.documentElement.style.setProperty(variableName, colorValue);
        }
      }
    }
  }
}
```

Listing 17: Palette Loading Vue App

5.6.1 CI/CD

For this purpose, each micro-frontend application will have a dedicated YAML file set up for the GitHub Actions workflow, enabling independent deployment. All the files follow a logic similar to that demonstrated for the container app in Listing 18. However, the container additionally conducts an end-to-end integration test on the development server of the GitHub Actions virtual machine before the application can be built and deployed. The workflows are automatically triggered by a push to the main branch of each directory in the monorepo, with the following steps:

1. **Dependency Installation and Application Build**
2. **AWS Integration:** Synchronizes the build output (dist directory) with an S3 bucket, as specified in the repository secrets.
3. **CloudFront Invalidation:** After updating the S3 bucket, an invalidation is created on an AWS CloudFront distribution. Invalidation instructs CloudFront to refresh its cache of specified files, in this case, the index.html file in the /container/latest path. This ensures that users always receive the most updated version

of the application. Without invalidation, CloudFront might continue serving an older, cached version of the files, delaying visibility of the latest updates until the cache expires naturally.

```
name: deploy-container
on:
push:
    branches:
    - main
jobs:
test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - run: npm install
    - run: npm run start:theme &
    - run: npm run start:marketing &
    - run: npm run start:auth &
    - run: npm run start:dafne &
    - run: npm run start:neighborhood &
    - run: npm run start:container &
    - run: sleep 5
    - run: npx cypress run
build:
    needs: test
    runs-on: ubuntu-latest
    defaults:
    run:
        working-directory: container
    steps:
    - uses: actions/checkout@v2
    - name: Install dependencies and build container app
        run: |
        npm install
        npm run build
        env:
        PRODUCTION_DOMAIN: ${{ secrets.PRODUCTION_DOMAIN }}
    - uses: shinyinc/action-aws-cli@v1.2
    - run: aws s3 sync dist s3://${{ secrets.AWS_S3_BUCKET_NAME }}/container/latest
        env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        AWS_DEFAULT_REGION: 'us-east-1'
    - run: aws cloudfront create-invalidation --distribution-id ${{ secrets.AWS_DISTRIBU
        env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        AWS_DEFAULT_REGION: 'eu-west-1'
```

Listing 18: Github actions workflow for application shell deployment

5.6.2 Medusa Client

As a complement to the CI/CD pipeline and version management, the Module Federation Dashboard Plugin has been integrated into the development server to support development activities (see Listing 19). This plugin extracts data from the Webpack build process and posts it on the Medusa Client's dashboard [das23]. It processes and presents this information in a user-friendly UI. This centralizes architectural information and visualizes crucial details about the micro-frontends, such as module versions and dependencies. Consequently, each build is saved as a distinct version on the dashboard, following a selected versioning strategy, and can be managed effectively.

```
new DashboardPlugin({
        versionStrategy: require("../package.json").version,
        // version equal to 'dafnePrimaryBlue' in package.json
        filename: 'dashboard.json',
        environment: 'development',
        dashboardURL: `${process.env.DASHBOARD_BASE_URL}/update?token=${process.env.DASHBO
        metadata: {
            baseUrl: 'http://localhost:8085',
            remote: 'http://localhost:8085/remoteEntry.js',
        },
    }),
```

Listing 19: Module Federation Dashboard Plugin of theme app

6 Evaluation

This section assesses the implemented micro-frontend architecture, utilizing Module Federation, in relation to the defined requirements and the developed system design that emerged from these requirements. In this evaluation, the requirements provide a structural framework, with each group of requirements forming a distinct subchapter.

6.1 Constraint: Implementation of Microfrontend Architecture for extensibility

This section assesses the outcomes of the implementation in comparison to Requirement Group 1, concentrating on the success and practicality of extending features within the Dafne app. The final results, showcased in Figure 6.1, confirm the successful feature extension in alignment with Requirement 1a. The Module Federation Dashboard Plugin, integrated to assist the development process, visualizes the relations between applications as seen in Figure 6.2. This visualization confirms that the initial system design depicted in Figure 4.1 was realized post-implementation, maintaining a hierarchical structure with unidirectional data flow and effectively splitting the Dafne app horizontally. It is worth underscoring that the relationships depicted are not dependencies but rather connections, as each application is decoupled and independently functional. The success of this extension, without compromising other features, is also attributed to the Error Boundary, which maintains app stability even when the Neighborhood Service is unavailable or errant, allowing other functionalities of Domain CSD-1 to function, as highlighted in Requirement 1e.

As for Requirement 1b, merging selected domains into a bounded context proved to be an effective micro-frontend identification strategy for a long-term, maintainable prototype. This approach, which limits fragmentation within the Dafne platform due to the high cohesion of Dafne elements, aims to isolate functions within the CSD-2 Use Case Explorer

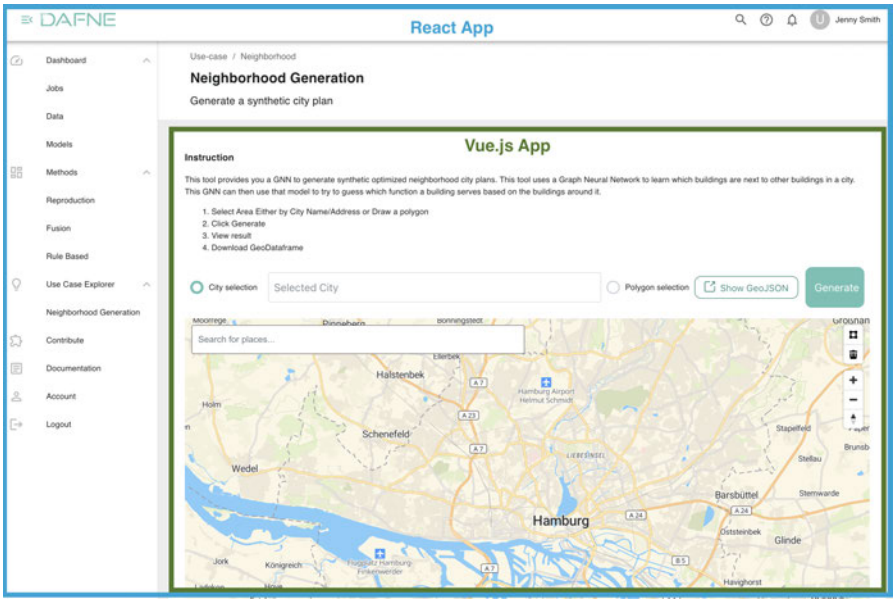

Figure 6.1: Neighborhood App Integrated into Dafne App

due to their potential for loose coupling. These features are unique to *inference as a service* models with unpredictable data models, lacking relevance to the entire platform. It suffices that job details only reference the dashboard, enabling the app to reload as needed.

In terms of integration flexibility referring to requirement 1d), any frontend application constructed using Webpack can theoretically be integrated into the Dafne App, regardless of the framework, as demonstrated with the integration of Vue.js. Nevertheless, the routing solution posed significant challenges. For instance, transitioning from React Router 5 to React Router 6 required considerable effort. Despite initial challenges in developing the NavigationManager, its adaptability to the system design allowed for its reuse across all React remote apps sharing the same React DOM and Routing Stack. However, for developers like those in the Neighborhood Generation who prefer simpler frameworks like Vue.js, the prospect of implementing a routing solution with Module Federation could be daunting, especially for those with limited knowledge of advanced front-end technologies. Module Federation - essentially an npm library for code sharing rather than a framework for micro-frontends - might not easily facilitate straightforward implementations in such scenarios. In the specific case of the Neighborhood Generation, no routing setup was needed in the Vue app, as it was designed as a single-page feature. While implementing a MemoryHistory-based routing solution in Vue for multiple routes

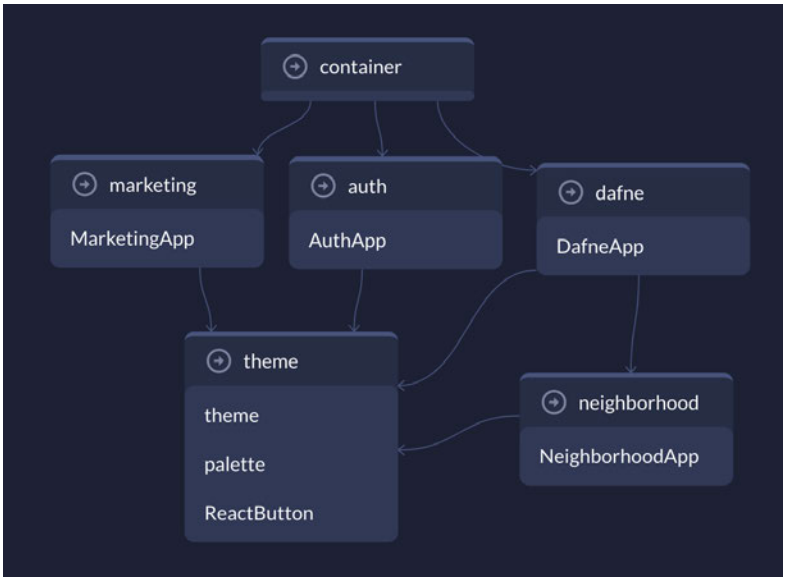

Figure 6.2: UML Diagram in Medusa Client showing the Federated Landscape

is feasible, it could be challenging for developers with limited knowledge of advanced frontend technologies.

In conclusion, while the implementation of micro-frontend architectures using Module Federation is feasible and offers flexibility in platform feature extension, exploring the use of micro-frontend-specific frameworks like single-spa might be beneficial. Single-spa offers framework-specific solutions to many challenges encountered in the Module Federation environment, along with a supportive community (e.g. with a Slack channel) and numerous examples to ease the learning curve [sin].

6.2 Consistency in User Experience

To ensure a consistent user experience, the design system of the application was integrated into the federated landscape as a remote app. While successful, this approach slightly deviates from the requirement of technological and framework neutrality. This is because Material UI, the chosen design system, is a React-based framework, whereas the architecture also integrates a Vue.js application. These Vue.js application utilizes a framework-specific component library named PrimeVue, which necessitates considerable manual effort for adaptation to the components used in the React applications.

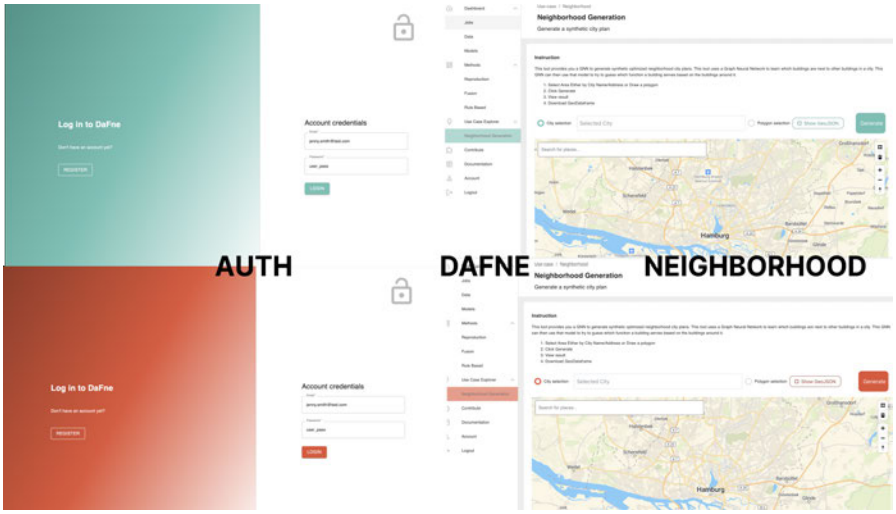

Figure 6.3: Switching between different Versions of the Design System

As a workaround, only the color palette from the remote design system was fetched to maintain at least color consistency across all apps.

Therefore, a framework-independent design system such as Bootstrap [boo], which allows identical components to be used in all applications, can provide a more viable approach to maintaining user interface consistency. The combination with the Module Federation Dashboard Plugin, managed through the central Medusa Client, illustrates the advantages of having a design system as a remote app. For instance, if there's a need to change or test a new primary color in the color system, modifications only need to be made and deployed in the remote theme. This eliminates the need for consuming apps to adapt to these changes manually. To test different versions of the design system, developers can simply switch between uploaded build versions using a dropdown selection UI in the Medusa Client, as shown in Figure 6.3. This approach demonstrates the flexibility and convenience of managing UI changes in a distributed micro-frontend architecture.

6.3 Performance

In assessing the performance, basic network traffic metrics for a specific user journey were measured. The user journey begins at the index route, loads the marketing page, proceeds to the authentication page upon user action, logs in the user, loads the Dafne dashboard, and finally loads the neighborhood app inside dafne. The measurements compare the

versions with and without module sharing, as shown in the table below. It is notable that the number of requests is significantly higher with shared dependencies, which also explains the larger size of uploads. This is likely due to shared dependencies consisting of more, but smaller, chunks for modules. Without module sharing, modules are bundled into larger files. The data shows that the download size, and therefore the total size of resources, is somewhat smaller with shared dependencies. However, no advantage in terms of loading times was observed. On the contrary, disadvantages with regard to the initial loading time were measured in several tests, which in any case requires further investigation for later development cycles.

Table 6.1: Performance Comparison

	With Shared Dependencies	**Without Shared Dependencies**
Number of requests	86	56
Downloaded size	13.7 MB	14.2 MB
Uploaded size	57.9 KB	7.03 KB
Initial page load	770ms	643ms
Size of all resources	17.9 MB	19.2 MB
Transferred data	116kB	33.3 kB

6.4 Communication and State Management

The evaluation of the implementation of the communication strategy based on the requirements shows that the requirements are met. The requirements were established based on micro-frontend principles and the Domain-Driven Design concept that communication between bounded contexts should be loosely coupled. While backend services typically communicate based on contracts or well-defined APIs, in this case, the frontend communication was facilitated through Custom Events, resulting in a reactive system. Although the requirements were met, the manual handling of events can quickly become cumbersome as there is no central event registry or event bus that the developers can access. Instead, they must find the events in the code at the appropriate location.

It's worth noting that alternative communication mechanisms, such as event buses or event streaming platforms such as Kafka [kaf], could potentially offer more streamlined and manageable event handling. Such systems might enable more efficient event processing and a cleaner separation of concerns.

Concerning backend communication, the API gateway pattern has successfully enabled interactions with backend services. Similarly, state management objectives were achieved without using to a global state. Each micro-frontend maintains its state independently, adhering to its chosen technological framework. The local storage emerges as the only centralized data store, aligning with the platform's decentralized architecture.

Examples from the Module Federation repository demonstrate the feasibility of shared contexts and technology-agnostic state management solutions [Exa]. When the Dafne platform evolves beyond the scope of an MVP, the exploration of such state management strategies could prove beneficial. The current reliance on Custom Events for state exchange, while effective, risks becoming convoluted as the platform scales. A transition towards a more structured and maintainable approach to state management and inter-application communication could therefore be a strategic consideration for future development.

6.5 Security and Authentication

Evaluating the implementation of security and authentication requirements indicates successful fulfillment. The application responds reactively to changes in authentication status, communicated via events and managed in local storage. A key element is the container setting up an authentication context, from which access to the Dafne platform is centrally protected and managed. Nevertheless, a discussion point arises regarding which micro-frontend executes the business logic for authentication. In this case, the Dafne app itself carries out the complete business logic for logout - internal state update but also removing the JWT token from the local storage - rather than triggering an event to be handled by either the auth app or the container. Achieving high cohesion in the auth app would ideally involve executing the authentication logic within it. However, since the remotes are designed to operate independently and not communicate horizontally, the most straightforward path for event communication would be for the container to handle the logic, specifically the removal of the token. Centralizing the logic in the container might also enhance flexibility since changes could then be implemented solely by responsible developers of each respective remote.

As of this writing, the final production authentication logic is not yet in place. Therefore, the current prototype proceeds without token validation and refresh logic. This

approach, however, lays a solid foundation for future design of the production version of the platform.

6.6 Search Engine Optimization

Evaluating the implementation against the requirements for the subdomain value proposition GSD-1 reveals that the objectives were not met. The goal was to develop an isomorphic application for the marketing app's landing page to optimize it for search engines. The strategy involved using Next.js for Static Site Generation to pre-render HTML at build time, subsequently serving it to the client. Unfortunately, within the constraints of the current setup and resources, this experiment did not succeed. Lighthouse analysis in Chrome Browser indicated good SEO performance for the standalone Next.js application (refer to Fig. 6.4). However, when integrated into the SPA container, the entire source code was reduced to the root div, leading to a significant decline in SEO performance (as shown in Fig. 6.5). It's important to note that while Lighthouse also highlighted performance issues, the primary focus here is on SEO.

This experiment required a deeper understanding of server-side technologies, and the available documentation for Module Federation proved inadequate for server-side beginners. Moreover, this approach lacked prevalent examples in the software development community.

Despite the setback, this experience suggests a potential path forward. For future developments, it is recommended to consider implementing the entire application with Next.js rather than React. Employing the NextFederationPlugin for the container app could facilitate a hybrid architecture that supports both server-side and client-side rendering. This approach could offer a viable solution for combining SEO-friendly landing pages with the interactive dynamics of SPAs.

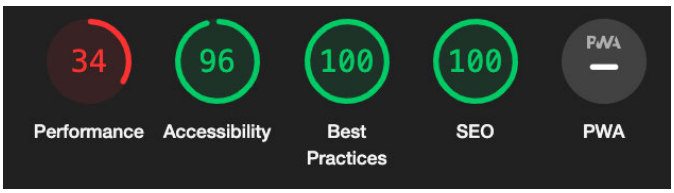

Figure 6.4: Lighthouse Analysis Output of Isolated Next.js App

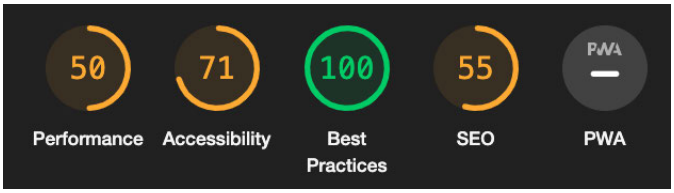

Figure 6.5: Lighthouse Analysis Output of Integrated Next.js App inside the Container App

6.7 Deployment and Maintenance

Evaluating the deployment and maintenance reveals that the implemented automated CI/CD pipeline successfully integrated all the independently developed apps into a cohesive platform. The planned system design utilizing Amazon S3 for storage and Amazon CloudFront as a CDN was effectively implemented. While the applications are managed in a centralized monorepo, the deployment for each individual app is isolated. This ensures that only the modified applications are updated, avoiding the need to process the entire application content through the pipeline. The applications operate independently, but a prerequisite end-to-end test in the container guarantees that they function as intended when composed together.

The Module Federation Dashboard currently serves as a development support tool, centralizing the management of micro-frontends and other remote modules. Its integration has proven immensely useful in development, offering the developer/architect a comprehensive overview of the system landscape, including versions and dependencies. However, the future growth of the platform should aim to manage production builds in the dashboard as well, fully leveraging its benefits. Implementing it for production would allow effortless switching between application versions.

Additionally, incorporating unit tests within each remote app into the pipeline could further strengthen the system for production. As the project scales and development teams expand, the discussion may also shift towards whether a polyrepo approach might be more suitable. Lastly, the current setup also supports flexible deployment strategies, adapting to evolving requirements such as potential Kubernetes cluster deployment for production.

7 Conclusion

This thesis explored the implementation of a micro-frontend architecture in the context of extending AI Software Services on the DaFne platform, which is a multidisciplinary research project. The research was grounded in Design Science Research methodology, aiming to create a practical prototype that enhances the platform's extensibility at the front-end level.

Key findings reveal the successful adaptation of the micro-frontend architecture using Module Federation and React. This integration enables the development of independent yet cohesive units that can be deployed separately. The research demonstrates the architecture's potential to enhance maintainability and extensibility, crucial for DaFne's nature as an open platform that accepts contributions of additional features. In the context of the paradigm shift from traditional cloud-native SaaS to AIaaS, investigating the accessibility of AI Software Services at various levels of software development, particularly in the frontend, becomes essential. The research shows that many concepts commonly associated with the backend in cloud-native environments, such as event-based communication or monolith modularization, are also applicable in the frontend and offer similar benefits.

The success of the architecture in improving extensibility was primarily validated by implementing a problem scenario described in Chapter 3. This scenario focused on aiding the AI *inference as a service* developer for the Neighborhood Generation feature. Their task was to integrate their use case with a frontend, developed with Vue.js, into a React-based host application.

The design of the prototype began with an analysis of the DaFne platform's system context, then applying Domain-Driven Design to segment it into distinct, loosely coupled domains. This segmentation was strategic, merging some domains into a single bounded context to ensure high internal cohesion and avoid unnecessary complexity from over-modularization. This methodology ensured that while domains functioned cohesively

internally, they remained independent, promoting the identification of micro-frontends. The chosen approach was particularly relevant for extending services with distinct domain models, such as use-case based inference as a service models like the Neighborhood Generation. Consequently, following remote applications alongside the container application were identified; the Marketing app, the Authentication app, the DaFne app, and the Neighborhood Generation app with an additional Theme app for the design system.

This method also facilitated the identification of distinct domain-specific requirements for a frontend application, such as the SEO requirements for the landing page. This analytical process, combined with the micro-frontend principles and challenges detailed in the Background Chapter, guided the formulation of non-functional quality requirements for the architecture. Subsequently, this formed the basis for the system's design. When usually with UI monoliths having to make trade-offs between the requirements of the different domains, each micro-frontend can now implement the requirements that are relevant to its domain. This approach also enables the use of different technologies and frameworks for each micro-frontend, as long as they are compatible with the container application.

Following to the implementation of this system design in Chapter 5, a systematic evaluation in Chapter 6 was conducted according to the defined requirements. The requirements were grouped into seven categories, each addressing different aspects of the architecture: extensibility of the platform, UX/UI consistency, performance, communication and state management, security and authentication, search engine optimization, and deployment and maintenance.

One notable advantage, especially in terms of UX/UI consistency, is the ability to share a design system across the entire system landscape. It's worth mentioning that the benefits are most evident in deployment and maintenance rather than any benefits for the user, particularly through the integration of a central dashboard that manages the decentralized frontend landscape.

Several challenges were encountered during implementation, including complexities in routing solutions that were ultimately resolved. Still, almost all defined requirements were met. One unresolved issue is the transition from an SPA to an isomorphic application that combines the advantages of client-side and server-side rendering to enhance SEO. Here, instead of having and SPA host, a hybrid architecture could be developed using the Module Federation plugin for Next.js, enabling both server-side and client-side rendering. An explanation for the experiment can be the developer's limited server-side

expertise and the absence of comprehensive Module Federation documentation. Given the growth of the Module Federation community and increased developer support, such as comprehensive documentation, Module Federation offers many advantages.

In the context of some complexities associated with Module Federation, the evaluation also highlights potential areas for future research. This includes exploring micro-frontend-specific frameworks like single-spa to enhance development efficiency, which could serve as a requirement basis for a potential next DSR cycle.

In conclusion, this work explicitly demonstrates the benefits of micro-frontend architecture by theoretically enabling the platform to integrate numerous isolated full-stack components, like the Neighborhood Generation, with the tech stack of choice. It maintains the appearence of a unified system while allowing for individual customization and extension. Overall, this study contributes to understanding how micro-frontend architectures can support an AI as a Software Service platform. It offers insights into practical implementation strategies and potential future improvements. However, it's worth noting that this architecture may not be ideal for smaller projects due to its complex nature, but has great potential in projects with a more complex organizational structure.

Bibliography

[ACCT21] Isil Karabey Aksakalli, Turgay Celik, Ahmet Burak Can, and Bedir Tekinerdogan. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 180:111014, 2021.

[Ang] Angular. Angular. https://angular.io/ (Accessed: 2023-06).

[bab] What is Babel? · Babel. https://babeljs.io/docs/. (Accessed: 30.11.2023).

[BDEM+18] Scott Boag, Parijat Dube, Kaoutar El Maghraoui, Benjamin Herta, Waldemar Hummer, KR Jayaram, Rania Khalaf, Vinod Muthusamy, Michael Kalantar, and Archit Verma. Dependability in a multi-tenant multi-framework deep learning as-a-service platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 43–46. IEEE, 2018.

[Bel20] Adam Bellemare. *Building Event-Driven Microservices*. " O'Reilly Media, Inc.", 2020.

[boo] Bootstrap. https://getbootstrap.com/. (Accessed: 19.12.2023).

[Clo] AWS | Amazon CloudFront CDN (Content Delivery Network). https://aws.amazon.com/de/cloudfront/. (Accessed: 11.12.2023).

[cyp] JavaScript Component Testing and E2E Testing Framework | Cypress. https://www.cypress.io/. (Accessed: 11.12.2023).

[das23] @module-federation/dashboard-plugin. https://www.npmjs.com/package/@module-federation/dashboard-plugin, November 2023. (Accessed: 18.12.2023).

[DFZ+15] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud C. Narendra, and Bo Hu. Everything as a service (xaas) on the cloud: Origins, current and future trends. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 621–628, 2015. doi:10.1109/CLOUD.2015.88.

[EK03] Amnon H Eden and Rick Kazman. Architecture, design, implementation. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 149–159. IEEE, 2003.

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[Exa] Module-federation-examples: Implementation examples of module federation , by the creators of module federation. https://github.com/module-federation/module-federation-examples. (Accessed: 21.11.2023).

[Fow14] Martin Fowler. Microservices - a definition of this new architectural term. https://martinfowler.com/articles/microservices.html (Accessed: 2023-06), 2014.

[FRSD21] Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani. *Software Architecture: The Hard Parts*. " O'Reilly Media, Inc.", 2021.

[Gee23] Michael Geers. Micro Frontends - extending the microservice idea to frontend development. https://micro-frontends.org/, July 2023. (Accessed: 10.07.2023).

[git] Grundlegendes zu GitHub Actions - GitHub-Dokumentation. https://ghdocs-prod.azurewebsites.net/_next/data/rm0AlolPpgDlGF7dxkBK3/de/free-pro-team@latest/actions/learn-github-actions/understanding-github-actions.json?versionId=free-pro-team%40latest&productId=actions&restPage=learn-github-actions&restPage=understanding-github-actions. (Accessed: 11.12.2023).

[God16] Micah Godbolt. *Frontend architecture for design systems: a modern blueprint for scalable and sustainable websites*. " O'Reilly Media, Inc.", 2016.

[Gor18] Elyse Gordon. *Isomorphic Web Applications: Universal Development with React*. Simon and Schuster, 2018.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in software engineering and knowledge engineering*, pages 1–39. World Scientific, 1993.

[HA10] Neil B Harrison and Paris Avgeriou. How do architecture patterns and tactics interact? a model and annotation. *Journal of Systems and Software*, 83(10):1735–1758, 2010.

[HJ20] Jack Herrington and Zack Jackson. *Practical Module Federation*. ScriptedAlchemy, 2020.

[HRLI17] Holger Harms, Collin Rogowski, and Luigi Lo Iacono. Guidelines for adopting frontend architectures and patterns in microservices-based systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 902–907, 2017.

[HSM14] Altaf Ahmad Huqqani, Erich Schikuta, and Erwin Mann. Parallelized neural networks as a service. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 2282–2289. IEEE, 2014.

[Ing18] Joseph Ingeno. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018.

[Jac] Jherr - Overview. https://github.com/jherr. (Accessed: 21.11.2023).

[kaf] Apache Kafka. https://kafka.apache.org/. (Accessed: 20.12.2023).

[KBC+22] Sabrine Khriji, Yahia Benbelgacem, Rym Chéour, Dhouha El Houssaini, and Olfa Kanoun. Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. *The Journal of Supercomputing*, pages 1–28, 2022.

[KKZS22] Pamela Kunert, Tom Krause, Olaf Zukunft, and Ulrike Steffens. A platform providing machine learning algorithms for data generation and fusion - an architectural approach. 2022.

[KMLS18] Mohamad Kassab, Manuel Mazzara, JooYoung Lee, and Giancarlo Succi. Software architectural patterns in practice: an empirical study. *Innovations in Systems and Software Engineering*, 14:263–271, 2018.

[Lan21] Ryan Lanciaux. *Modern Front-end Architecture: Optimize Your Front-end Development with Components, Storybook, and Mise En Place Philosophy.* Apress, Berkeley, CA, 2021. doi:10.1007/978-1-4842-6625-0.

[LPT+21] Sebastian Lins, Konstantin D Pandl, Heiner Teigeler, Scott Thiebes, Calvin Bayer, and Ali Sunyaev. Artificial intelligence as a service: Classification and research directions. *Business & Information Systems Engineering*, 63:441–456, 2021.

[Mez21] Luca Mezzalira. *Building Micro-Frontends.* " O'Reilly Media, Inc.", 2021.

[MG+11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[MGM+18] Davi Monteiro, Rômulo Gadelha, Paulo Henrique M Maia, Lincoln S Rocha, and Nabor C Mendonça. Beethoven: an event-driven lightweight platform for microservice orchestration. In *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*, pages 191–199. Springer, 2018.

[Mic06] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2(12):10–1571, 2006.

[MUI] Overview - Material UI. https://mui.com/material-ui/getting-started/. (Accessed: 12.12.2023).

[Nexa] Docs | Next.js. https://nextjs.org/docs. (Accessed: 12.12.2023).

[Nexb] Getting Started: Project Structure | Next.js. https://nextjs.org/docs/getting-started/project-structure. (Accessed: 12.12.2023).

[Nex23] @module-federation/nextjs-mf. https://www.npmjs.com/package/@module-federation/nextjs-mf, August 2023. (Accessed: 12.12.2023).

[PAMM20] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. Micro-frontends: application of microservices to web front-ends. *J. Internet Serv. Inf. Secur.*, 10(2):49–66, 2020.

[PMT21] Severi Peltonen, Luca Mezzalira, and Davide Taibi. Motivations, benefits, and issues for adopting micro-frontends: a multivocal literature review. *Information and Software Technology*, 136:106571, 2021.

[PPS21] Y Prajwal, Jainil Viren Parekh, and Rajashree Shettar. A brief review of micro-frontends. *United International Journal for Research and Technology*, 2(8), 2021.

[Pri] PrimeVue | Vue UI Component Library. https://primevue.org/. (Accessed: 15.12.2023).

[reaa] Feature Overview v6.20.0. https://reactrouter.com/en/main/start/overview. (Accessed: 01.12.2023).

[Reab] React. React. https://react.dev/ (Accessed: 2023-06).

[Ric23] Chris Richardson. Pattern: Microservice architecture. https://microservices.io/patterns/microservices.html (Accessed: 2023-06), 2023.

[S3A] Amazon Simple Storage Service S3 – Cloud Online-Speicher. https://aws.amazon.com/de/s3/. (Accessed: 11.12.2023).

[S3C18] Amazon S3 + Amazon CloudFront: A Match Made in the Cloud | Networking & Content Delivery. https://aws.amazon.com/blogs/networking-and-content-delivery/amazon-s3-amazon-cloudfront-a-match-made-in-the-cloud/, June 2018. (Accessed: 11.12.2023).

[SAP] Documentation - Luigi - The Enterprise-Ready Micro Frontend Framework. https://docs.luigi-project.io/docs/general-settings. (Accessed: 21.11.2023).

[Sas] Sass: Documentation. https://sass-lang.com/documentation/. (Accessed: 15.12.2023).

[Sch] Heiko Schröder. Frontends with Microservices. https://www.otto.de/jobs/de/technology/techblog/artikel/frontends-mit-microservices.php. (Accessed: 20.11.2023).

[SGHA17] Roland H Steinegger, Pascal Giessler, Benjamin Hippchen, and Sebastian Abeck. Overview of a domain-driven design approach to build microservice-based applications. In *The Thrid Int. Conf. on Advances and Trends in Software Engineering*, 2017.

[sin] Help | single-spa. https://single-spa.js.org/help/. (Accessed: 20.12.2023).

[ssa] single spa. Getting Started with single-spa | single-spa. https://single-spa.js.org/docs/getting-started-overview. (Accessed: 21.11.2023).

[ssb] single spa. Parcels | single-spa. https://single-spa.js.org/docs/parcels-overview. (Accessed: 21.11.2023).

[Tay09] Hugh Taylor. *Event-driven architecture: how SOA enables the real-time enterprise*. Pearson Education India, 2009.

[Tha20] Mohit Thakkar. Building react apps with server-side rendering. *Use React*, 2020.

[Tho16] ThoughtWorks. Micro frontends. Available: https://www.thoughtworks.com/de-de/radar/techniques/micro-frontends [Accessed June 2023]], November 2016.

[TM22] Davide Taibi and Luca Mezzalira. Micro-frontends: Principles, implementations, and pitfalls. *ACM SIGSOFT Software Engineering Notes*, 47(4):25–29, 2022.

[TS20] Davide Taibi and Kari Systä. A decomposition and metric-based evaluation framework for microservices. In *Cloud Computing and Services Science: 9th International Conference, CLOSER 2019, Heraklion, Crete, Greece, May 2–4, 2019, Revised Selected Papers 9*, pages 133–149. Springer, 2020.

[Vue] VueJS. Vuejs. https://vuejs.org/ (Accessed: 2023-06).

[Weba] Webpack. Awesome webpack. https://webpack.js.org/awesome-webpack/. (Accessed: 27.07.2023).

[Webb] Webpack. Concepts. https://webpack.js.org/concepts/. (Accessed: 27.07.2023).

[Webc] Webpack. Loaders. https://webpack.js.org/concepts/loaders/. (Accessed: 21.11.2023).

[XSCIV19] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. In *Advances in Neural Information Processing Systems*, 2019.

[Yal] YalesRios/DaFne-GNN: GNN for Urban Land Use. https://github.com/YalesRios/DaFne-GNN. (Accessed: 04.12.2023).

[Zac] ScriptedAlchemy - Overview. https://github.com/ScriptedAlchemy. (Accessed: 21.11.2023).

[ZFZ+17] Wei Zhang, Minwei Feng, Yunhui Zheng, Yufei Ren, Yandong Wang, Ji Liu, Peng Liu, Bing Xiang, Li Zhang, Bowen Zhou, et al. Gadei: On scale-up training as a service for deep learning. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 1195–1200. IEEE, 2017.

A Appendix

```
<!DOCTYPE html>
<html lang="en">
<head>
    <style data-next-hide-fouc="true">
        body {
            display: none;
        }
    </style>
    <meta charSet="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>Test next app</title>
    <meta name="description" content="Generated by create next app" />
    <meta name="next-head-count" content="4" />
    <noscript data-n-css=""></noscript>
    <script defer="" nomodule="" src="/_next/static/chunks/
    polyfills.js?ts=1694006009801"></script>
    <script src="/_next/static/chunks/webpack.js?ts=1694006009801"
    defer=""></script>
    <script src="/_next/static/chunks/main.js?ts=1694006009801"
    defer=""></script>
    <script src="/_next/static/chunks/pages/_app.js?ts=1694006009801"
    defer=""></script>
    <script src="/_next/static/chunks/pages/index.js?ts=1694006009801"
    defer=""></script>
    <script src="/_next/static/development/_buildManifest.js?ts=1694006009801"
     defer=""></script>
    <script src="/_next/static/development/_ssgManifest.js?ts=1694006009801"
    defer=""></script>
    <noscript id="__next_css__DO_NOT_USE__"></noscript>
</head>

<body>
    <div id="__next">
        <div>
            <main>
                <style data-emotion="css o2w69a-MuiTypography-root">
                    ...
                </style>
                <h1 class="MuiTypography-root MuiTypography-h1
                css-o2w69a-MuiTypography-root">NextJS App Heading</h1>
            </main>
            <style data-emotion="css 1sra7t5-MuiTypography-root">
                ... <h2 class="MuiTypography-root MuiTypography-h2
                css-1sra7t5-MuiTypography-root">
                                NextJS Component Heading</h2>
            </style>
            <style data-emotion="css ahj2mt-MuiTypography-root">
                ...
            </style>
            <p class="MuiTypography-root MuiTypography-body1
            css-ahj2mt-MuiTypography-root" style="color: #1976d2">
                Lorem Ipsum</p>
        </div>
    </div>
    <script src="/_next/static/chunks/react-refresh.js?ts=1694006009801"></script>
    <script id="__NEXT_DATA__" type="application/json">
    {"props":{"pageProps":{}},"page":"/","query":{},"buildId":
    "development","nextExport":true,"autoExport":true,"isFallback":false,"scriptLoader":[]}
  </script>
</body>
</html>
```

Listing 20: Next.js page isolated from SPA container

```
describe('Routing Test', () => {
const container_base_url = 'http://localhost:8080';


const login = () => {
    cy.get('button[type="submit"]').should('be.visible').click();
};

it('Visits the Marketing Microfrontend on index', () => {
cy.visit('localhost:8080');
cy.url().should('eq', `${container_base_url}/marketing/`);
});
it('Visits the Marketing Microfrontend on /marketing', () => {
cy.visit(`${container_base_url}/marketing`);
cy.contains('Marketing App');
});
it('Navigates to auth from marketing, logs in and navigates to dafne', () => {
cy.visit(`${container_base_url}/marketing`);
cy.contains('auth').click();
cy.url().should('eq', `${container_base_url}/auth/login`);

    login()

    cy.window().then((window) => {
        const token = window.localStorage.getItem("jwtToken");
        expect(token).to.exist;
    });

    cy.url().should('eq', `${container_base_url}/dafne/dashboard/jobs`);
  });

it('Checks if protected /dafne route can be accessed when visiting it
in unauthenticated mode', () => {
cy.visit(`${container_base_url}/dafne`);
cy.window().then((window) => {
    const token = window.localStorage.getItem("jwtToken");
    expect(token).to.be.null;
});
cy.url().should('eq', `${container_base_url}/auth/login`);
  });
it('Checks correct redirect to protected route after successfull login', () => {
cy.visit(`${container_base_url}/dafne/methods/reproduction`);
cy.window().then((window) => {
    const token = window.localStorage.getItem("jwtToken");
    expect(token).to.be.null;
});
cy.url().should('eq', `${container_base_url}/auth/login`);
login()
cy.url().should('eq', `${container_base_url}/dafne/methods/reproduction`);
});
});
```

Listing 21: Shell Integration Test

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original