INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

A Micro Frontends Solution - Analyzing quality attributes

RICARDO ALEXANDRE PINTO DA SILVA

Junho de 2021

POLITÉCNICO
DO PORTO

A Micro Frontends Solution

Analyzing quality attributes

Ricardo Alexandre Pinto da Silva

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Supervisor: Isabel de Fátima Silva Azevedo

Porto, June 2021

Dedication

This thesis is dedicated to my beloved parents and sister, who have been the source of encouragement and inspiration throughout my entire life.

To my girlfriend for the incredible support during the hard times and to help me become better everyday.

To my friends for being by my side during this path and making my academic journey full of joy and amazing moments.

To my family for always being there.

A special thank you to my thesis supervisor, Isabel Azevedo for guiding me through this journey.

Resumo

A arquitetura de *micro frontend*s propõe a decomposição de uma aplicação web dividida em módulos menores por página ou funcionalidade, sendo que cada módulo é propriedade de uma equipa multifuncional com responsabilidade *end-to-end*. Essa abordagem visa trazer os benefícios dos micro-serviços para o *frontend*, permitindo o desenvolvimento de pequenos *frontends* independentes, com reduzida complexidade e dependência da equipa, aumentando a escalabilidade no geral.
Os *Micro Frontends* estão a tornar-se populares dia após dia. Embora não seja em demasia, a documentação sobre o assunto aumenta a cada dia e é possível encontrar relatos de casos de uso de sucesso de grandes empresas e exemplos da aplicação da arquitetura.
Um dos principais desafios dos *micro frontends* é a composição do lado do cliente. A possibilidade de criar uma *single-page application* composta de módulos importados remotamente durante o tempo de execução é possível e facilitado com o surgimento de frameworks como *single-spa* e a nova técnica: Module Federation. No entanto, a análise de tais soluções com base em atributos de qualidade é escassa.
O objetivo deste trabalho é realizar uma análise em *micro frontends* com base nos atributos de qualidade de software da ISO-20510, nomeadamente: Manutenibilidade, Eficiência de Desempenho, Escalabilidade, e Testabilidade.
Esta dissertação fornece uma visão geral do estado atual dos *micro frontends* até ao momento, estruturada utilizando a metodologia DSRM para organizar o estudo sobre a nova abordagem arquitetural. Esse estudo inclui os seus benefícios e problemas, casos de uso atuais, padrões comuns, técnicas, *frameworks*, arquitetura e uma visão geral da organização de equipas.
A prova de conceito é desenvolvida usando uma abordagem de *Module Federation*, com base em um *application shell* unificado para composição do lado do cliente.
A validação da solução é baseada na metodologia GQM para estruturar a sua análise em relação à Manutenibilidade, Eficiência de Desempenho, Escalabilidade, e Testabilidade.

Palavras-chave: *micro frontends*, Composição, *module federation*, arquitetura de software, micro-serviços no frontend.

Abstract

Micro frontends architecture proposes the decomposition of a web application divided into smaller modules by page or feature, whereas each piece is owned by a cross-functional team with end-to-end responsibility. Such an approach aims to bring the benefits of microservices to the frontend side by allowing the development of small self-contained and self-deployable frontends, with reduced complexity and team dependency, increasing overall scalability.

Micro Frontends are becoming popular day by day. Although not superabundant, documentation on the subject is increasing every day and it is possible to find reports of successful use cases of large companies and examples on the application of the architecture.

One of the main challenges of micro frontends is client-side composition. Being able to create a single-page application composed of remote pieces imported at runtime is possible and facilitated with the appearance of frameworks such as single-spa and the newly Module Federation. Nevertheless, analysis of such solutions based on quality attributes is scarce.

The objective of this work is to perform an analysis on micro frontends based on ISO-20510 software quality attributes, namely: Maintainability, Performance Efficiency, Scalability, and Testability.

This dissertation provides an overview of the current state of micro frontend to date, structured using DSRM methodology to organize the study on the architectural approach. Such study includes its benefits and caveats, current use cases, common patterns, techniques, frameworks, architecture, and a view on team organization.

The proof of concept is developed using a module federation approach, based on a unified app shell for client-side composition.

Solution validation is based on the GQM approach to structure its analysis regarding maintainability, performance efficiency, testability, and scalability.

Keywords: micro frontends, client-side composition, module federation, software architecture, frontend microservices.

Table of Contents

1 Introduction 1

1.1 Context 1

1.2 Problem 3

1.3 Objectives 4

1.4 Research methodology 4

1.5 Document Structure 5

2 UI composition 7

2.1 Micro Frontends 7

2.1.1 Benefits 8

2.1.2 Downsides 9

2.1.3 Use Cases 9

2.1.4 Spotify 10

2.1.5 Zalando 10

2.1.6 Dazn 13

2.1.7 Ikea 14

2.1.8 Facebook 15

2.1.9 Hello Fresh 16

2.1.10 Summary 19

2.2 Patterns 20

2.2.1 Client-Side Composition 20

2.2.2 Server-side Composition 20

2.2.3 Edge-side Composition 21

2.3 Related Patterns 22

2.3.1 API Gateway 22

2.3.2 Backends For Frontends (BFF) 22

2.4 Techniques 24

2.4.1 Separate Runtime 24

2.4.2 Shared Runtime 25

2.5 Frameworks 29

2.5.1 Single-SPA 29

2.5.2 PuzzleJS ... 32
2.5.3 Tailor ... 32
2.5.4 Podium ... 33
2.5.5 Piral ... 34
2.5.6 Luigi ... 34
2.6 Team Organization ... 35
2.6.1 Component Team ... 35
2.6.2 Feature Team ... 36
2.6.3 Conclusion ... 37
2.7 Quality attributes ... 40
2.7.1 Maintainability ... 40
2.7.2 Performance Efficiency ... 41
2.7.3 Scalability ... 41
3 Value Analysis ... 43
3.1 Business Process & Innovation ... 43
3.2 Customer Value ... 47
3.3 Value Proposition ... 48
3.4 Canvas Business Model ... 49
3.5 AHP ... 51
4 Techniques Comparison ... 59
4.1 Route Based ... 60
4.2 IFrames ... 61
4.3 Native WebComponents ... 62
4.4 Framework-based Web Components ... 63
4.5 Server-side Transclusion ... 64
4.6 Edge-side Transclusion ... 65
4.7 Module Federation ... 66
4.8 Summary ... 67
5 Solution Design ... 69
5.1 Requirements Specification ... 69

5.2 Team Organization & Technology Stack 71
5.3 Architectural Alternatives 71
5.4 Architecture In Detail 74

6 Solution Implementation 79

6.1 Overview 79
6.2 Module Federation 80
6.3 Micro Frontends 83
6.3.1 App Shell 83
6.3.2 Navbar 84
6.3.3 Catalog 84
6.3.4 Purchases 85
6.3.5 Cart 86
6.4 Other Aspects 87
6.4.1 Routing 87
6.4.2 State Handling 88
6.4.3 Error Handling 89
6.5 Solution Verification 90

7 Solution Validation 93

7.1 Design 93
7.2 Experiments 97
7.2.1 Maintainability 97
7.2.2 Scalability 98
7.2.3 Performance 100
7.2.4 Testability 101
7.3 Summary 102

8 Conclusions 105

8.1 Contributions 105
8.2 Challenges and Limitations 105
8.3 Achieved Goals 107
8.4 Future Work 107
8.5 Final remarks 108

Annex A – Solution Experimentation Results 118

Annex B – Solution Validation Results 129

List of Figures

Figure 1 - Frontend technologies evolution timeline. Source: (Geers, 2020) ... 2

Figure 2 - Zalando architechure. Source: (CodeTalks, 2017) ... 12

Figure 3 - Sketch of the architecture at DAZN. Source: (Mezzalira, 2019) ... 14

Figure 4 - Facebook page composed of different *pagelets.* Source: (Facebook.com, 2010) ... 16

Figure 5 - Demo view of the Hello Fresh infrastructure. Source: (Senders, 2017) ... 18

Figure 6 - Client-side Composition example. Source: (Geers, 2020) ... 20

Figure 7 - Architecture example for Server-side Composition. Source: (Geers, 2020) ... 21

Figure 8 - Main patterns comparison for Micro Frontends. Source: (Peltonen et al., 2020) ... 21

Figure 9 - Example use of BFF. Source: (Newman, 2015) ... 23

Figure 10 - Example of Bidirectional hosts. Source: (Mrowetz, 2020) ... 27

Figure 11 - ESI application example using Apache server. Source: (Newman, 2019) ... 29

Figure 12 - Component Teams. Source: (Gidlund, 2016) ... 36

Figure 13 - Feature Teams. Source: (Visual-paradigm.com, 2020) ... 37

Figure 14 - Diagram of the innovation process. Source: (Belliveau, et al.,2002) ... 44

Figure 15 - NCD Model. Source: (Koen, et al, 2014) ... 45

Figure 16 - Canvas Business Model ... 50

Figure 17 - Hierarchical Decision Tree - Selection of the Micro Frontend pattern. ... 52

Figure 18 - Architecture Diagram for Server-side transclusion Approach ... 72

Figure 19 - Architecture Diagram for Client-Side Composition approaches ... 73

Figure 20 - Architecture Diagram of the Solution ... 75

Figure 21 - Deployment Diagram of the solution. ... 77

Figure 22 - Purchases Micro Frontend Sandbox – for testing purposes. ... 83

Figure 23 - Home Page showing the micro frontend composition. ... 84

Figure 24 - Custom Event fired when clicking the buy button. ... 85

Figure 25 - Checkout Page ... 86

Figure 26 - Purchase History Page ... 86

Figure 27 - Cart Button and Drawer ... 87

Figure 28 - Navbar Fallback for Cart Micro frontend failure – Cart button will not appear. ... 89

Figure 29 - App Shell fallback to remote importing failure. ... 89

Figure 30 - Summary of the errors for 1.000 VU and no scalling load test. ... 99

Figure 31 - Summary of Maintainability Rating for AppShell Micro Frontend. ... 118

Figure 32 - Summary of Maintainability Rating for Purchases Micro Frontend....................... 119
Figure 33 - Summary of Maintainability Rating for Catalog Micro Frontend.......................... 119
Figure 34 - Summary of Maintainability Rating for Navbar Micro Frontend. 120
Figure 35 - Summary of Maintainability Rating for Cart Micro Frontend. 120
Figure 36 - Scalability experiment with 1 virtual user and no scaling performed. 121
Figure 37 - Scalability experiment with 50 virtual users and no scaling performed................ 121
Figure 38 - Scalability experiment with 500 virtual users and no scaling performed............. 121
Figure 39 - Scalability experiment with 1.000 virtual users and no scaling performed.......... 122
Figure 40 - Scalability experiment with 1 virtual user and scaling up to 3 replicas. 122
Figure 41 - Scalability experiment with 50 virtual users and scaling up to 3 replicas............. 122
Figure 42 - Scalability experiment with 500 virtual users and scaling up to 3 replicas............ 123
Figure 43 - Scalability experiment with 1.000 virtual users and scaling up to 3 replicas......... 123
Figure 44 - Kubernetes dashboard shows activity peak on each replica while load testing.... 123
Figure 45 - Performance testing summary with only Simulated Throttling............................ 124
Figure 46 - Performance testing summary without constraints applied. 124
Figure 47 - Performance testing summary with only Clear Cache applied. 125
Figure 48 - Performance testing summary with Simulated Throttling and Clear Cache.......... 125
Figure 49 - Cyclomatic Complexity results for Purchases micro frontend. 126
Figure 50 - Cyclomatic Complexity results for Catalog micro frontend. 126
Figure 51 - Cyclomatic Complexity results for AppShell micro frontend. 127
Figure 52 - Cyclomatic Complexity results for Cart micro frontend.. 127
Figure 53 - Cyclomatic Complexity results for Navbar micro frontend................................... 128
Figure 54 - “Buy a product” end-to-end test result, using cypress GUI. 129
Figure 55 - Navbar and Cart Integration tests, using Cypress GUI. 129

List of Tables

Table 1 - Comparison between team organizations. Source: (Less.works, n.d.) 38
Table 2 - Longitudinal perspective of value 48
Table 3 - Fundamental Scale. Source: (Saaty, 1990) 52
Table 4 - Comparison Matrix between the criteria 53
Table 5 - Normalized Comparison Matrix and Relative Priority Vector 53
Table 6 - Random Consistency Values. Source: (Nicola, 2019) 54
Table 7 - λ-max, Consistency Ratio and Consistency Index results 55
Table 8 - Comparison Matrices between Alternatives and the Criteria. 56
Table 9 - Normalized Matrices for Alternative-Criteria Comparisons and Local Priority 57
Table 10 - Criteria/Alternatives Classification Matrix and Composite Priority 58
Table 11 - Summary Table of the patterns 67
Table 12 - Use Cases of the solution 71
Table 13 - Correspondence between Micro frontends and Microservices 80
Table 14 - Goals and Questions defined for each Quality Attribute. 94
Table 15 - Relation between metric, rating and evaluation for Maintainability. 95
Table 16 - Relation between metric, possible outcome and evaluation for Scalability............. 96
Table 17 - Relation between metric, rating and evaluation for Performance. 96
Table 18 - Relation between metrics and evaluation for testability... 97
Table 19 - Summary of Maintainability analysis results. 97
Table 20 - Average HTTP request duration on each load test. 99
Table 21 - Rate of failure of HTTP requests on load tests. 99
Table 22 - Summary of the performance tests results. 100
Table 23 - Summary of testability results 101

List of Code Excerpts

Code 1 - Webpack configuration example for federated modules. Source: (Herrington, 2020a) 26

Code 2 - Example of composing a page using two different Single Page Applications. 31

Code 3 - Webpack configuration example for the host application. 81

Code 4 - Webpack configuration example for Remote application.. 82

Code 5 - *index.html* example for remote application importing. 82

Code 6 - Router example.. 88

Code 7 – A simple test to assert if AppShell is loading Navbar component. 91

Acronyms

SPA	Single Page Application
CDN	Content Delivery Network
DOM	Document Object Model
W3C	World Wide Web Consortium
HTML	Hypertext markup Language
AJAX	Asynchronous JavaScript and XML
SEO	Search Engine Optimization
ESI	Edge-side Includes
SSI	Server-side Includes
JSON	JavaScript Object Notation
URL	Unified Resource Location
BFF	Backend for Frontend
JS	JavaScript
AHP	Analytic Hierarchy Process
URI	Unique Resource Identifier
VU	Virtual User(s)

1 Introduction

This chapter is dedicated to giving an overview of the document structure, the current context of this work, and explaining the problem this dissertation is based on. Posteriorly, the objectives, the approach, and development process are defined.

1.1 Context

Nowadays, companies have many web projects with multiple teams working in the same codebase regarding the web application frontend. The web application communicates with a server-side layer containing all the microservices designed according to the projects' domains and feature needs. Every microservice has either a database or communicate with external/internal APIs to fetch data and serve the web application. The teams work in an agile mindset where incremental value delivery is needed for continuous review by the stakeholders by the end of each sprint.

Monolith frontend web applications are seen as a bottleneck (Dhiman, 2019). As codebases become larger, a small change in each feature means having to trigger a pipeline for the whole application, where all the tests will have to be re-run, and the deployment must happen as a whole. This difficulty in scaling and maintaining results in a continuous increase of dependencies from different teams responsible from the different areas, as the application continues to grow, the complex business domain boundaries start to get blurry and the codebase more extensive,

resulting in decreased flexibility on changes, where a small change to the codebase can impact other features or teams due to high coupling and consequently it's harder for a team to focus on their unique subdomains autonomously. These shared concerns between features usually result in an unpredictable application where it is harder to keep track of the possible points of failure and less flexible to change. Having these drawbacks in mind, modifying the application means gather everyone in a room on a series of formal meetings to keep everyone on track and having consensus on choosing the best next step to follow, which ends up being very resource and time-consuming overall.

Moreover, excessive meetings have a huge financial and personal impact on companies around the globe (Doodle, 2019), where millions of dollars are wasted every year in excessive or poorly organized meetings and for increased frustration and time wasted of their employees.

Web development technology is also another part of the problem, having a monolith frontend means a single stack of technology is being used in the development of each layer, means for instance, if the web application is developed in AngularJS, migrating for VueJS means having to migrate a huge codebase to the new technology. There are high chances of a company facing this problem due to the fast-paced evolution of frontend technologies (Figure 1) and consequently, expecting lifetime support for a given technology is not currently realistic - as an example - AngularJS, a popular JavaScript framework whose support is near its end (Convective, 2018).

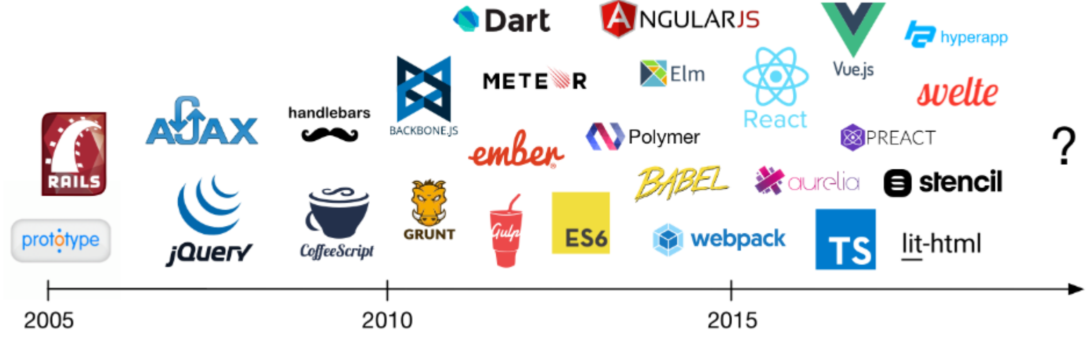


Figure 1 - Frontend technologies evolution timeline. Source: (Geers, 2020)

From a business point of view, where information technology plays a major role nowadays, there is a demand for a quick time to market. Flexibility on change and easier scaling are crucial for a company to stay ahead of the competition and deliver the best product to its customers.

Having the previously described problems in mind, an approach of micro frontends will be explored so that frontend development can take advantage of the benefits of having a modular

architecture on the client-side (as seen in the server-side with microservices) such as the increased autonomy of the teams, improved continuous delivery and decoupling between features, allowing a more efficient development, increasing the productivity and time to market to try to tackle the current weaknesses in this area.

1.2 Problem

Micro Frontend proposes not separate frontend and backend teams, but cross-functional teams able to deal with their parts of interface: "a web application is broken up by its pages and features, with each feature being owned end-to-end by a single team" (Thoughtworks, 2019). There have been some proposals for micro frontend implementation (Geers, 2019), but with few details.

Nowadays, companies all over the world comprise multiple teams working on extensive web products. As existing architectural options for web development can become cumbersome, most of these applications end up being large monolithic frontends. Consequently, as web applications grow, there is a need to scale development as numerous different teams need to work in the same codebase and that is when the problems arise. Although it is possible to see early adopters of Micro Frontends, many companies are still hesitant to adopt the technique due to the lack of knowledge concerning it, which online documentation can be confusing and contradictory at times (Peltonen et al., 2020).

Micro Frontends promises to enable the decomposition of monolithic frontends into individual and independent frontends, reducing overall complexity and increasing team independence, obtaining the same benefits as microservices on the backend side, namely: Incremental Upgrades, decoupled codebases, independent deployment, and increase teams' autonomy (Peltonen et al., 2020) (Jackson, 2019). To effectively study this approach, a prototype is required to address the challenges, benefits, and perks of implementing such architecture, allowing to understand the improvements or drawbacks in terms of quality attributes such as scalability, maintainability, testability and performance efficiency.

1.3 Objectives

The main objective is to evaluate a micro frontend solution in terms of the ISO-20510 Software Quality Attributes seen as problematic in the monolithic approach, and understand how a modular approach can tackle such issues.

This objective will be accomplished by following the steps:

- Study the implementation of micro frontends regarding actual use cases, patterns, techniques, and frameworks to address its advantages and drawbacks.
- Understand the main quality attributes seen as problematic in the monolithic approach that a micro frontend architecture can tackle.
- Develop a proof-of-concept solution based on a micro frontend architecture, including a backend.
- Analyze the micro frontend solution in terms of the main quality attributes: Maintainability, Scalability, Testability, and Performance Efficiency to draw conclusions based on metrics.

1.4 Research methodology

This dissertation is following the Design Science Research Methodology (DSRM), aiming to improve the relevance and the quality of the solution to the problem previously stated. DSRM is a framework that provides the needed guidelines for evaluating and performing design science research in information systems by defining and analyzing the constraints, objectives, processes, and results to be presented posteriorly in the clearest terms with legitimate value to the audience (Peffers et al., 2007).

This methodology consists of six activities (Peffers et al., 2006), applied in the context of this dissertation as follows:

- Problem identification and motivation – consists of defining the problem that is being the target of the research and to justify how the proposed solution will be valuable in this context. In this document, the problem is described in section 1.2 and the value of the proposed solution in section 2.
- Objectives of a solution – consists of defining the objectives based on the problem's constraints. This requires the study of several existing approaches and their assets or

drawbacks to correctly choose the objectives, which can be quantitative (in the means of how a specific solution can be more desirable over others) or qualitative (how a given artifact can be the solution to a given constraint). This activity is described in section 1.3.

- Design and development – Create and implement the chosen solution. Described in sections 5 and 6. The design is performed after analyzing the current state-of-the-art frameworks, patterns, and techniques to develop a *proof of concept* that faithfully implements this architecture considering all drivers. An e-commerce application will be developed as a single-page application using a micro frontend approach.
- Demonstration – consists of describing how effectively the solution solves the problem. Involves experimentation and some type of evidence in how well the implemented solution tackles the faced concern. This is described in section 7.
- Evaluation – consists of measuring how the implementation holds the solution for the problem using metrics. This is also described in section 7 by comparing the initial objectives with the outputs of the Demonstration activity.
- Communication – this final step consists of the communication on how effective the outputs of the proposed solution are in tackling the faced problem, the utility in the context of the area, and how it meets the expected value. Regarding this dissertation, communication is described in the final Section 8 - Conclusions, and the final presentation to the evaluation audience.

1.5 Document Structure

Chapter 1 concludes with the document structure subchapter, aiming to describe each chapter composing the document. This document organizes into 8 chapters, namely:

- Chapter 1 presents the context of the subject, the problem statement, the objectives of this document, and the research methodology.
- Chapter 2 provides an overview of the study of the current state-of-art on micro frontend architecture, including its benefits, drawbacks, related architectural patterns, techniques, frameworks, and actual use cases. This chapter ends with an overview of the quality attributes closely related to micro frontends, which are the target of the study.

- Chapter 3 is dedicated to the value analysis, divided into the business process and innovation subchapter, customer value, and value proposition presented using the canvas business model. To conclude, AHP methodology is used to aid in the multicriteria decision of choosing the composition pattern to develop the proof of concept, based on a set of criteria.
- Chapter 4 presents a comparison of all micro frontend techniques studied in previous chapters, using 6 characteristics inherent to the approach.
- Chapter 5 presents the design of the solution, describing requirements, use cases, technology stack, architecture alternatives, detailed architecture and deployment of the solution.
- Chapter 6 presents the solution implementation, providing a detailed overview of the implementation. Specific characteristics of the solution are explained such as how module federation is implemented or the communication between micro frontends is achieved.
- Chapter 7 refers to the Validation of the Solution. In this chapter, the validation is assessed based on the Goal Question Metric approach and tests are performed on the implemented solution based on the quality attributes specified in previous chapters. To conclude the chapter, the subsection of summary provides conclusions based on such experiments.
- Chapter 8 is the final chapter and refers to the Conclusion of the overall document, describing conclusions on achieved goals, found challenges, a global appreciation, and future work on the subject.

2 UI composition

This chapter is dedicated to showing the current state of the art, where known use cases of companies implementing a UI composition, as well as the patterns, techniques, and frameworks for Micro Frontends are studied.

2.1 Micro Frontends

Micro Frontends is an architectural approach for frontend applications, characterized by the composition of features independently delivered (Jackson, 2019).

This approach can be implemented by composing the frontends on the client-side, edge-side, and server-side, or as a combination of these by making use of a variety of techniques. Composition patterns and techniques are detailed in sections 2.2 and 2.4.

This approach is usually associated with cross-functional teams that usually work in a single or reduced number of business areas, owning and developing a feature end-to-end (Geers, n.d.).

Micro frontends are seen as the microservices on the frontend side due to the similarities, as they are based on the same principles (Peltonen et al., 2020):

- Automation – As the approach will be based on having several pieces that are composed together will create the website, it is important to automate everything from testing to deployment to create confidence and deliver faster.
- Decentralization – Each team owns a single piece of the complex "puzzle" that is a web application, so decisions need to be made locally and based on the specific feature's

problem, instead of deciding on the overall system that can have an unexpected impact on other parts.

- Independent Integration and Deployment – Each Micro Frontend can be developed and deployed independently, without impacting other teams.
- Failure Isolation – If a Micro Frontend fails, other pieces should still work together and should not have an impact or be able to allow for alternative content. Also, the ability to lazy load can provide the additional benefit of loading the pieces exclusively when necessary.
- Organized around Business Domains – Like Microservice, Micro Frontends should be based on the principles of Domain-Driven Design, as they should be created according to the business and technical needs according to the business domain of a problem, and accordingly to the ubiquitous language.

2.1.1 Benefits

Related benefits of Micro Frontends are the following (Jackson, 2019) (Peltonen et al., 2020):

- Incremental Upgrades – A benefit when migrating, in the means that it can be done incrementally instead of all at once, and also when introducing changes, coping entirely with an agile mindset.
- Loosely Coupled Codebase – Separating a monolithic codebase into multiple smaller modules according to the business sub-domains decreases dependencies between the codebase and teams, consequently increasing autonomy.
- Independent development and delivery – Micro Frontends are developed autonomously and can be delivered independently at their own pace, without impacting or creating dependencies between teams.
- Autonomy in team organization – Each team owns a module end-to-end and can be organized and specialized according to the needs.
- Technology Agnostic – Each module can be developed in different technologies according to the need.
- Improved Testability – Loosely coupled codebase means testing each module independently, becoming not only easier to test but faster when automating tests.

- Increased Overall Scalability – Easier to introduce new features or new teams due to the loose coupling nature, up-scale is also facilitated as there's no need to scale the entire system.
- Improved fault isolation – A Micro Frontend failing will not impact others as there is the possibility to show alternative content.
- Improved Migration – By Having the system separated into smaller modules, not only is easier to introduce new features as it is easier to replace existing modules.

2.1.2 Downsides

As a relatively new approach, micro frontends have drawbacks. Although some solutions or techniques can be used to overcome or at least reduce the effect of such problems, the following are the downsides related to the approach (Jackson, 2019) (Peltonen et al., 2020):

- Environment differences – As each Micro Frontend can be developed, tested, and deployed independently, behavior can be unexpected when shipping to production, this underlines the importance of automation and fault isolation to mitigate this issue.
- Payload Size – While technology agnosticism can be a benefit as increases the autonomy of teams, it ultimately increases payload sizes and can become a critical problem in the micro frontend approach, consequently increasing complexity and degrade performance.
- Increased overall complexity – Organizing teams, integrate a wide variety of technologies, compose the distributed modules, and monitoring all the small pieces will be challenging and will introduce the overall complexity of the system.
- Increased duplication and shared dependencies – Need to be managed carefully, as each piece is independent, code duplication will eventually happen, sharing dependencies such as libraries can tackle the problem and also become a new problem as having multiple codebases depending on a single library will ultimately increase coupling.
- Inconsistent User Experience – With the increase of autonomy and isolation, having a consistent User Experience can become challenging, since all the Micro Frontends should not rely on a single shared stylesheet, for instance, maintaining the loose coupling.

2.1.3 Use Cases

In this sub-section, the use cases of companies applying micro frontend approaches are presented.

2.1.4 Spotify

The Music Streaming company, Spotify, relies on Iframe Composition to implement its Micro Frontend Solution and an event Bus for handling the events across these iframes (Mezzalira, 2019).
This desktop client User Interface (UI) is built in JavaScript, on top of a C++ core also used by iOS and Android Apps. This core is responsible for features such as playback or other offline feature and rendering and interaction with the UI is done using JavaScript. The UI is developed in HTML and CSS. The communication between the C++ Core and UI is performed by a messaging interface called the bridge. The UI is composed of many small web applications called *Spotlets*, running inside a Chromium Embedded Framework, where each *Spotlet* is embedded in its iFrame, giving to the teams full autonomy on choosing their frameworks. Although this approach is good in terms of team autonomy, also has the disadvantage of duplicating different versions of libraries. They also share functionalities such as their CSS Framework (called *GLUE*), since they want them to work the same way across all the micro applications (Johansson, 2015).
So that every UI View can be developed autonomously, Spotify uses an internal framework named 'Stitch' to allow any team to work in their feature's view isolated and run in a sandboxed iframe, to guarantee no other feature is broken in the process (Blixt, 2013).
As for team organization, Spotify is internally divided into small teams of 3 to 12 people called *squads*, who own and maintain a feature end-to-end with iOS, Android, Web, and Backend developers to be fully independent and not needing to ask for permission or depending on any other squad (Johansson, 2015).

2.1.5 Zalando

The e-commerce company Zalando was one of the pioneers in the micro frontend architecture with Project Mosaic (CodeTalks, 2017). It is an ecosystem of libraries used to implement Zalando's approach on micro frontend architecture, such as:

- *Skypper* - HTTP router.

- *Tailor* – layout service.
- *Shaker* – show the UI components.
- *Quilt* – storage of the templates used posteriorly by Taylor.
- *InnKeeper* – storage of routes.
- *Tessellate* – server-side render responsible for creating the static HTML.

As a company operating in 15 countries and with 1.600 tech employees, Zalando wanted a solution to be able to scale the development teams and the architecture. In this case, growing the frontend team meant new features to existing products or new products to be built.

The reasons behind the scale of the teams were:

- Less time in deployments or less time dealing with dependencies between different teams.
- Create diversity to be able to see different perspectives to the same problems – fresh ideas led to innovation.
- Encourage overall innovation.

This led to an exponential growth in the number of produced apps and development teams. To make this possible, Zalando introduced Radical Agility – a way of running the business based on 3 main principles: Purpose, Autonomy, and Mastery, or in one sentence – "trust instead of control". The focus was to increase innovation and productivity.

This consisted in organizing the developers into small and autonomous teams which are trusted and accept responsibility, they are given the freedom to self-organize and a clear purpose. In consequence of reorganizing the team, they had to reorganize the architecture. Zalando intended:

- Team autonomy: many teams working independently and in parallel.
- Independent Release Cycles: possibility of continuous deployments in their feature.
- Mix different technologic stacks: tool diversity.
- Easy a/b Testing: adding and removing features fast.

Zalando realized these were the principles from microservices and wanted to use them to tackle the monolithic frontend problem.

They had their frontend teams dependent on each other working on a monolith, where they could not choose different technologic stacks beside the common framework used on the web app and they could not deploy independently.

Bearing this in mind they introduced Project Mosaic. Mosaic is a set of services and libraries mainly open-source allied to a specification that defines how its components interact with each other. The main goal is to support a micro-service-based architecture for large-scale websites.

Mosaic addresses this architecture with the use of *fragments*. *Fragments* are services with different purposes such as to render HTML, later composed together at runtime according to template definitions.
These fragments are frontend microservices owned by different and independent teams and do not have necessarily to be visible to the user in the final product.
Besides rendering HTML, it is possible to fetch data and handle AJAX requests as well.
Business logic in the fragments is not supposed, but everything is flexible and can be chosen by the development team.
The result was the ability to iterate rapidly through the fragments, more flexibility in technology choices and development decisions.
The fragments are then rendered by the layout services (Tailor) which is a layer on top of the fragments, this service assembles the content render with the fragments and uses templates to know how to assemble them.

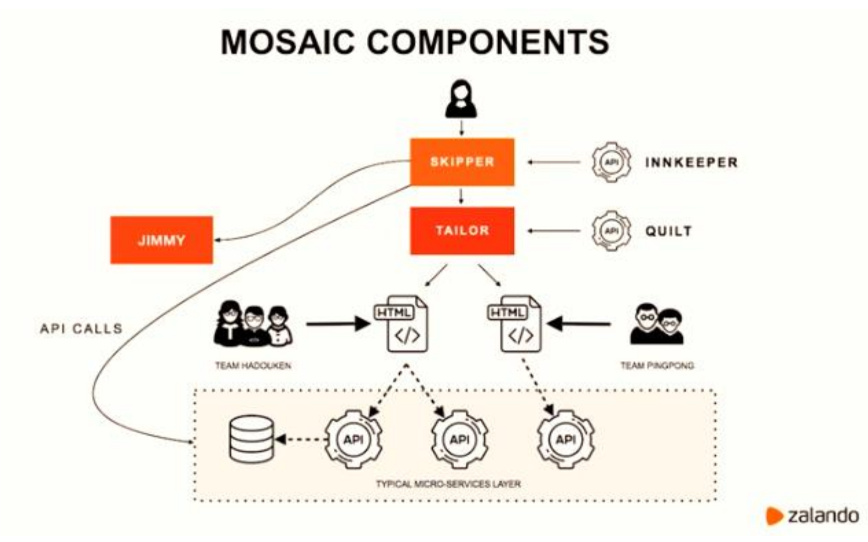


Figure 2 - Zalando architechure. Source: (CodeTalks, 2017)

The assembler content is then streamed to the client using a router (*Skipper*). This is the entry point from every request from the browser, excluding the static resources. It works as a reverse proxy and delegates the request based on an internal route to a microservice or a legacy system – *Jimmy* - in case of migration.

As for the layout service, *Taylor*, a service that uses streams to be able to compose a page from fragments, then loaded in parallel from the template. Also provides error handling capabilities and fallback features.
This content is assembled on the Server-side for SEO and performance reasons.
Mosaic is live and it is stated to already have traffic dimensions of 20.000 requests per second on *skipper* in production on a Black Friday day.

2.1.6 Dazn

DAZN needed a new architecture to comply with hundreds of developers across the world working at the same codebase (Mezzalira, 2019). Initially, having started with a database shared across multiple APIs, this API layer was a monolith containing all the services and integration with third-party companies, connecting to a Single Page Application on the clientside. As a startup company, it was crucial for DAZN was to get feedback from potential customers.
As the company grew, they saw the need for breaking apart the database and the monolith backend into a microservices architecture, where each communicates with a single database. The result of this migration for the teams was the use of the same codebase at the frontend, communication overhead in managing different parts of the UI and microservices, meaning the frontend stood like a monolith Single Page Application.
They started to use the same principles of microservices, such as Domain-Driven Design, and applied it to the frontend as well. The impact to the teams with this new paradigm was End-to-End autonomy with the business domain, freedom, responsibility, and innovation without affecting the whole application. The main purpose of the use of the micro frontends was also to scale teams by using cross-functional teams who handle the frontend, backend, and infrastructure layer in full autonomy, respecting the contracts between the rest of the teams and the possibility of using different technology stacks.
According to Luca Mezzalira (Mezzalira, 2019), chief architect at DAZN, another key concept of micro frontends is sharing nothing. Usually, in frontend development, it is common to create Component libraries to share components between the features. This can make the code more complex to maintain in long term and should be avoided, communication overhead can be the bottleneck in organizations.
DAZN's approach on micro frontends was to start to break the domain into several subdomains such as “Authentication”, “My Account”, “Customer Support” and others. Each of these

subdomains is a different single-page application project and some have a different technologic stack. To avoid scalability issues and to be customized for the user, they created a layer called Bootstrap, which is the first layer downloaded when entering the DAZN website.

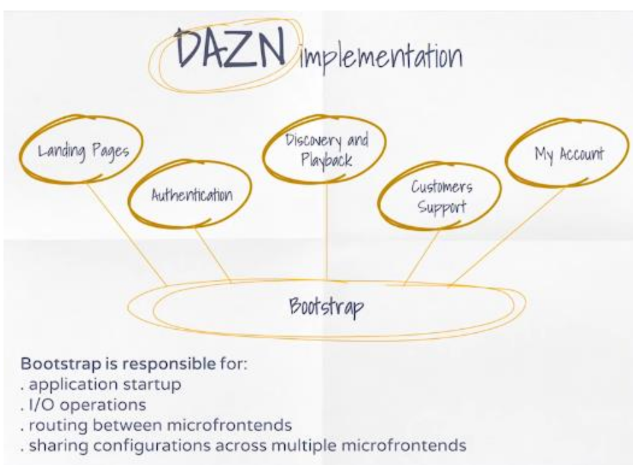


Figure 3 - Sketch of the architecture at DAZN. Source: (Mezzalira, 2019)

This architecture allows DAZN to deploy the chosen micro frontends in specific countries and have an independent deployment for each, also the teams can make their own decisions without having an impact on others since there are minimal dependencies.

They were able to onboard 5 teams in the same projects in 3 weeks and after the first week they were already contributing to the codebase, since with easy-to-understand small subdomains and independence, they did not have to acknowledge a huge architecture, all it mattered was the features. DAZN also has over 100 developers working on the same project.

The bootstrap component used in Figure 3 is an internal project, but two frameworks that do a similar job are Single-SPA and FrintJS. Mezzalira mentioned Single-SPA as a very similar tool that he discovered about 1 year after they started working in the new architecture.

2.1.7 Ikea

Gustaf Nilsson Kotte, (Stenberg, 2018) (Kotte, 2017) web architect for IKEA, needed an architecture based on 3 main principles: performance, Scalable development, and evolvability. Kotte started to find the same problems with the frontend monoliths they had with the backend before dividing the architecture into smaller services. By using micro frontends, the frontend is

divided into smaller parts to allow teams to develop autonomously, so they can deliver frontends continuously.

By dividing the system vertically into self-contained systems, with the backend and frontend owned by the same team, teams up to 10-12 people do an efficient job of delivering value to the final user, although these feature teams can have their problems such as collaboration or component reuse. A team can also work in more than one service if they are small enough.

The fact that microservices are a heterogeneous architecture allowing different technologic stacks should also be possible in the frontend, and while frontend technology is in constant change, Kotte has observed large rewrite frontend projects with high costs, and a 2 to 4-year rewriting cycle is not optimal for any business. A diverse technology stack can be the answer to this problem.

The solution found by Kotte to be able to use several different frameworks and not tie to one single framework is the use of Transclusion on the server-side with the addition of Edge-side Includes, a technique based on containing a source attribute and a URL pointing to a resource to be included in the final document. Server-side Transclusion became fundamental to IKEA's micro-frontend architecture. Each team is responsible for several pages and fragments, which pages include ESI references for several fragments and these fragments can be used by other different teams to guarantee a consistent user interface. The fragments can be developed using different frameworks and the teams used a technique named self-contained fragments to allow CSS and JavaScript to be included inside the fragments so they can fit together. Meaning no team needs to bother with dependencies, and just include ESI for the style or the scripts. Kotte states this technique brings complexity to the server-side, but it is then simple for each team to develop its work.

2.1.8 Facebook

The company Facebook, popular for its social Network platform, needed to increase performance in its website (Facebook.com, 2010). The traditional dynamic web page serving system was outdated, so due to several technical achievements, Facebook was able to accomplish its objective with a major emphasis on BigPipe. The idea behind this technology is to decompose the web pages into small fragments called pagelets which go through several steps of a pipeline on the server-side. The web server assembles the pagelets into HTML which is then transferred to the browser. The browser also downloads the required CSS styles for the

page, does the DOM Tree construction, associates the CSS style rules and the related JavaScript code. The final step is the execution of this code by the browser and the page is finally presented. The result is a full web page composed of the different fragments represented by each light blue rectangle in Figure 4.

Figure 4 - Facebook page composed of different *pagelets.* Source: (Facebook.com, 2010)

This implementation is similar to the Server-Side Composition pattern in section 2.2.2 and helped Facebook increased the performance of the website to become about twice as fast.

2.1.9 Hello Fresh

Hello Fresh[1] is a company based on Meal Kit Delivery Service containing a website. As stated by the company's take on Micro Frontend described by Pepijn Senders in Hello Fresh's Blog (Senders, 2017), started initially with an idea based on iFrames by a company's developer which later on, a solution would be presented based on this initial idea. The company started by defining its own concept for frontend microservices:

> "*A front-end microservice listens on one or more HTTP hooks (endpoints) and serves the page HTML and links to scoped CSS and JavaScript.*

The technology should be free to choose; there should be no external critical dependencies. This way we can serve React apps on one page, plain HTML on the other, and maybe a Vue app on another page."

Some additional requirements were also necessary such as an isolated development cycle, run in an isolated environment, preference on HTML server-side rendering, and priority on a fast response time.

Regarding architecture, the component nomenclature was defined:

- *Fragments* or *florp* - a server responsible for serving the entire page, non-legacy fragments are usually based on isomorphic React applications served by an Express [1] Server.
- *Particle* or *glorp* - shared parts of the page that are intended to be loaded synchronously – intended to be applications shared across the platform and rendered on server-side, such as the header or footer of the page. The particle returns a JSON payload containing the HTML, CSS, state of the application, and the link to the script file allowing it to be mounted on a parent application.
- *Tag* - shared parts of the page to be loaded asynchronously. An example provided is a modal to be loaded asynchronously on a button press, this loading is deferred[2]. Tags provide a window function to be the interface.

For the connection between the Frontend microservices to work, an NGINX server routes traffic via consul[3] to facilitate the launch and discovery of the new servers. The errors are also intercepted in the process and proxied to other frontend microservices to serve the error page.

Figure 5 shows how the components work together.

[1] https://expressjs.com/

[2] https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/deferred-versus-immediate-loading

[3] https://www.consul.io/

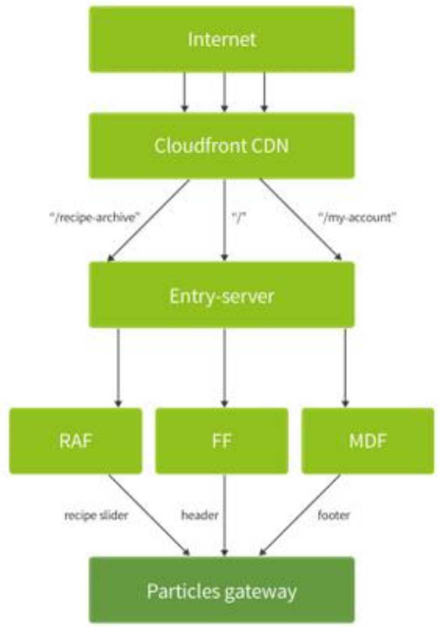


Figure 5 - Demo view of the Hello Fresh infrastructure. Source: (Senders, 2017)

The clients send a request for a page, the CDN checks if a recent copy is cached to be instantly delivered, if not, the request will be redirected to the entry servers and forward to one of the following fragments presented in the figure (example fragments):

- *Recipe Fragment* (RAF).
- *Funnel Fragment* (FF).
- *My Delivery Fragment* (MDF).

These fragments will request particles from the particle gateway to fully render the page.
Since fragments are usually cached in the CDN, the performance is fast.
After a year of experiencing this architecture, the result was:

- Regarding development speed, it was slow in the initial phase due to a higher learning curve since it was mandatory to own the project end-to-end. Passing this phase, the development speed increased as the new HelloFresh shop took 4 weeks to develop by 3 people.
- Code Isolation, although it is possible to develop and integrate projects in numerous JavaScript frameworks such as Vue, React, or Angular, it became a common practice to use React. Yet, it was considered a major benefit to being able to have this flexibility.

- Error Tracing - Logging is said to be "consistent and unified" and the speed increased in triaging problems, so it was a major advantage as well.
- Independent development was the main benefit of this migration since every project can be run on its server and in isolation. Also, another important point is the fact that external dependencies can be configured and be integrated with the production or staging environment to test or debug problems for instance.

2.1.10 Summary

After the study of the several use cases presented in this section, similarities were found in the means that most companies adopting a micro frontend architecture have opted to use a server-side composition approach, with some of the companies, such as Ikea, with the particularity of using Edge-side composition additionally. With the constant effort put on evolving the micro frontend architecture and according to some slightly different requirements, companies like DAZN are focusing on a Client-Side Composition approach expecting to bring the advantages of microservices and Single Page Applications to the frontend side.
In terms of benefits, the following were mainly found across most of the use cases:

- Increased Autonomy of teams translated into faster onboardings and development.
- Increased Scaling of teams and codebase
- Overall loose coupling in codebases allowing independent release cycles, faster integration, and deployments.
- Increased testability of the solution.
- Technology agnosticism, examples like Hello Fresh can develop micro frontends in different JavaScript Frameworks, although usually reducing the number of technologies used.

Nevertheless, some difficulties have been found in maintaining the User Interface consistent across all the modules and with existing duplicated code.

2.2 Patterns

This section examines the main composition patterns of micro frontends to allow the integration of the UI modules on a single page.

2.2.1 Client-Side Composition

Client-Side composition consists of assembling in parallel and updating the page directly in the browser. This approach allows for more reactive websites, similar to monolith single-page applications, where the page reacts almost immediately to the user input (Geers, 2020). By default, this approach consists of an integration component downloading the bundles from different deployed components and assembling them on the same page in runtime, such as in Figure 6.

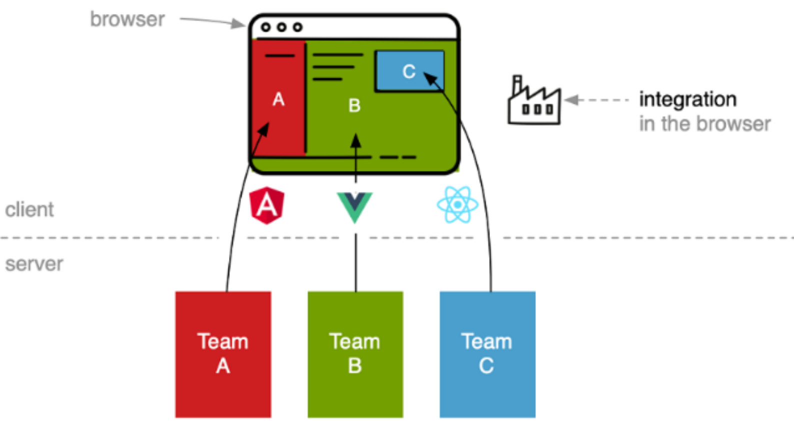


Figure 6 - Client-side Composition example. Source: (Geers, 2020)

2.2.2 Server-side Composition

Server-side composition consists of service on the server-side responsible for assembling pages by using the pre-rendered sources from other applications also placed on the server, which the client will then download and display (Geers, 2020). This service is a render engine located usually between the client-side and the server-side applications, within the server boundary as displayed in Figure 7.

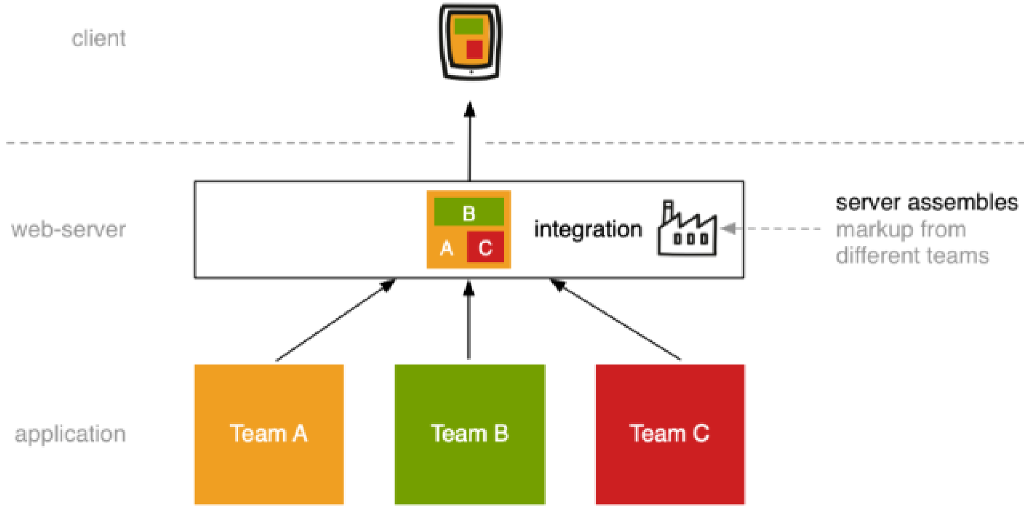


Figure 7 - Architecture example for Server-side Composition. Source: (Geers, 2020)

2.2.3 Edge-side Composition

Edge side composition consists of assembling pages at the edge, a series of CDNs or proxies around the world responsible for caching and delivering the content closer to the user region. It usually relies on a markup language called Edge-side Includes (ESI), a standard by Akamai.

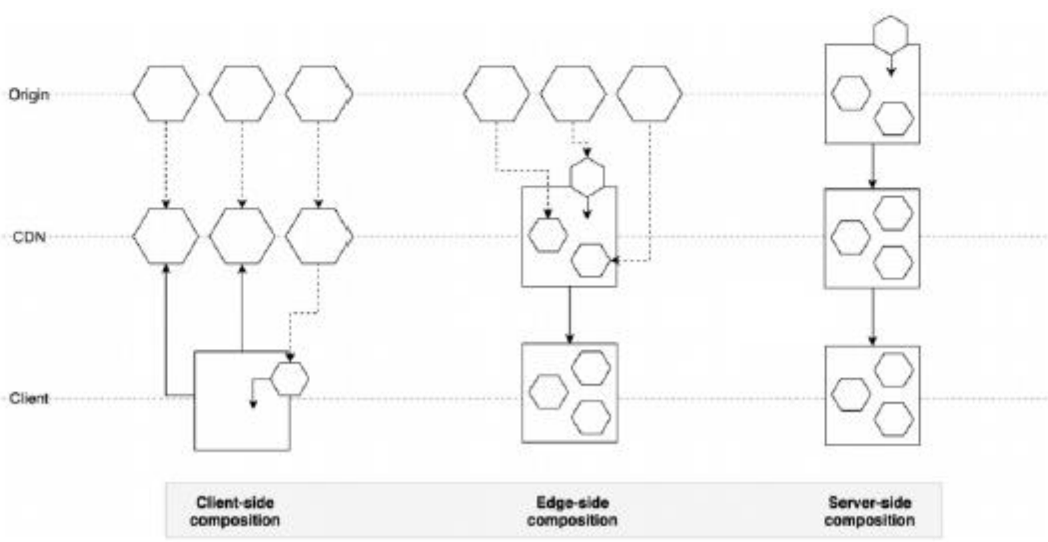


Figure 8 - Main patterns comparison for Micro Frontends. Source: (Peltonen et al., 2020)

2.3 Related Patterns

This section addresses patterns that use is related to micro frontends architecture.

2.3.1 API Gateway

The API Gateway pattern is characterized as being a service that sits between the client and the server, providing a single entry point for the clients. Usually routes client requests to a specific microservice, or retrieves data from multiple microservices on a single client request.
It allows several advantages (Rousos et al., 2021):

- Decreased coupling – Clients will not be coupled to a specific or multiple microservices and be coupled to the gateway instead, becoming easier to replace a microservice reducing risk of breaking changes.
- Decreased number of round trips – If a client needs to fetch data from multiple microservices, only needs to perform a single request to the API Gateway, which will fetch the data from the microservices and retrieve it back to the client.
- Increased Security – Microservices will not be exposed to the network, API gateway will be a single point where concerns such as Authorization and SSL will be performed.

2.3.2 Backends For Frontends (BFF)

Backend for Frontend (BFF) (Newman, 2015) is a pattern in which use is increasingly related to micro frontends. It is characterized as being an API placed on the server-side, that will contain all the logic that once would be in the backend of the web application. It intends to strip all the business logic associated with the frontend component to increase lightness, decrease the bundle size, responsibility and ultimately try to obtain more autonomy for the client applications since there is no need to request changes on the microservices in terms of data, the BFFs will be able to format according to its needs.
In (Plotnicki, 2015), mentioning the use case of the company SoundCloud, the BFF can be owned by the Frontend developers, and a BFF can be developed to assure the needs of a specific use case or feature to act as a facade. BFFs can make the changes on the displayed information less

invasive to the microservices and the client applications where the changes can be done without having to modify both. This granted more autonomy and faster delivery since it supported evolutionary design and less coupling between the layers.

A major advantage of the use of the BFFs is, instead of doing multiple REST requests to different endpoints to fetch the needed data for a singular feature and then merge it in a single object to be shown in the interface, as this would all be done in the backend side of the client web application, the BFFs will allow the aggregation of the requests to the microservices also on the server-side, merging all the data from several different microservices and prepare the object for the views, where the web application would only need a single request to parse the data for the UI to show information to the user. The JSON payload decreases in size since the BFFs will work on the data provided by the microservices and only use what is needed by the client applications. Also, the URI should be more uniform when used in API requests by the client application. Figure 9 shows an example of the BFF pattern on mobile clients, where a BFF is used to receive requests by its related client and perform requests to different microservices.

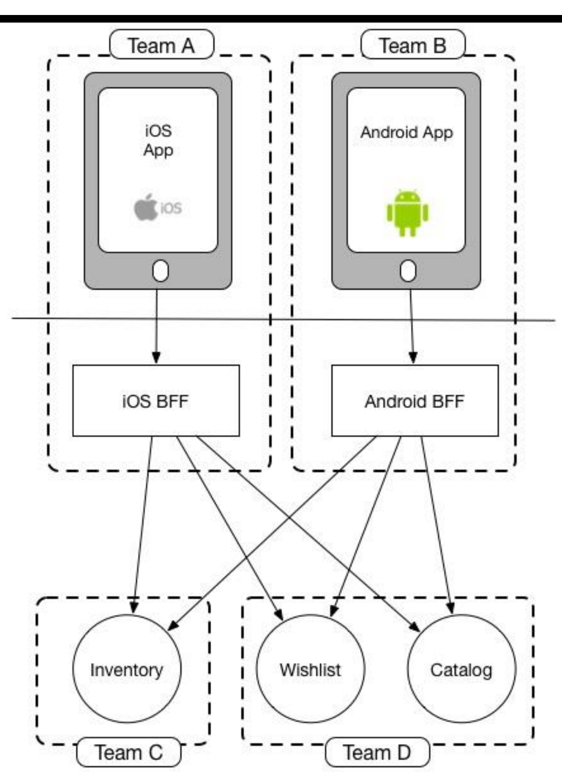


Figure 9 - Example use of BFF. Source: (Newman, 2015)

2.4 Techniques

In this section, techniques for implementing the micro frontend architecture based on the patterns mentioned in section 2.2 will be presented.

2.4.1 Separate Runtime

The techniques presented in this section are related to separate runtime, i.e., applications that run completely independently without sharing resources or events by default.

2.4.1.1 MicroApp Route Distribution

In MicroApp Route Distribution, every team builds their feature as a standalone and isolated micro application that run completely independently from each other and then distributes them by routing between the applications, where each link takes the user to a different micro application from this ecosystem. This is usually implemented using a reverse proxy or by the framework's routing. To emphasize the illusion of delivering one single web application to the user, it's important to have the same style and theme, for that matter usually the use of a centralized CDN to share UI resources is used (Dhiman, 2019). Also, the use of Single Sign-On mechanism is used so that one only needs to log in to a single page or dashboard and share the authentication throughout all pages (Geers, 2020) (Yang et al., 2019).

2.4.1.2 IFrames

Each team develops its feature in an application completely isolated and independent from one another and by making use of the iFrame Html element to embed a page into another, obtaining SPA composition with a nested context while maintaining loose coupling and isolation in the features. (Geers, 2020) This isolation means the applications are completely autonomous and existing in their context without shared resources or memory by default.

2.4.2 Shared Runtime

The techniques presented in this section are related to shared runtime, i.e., components from the application can have shared resources or events by default.

2.4.2.1 Native Web Components

In this pattern, every team develops its feature in the form of web components, isolated from each other. These web components are reusable custom elements with encapsulated functionality that can be imported into the main application by loading the JavaScript bundle and registering the components to be loaded on-demand in the DOM (Yang et al., 2019).
After the integration, the memory and resources are shared between the components, therefore, a shared runtime integration is obtained with the use of the Web Component Specification. It consists of several W3C Standard specifications (Webcomponents.org, n.d.):

- *Custom-Element* - Allows the creation of new HTML tags or extend other elements.
- *Shadow DOM* - Encapsulates the DOM within the component, allowing the encapsulation of the micro frontend itself, its own behavior and style (Oh, Ahn and Kim, 2017).
- *HTML Template* - Defines the declaration of the contents of the application, used by the browser to render the micro frontend components instantiated at runtime.
- *HTML Imports* - Import HTML documents to be used in other HTML documents.
- *ES Module Specification* - Import JavaScript documents in other JavaScript Documents, allows the components to be modular and have an interface defined. They can then be merged into a file in client-side.

2.4.2.2 Framework Based Web Components

Like the Native Web Components approach in the section before, Framework Based Web Components instead relies on a 'contract' established by a framework for the integration of the micro frontends in a web application. This allows the abstraction of integration logic and easily integrates components developed in different JavaScript Frameworks (Dhiman, 2019) (Geers, n.d.).

2.4.2.3 Module Federation

Invented by Zack Jackson and co-authored by Marais Rossouw and Tobias Koppers and implemented in webpack version 5 released in October 2020, Module Federation allows a JavaScript application to load and run code dynamically from another web application, where each application can be in its repository and independently deployed (Jackson, 2020). The imported code can be seemingly used as an ES6 import as if the code were present in the same repository, the only difference is that the module has been remotely loaded, as the code will not need to be wrapped in any way as it would by using a specific framework.

The nomenclature (Jackson, 2020) used in this approach is the following:

- *Host* - An independent web application that will dynamically consume a module from another.
- *Remote* - An independent web application consumed by the host.
- *Bidirectional host* - When a web application acts as a host and a remote at the same time.
- *Omnidirectional hosts* - As the concept of host and remotes is not known by the application itself on startup, webpack is able to change the vendors using semantic versioning allowing multiple versions.

Module federation is implemented simply by configuring a webpack plugin in the project's *webpack.config* file as the example in Code 1:

```
new ModuleFederationPlugin ({
name: "home",
filename: "remoteEntry.js",
remotes: {
nav: "nav@http://localhost:3003/remoteEntry.js",
},
exposes: {
"./ProductCarousel": "./src/ProductCarousel"
},
shared: [‘react’, ‘react-dom’]
}),
```

Code 1 - Webpack configuration example for federated modules. Source: (Herrington, 2020a)

The following tags are used:

- *Remotes* - it identifies the code being imported.
- *Exposes* - represents what is being exposed to an external application.
- *Shared* - dependencies to be exported along with the Remotes.

Module federation offers features such as bi-directional share of resources, ability to share anything that can be bundled by web pack (such as CSS, images, code, and libraries such as globalization strings, analytics, etc.), can be used for client-side and server-side rendering, is an optimized solution in terms or performance and scalability comparing with externals and import mapping since it doesn't need centralized dependency in a file. It also provides fallbacks, as if an application consuming a remote does not contain a given dependency, webpack will download from the origin enabling less code duplication (Herrington, 2020c) (Jackson, 2020).

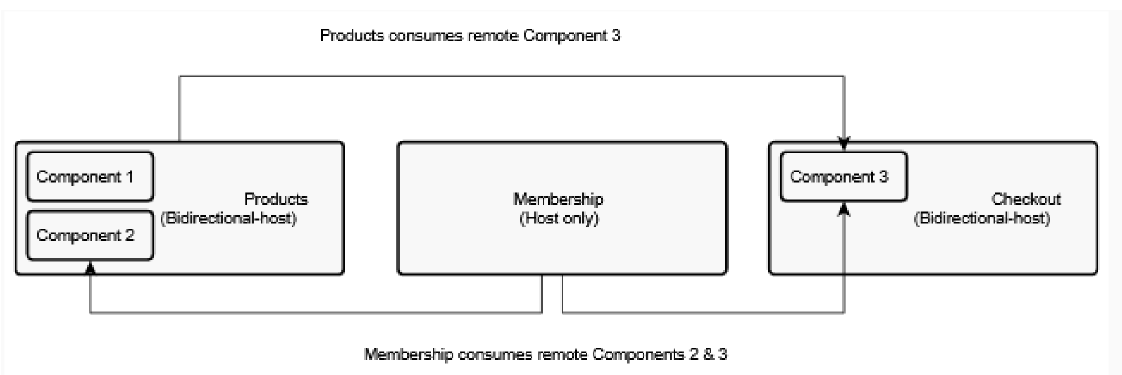


Figure 10 - Example of Bidirectional hosts. Source: (Mrowetz, 2020)

Module federation itself cannot be used to integrate agnostic components developed in different JavaScript Framework - there is the possibility to use ReactJS + Vanilla JS for instance, but not ReactJS + VueJS - this can be possible by integrating with a micro frontend framework or by using native web components (in this case, the problem regarding the bundle size might be present). This is because Module Federation is a new approach to loading modules dynamically from remote hosts (acting as a transport layer), similar to SystemJS and Import Maps widely used in the Single-SPA framework, but it does not rely on any additional frameworks to do so (Harrington, 2020b).

For State sharing, this approach can also use a global approach such as custom events, but it is also possible to load the components as they are implemented in the remote application and it should be working (Harrington, 2020b).

In terms of dependency versioning, Module Federation also addresses, as it always attempts to provide the higher compatible version if possible or download both incompatible versions to be used as a fallback. In case of completely incompatible versions, there is the possibility to fail fast and throw an exception or accept a version range upon configuration (Manfred, 2020).

2.4.2.4 Server-side Transclusion

The Server-side transclusion technique is based on the Server-Side composition pattern.

Consists of an engine on the server-side responsible for assembling the page by composing several different fragments. After the page is pre-rendered, is then sent to the client browser, so the page is already assembled before reaching the client-side. It is a good approach in terms of performance since it depends on the capabilities of the server to assembly the page and only needs the client capabilities for the download of that same page, there is no need to rely on the client's JavaScript (Dhiman, 2019). An example of aggregation style can be Server-Side Includes (SSI), configured in the origin server to allow the composition.

2.4.2.5 Edge-side Transclusion

Edge-side Transclusion is a technique to apply the Edge-side Composition pattern. An example of a web standard for this approach is the Edge-side Includes (ESI) (Dhiman, 2019). This consists of include tags in the HTML pages to get content from different origin servers, which can have their web applications. The content of these applications is then be assembled at the main page by an ESI processor/engine at the CDN level, which picks all include tags and fetches their content, an example can be seen in Figure 11.

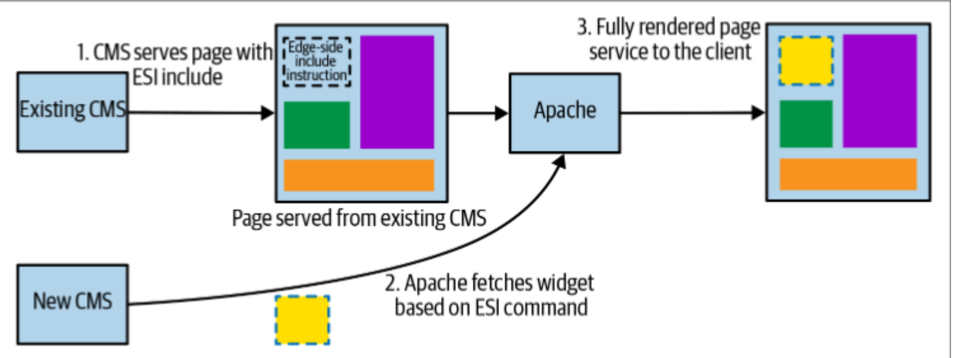


Figure 11 - ESI application example using Apache server. Source: (Newman, 2019)

By using cache and placing CDNs next to the user to deliver content, can improve performance by horizontally Scaling in multiple points around the world and decreasing the load of the origin server. ESI also contains additional features such as using variables, conditions, and error handling.

The drawback of this approach is that each CDN can have its implementation, a multi-CDN approach can increase complexity. Additionally, using a managed service has limitations, for instance, in response size or the number of resources.

This approach is seen as having main benefits when delivering static pages such as news websites or catalogs (Mezzalira, 2020).

2.5 Frameworks

In this section, Frameworks for implementing web applications with a micro frontend-based architecture are presented.

2.5.1 Single-SPA

This framework implements framework-based web components pattern and it is mentioned and recommended by many authors (Mezzalira, 2019) (Dhiman, 2019) of micro-frontend material to easily implement this architecture in a very efficient way, that will be covered next.

Single-SPA is a framework created by Canopy-Tax, it's essentially a top-level router, it acts as an interface contract which the micro frontends will follow (similar to the web component spec

specified in the patterns sub-section) so that the app shell (or bootstrap layer, a tiny application responsible for loading the application code and routing between them) to load on-demand the applications' bundles using lazy loading. Note that the micro-frontend applications can be built in any framework such as Angular or VueJS, Single-SPA is currently compatible with the main used vendors available. (Geers, 2020)

This app shell is where single-spa javascript code exists, contains an HTML page as the stitching layer and the code responsible for registering the applications. The registry is done by specifying only 3 parameters such as *name*, which must be different for all the applications, and will allow single-spa to instantiate the applications and place them in the DOM, so there are no style conflicts, for instance, since everything is self-contained under the registry that was done. To control the lifecycle of the applications, single-spa uses the methods *mount(), unmount(),* and *bootstrap* (in contrast to the custom elements' *constructor, connectedCallback* and *disconnectedCallback* methods), where it mounts a given application when it's to be used and unmounts when it's no longer needed. These methods are asynchronous, and together with the lazy loading, it's a great advantage over the native web component spec, where for this to be possible needs extra effort from the development team since it's not *out of the box*. These two advantages help single-SPA to improve performance. (Geers, 2020)

To help Single-SPA do its job more efficiently, it uses a library called SystemJS (also by Canopy-Tax) a module loader that sits in a layer underneath, with the function of loading the component scripts to the page, so that single-SPA can instantiate them.

Example code for the app shell to compose a webpage using multiple components using System JS, is shown in Code 2.

```
<script type="systemjs-importmap">
{
"imports": {
"app1": "http://localhost:8081/app.js",
"app2": "http://localhost:8082/app.js",
"single-spa":"https://cdnjs.cloudflare.com/ajax/libs/single-
spa/4.3.7/system/single-spa.min.js",
"vue": "https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js",
"vue-router":
"https://cdn.jsdelivr.net/npm/vue-router@3.0.7/dist/vue-router.min.js"
 }
}
</script>
singleSpa.registerApplication(
  'app1',
 () => System.import('app1'),
 location => location.pathname.startsWith('/app1')
)
singleSpa.registerApplication(
'app2',
() => System.import('app2'),
   location => location.pathname.startsWith('/app2')
)
```

Code 2 - Example of composing a page using two different Single Page Applications.

In Code 2, the Single-SPA framework is loading app1 and app2 deployed in *http://localhost:8081* and *http://localhost:8082* respectively, and downloading the bundles (*app.js*) which contain all the information about the application. The bundles are created with Webpack[4]. Then, by using *singleSpa.registerApplication()* the applications can be imported into the main application's page and used as components.

[4] Webpack: https://webpack.js.org/

2.5.2 PuzzleJS

PuzzleJS implements server-side transclusion pattern and uses the concepts of gateways and storefront illusive to components and app shell respective. The gateways can be composed into separate fragments and the storefront will be responsible for creating the pages composed by the gateways which are declared using a configuration file. The gateways expose the fragments and all their information working as an API for the storefront project.

The way it works on compile-time, the storefront fetches the data from the gateways previously registered in the configuration file, where all the fragments and other assets will be cached. The HTML template is compiled as JavaScript functions in-memory. When the browser requests the server, the storefront is responsible for sending a response with the compiled function as a template containing static content and placeholders while requesting the gateways for the rendered content. After the response from the gateways, the storefront will then replace the placeholder with the fragment content and finish the page. The connection between the browser and the server is only terminated when this process finishes (PuzzleJS, 2020).

PuzzleJS (PuzzleJS, 2020) claims to provide the following features:

- Quick First Time To Byte.
- Works well with Search Engine Optimization (SEO).
- Extensibility for custom functions due to good performance.
- Easy to use and configure.
- Technology agnostic.
- Scalable since the modules are independent.
- Fail-Safe when a fragment is down.

2.5.3 Tailor

The framework by Zalando as specified in the Use Case subsection, Tailor, implements micro frontends with server-side transclusion, and is part of their ecosystem – Project Mosaic, and is open source.

Tailor uses the HTML of the pages as fragment-tags to fetch its content, where the content is replacing the tags to compose the final page. These tags contain parameters such as *scr* to specify the source of the content (the applications deployed in a web server), *timeouts,* and

fallback content, which will be used instead of the original *src* URL in case of error or timeout. It also contains the style and JavaScript scripts associated with the page in the header, Tailor will use this header to add these scripts and sources to the page. The developer does not control this association so this can be a downside (Geers, 2020).
Tailor sends a page template to the client as data where the fragments will be loaded then, this is a benefit as will increase the time to the first byte as they are loaded in parallel.
The team behind Tailor is currently working on a replacement (Colin, 2018) based on the fragments approach called "Interface Framework" (that will be composed by a rendering engine (orchestrator), renderers (the fragments of views), recommendation system (responsible to decide which render to show in a specific point) and a Fashion Store API (an aggregation layer using GraphQL[5] – a query language for APIs).

2.5.4 Podium

Podium is a framework that uses server-side transclusion to implement the micro frontends, similar to Taylor. It uses the terms *podlets* and *layout* to identify fragments and pages respectively and essentially ensembles the pages using *podlets* on the server-side, but unlike ESI, it's done inside the node.js application that implements podium, responsible for being the stitching layer and serve the pages to the client, making Podium a simpler solution than ESI.
For this approach to work, a *podlet* is developed by each team using the *podlet library* and generates the *podlet manifest*, a JSON file containing information such as the name, version, endpoint URL, fallback content, javascript, and CSS assets. This manifest file acts as a contract between the podlet itself and the integrator application.
On the side of the integrator app, that uses the *layout library* responsible for assembling the page by composing the needed fragments/*podlets* into one final page and deliver to the client-side, it can use cache to make this process better.
Podium can use the web server the developer wants as it doesn't impose any and it's an easy solution to configure unlike other ways of implementing server-side transclusion.
As to fallbacks, they are provided in the manifest file as stated, so each team is responsible for developing their own. (Geers, 2020)

[5] graphQL: https://graphql.org/

This is a solid solution and can obtain good first load performance since the server is responsible for the formation of the page.

2.5.5 Piral

Piral is a framework that applies micro frontends using the Client-Side Composition pattern by default but can also apply the Server-side Composition.
The web application is abstracted as a piral instance and is a layer on top of piral core (a subset of the full piral framework) and the modules/components that are named pillets that will be assembled on the main page. Piral primary API is based on React JS, so browser compatibility is according to this framework. The pillet contains the contract needed for the piral instance to fetch and show in the browser. The full piral npm package contains the full framework that abstracts all the needed functionalities, but there's also the possibility to use only the Piral Core for simplicity. In terms of compatibility, Piral works with the most popular browsers if updated to recent versions and has the following features (Piral, n.d.):

- Modularity.
- Framework agnostic.
- Integrated translation system.
- Supports asset bundling.
- Contains global state management.
- Independent development and deployment.

2.5.6 Luigi

Luigi is an open-source framework developed by SAP that applies micro frontends using client-side composition. Developed in Svelte, it provides a way to easily develop a web application with features such as Standardized UI, Responsive Design, Authorization, Notification Management, multi-language support, and technology agnostic, enabling the development of micro frontends in different frameworks, such as React or AngularJS (Luigi-project.io, n.d.).
The nomenclature of the components in the Luigi framework are the following (Simeonova, 2020):

- Luigi Core - Main Application in which micro frontends are displayed.
- Luigi Client - Client API from which the Micro Frontends are configurable.
- Parameters - Elements used in the configuration of the Luigi Application.
- Navigation nodes - Links of the side navigation.
- Contexts - Luigi Parameters used to transfer objects to a Luigi Micro frontend.
- Views - Luigi's Micro Frontends.

2.6 Team Organization

This section approaches different types of team organization to understand how it would fit on a micro-frontend-based project.

2.6.1 Component Team

A form of team organization still very common in today's companies, component teams are Agile Team that focuses on specific components of a system. All these components together are delivered in their final form as the product. Each team is assigned to a component with a focus on reliability, separation of concerns, re-use, and testability. A separate component is irrelevant for the end-user in terms of value, and this is one of the main disadvantages, the value flow is slow since it needs to be integrated with the rest of the components so that value is delivered. This leads to dependencies that require cooperation across teams to build, deploy and release. Teams also waste too much time on integration testing and with dependencies with other teams (Visual-paradigm.com, 2020) (Gidlund, 2016).

Since each team owns a component, usually we have a team of backend developers, one team of frontend developers, one team for database, one team of testers, and so on, such as Figure 12. This means if there is a problem or need for a change that implies crosscutting between several different components, several teams must organize and synchronize between themselves to fix such issue and successfully deploy the solution.

Figure 12 - Component Teams. Source: (Gidlund, 2016)

2.6.2 Feature Team

An approach to team organization that is gaining popularity is feature teams or cross functional teams. This organization focuses on features instead of a technology stack as the previous team organization, a team who owns a feature end-to-end and is responsible for the infrastructure, backend, fronted, and tests of this given feature. This is optimized for continuous delivery since it favors the incremental delivery of value to the end-user, now that one single team is completely independent and must only rely on itself to integrate all the layers of a feature in a product, unlike component teams that only are responsible for a layer of that specific feature. This also means, from an agile point of view, their user stories should not have dependencies with other teams in delivering value. These user stories should be broken into small pieces that are valuable and can be demonstrated to the end-user.

Feature teams are also capable of daily build and this can decrease lead time, increase productivity and quality due to the teams having full knowledge of their domain and more efficient coordination (Gidlund, 2016).

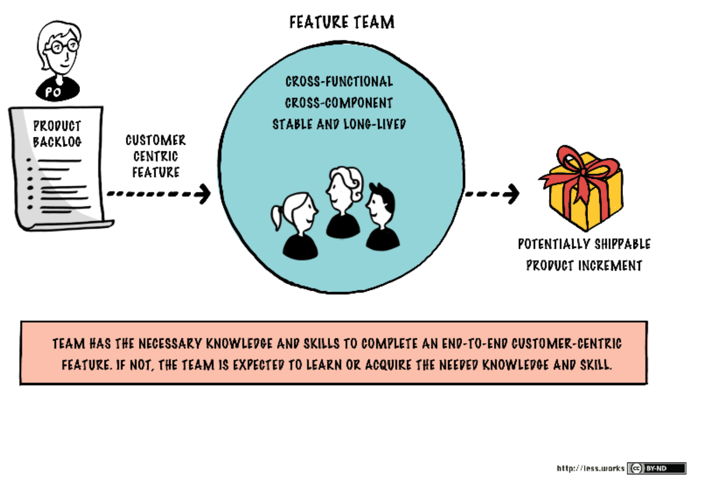


Figure 13 - Feature Teams. Source: (Visual-paradigm.com, 2020)

The ideal feature teams has the following attributes (less.works, n.d.):

- Long-lived team, i.e., the team remains the same or partially the same for increased performance and to take on new features over time.
- Cross-functional/ cross-component.
- Co-located, i.e., the end-to-end team should be together.
- Work on an end-to-end customer-centric feature, across all components and areas (infrastructure, development, testing, etc.).
- About 7 members.

2.6.3 Conclusion

The use of feature teams copes well with the use of a micro-frontend architecture. A single team capable of owning a feature end-to-end and customer-centric, with developers for the frontend micro components, developers for the backend side and infrastructure and, in an agile mindset, the inclusion of a scrum master and a product owner will allow complete autonomy of the team, decreasing dependencies and increase the pace to deliver value. According to (Gidlund, 2016) the team holds the necessary expertise to deliver a product that is ready to be deployed in production in each sprint by developing and testing the same feature. As companies

work in a fast-changing market, they need fast time to market and adaptation to change, a cross-functional team is highly beneficial in these scenarios (Gidlund, 2016).

Table 1 compares the traditional component teams and feature teams.

Table 1 - Comparison between team organizations. Source: (Less.works, n.d.)

component team	feature team
optimized for delivering the maximum number of lines of code	optimized for delivering the maximum customer value
focus on increased individual productivity by implementing 'easy' lower-value features	focus on high-value features and system productivity (value throughput)
responsible for only part of a customer-centric feature	responsible for complete customer-centric feature
traditional way of organizing teams — follows Conway's law	'modern' way of organizing teams — avoids Conway's law
leads to 'invented' work and a forever-growing organization	leads to customer focus, visibility, and smaller organizations
dependencies between teams leads to additional planning	minimizes dependencies between teams to increase flexibility
focus on single specialization	focus on multiple specializations
individual/team code ownership	shared product code ownership
clear individual responsibilities	shared team responsibilities
results in 'waterfall' development	supports iterative development
exploits existing expertise; lower level of learning new skills	exploits flexibility; continuous and broad learning
works with sloppy engineering practices—effects are localized	requires skilled engineering practices—effects are broadly visible
contrary to belief, often leads to low-quality code in component	provides a motivation to make code easy to maintain and test
seemingly easy to implement	seemingly difficult to implement

According to (Larman and Vodde, 2009), feature teams have the following advantages:

- Increased value throughput - increased delivery of user-centric value.
- increased learning - team contains specialists in all areas, and everyone will have increased responsibilities be in the same location, consequently, increase the learning of each member in different areas of expertise.
- simplified planning - easier coordination since the same team works in a feature end-to-end.
- reduced waste of handoff - the feature works on all areas (i.e., UI/UX, code, testing, etc.).

- decreased waiting means faster cycle time - a feature does not require multiple teams working in coordination for value delivery, that means fewer dependencies and faster delivery.
- self-managing, improved cost, and efficiency - as (Gidlund, 2016) states “feature teams do not require project managers or matrix management for feature delivery (…) Data shows an inverse relationship between the number of managers and development productivity, and also that teams with both an internal and external focus are more likely to be successful”. Also, feature teams, since they are lightweight and without the need of additional stakeholders such as project managers, become less expensive.
- better code quality - a smaller team with more knowledge and context of their feature will pay more attention to code quality and following the standards imposed by the organization and with increased testability.
- increased motivation - according to (Larman and Vodde, 2009), if a team feels they have completed an end-to-end task focused on customer value and they have the possibility to deliver that value after its completion, there’s higher motivation.
- decreased team dependencies - the team operates the whole stack of their feature so there is no need to ask another team to do a change in another interface.
- flexibility in change- change becomes flexible since there is only the need to coordinate one single team.

Additionally, features team is beneficial in terms of onboardings of new teams or team member(s) of an existing team, since there’s no need to learn about the whole domain of the organization and to adapt to all the dependencies, the developer only has to focus in its technology of expertise (which will probably be the same as the rest of the team, not needing to be the same as the other teams) and to focus on the sub-domain of his feature. This will eventually lead to decreasing costs for a company.

2.7 Quality attributes

The purpose of this dissertation is to analyze a micro frontend approach based on a series of quality attributes standards.

The standard is ISO 20510, a standard of the series of ISO/IEC 25000 which primary goal is the creation of a framework for the evaluation of software quality, and it refers to the quality model division – the quality characteristics considered when evaluating software. (ISO25000, 2020)

From previous sub-sections, where an overview of the current state of the art regarding the micro frontend was studied, it was concluded that most of the benefits of the approach are mainly observed in the Maintainability and Scalability attributes.

Performance Efficiency will be studied as well. Although not being a benefit of a micro-frontend approach, is seen as a critical attribute as it will impact several areas such as perceived user experience (MDN, 2021). It is important to understand if there is any performance degradation and to what degree when using such an approach.

2.7.1 Maintainability

Maintainability is the effectiveness and efficiency a product can be modified to further improvement, correction, or adaptation to changes in environment or requirements (ISO25000, 2020). Within this quality attribute, the sub-attributes of focus are the following:

- Modularity – How a system is composed of discrete components so that a change in each component has minimal impact on others (ISO25000, 2020).
- Modifiability – How easy a system can be modified to changes in its environment or requirements without introducing defects or deteriorate quality (Bengtsson et al., 2005).
- Testability – Establishment of test criteria for a system and how tests can be performed to meet these criteria (ISO25000, 2020). This can be achieved with metrics such as test scenario coverage, number of test cases, or time duration for instance (ProfessionalQA, 2020).

2.7.2 Performance Efficiency

Performance Efficiency - represents the degree of performance relative to the number of resources needed under stated conditions (ISO25000, 2020) and impacts major areas such as (MDN, 2021) overall load time, perceived performance, or smoothness and interactivity for instance. The sub-attributes from Performance Efficiency that will be a target of study in this dissertation are the following:

- Time behavior – Response, processing times, and throughput rates needed for a product or system to perform its functions (ISO25000, 2020).
- Resource Utilization – Number of resources used by a product or system to perform its functions (ISO25000, 2020).

2.7.3 Scalability

Scalability is characterized as the system's ability to handle an increase in load (Gan et al., 2005) by changing in size when used above its capacities. This attribute will be assessed to analyze how the micro frontend approaches will behave when attempting to horizontally scale.

3 Value Analysis

This chapter refers to the value analysis, starting with a section about business process and innovation where the NCD model will be applied, following with the customer value, the value proposition with the inclusion of the canvas business model, and an application of the Analytic Hierarchy Process (AHP) for multicriteria decision to aid choosing the best micro frontend pattern approach to focus.

3.1 Business Process & Innovation

The innovation process is divided into three parts: *Fuzzy Front End (*or *the Frontend of Innovation*), *New Product Development* (NPD), and *Commercialization* (Belliveau, et al.,2002). Fuzzy Front End is seen as one of the best opportunities for improvement in the whole innovation process, as the choices made in this part will impact the subsequent parts, so it is receiving increased attention to increase the value and quality of its outputs. Yet, it is seen as an experimental part of the process of innovation where it reigns ambiguity and can be unpredictable. It is regarded as a process of three main sequential steps, the first step being the pre-work and its where new opportunities are found, the second step is the scoping stage where are made marketing evaluations and the third and final step is where a business case it is accomplished (Koen, et al, 2014).

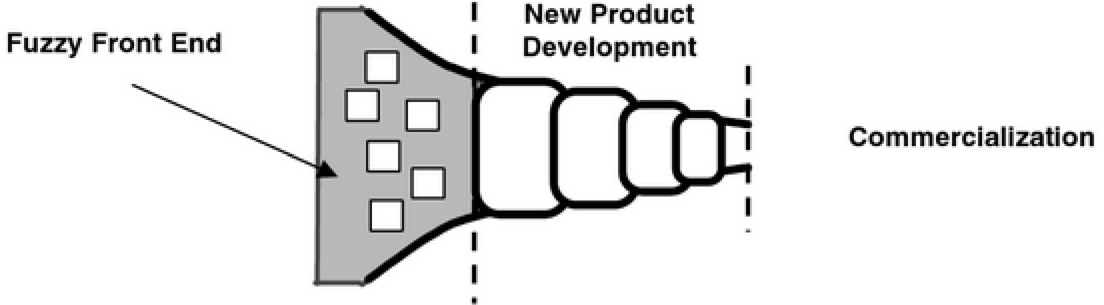


Figure 14 - Diagram of the innovation process. Source: (Belliveau, et al.,2002)

As an approach to reduce ambiguity (or Fuzziness), the essential steps and attributes that had a major role in the Frontend of the innovation process were identified.
According to (Koen, et al, 2014), New Concept Development Model is a holistic framework that identifies "the most effective practices in managing the front end of innovation".

The NCD model is divided into three areas (Koen, et al, 2014):

- *Engine* – (center) contains the organizational attributes that drive the process such as vision and strategy.
- *Wheel* – (Inner) contains the five activities of the Front End: *Opportunity Identification, Opportunity Analysis, Idea Generation & Enrichment, Ideas Selection,* and *Concept Definition*.
- *Rim* – (exterior) Includes all the external factors of the surroundings that influence the engine and the wheel's activities such as competitors, customers, trends, or regulations.

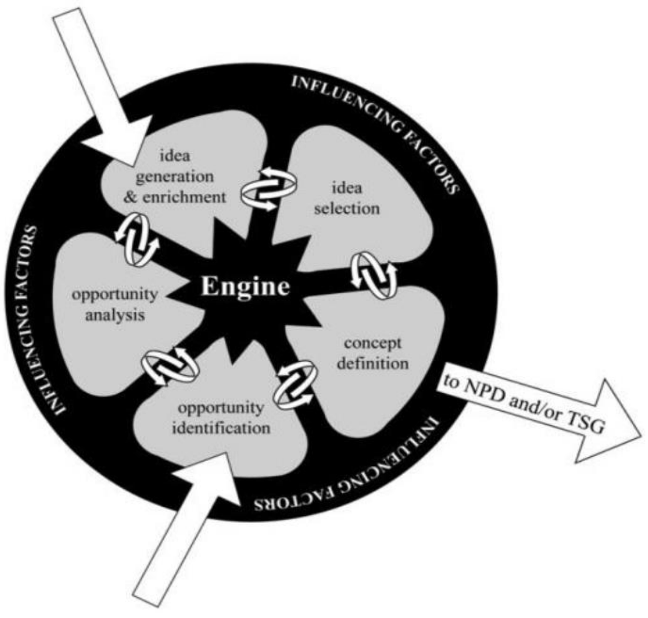


Figure 15 - NCD Model. Source: (Koen, et al, 2014)

This model is pictured as circular to indicate the ideas flow across all the five elements and the arrows pointing inside the *opportunity identification and idea generation & enrichment* phases means that these are the usual starting point for projects and ideas. (Koen, et al, 2014)
Based on this analysis of the NCD model, it is possible to conclude that the idea for this approach was in the *Opportunity Identification* phase.

The opportunity to provide this approach came from the current state of the web projects in the companies to be mostly monoliths.
In the 14th Annual State of Agile Report, where 95% of the over 40,000 respondents stated their companies use Agile Development Methodologies, and the top reasons were (digital.ai, 2020):

- Increase delivery speed.
- Increased Flexibility to change.
- Increase Efficiency in Business Alignment.
- Increase overall quality.

In the Survey State of Microservices 2020 from The Software House (The Software House, 2020), responded by around 650 tech-leads from all over the world, more than 50% of respondents stated to use Microservices for more than a year.

The main benefits of its use were the following:

- Enhanced Maintenance.
- Work Efficiency.
- Increased Scalability.
- Increased Performance.
- Improved Teamwork.

It is possible to conclude that companies all over the world are seeking to increase performance, maintainability, and scalability on their projects using Agile methodologies allied with a microservice approach on the server-side. With the continuous growth and known benefits from a micro-frontend approach as seen in Sections 1 and 2, this is a clear opportunity to introduce this new architectural approach, promising to bring the microservices benefits to the frontend side and tackle the last monolith.

In the *Opportunity Analysis* phase, there was research to understand the current use cases of early adopters of the micro-frontend approach as seen in section 2.1.3. It is possible to conclude about the needs each company had that led them to start with such a new approach and the benefits or drawbacks obtained as a result.

This led to the analysis of the approaches used by other companies, a deeper study on the concepts to understand which quality attributes are in focus with the micro-frontend architecture. The output led to the *Idea generation and Enrichment* phase where different approaches were gathered for posterior analysis and study in the *Idea Selection* phase. This phase consisting of selecting the techniques that fitted the needs of the proof of concept and could increase the productivity and the quality of the value delivered.

The output of this phase led to the *Concept Definition* phase where the pattern and architectural approach was established and ready for the development of the Proof of Concept to prove how this idea works, how it can solve the problems addressed in the monolith frontend approach, and how it compares in terms of the main attributes for quality.

3.2 Customer Value

The meaning of *Value, Perceived Value,* and *Value for the customer* are explained in this section and related with the project to understand the concepts and advance to the value proposition.

- *Value* – essentially defined as needs, desire, interest, beliefs, attitudes, and preferences (Nicola, et al, 2012).
- *Perceived value* – how a customer sees the utility of the product based on perceptions of what is received and what is given (Zeithaml, 1988).
- *Value for the Customer* – can be an ambiguous concept since it is defined based on how a customer interacts with a given product and its experience in fulfilling its needs (Woodroof, 1997).

Value for the customer can be divided in 4 parts (Nicola, et al, 2012):

- *Pre-purchase* – predict how people perceive the value.
- *Point of trade* – quality of the product and associated costs.
- *Post-Purchase* – how the delivered value is seen from the point of view of the customer experience.
- *After-Use* – Reflection about the point of sale.

With these concepts in mind, and relating them to benefits and sacrifices, Table 2 reflects how they can be applied in the context of this project and understand how the customer perceives the value proposition.

Table 2 - Longitudinal perspective of value

	Benefits	Sacrifices
Pre-purchase	Increase efficiency of teamwork. Quality of value delivery. Increased flexibility to change. Possible reduction of associated costs with the project.	Possible cost and time to be spent on studying and implementing a new Micro Frontend approach.
Point of Trade	Quality of teamwork. Quality of the value delivered.	Cost of migration of the current approach to the new.
Post-Purchase	Better time to market on introducing change in the products. Team processes more efficient. Consequent cost reduction.	
After-Use	Satisfaction with the approach. Increased quality and speed of the value. Reduced costs at medium-long terms.	

3.3 Value Proposition

The Value Proposition is how a company presents to its client segments the advantages of a given item of value (i.e., product(s) and/or service(s)) and why the clients choose the company's product as it stands out from the competition (Osterwalder & Pigneur, 2003).
The value proposition that serves as a base for this document is an implementation of a new architecture approach focusing on web development. This Architecture consists of breaking the web frontend monolith into several modules by business domain and features, called Micro Frontends. This new approach provides teams an architecture with autonomy, reducing dependencies to other teams where every feature can be developed and deployed independently, with smaller, easier to maintain codebases and more flexible to change, technology agnosticism, a more robust solution where a problem in one component will not affect other team's component. Also, increased autonomy means less time and consequently

reduced costs in decision-making processes such as meetings. By delivering faster, teams will be able to gather faster feedback from the users and stakeholders and combine different areas of specialization to reduce onboarding time and to increase learning and quality of the delivered value. This will lead to reduced time and increased flexibility in development.

3.4 Canvas Business Model

The Business Model is "a conceptual tool that contains a set of elements and their relationships and allows expressing the business logic of a specific firm" (Osterwalder & Pigneur, 2003).
To easily create the business model, the Canvas Business Model will be used. Canvas Business Model is divided into 9 sections crucial for the creation of the business model. These are the sections to be presented:

- Key Partners: The essential partnerships of the business.
- Key Activities: The main activities the company needs to make the value possible.
- Key Resources: The essential resources to make the value possible.
- Value Propositions: What will deliver the value to the end customer, a product, or service that will them solve a specific problem.
- Customer relationships: The relationship the company expects to maintain with its customers.
- Channels: How the company is reaching the customers.
- Customer Segments: For whom the company will deliver value.
- Revenue Streams: How will the value delivered generate revenue.
- Cost Structure: The related costs.

The model is presented in Figure 16.

Key Partners	Key Activities	Value Propositions	Customer Relationships	Customer Segments
companies, the intervenient that can make this implementation possible. Namely, the CTO, the development team, the Product Owner, the Scrum Master, and the Release manager.	Choosing the right pattern for implementing the solution. Development of the Proof of Concept itself, where this architecture is implemented. **Key Resources** **Human Resources:** Development team, product owner, and Scrum Master, the team that develops the product. **Financial Resources:** Cost of onboarding in a project and cost of migration for existing projects **Physical Resources:** Possible IDEs needed for developing the solutions, Servers for deploying the solutions, and infrastructure.	Team ownership end-to-end of a feature. Increased team performance, autonomy, and scalability. Smaller, easier to maintain codebases. Independent and faster deployments. Increased flexibility to change. Technology agnosticism. Robust Solution. Autonomous Codebases. Reduced time and costs in development. Faster feedback. Faster time to market.	Communication must be constant like e-mail and Microsoft Teams. Also, Confluence for documentation. **Channels** Email, Meetings.	Software companies with a product development focus on web development for user interfaces in big projects that requires multiple teams working on the same codebase.

Cost Structure	Revenue Streams
If a migration is made, companies have possible losses while implementing this new approach to an existent project. Cost for the time needed to onboard a new team in a new project. Cost for the time needed to onboard a new member in an existing team and project.	Product delivered to the end customer.

Figure 16 - Canvas Business Model

3.5 AHP

To aid in the multicriteria decision, a methodology called Analytic Hierarchy Process (AHP) will be used. Since this point is an initial phase of the work being developed, this methodology will be applied in the comparison between the three main micro frontend patterns researched in Section 2.2.

The Analytic Hierarchy Process (Saaty, 1990) (Podvezko, 2009) is a mathematically grounded method for quantitative multicriteria evaluations relying on numeric techniques to compare a set of alternatives with the defined evaluation criteria through pairwise comparisons with the aid of a scale of absolute judgements that assigns each criterion to a higher or lower level of importance. This method relies on the judgement of the decision-maker to assign the priority scale. It is implemented in 7 phases (Nicola, 2019):

- Defining the problem (with a Hierarchical Decision Tree).
- Comparison of alternatives and criteria (using a comparison matrix).
- Obtain the Relative Priority Vector for each criterion.
- Evaluate the consistency of the decisions.
- Pairwise comparison matrix for each criterion according to each alternative.
- Obtain the Composite Priority Vector for the alternatives.
- Selection of the alternative using the Composite Priority Vector.

The problem in analysis is *selecting an architectural pattern for implementing the micro frontend approach*. The alternatives are Server-side Composition, Client-Side Composition, and Edge-Side Composition. The criteria for the decision are User Experience, Autonomy, Scalability, Technology Agnosticism, and Performance.

In Figure 17, the problem is defined as a hierarchical decision tree for easy understanding.

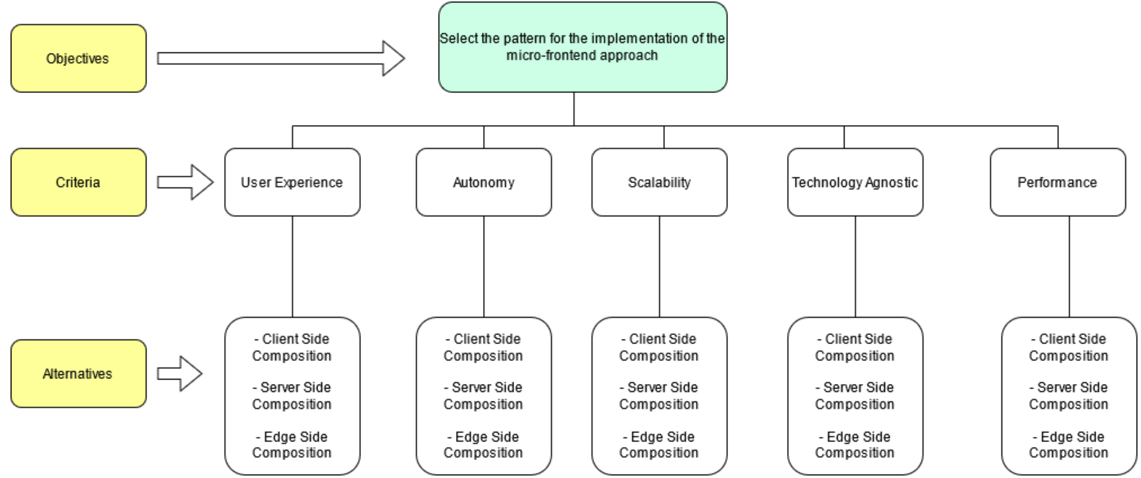


Figure 17 - Hierarchical Decision Tree - Selection of the Micro Frontend pattern.

At this phase, the priorities of the criteria are defined by performing pairwise comparisons to differentiate the criteria by their importance against each other. This is done with the help of the fundamental scale (Saaty, 1990) and its values are shown in Table 3.

Table 3 - Fundamental Scale. Source: (Saaty, 1990)

Intensity of importance on an absolute scale	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance of one over another	Experience and judgment strongly favor one activity over another
5	Essential or strong importance	Experience and judgement strongly favor one activity over another
7	Very strong importance	An activity is strongly favored and its dominance demonstrated in practice
9	Extreme importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgments	When compromise is needed
Reciprocals	If activity i has one of the above numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i	
Rationals	Ratios arising from the scale	If consistency were to be forced by obtaining n numerical values to span the matrix

The comparison in the context of this problem is performed in Table 4.

Table 4 - Comparison Matrix between the criteria

	User Experience	Autonomy	Scalability	Technology Agnostic	Performance
User Experience	1	2	2	3	3
Autonomy	1/2	1	2	3	3
Scalability	1/2	1/2	1	3	3
Technology Agnostic	1/3	1/3	1/3	1	2
Performance	1/3	1/3	1/3	1/2	1

After the comparison, results are normalized to match the values assigned to each criterion to the same unit. The relative priority vector is then calculated to define the importance of each criterion over the others. The results are shown in Table 5.

Table 5 - Normalized Comparison Matrix and Relative Priority Vector

	User Experience	Autonomy	Scalability	Technology Agnostic	Performance	Priority Vector
User Experience	0,38	0,48	0,35	0,29	0,25	0,35
Autonomy	0,19	0,24	0,35	0,29	0,25	0,26
Scalability	0,19	0,12	0,18	0,29	0,25	0,20
Technology Agnostic	0,13	0,08	0,06	0,10	0,17	0,11
Performance	0,13	0,08	0,06	0,05	0,08	0,08

According to the results, the most critical criterion of the proof of concept is User Experience, as there it is expected to simulate a web application with characteristics such as high interactivity and responsiveness. Afterward, autonomy and scalability, seen as two of the main aspects of micro frontends in terms of team organization and development. The next is Technology Agnosticism, as this characteristic can be an advantage of micro frontends to provide the flexibility for each team to choose the technology they find better to develop, it can also be a downside as it can lead to an anarchy of the use of a wide range of different technologies (Thoughtworks, 2020). Performance is seen as the least critical characteristic,

albeit it will be a target of study in this dissertation to understand how it will behave according to the performance metrics. There are no expectations of an increase in the overall performance of the web application, as it is not the main focus of micro frontends as seen in previous sections. In the next step, to measure the consistency of the judgements, the consistency ratio (RC) is calculated. The judgements are considered reliable if the value of RC is below 0,1. (Nicola, 2019).

First, λ-max is calculated using the formula:

$$Ax = \lambda_{max}x \tag{1}$$

Where *A* is the criteria comparison matrix and *x* the priority vector. The result of *λ-max* is approx. 5,20.

With this value it is possible to calculate the Consistency Index (IC) using the following formula:

$$IC = \frac{\lambda_{max} - n}{n - 1} \tag{2}$$

Where *n* is the number of criteria. The result of *IC* is approx. 0,049. Then, the Consistency ratio can be calculated using the formula:

$$RC = \frac{IC}{IR} \tag{3}$$

IC is the previously calculated value and IR is the random consistency value for our matrix. The *IR* value is a constant value defined in Table 6.

Table 6 - Random Consistency Values. Source: (Nicola, 2019)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

As this matrix is a 5x5 matrix, the Random Consistency value (*IR*) for order 5 matrices is 1,12 according to Table 6. The value of *RC* is approx. 0,044. Since this value is below 0,1, the judgements are considered reliable. Table 7 shows a summary of the values obtained.

Table 7 - λ-max, Consistency Ratio and Consistency Index results

λ-max	5,20
Consistency Index (IC)	0,049
Consistency Ratio (RC)	0,044

The next step is to create a comparison matrix for each criterion, considering both alternatives, to find the best approach based on the criteria after calculating the global priority vector. The process is done by repeating the creation of the comparison matrices (this time, between each criterion and alternative), the respective normalization matrix, and calculating the priority vector. The initial comparison using the fundamental scale (Table 3) is shown in Table 8.

Table 8 - Comparison Matrices between Alternatives and the Criteria.

User Experience			
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**
Client-side Composition	1	5	5
Server-side Composition	1/5	1	1/3
Edge-side Composition	1/5	3	1
Autonomy			
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**
Client-side Composition	1	5	7
Server-side Composition	1/5	1	5
Edge-side Composition	1/7	1/5	1
Scalability			
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**
Client Side Composition	1	1/3	1/3
Server-side Composition	3	1	1/2
Edge-side Composition	3	2	1
Technology Agnostic			
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**
Client-side Composition	1	1	5
Server-side Composition	1	1	5
Edge-side Composition	1/5	1/5	1
Performance			
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**
Client-side Composition	1	1/5	1/5
Server-side Composition	5	1	1/2
Edge-side Composition	5	2	1

Then, the matrices are normalized and the respective local priority vector is calculated. The result is shown in Table 9, along with the local priority vector.

Table 9 - Normalized Matrices for Alternative-Criteria Comparisons and Local Priority

User Experience				
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**	**Local Priority**
Client-side Composition	0,71	0,56	0,79	0,69
Server-side Composition	0,14	0,11	0,05	0,10
Edge-side Composition	0,14	0,33	0,16	0,21
Autonomy				
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**	**Local Priority**
Client-side Composition	0,74	0,81	0,54	0,70
Server-side Composition	0,15	0,16	0,38	0,23
Edge-side Composition	0,11	0,03	0,08	0,07
Scalability				
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**	**Local Priority**
Client-side Composition	0,14	0,10	0,18	0,14
Server-side Composition	0,43	0,30	0,27	0,33
Edge-side Composition	0,43	0,60	0,55	0,52
Technology Agnostic				
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**	**Local Priority**
Client-side Composition	0,45	0,45	0,45	0,45
Server-side Composition	0,45	0,45	0,45	0,45
Edge-side Composition	0,09	0,09	0,09	0,09
Performance				
	Client-side Composition	**Server-side Composition**	**Edge-side Composition**	**Local Priority**
Client-side Composition	0,09	0,06	0,12	0,09
Server-side Composition	0,45	0,31	0,29	0,35
Edge-side Composition	0,45	0,63	0,59	0,56

The results of Table 9 are then compiled into a new matrix along with the relative priority vector from Table 5. With these values, it is possible to calculate the Composite Priority Vector by multiplying the result from each alternative with the priority vector. This vector will represent the importance of the alternative based on the criteria. The results are shown in Table 10.

Table 10 - Criteria/Alternatives Classification Matrix and Composite Priority

	User Experience	Autonomy	Scalability	Technology Agnostic	Performance	Global Priority
Client-side Composition	0,69	0,70	0,14	0,45	0,09	0,51
Server-side Composition	0,10	0,23	0,33	0,45	0,35	0,24
Edge-side Composition	0,21	0,07	0,52	0,09	0,56	0,25

Concluding this process, according to the AHP method and the respective results of the global priority vector from Table 10, the Client-Side Composition is the approach of choice, following by Edge-side Composition and Server-side Composition Approach.

4 Techniques Comparison

In this chapter, it is possible to see a comparison of all the techniques using 6 characteristics seen as benefits of a micro frontend approach (Autonomy, Technology Agnosticism, User Experience, Value Delivery, Performance, and Resource Sharing) and a summary table at the end. There is also a comparison between architectural alternatives and frameworks to decide which to use in the Design and Implementation of the solution.

To correctly evaluate the adequacy of some patterns, the techniques have to be compared according to the characteristics that define micro frontends and web development in general and see how they cope with these characteristics. After the analysis, a table summary will be presented using a scale of three colors, green for “works well with”, yellow for “works but with issues”, and red for “doesn’t work well with”.

Essentially, the main characteristics provided by micro frontends are as follows:

1. **Autonomy** – How a given approach can cope with team and codebase autonomy, for instance, one of the key elements in implementing a micro-frontend approach.
2. **Technology Agnosticism** – How the micro-frontend pattern can cope with technology agnosticism since this is a key element in the development of web applications on an enterprise and analysis in this document.
3. **User Experience** - How the app reacts to a user keystroke or the consistency of the style of the fragments that together, give the illusion of one single application overall, essentially how micro frontends can cope with user experience.
4. **Value Delivery** – The quality of the value delivery when on developing a given approach.

5. **Performance** – How well the solution works in terms of initial page loading, transitions, among others.
6. **Resource sharing** – How the approach copes with resource sharing in terms of context, memory, etc.

Note that for scalability, all approaches have independent deployments in their micro frontends, so it should be easy enough to scale a given micro-application in terms of infrastructure as well as extend the code of the given feature if needed. As for code or resource duplication (such as style), a centralized library or CDN can be created as a repository of resources but this will create a new dependency which is something micro frontends are trying to avoid. For these reasons, these attributes will not be considered in this analysis since they can be shared across all the approaches.

4.1 Route Based

1. **Autonomy** – The fact that this approach is based on routing through different and isolated web applications, makes it very good in terms of autonomy. Each team develops in its codebase and environment and the applications can be developed without dependencies, applying the micro frontend principles easily regarding autonomy. As for the codebase, there will not be any conflicts of styles for instance, simply because there are no shared resources whatsoever. A downside is the fact that each team must communicate the *URL* of their page or find an agreement in a *URL* standard, so it does not create conflicts and to each team know which URL to call to a route (Dhiman, 2019).
2. **Technology Agnosticism** – This approach deals well with different technologies since every app is isolated, so there is no problem regarding this characteristic.
3. **User Experience** – Since they are different applications that jump through routing, the page must be refreshed, and there will be a white screen between the pages. This white screen issue can be softened with the use of third-party libraries. Yet, this is a major downside. Also, the fact that we cannot embed apps, can lead to a poor user experience.
4. **Value Delivery** – This a downside in this approach, since we cannot embed different pages inside another using the routing and will limit our possibilities. Essentially, there must be

worked to ensure the user has the idea that all the apps are delivered has a single app, such as the use of single sign-on for only need to authenticate on one single app and make sure all the applications share this state, for instance.

5. **Performance** – The performance in this approach is fine because the separate applications are supposed to be micro-apps deployed in a web server, so they should be lightweight enough to provide a performance.
6. **Resource Sharing** – There are no shared resources in this approach, this can be a downside. Yet, it can be worked out, using event bus, for instance, to enable communication between the pages. Also, an error on one page does not affect the rest (Geers, 2020) since they do not share resources.

4.2 IFrames

1. **Autonomy** – Iframes are completely isolated web applications so there will not be any problem regarding the autonomy of teams and the application itself.
2. **Technology Agnosticism** – As for agnosticism, Iframes work well since they are also completely independent applications such as route based micro frontends.
3. **User Experience** – There might be an issue with sizing for instance, and we end up with scrolls in the Iframes that will ruin the UX and create layout coupling. Also, if we care about accessibility, for instance, the browser does not cope well with Iframes for screen readers, since it breaks semantics (Geers, 2020), so the browser will have a hard time identifying the different components that compose the page. In case of errors, we also have a poor UX since some browsers display the error of the Iframe as grey background for instance, and there's no way to replace this behavior (Geers, 2020).
4. **Value Delivery** – As an advantage against the route-based, IFrames allows the use of embedded applications within a page, which is something we are interested in when using a micro-frontend approach. The browser also supports well this technology so there will not be compatibility issues related. If SEO is a major concern Iframes do not cope well with crawlers as they will index embed pages as different pages (Geers, 2020).
5. **Performance** – If the main page has to load several Iframes with different javascript frameworks, performance problems can exist due to heavy resources loaded (repeated libraries for instance) in memory and CPU (Dhiman, 2019).

6. **Resource Sharing** – Although there is separated runtime in this approach, yet since this is a native HTML tag, window events can be used to communicate between Iframes, so it's an advantage over the route-based micro-frontend approach. Yet, having separate runtimes, means all the applications are completely isolated from each other, and there are also some advantages with the approach since there won't be any conflicts between resources such as namespaces or styles.

4.3 Native WebComponents

1. **Autonomy** – As seen in previous sections, the use of technology included in the Web Component spec such as ShadowDOM (Oh, Ahn and Kim, 2017), makes sure this approach has isolation and contributes to the full autonomy of the teams.
2. **Technology Agnosticism** – This might be a problem in this approach, if we want to create custom web components using different JavaScript Frameworks (such as using angular elements, to create web components using Angular framework) we might end up with compatibility issues with some vendors now, such as React (Dodson, n.d.). Yet, since this uses the web component spec standard, as time goes by this problem should be addressed and we eventually will end up with full compatibility with all the major vendors.
3. **User Experience** – As micro frontends will be rendered in the page as native custom elements, we do not have issues we had in Iframes, for instance, this will work the same way as the components were developed in the same application, the DOM will load it normally.
4. **Value Delivery** – With this approach, we can embed several components to compose a page rendered as an HTML component. Also, browser compatibility is an issue since older browsers do not provide support for custom elements, yet pollyfills can be used to address this issue but that will decrease performance (Geers, 2020).
5. **Performance** – This approach has great performance since the native components are more lightweight compared to the framework-based web components. Yet, if one develops the custom elements using different frameworks, the size of the bundles can increase and become resource-heavy (Dhiman, 2019).

6. **Resource Sharing** – There's context shared across all the components that form a page, which is a major advantage since we don't need to rely on third party technology for communication for instance (browser events can be used), yet this shared resource is encapsulated using technology such as Shadow DOM (Oh, Ahn and Kim, 2017), resulting in having the advantages of both a shared and a separate runtime. There is also no need to have third-party technology for component communication.

4.4 Framework-based Web Components

1. **Autonomy** – The frameworks grant autonomy as seen in previous sections.
2. **Technology Agnosticism** – The use of meta-frameworks such as Single-SPA handles the technology agnosticism well, providing interfaces to easily integrate different technologies in the same final web application and compatibility with all the major JavaScript Framework vendors in the market. Some of the frameworks have huge support from the open-source community which can increase the speed of development of new feature updates and improvements. Yet, a developer ends up tied to a framework to implement the micro-frontend solution, this can be problematic if the support from the community is over, consequently might result in migrations.
3. **User Experience** – Frameworks such as Single-SPA are reliable in terms of page composition.
4. **Value Delivery** - With this approach It is also possible to embed several components to compose a page rendered as an HTML component and deliver a complete single-page application without noticing the separation of the components, such as Native web components.
5. **Performance** – A possible problem with this solution is the fact that, if we have many different frameworks to be integrated, all vendor bundles need to be downloaded and consequently the overall bundle size will increase, this can lead to performance issues and must be worked by the developers to improve. Anyway, some meta-frameworks, such as single-SPA, already address this issue and provide ways for performance tuning with the use of lazy loading as the bundles will only be downloaded when they are needed, softens the problem to become a solid solution.

6. **Resource Sharing** – There is also resource sharing in this approach which is a major advantage on the communication between the components.

4.5 Server-side Transclusion

1. **Autonomy** – This might be a problem depending on the technology used since every developer needs a web server with SSI support for local development, but the use of frameworks such as podium can help in this issue.
2. **Technology Agnosticism** – Most server-side transclusion implementation of micro-frontend can cope well with different technologic stacks, including the use of different frameworks for the frontends. Yet, the fact that there is no native way of implementing this pattern will be a downside as there is a dependency regarding some third-party software or frameworks and how they are configured (Dhiman, 2019).
3. **User Experience** – If a company decides to develop a reactive web application with an almost immediate response with user interaction, this pattern will have problems as the browser will always need to communicate with the server and download the pages, resulting in a delay of the actions and if the product meant to be a reactive web application, the user experience will be a major drawback.
4. **Value Delivery** – good for SEO, crawlers cope well with this approach since it does not require lots of client-side code and has good performance, which will help on the search engine rankings (Geers, 2020). There is also an advantage regarding this criterion mostly versus a client-side composition approach, in the client the browser needed to fetch heavy resources by default and the user would experience a slow loading time, server-side transclusion on the other hand addresses this problem by loading the heavy resources in the server-side where there is more computing power if the server is properly set up. (Dhiman, 2019).
5. **Performance** – First Page Loading is fast due to the page being formed on the server-side, also for heavy work this might be the best solution since it can do all the composition in server-side where there's more computing power and only relies on the client-side browser to download the page, this is an advantage mostly versus a client-side composition approach as in the client the browser needs to fetch heavy resources by default and the user would experience a slow loading time. Yet, if the server is not properly set up, it can

provide a slow page delivery to the client as well. Additionally, some frameworks provide a good time to first byte since the page is initially sent as a template and then it loads the components for better performance and the use of parallel loading of fragments (Geers, 2020) (Dhiman, 2019).

6. **Resource Sharing** – There's still resource sharing between the different components as in web components since the rendering engine will compose the page.

4.6 Edge-side Transclusion

1. **Autonomy** – This relies mainly on the CDN and proxies to implement ESI, where each CDN has its own implementation, decreasing autonomy.
2. **Technology Agnosticism** – Similar to Server-Side transclusion, it can cope well with different technologies on implementing micro frontends but will still be limited to third parties when implementing using CDNs or proxies.
3. **User Experience** – Using Cache can increase user experience, yet, the use in static content delivery is the main benefit of this approach.
4. **Value Delivery** – Similar to Server-side transclusion, Cache features and delivering at the edge can increase the quality of value delivery. Limitations with managed services can be a drawback in terms of value delivery.
5. **Performance** – Although relying on a server to fetch content, using Caching at the edge level will increase performance.
6. **Resource Sharing** – There is resource sharing between the different components after composing the page.

4.7 Module Federation

1. **Autonomy** – Since the modules will be deployed independently to be then provided and loaded remotely and dynamically, Module Federation allows autonomy.
2. **Technology Agnosticism** – Might need the integration with a framework or implementation of more code to be possible integrating different JavaScript frameworks, still blurry. In terms of applying the architecture itself, although webpack is a very popular bundler, it is a mandatory dependency.
3. **User Experience** – As modules will be dynamically loaded, as seemingly all modules are in the same repository, it will allow the development of reactive web applications with the advantages of single-page applications.
4. **Value Delivery** - With this approach It is also possible to embed several components to compose a page rendered with different components at runtime previously compiled by other deployed web applications that provide the modules remotely, so value delivery is like native and framework web components.
5. **Performance** – This approach tackles some issues from other approaches mainly in the framework web components approach, such as the popular SystemJS, offering a performance tuning as the content downloaded from bundles will be optimized. In terms of implementing different frameworks using module federation, increased bundle size might be an issue at some point and will need to be studied further, but performance overall should be better due to the optimization.
6. **Resource Sharing** – Resource sharing is not only possible but also sharing other artifacts between micro frontends such as CSS, images, configuration, etc., is facilitated as anything that can be bundled can be shared.

4.8 Summary

Table 11 - Summary Table of the patterns

	Autonomy	Tech Agnostic	User Experience	Value Delivery	Performance	Resource Sharing
Route	Green	Green	Red	Red	Green	Red
IFrames	Green	Green	Red	Yellow	Red	Green
Web Components	Green	Yellow	Green	Green	Yellow	Green
Framework Web Components	Green	Yellow	Green	Green	Yellow	Green
Server-side transclusion	Yellow	Yellow	Red	Green	Green	Green
Edge-Side Transclusion	Yellow	Yellow	Red	Yellow	Green	Green
Module Federation	Green	Yellow	Green	Green	Green	Green

Legend:

Green = “works well with”

Yellow = “works but with slight drawbacks”

Red = “Doesn’t work well with”

5 Solution Design

According to the objectives, a proof of concept using a micro-frontend approach is developed. In this chapter, the design of the solution to be implemented is described, first by considering the functional and non-functional requirements for the proof of concept and then creating use cases.

The next steps are to assign each team to a use case to develop its micro frontend and choosing the technology stack.

Then, the architecture of the solution was conceived with an explanation of the architectural patterns used, and each module of the solution. A deployment diagram is designed to show an example on how the implementation could be deployed.

5.1 Requirements Specification

The *Proof of Concept* to implement the micro-frontend approach studied in the previous sections, consists of a web application containing several features to show how to apply the micro frontend architecture when facing a specific situation.

The process started with a requirement specification phase, where both functional and non-functional requirements are used to describe the features to be implemented and to apply the principles of the micro-frontend practices. Since this is a *proof of concept*, increased priority is

given to the non-functional requirements, to show how a web application would benefit from the use of micro frontends.

The Functional Requirements are as follows:

- The system must allow the user to browse the existing products.
- The system must be able to show the shopping cart of a user.
- The system must be able to edit the shopping cart of a user.
- The system must contain a menu to easily navigate through the features.
- The System must have a bootstrap/app shell layer to handle the composition of all the micro frontends.
- The System must be able to show the purchase history to the user.
- The system must be able to show the product list at checkout.
- The system must be able to allow the user to choose a payment method.

The Non-Functional requirements are the following:

- The system must allow teams to work in each feature in isolation to reduce dependencies.
- All the components of each feature must be isolated and easily scalable.
- The components must have a smaller codebase.
- The components must allow being previewed independently and locally.
- The components must be able to be deployed independently.
- The overall application must be robust and autonomous, an error in one component should not affect another team’s component.
- The components must be tested independently.

Table 12 - Use Cases of the solution

ID	Use Case	Priority
UC001	Browse products	1
UC002	Show shopping cart	1
UC003	Edit shopping cart	1
UC004	Purchase a Product	1
UC004	Select Payment Method	1
UC005	Show Purchase History	1

5.2 Team Organization & Technology Stack

The use cases were mapped into different teams, and the technology stack was chosen.

Team A – Catalog component, Catalog microservice, FakeStoreApi (external).

Team B – Cart Component, Cart Micro Service, Cart Database.

Team C – Checkout Component, Purchases Microservice, Purchases Database.

Team D – Navbar Component

Team Core – App Shell component

All the components will communicate with the backend using HTTP and JSON. Since the focus in this *Proof of Concept* is the frontend side, the backend side will be developed using netcore 3.0 as chosen technology due to the author's experience on the language to increase development speed. Anyway, any technology could be used in that layer such as Java EE. Microservices are already a mature concept.
The Micro frontends will be developed using ReactJS Framework and Typescript.

5.3 Architectural Alternatives

This section presents architectural proposals that allow implementation of the micro frontend solution.
Bearing in mind the analysis of the techniques in previous sections, Server-side Transclusion, Web Components (native and framework based), and Module Federation are the approaches seen as containing the most advantages.

From this conclusion, and by studying the patterns specified previously and use-cases by existing companies, the architectural approaches regarding the 3 patterns are presented in Figure 18 and Figure 19.

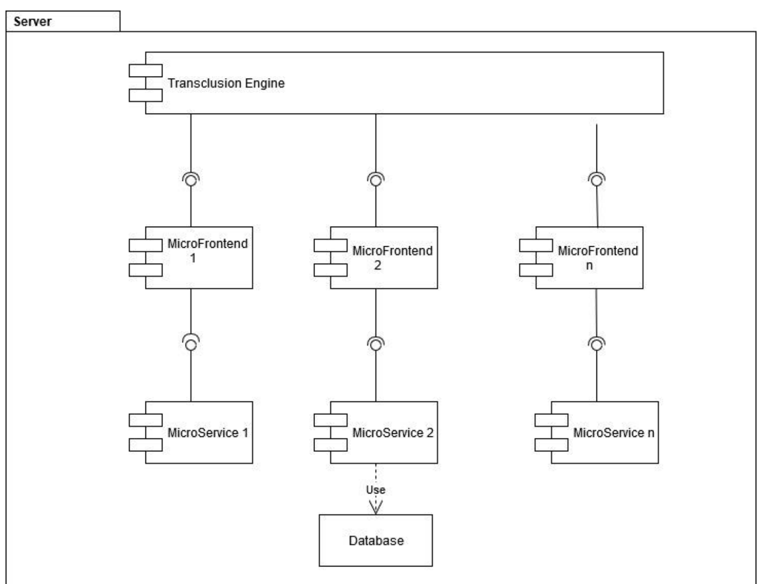


Figure 18 - Architecture Diagram for Server-side transclusion Approach

The approach in Figure 18 is designed to implement the server-side Transclusion pattern, based on using a JavaScript Framework, such as Podium as the Transclusion engine, to take advantage of its many features. The browser at runtime will communicate with the server to request the pages to the Transclusion Engine, which will provide the page template containing the placeholders for the micro frontends that will be pre-rendered and send so that the browser can obtain a good performance on First Page Load.

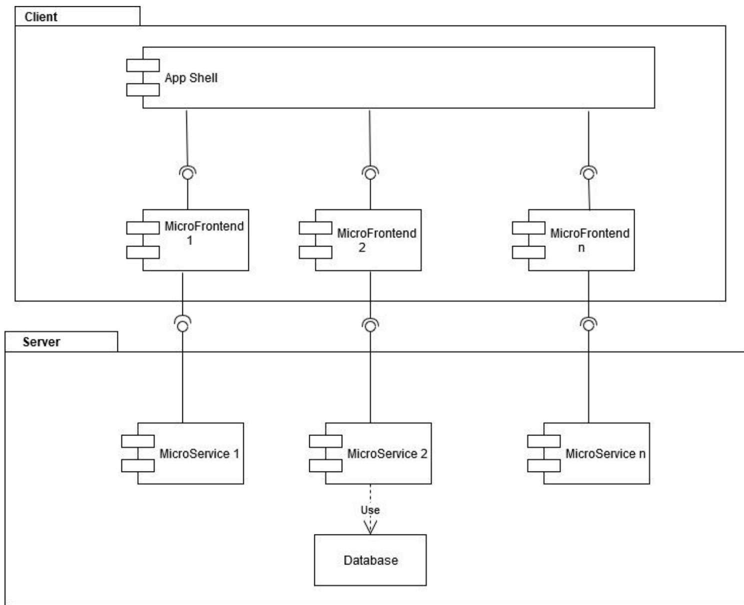


Figure 19 - Architecture Diagram for Client-Side Composition approaches

The approach in Figure 19 serves the purpose of both Native Web Components, Framework-Based Web Components, and Module Federation. Depending on the choice between these patterns, the App Shell implementation will be slightly different, and some small changes will have to be applied in the Micro Frontends so it can adapt to the lifecycle interface of the Web Component specification (used in native web components) or the custom interface provided by the framework (in Framework-Based Web Components approach). Using of module federation, the app shell would need to have a configuration in webpack regarding the module federation plugin to be able to import the existing micro frontends, as it would be the same as implementing a normal single-page application by importing the modules.

App Shell component should be regarded as a tiny layer and with the responsibilities kept to a minimum to decrease the dependencies and facilitate in case of maintenance or migration. At most, there will be the attempt of only the routing code placed there since this layer will orchestrate all the existing micro frontends.

Both solutions can implement a micro frontend approach successfully with advantages depending on the Use Cases the web application needs. Since the idea of this Proof of Concept is to focus on user experience by creating a reactive web application that can quickly respond to a user action and will not contain the loading of heavy resources, the server-side transclusion approach will not be used.

Since an edge-side transclusion solution would rely mainly on a proprietary CDN, and its benefits are obtained on more static content delivery (opposite of what is expected in the proof of concept), this approach will not be considered as well.

If the idea was to demonstrate full compatibility between different technology (in the case of JavaScript Frameworks), the Framework-Based Components approach using the Single-SPA framework would be a good choice. This framework is widely popular and seen as one of the most mature to date, in the Micro frontend scene.

Considering all aspects discussed so far, the Module federation approach will be used in the Proof of Concept since It brings many potential benefits discussed in previous sections, although still very recent. The Proof of Concept will be based on a single JavaScript Framework – React – expecting a more lightweight approach.

The use of a single JavaScript Framework is emphasized by the recent findings regarding micro frontends implementation in developing a solution, along with the benefits of a single-page application. As the micro frontend concept is becoming more mature, bad practices are also being found, such as the abusive use of many different technologies when developing the micro frontends (Thoughtworks, 2020), which is important to understand the benefits or drawbacks between a micro-frontend approach containing multiple technologies *versus* a solution containing a reduced number of technologies or libraries.

5.4 Architecture In Detail

The chosen architecture is based on the Micro Frontend architectural approach using client-side composition with module federation. The following points describe the architecture in Figure 19 in detail:

- Each micro frontend is a completely isolated component, it is possible to have its style, and its dependencies without conflicts with other components.
- Independent Continuous Integration and Continuous Delivery, where each micro frontend has its pipeline, for testing and deployment. Allowing increased scalability, flexibility, and modifiability.
- Each Feature is autonomous and composed of its micro frontend, its microservice, and, if needed, its database.

- Each component has a small codebase related to a sub-domain of the application's domain.
- Each component or module can be migrated separately without the need to migrate the whole application at once.
- Each component can be maintained separately by a single team specialized in its context, also codebases are smaller and easy to modify.

A **sensitive point** in the architecture might be performance. The App Shell imports dynamically each micro frontend at runtime by downloading its remote bundles. Although this is a central piece of the architecture, dependency management and a good separation of concerns must be performed, along with features such as lazy loading. It is important to keep bundle sizes as low as possible to increase performance.

With the increase complexity of the frontend modules, there is a **risk** with the codebase increasing over time. This issue can be mitigated using the Backend-for-Frontend Pattern. This new component would be tightly coupled with the micro frontend and will handle the job of separating the logic related to the microservices from the presentation layer, thus decreasing the codebase on the frontend side. This pattern is described in Section 2.3.2. An architectural design using BFF pattern can be seen in Figure 20.

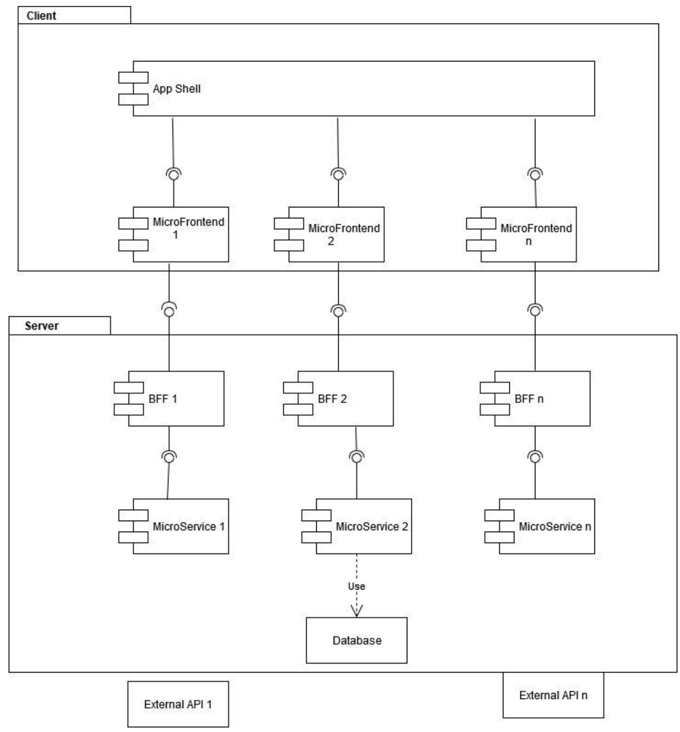


Figure 20 - Architecture Diagram of the Solution

Since this proof of concept will not have such level of complexity, the architecture in Figure 19 is used in the development.

The main module of the architecture in Figure 19 is the app shell.

The pages are composed of micro frontends on the bootstrap/app shell layer, featuring Typescript + ReactJs + Webpack 5 in the module federation approach, used to compose the fragments and then render them on the body of the document. This component is the first point of contact for the incoming requests and is also responsible for the routing of the application and mounting/unmounting the micro-frontend modules, according to needs.
This layer is a crucial part of the client application architecture, so it is also a single-point-of-failure, it should be carefully analyzed and implemented with quality, avoiding business logic to be tiny, not create dependencies, and being easier to maintain. Some cross-cutting concerns such as Authentication and Analytics (Geers, 2020) can be implemented in this layer.
For the communication between all the components on a page, custom events will be used.
All the micro frontends will communicate with several different microservices the infrastructure currently holds previously developed and already used in production to access the data needed for each feature. These Micro Services will then communicate with their database or external API as data sources.

By using micro frontends, makes it easier to migrate a codebase, simply because each feature is isolated, with only the need to migrate these tiny fragments at a time, not needing to migrate the whole application at once.
An example for the solution’s deployment is shown in Figure 21.

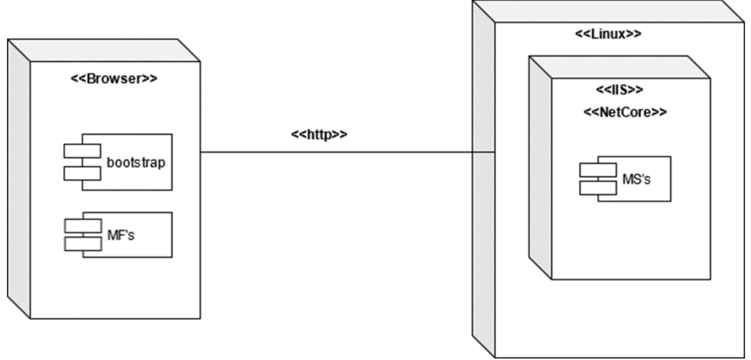


Figure 21 - Deployment Diagram of the solution.

The client-side will be accessed by a browser and will contain the bootstrap/app shell layer and the micro frontends. This layer communicates with the Microservices (MS) on the server-side.

6 Solution Implementation

In this chapter, the solution implementation is described starting with an overview of how module federation was implemented. Afterward, a description of each micro frontend implemented followed by State Management, Error Handling, and testing. Source code is available on Github (Silva, 2021).

6.1 Overview

As stated in previous chapters, this proof of concept refers to an e-commerce website. Table 13 summarizes the correspondence between each micro frontend and microservice.

Table 13 - Correspondence between Micro frontends and Microservices

Micro frontend	Microservice
App Shell	-
Navbar	-
Cart	cart-service
Catalog	catalog-service
Checkout	Purchases-service

Although the use of standards is usually the safer way to go - as it will continue consistent over time - the possibility of being discontinued always remains – currently, using a framework for micro frontends is a very solid solution due to compatibility with major vendors such as Angular or React. Nowadays, the use of native web components still has compatibility issues to date with, for instance, React (Dodson, n.d.) – a widely used solution in designing frontend solutions. The fact that JavaScript frameworks are also widely used in the development and carry that luggage of the possibility of being discontinued at some point, the compatibility with the vendors weights the decision of using a meta-framework on the bootstrap/app shell tiny layer as a stitching point for all the micro frontends, and single-SPA does a good job in that task. Nevertheless, both solutions are being improved fast over time and although the use of web component specification can be a solid solution, the proof of concept will make use of Module Federation (webpack 5) + react for the implementation due to benefits found on section 2.4.2.3.

6.2 Module Federation

Module federation is used in this proof of concept. The concept is explained in section 2.4.2.3. It’s implemented in the webpack.config.js file where most of the work such as referencing the remote applications to be imported is made.

```
new ModuleFederationPlugin({
name: "host",
library: { type: "var", name: "host" },
remotes: {
Catalog: "Catalog",
Navbar: "Navbar",
 (…)
},
shared: {
...deps,
 react: { singleton: true, eager: true, requiredVersion: deps.react },
 "react-dom":
{ singleton: true, eager: true, requiredVersion: deps["react-dom"] },
 (…)
  },
}),
new HtmlWebPackPlugin({
 template: "./src/index.html",
 CatalogRemoteEntry: "http://localhost:4000/remoteEntry.js",
(…)
    }),
```

Code 3 - Webpack configuration example for the host application.

Code 3 shows an exemple of module federation configuration in App Shell Micro frontend. At the plugin entry, *ModuleFederationPlugin* and *HtmlWebPackPlugin* are used. *Name* is how the app will be named remotely.

The part *Remotes* is what is being imported by the App Shell. In this example, *Catalog* and *Navbar* are imported. Both are being imported using the on the entry *CatalogRemoteEntry*. The shared entry shows the shared dependencies needed for the import, in the example react and react-dom.

```
new ModuleFederationPlugin({
name: "Purchases",
library: { type: "var", name: "Purchases" },
filename: "remoteEntry.js",
remotes: {
},
exposes: {
"./PurchaseHistoryTable":"./src/Components/PurchaseHistoryTable",
"./CheckoutItems":"./src/Components/CheckoutItems"
},
```

Code 4 - Webpack configuration example for Remote application

Code 4 shows the configuration for the remote application. This configuration is similar to the host application, the main difference is the *exposes* and *filename* entries, used to describe the component to be exposed for importing..

The remote application is then referenced on the index.html file as a script, as shown in Code 5.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
<div id="root"></div>
<script src="<%= htmlWebpackPlugin.options.NavbarRemoteEntry %>"></script>
(...)
</body>
</html>
```

Code 5 - *index.html* example for remote application importing.

As only chosen fragments or pages are remotely shared from each micro-frontend, this allows creating a sandbox for running and testing each micro frontend. Figure 22 shows the Purchases Micro frontend sandbox, which can be used for testing the checkout, purchase history, and the route to a test page on pressing the confirm button.

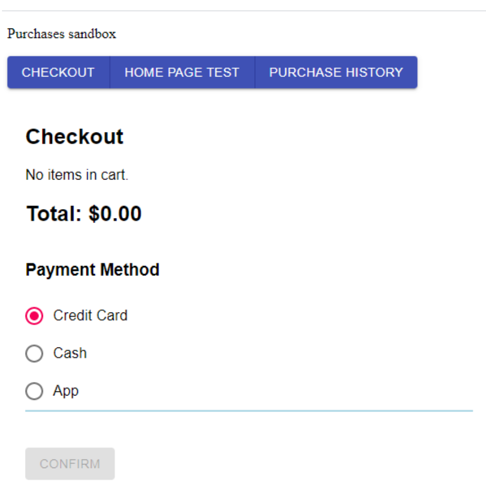


Figure 22 - Purchases Micro Frontend Sandbox – for testing purposes.

6.3 Micro Frontends

The five micro frontends developed in this proof of concept are App Shell, Navbar, Cart, Purchases, and Catalog. This section provides insight into its purpose and some of its particularities.

6.3.1 App Shell

The App shell is the micro frontend responsible for cross-cutting concerns, in this case: routing and importing all other micro frontends. This module serves as the first point of contact for the user to access the website and a single point of failure (Geers, 2020).

Figure 23 shows the Home Page and the division of each micro frontend composing the page: Yellow for Cart, Green for Catalog, and Red for Navbar.

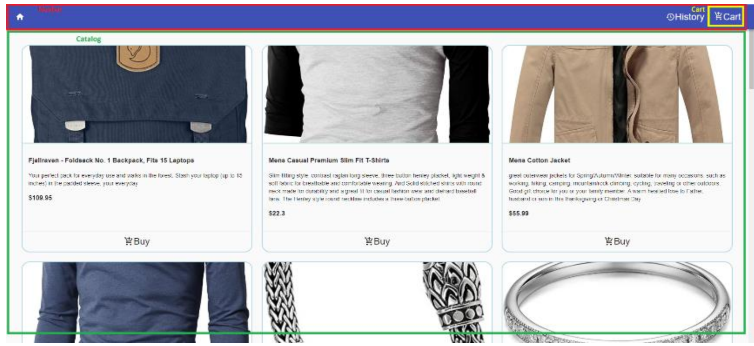


Figure 23 - Home Page showing the micro frontend composition.

6.3.2 Navbar

The Navbar choice to develop on a separated micro frontend concerns creating a more lightweight app shell component, separation of concerns, and allowing to do remote adjustments on the navbar without the need to work on another micro frontend (app shell, for instance).

This component is fixed on top of the page and contains a button for the homepage - which links to the catalog micro frontend – a button for the purchase history and an entry point for the Cart feature. The cart feature integration is performed by consuming the cart micro frontend to compose the navbar with the Cart Button and Drawer.

6.3.3 Catalog

Catalog Micro Frontend presents the user with the store catalog, each product containing a BUY Button, allowing them to add the product to the cart. The communication between the catalog item and the cart is performed using custom events, where a product-added-to-cart event is fired and subsequently listened to by the Cart Micro frontend that will add the product on the cart and show the number of products on the Cart Button.

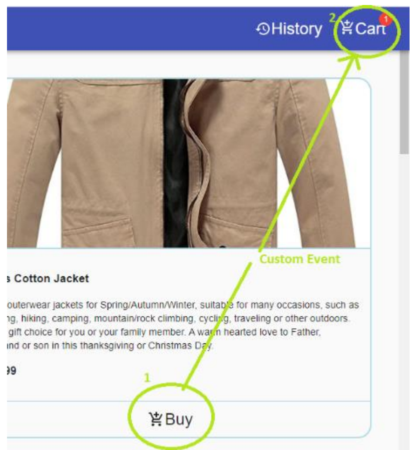


Figure 24 - Custom Event fired when clicking the buy button.

Figure 24 shows an example of a custom event, on click the Buy Button, the event is fired and updates the cart, showing the number of total itemson the button as well.
The catalog is also communicating with the Cart Microservice which fetches data from an open-source API called FakeStoreApi (Keikavousi, 2021) to provide mock data.

6.3.4 Purchases

Purchases Micro Frontend provides the checkout page and purchase history for the app-shell. The checkout page communicates with the cart microservice with a /GET HTTP Request to fetch the current products in the cart. If using the Backend-for-frontend pattern, this call could be done on the feature BFF to fetch the content from another microservice and prepare the view model.
After obtaining the cart data, the user is allowed to select the payment method and confirm the purchase. This purchase is then committed to the Purchases microservice that will update the list of purchases. The action of confirming a purchase will also fire a *purchase-completed* custom event, posteriorly listened by the cart micro frontend to clear the cart and update the cart microservice with a new cart state.

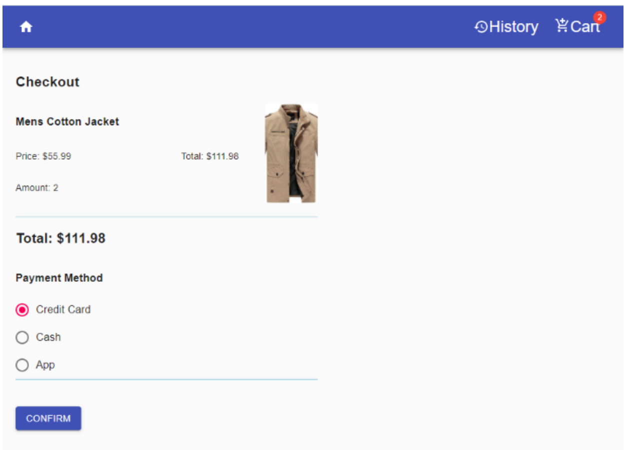


Figure 25 - Checkout Page.

The purchase history is shown by fetching the purchase data from the Purchases Micro Service and present the user.

Figure 26 - Purchase History Page.

6.3.5 Cart

Cart Micro Frontend consists of a Cart Button that will display the number of items currently in the cart. On Click, the button opens a drawer to display the cart details, such as the product name , image, price, number of items, and total price. Cart Details contains a button that serves as an entry point to the Checkout page (Purchases Micro Frontend).

The cart microservice is constantly being updated with an HTTP request to persist the user's cart.

Figure 27 - Cart Button and Drawer

6.4 Other Aspects

This section provides insights into other relevant aspects of the proof of concept development, equally fundamental for the correct functionality. The aspects addressed are the following: Routing, State Handling, and Error Handling.

6.4.1 Routing

As stated in previous subchapters, application routing is done on the app shell for this particular proof of concept, using React Router. A default route is set to the homepage if attempting to write random routes.

In more complex scenarios, another approach could be done with two-level routing (Geers, 2020), meaning the app shell would do the first level routing to a specific micro frontend, subsequently responsible to route to the actual page.

```
<HashRouter basename="/">
<RemoteNavbar />
  <Switch>
  <Route exact path="/">
  <RemoteCatalog />
  </Route>
  <Route path="/checkout">
<RemoteCheckout />
</Route>
(…)
 </Route>
</Switch>
</HashRouter >
<Button variant="contained" color="primary"
onClick={() => {this.setState({ isCartOpen: false })} }
component={Link} to="/checkout">
 Checkout
</Button>
```

Code 6 - Router example.

Code 6 shows an example of the router on AppShell Micro Frontend, and a Router Link on a button placed on a different Micro Frontend. In runtime, when the App Shell imports the module, the router link will be recognized.

6.4.2 State Handling

State handling is local on each component as recommended (Geers, 2020). This option was chosen due to the possibility of increasing coupling between features by using Global State, although it looks possible with the use of module federation. Cart, Purchases, and Catalog Micro Frontends use local state in their components. The use of global or shared state was mitigated using Custom Events for communicating between the fragments. After listening to an event, the local state is then updated.

6.4.3 Error Handling

When dynamically importing code from a remote application, one will come across errors now and then. This specific use case was thought in this proof of concept, although in a simple and not so realistic manner. Error handling was applied in two different situations using React's Error Boundaries feature (ReactJS, 2021) (Bvaughn, 2021):
In Cart button integration with the navbar, in a scenario where the Cart Micro frontend is not available, Navbar Micro Frontend wraps the Cart component into an Error Boundary, which causes the Navbar to load without the Cart Button normally – showing it is possible to continue running the application even if noncritical features fail.

Figure 28 - Navbar Fallback for Cart Micro frontend failure – Cart button will not appear.

In another scenario, the app shell wraps the loading of all components into an Error boundary, case of fails to load, App Shell will load a simple component stating "Website not available" as a fallback.

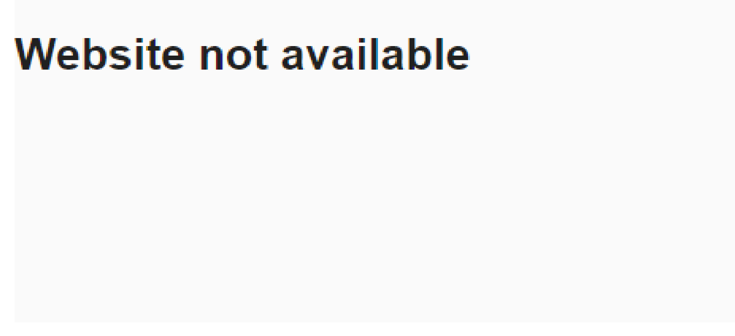


Figure 29 - App Shell fallback to remote importing failure.

Possible additional solutions for this use case can go through loading a fallback component in case of failure, and one could also handle errors differently or do some additional steps to guarantee more availability on each micro frontend. Anyway, for this specific proof of concept, this serves as a demonstration.

6.5 Solution Verification

To ensure that the application was built according to the specifications, and to provide an example on testing, several unit, integration and end-to-end tests were conducted.
In micro frontends, each module should be tested in isolation (unit tests, for instance) and in integration (integration or end to end tests - less quantity) such as any other monolith web application (Geers, 2020) to allow testing automation. There are no major differences using Module Federation, as each frontend module is self-contained, allows testing in isolation for each unit and feature. Tests were performed using the Jest library and React Testing Library. Some integration tests and ent-to-end tests were performed using Cypress framework, allowing live testing with all micro frontends and microservices running at the same time.
Cypress also allows performing the tests on Graphical User Interface (GUI) , making use of a real browser, or using a headless browser optimal to use in CI/CD pipeline.
Annex B shows screenshots of integration and end-to-end testing using cypress on graphical user interface.

The end-to-end test covers a scenario since the point a user buys a product, going through the entire process until checkout and opening his purchase history.
Integration tests exemplified are performed on Navbar, where the integration with the Cart microfrontend is tested, by asserting that the Cart button is placed on the Navbar when both micro frontends are available.

```
it("main page will load navbar",async ()=>{
  render(<App />);
  const element = await waitFor(()=> screen.getByText(/Navbar/i));
  expect(element).toBeInTheDocument();
 });
```

Code 7 – A simple test to assert if AppShell is loading Navbar component.

7 Solution Validation

As stated in Section 1.3, the focus of this document is on the main attributes for quality: Maintainability, Testability, Performance Efficiency, and Scalability. In this section, an analysis is conducted by validating the proof of concept based on the quality attributes. The section starts with a Design sub-section, explaining how the analysis will be conducted based on the GQM approach, as well as a description of the metrics used and the evaluation criteria. The following sub-section, Experiments, shows an insight into the experiments carried out. The last sub-section, Summary, provides the conclusions on the results for each experiment.

7.1 Design

Solution Validation is performed based on the Goal, Question, Metrics (GQM) approach.
GQM is a method used to measure a specific goal based on a set of metrics, performed in three levels (Calabrese et al., 2018):

- Conceptual Level (Goal) – The first step is to define a goal for each quality attribute studied, based on the proof of concept developed.
- Operation Level (Question) – To understand how the defined goals can be met, a set of questions are introduced.
- Quantitative Level (Metric) – Last level consists of defining the metrics (objective or subjective) to answer the questions and conclude how the goals are met.

Table 14 - Goals and Questions defined for each Quality Attribute.

Qual. Attribute	Goals	Questions
Maintainability	The solution needs to be maintainable.	Is the application easily maintainable?
Scalability	The solution needs to be easily and highly scalable.	Can HTTP request duration decrease when scaling only the needed micro frontends for a specific feature?
Performance	The solution needs to have an acceptable performance.	Is the solution's performance score acceptable when testing performance using a popular tool?
Testability	The solution needs to be easily testable.	How complex is the solution in terms of testing?

After defining the Goals and Questions in Table 14 , the metrics are established along with the specific evaluation criteria.

Maintainability metric: SonarQube's Maintainability Rating

Maintainability Rating considers a subset of metrics such as the technical debt ratio, which in turn considers remediation costs, the cost to develop 1 line of code, and total lines of code. Additionally, the cost to develop a line of code is stated as 0.06 days by the tool's documentation (docs.sonarqube.org, 2021).
Therefore, it is a good maintainability measure for this proof of concept as it considers the complete project instead of each separate file, therefore the analysis will be conducted on each micro frontend. Table 15 shows a relation between metric, rating, and evaluation for Maintainability.

Table 15 - Relation between metric, rating and evaluation for Maintainability.

Metric	Evaluation
A	Acceptable
B	
C	
D	Not Acceptable
E	

Scalability metric: Average HTTP request duration

Average HTTP request duration comparison between 1, 50, 500, and 1.000 virtual users (VU), performing requests to the web application while running each component of the application in 1 pod and then repeating the tests by scaling only the needed micro frontends up to 3 pods/replicas on kubernetes. The tool used on the load testing is k6. The average of the HTTP request duration will be used since a huge amount of HTTP requests are performed on the load tests.

Important metrics on the load tests and their meanings are (k6.io, 2021b):

- *http_req_duration* – shows the total time for the request by the server to process the request and respond (*http_request_sending* + *http_req_waiting* + *http_req_receiving*).
- *http_req_failed* – shows the rate of failure.
- *http_reqs* – shows how many HTTP requests were performed in total.

Table 16 shows the relation between metric, possible outcomes and evaluation for Scalability Quality Attribute.

Table 16 - Relation between metric, possible outcome and evaluation for Scalability

Metric	Possible Outcome	Evaluation
Average HTTP request duration	Average HTTP request duration when scaled is **less or equal** than the average HTTP request duration when not-scaled.	Acceptable
	Average HTTP request duration when scaled is **higher** than the average HTTP request duration when not-scaled.	Not Acceptable

Performance metric: Google Lighthouse's performance score

Google Lighthouse provides performance metrics based on a set of criteria such as First Contentful Paint, Speed Index, or Time to Interactive.

Table 17 - Relation between metric, rating and evaluation for Performance.

Metric	Rating	Evaluation
0 - 45	Red	Not Acceptable
50 – 89	Orange	Acceptable
90 - 100	Green	

Testability metric: Cyclomatic Complexity

Cyclomatic Complexity is a metric that measures the number of independent paths through a program, providing a quantitative measure for the logical complexity, usually associated with test planning and test-case design (Pressman, 2014). As a means to evaluate testability, the higher the cyclomatic complexity, higher the number of test cases and consequently, lower the testability (Kalagara, 2020).
The metrics are then related with the evaluation, to be used on validating the proof of concept, as shown in Table 18.

Table 18 - Relation between metrics and evaluation for testability

Metric	Description	Evaluation
1 – 10	Minimal Complexity	Acceptable
10 – 20	Moderate Complexity	
21 – 40	High Complexity	Not Acceptable
>40	Very High Complexity	

The experiment is conducted by analyzing the source code for complexity using SonarQube and the evaluation is then applied to the file with the higher cyclomatic complexity.

7.2 Experiments

This section provides an insight into the results of the experimetns performed on each quality attribute.

7.2.1 Maintainability

Maintainability experiments were performed using SonarQube. This tool is seen as one of the main tools used to perform static analysis (Baldassarre et al.,2020).

Annex A shows the complete results of the Maintainability analysis. Table 19 provides a summary of the analysis.

Table 19 - Summary of Maintainability analysis results.

Micro Frontend	Maintainability Rating
App Shell	A
Navbar	A
Catalog	A
Purchases	A
Cart	A

On the analyzis results, It is possible to notice there is technical debt on some of the files that could easily be fixed. Other than that, SonarQube is rating maintainability on each project is A, resulting in an average Maintainability Rating A for the overall proof of concept.

7.2.2 Scalability

Scalability experiments were performed using docker containers, Kubernetes (K3S), and K6 load testing.

The Test case is to open the purchase-history page, which requires:

- Micro Frontends: App Shell, Navbar, Purchases, and Cart (only used to show the cart entry point on Navbar).
- Micro Services: Purchases.

The biggest challenge found when conducting an experiment to simulate a user accessing the website is to attempt on simulating the actual browser behavior. As the test was performed on a localhost environment, using frameworks that work with an actual browser, such as Selenium, was not seen as a good choice due to computational limitations. The choice was to k6's browser recorder, to obtain all the HTTP requests performed by the browsers and import them as a *har* file. This file extension can be then converted by k6 to generate a test script. This initial test script was then adapted to create 4 different test scripts to simulate load testing by 1, 50, 500, and 1.000 virtual users. The tool used for integration and end-to-end testing as stated in Section 6.5, Cypress, might have been another possibility due to its headless browser capability when running test scripts.

In detail, the first test script is configured to start with a test of a single Virtual User. The next test starts with 5 virtual users, ramp up until 50 users for 30 seconds and stay flat in 50 users for 10 seconds. Then, there are two similar tests that increase the number of VU up to 500 and 1.000 users, under the same conditions.

Table 20 shows the result of the average HTTP request duration for each test. Tests were performed by deploying each micro frontend and its specific microservice in 1 pod and then scaling up to 3 pods/replicas.

Table 20 - Average HTTP request duration on each load test.

	Not Scaled	Scaled – 3 Replicas
1 user	30,29ms	32,48ms
50 users	161,83ms	159,1ms
500 users	2,9s	2,91s
1000 users	8,56s	6,22s

Annex A contains the full results of the load tests.

Table 21 - Rate of failure of HTTP requests on load tests.

	Not Scaled	Scaled – 3 Replicas
1 user	0%	0%
50 users	0%	0%
500 users	0%	0%
1000 users	3,66%	0,02%

Figure 30 - Summary of the errors for 1.000 VU and no scalling load test.

Some of the errors were related to "Context deadline exceeded", which according to the official documentation, means k6 can send a request but the target system did not respond (k6.io, 2021a).

From Table 20 and Table 21, it is possible to conclude the following: on the tests with 1 VU, 50 VU and 500 VU, differences are residuals, only a few milliseconds difference. Yet, the main difference is in the load test with 1.000 virtual users, where scaling resulted in an improvement of around 2 seconds on the average HTTP request duration, and less 3,64 % of failed requests. Due to computational resource limitations, it was not possible to perform load tests in a higher number of virtual users, but it is expected to have better results as the numbers increase.

To perform the tests, only 4 components out of a total of 8 were needed to scale. The result is very satisfatory as only half of the components were needed for the feature to work, namely:

- Micro frontends: AppShell, Navbar and Purchases.
- Microservices: Purchases.

7.2.3 Performance

Performance experiments were performed using Google Lighthouse. The constraints used on the tests are *Clear Cache* and *Simulated Throttling*.
Clear Cache simulates opening the page without caching resources, and simulated throttling simulates a slower page loading based on data from the initial unthrottled load (lighthouse, 2021).
The tests were performed in *production mode*, as webpack provide several features to increase performance such as bundle minification (i.e., bundles decrease size) or lighter source maps (webpack.js.org, 2021). This change leads to a more faithful validation of performance, simulating a real production environment.

The test is performend on the main page. Table 22 shows a summary of the performance tests results, with different combinations of the constraints.

Table 22 - Summary of the performance tests results.

Test case	Rating
Only Simulated Throttling active	97 (green)
Only Clear cache active	91 (green)
No Constraints Applied	97 (green)
Clear Cache and Simulated Throttling active	23 (red)

Annex A shows a summary of the performance tests scores from Google Lighthouse and the full report can be seen on the GitHub repository (Silva, 2021), in HTML format.

The main issues in the performance audit are related to:
- Large network payloads in remoteEntry.js files, downloaded to import the code from other remote micro frontends.
- Lack of caching mechanisms in serving static assets (images, remoteEntry.js files, etc.).

- Poor image asset file handling - not properly sized, not lazy loading.
- JavaScript execution time, the web application relies on large amounts of JavaScript being executed, decreasing performance.

Image file handling issues can be fixed in code, whereas issues related to caching, network payloads and asset loading can be fixed using better caching mechanisms – this is the reason for degrading performance once the cache is cleared, resulting in poor performance on the first time page loading.

The fact the web page is heavily relying on JavaScript bundles, which can increase in size over time, can be a problem of the approach and work would have to be performed to solve these issues.

7.2.4 Testability

Testability experiments were performed using SonarQube. This experiment was conducted on the whole source code on each micro frontend, but with a focus on the file with the higher cyclomatic complexity, which its meaning was explained in Section 7.1. Table 23 shows a summary of the testability results.

Table 23 - Summary of testability results

Micro frontend	File	Cyclomatic complexity
Purchases	checkoutItems.tsx	12
Catalog	Catalog.tsx	6
Appshell	app.tsx	2
Cart	cartComponent.tsx	28
Navbar	Navbar.tsx	3

The results were overall acceptable as the max cyclomatic complexity on each file of the developed micro frontends is usually under 20, meaning lower or medium complexity. The unique case of high complexity is found on Cart micro frontend, on the file *CartComponent.tsx*.

Due to the higher level of logic on this component, it is difficult to test as a unit, and it would be necessary to split some of its logic to lower cyclomatic complexity, increasing testability.
In this particular scenario, it is possible to conclude that testability is mainly dependent on the code itself, at least while using the module federation technique. As the code can be tested separately as a unit, by making use of mocks to simulate the behavior of other components or integrating with the other micro frontends and microservices in an end-to-end manner, this allows testing the application the same way a monolith application would be tested.

7.3 Summary

The results from the experiences are considered overall acceptable.
On the scalability side, micro frontends allowing scaling each component separately is a major benefit, regardless of the response times. A scenario where only certain features have an increase in load by users' access peak at a certain point in time, allows the automatic scaling of specific components instead of the web application as a whole if a good separation of concerns is performed by design. This can be a major benefit in terms of costs and availability when dealing with large enterprise web applications.
Performance, on the other hand, is seen as a possible downside of this approach. Although not the focus on this document, performance is still a very important concern on a website, and if there is no use of caching mechanisms, for instance, performance (mainly first page loading time) can degrade on the default implementation of micro frontends, at least on a client-side composition approach. Considering the limited time for the development of this proof of concept, increasing the performance of the proof of concept was not a primary focus since the beginning. Thus, it should be possible to adapt the code to increase performance, and module federation likely provides tweaks for improving overall performance, as it is being enhanced day by day by the community. Anyway, this proof of concept serves as a demonstration of the functionality of module federation.
In terms of maintainability, the reduced number of lines of code in the source code of each component is a benefit. Moreover, a good separation of concerns also acts as an obligation of not coupling the micro frontends (although, ways of creating coupling, such as shared state, are possible but should be avoided) benefits on creating self-contained components, consequently increasing overall maintainability. Experienced developers should even be able to increase the

maintainability of each component as seen in microservices approaches on the backend side, with a focus on good separation of concerns and good code practices with the aid of tools such as SonarQube together with continuous delivery and continuous integration.

On the testability side, although there are fewer lines of code, if the code is not properly implemented, cyclomatic complexity increases and consequently, testability degrades. Regarding this proof of concept, mainly due to the author's experience on web development and the limited duration of the thesis, cyclomatic complexity is relatively high in some classes, and that should be a topic that requires special care in a real development scenario. While Micro frontends provide the benefit of having smaller overall codebases that should help in increasing testability and maintainability, the developers' experience on keeping a lower complexity in each file should be an important aspect taken into account.

Although the majority of documents found are related to the comparison between monolith and micro frontends, this document differs by focusing on analyzing a micro frontend solution in terms of quality attributes. It is possible to find on other documents the following conclusions on micro frontends (Kroiss, 2021):

- Micro frontends performance degrades on first page-load without caching resources.
- Although requiring increased effort to be implemented, the overall micro frontend solution is highly maintainable.
- Independent releases results in faster development cycles.

Comparing each conclusion with the conclusions taken from the quality attribute analysis, it is possible to agree on each statement:

- Performance analysis confirms that performance is slower when the cache is clear.
- High Maintainability is confirmed due to the high Maintainability Ranking obtained during maintainability analysis.
- Due to the high modularity resulted from having separate and self-contained micro frontends, it should possible to perform independent releases and faster development cycles.

In scenarios where performance is a main focus, a server-side composition approach could be a better option for micro frontends, despite having possible losses on usability for instance, as it would loose the benefits of a single-page application approach.

As an example, companies such as Amazon rely on a server-side rendering micro frontend approach, as performance is an important aspect (Kroiss, 2021).

8 Conclusions

8.1 Contributions

This document supplies contribution to the micro frontend architecture topic focusing on client-side composition, using a technique called Module Federation implemented with webpack version 5. The developed proof of concept used analyzed in this document provides a demonstration on the implementation with this technique using React and TypeScript language, as well as insights on several use cases such as dynamic import of remote applications using an app shell component, micro frontend communication, testing, and container orchestration. The document focus on evaluating the solution based on quality attributes, namely maintainability, performance efficiency, scalability, and testability.
Source code is available on GitHub (Silva, 2021), fully open source, for testing or continuing the work developed so far. The author grants permission to change or replicate the code according to the needs.

8.2 Challenges and Limitations

While developing this document, the author faced several challenges. Micro Frontends is a very recent topic, still being assessed by many authors and companies as a client-side composition

approach, where most of the use cases found are centered on server-side composition approaches.
Meanwhile, Module Federation, a technique implemented on webpack version 5 was released around the beginning of the development of this document, meaning the author was faced with an overall shortage of detailed documentation, scientific articles, and detailed examples of implementation on some of the faced use cases. This meant relying mostly on webpack official documentation or articles by few authors to understand the details and caveats of the implementation.
As the technique is still recent, there is still a shortage of reports about use cases of companies using it, which would be very important to assess its viability and drawbacks of its use in a real production environment.
Nevertheless, it is important to praise the dedication of several authors referenced in this document in spreading the word by creating enriching documentation such as books or articles, some of which are still to be released at this date, that provides important insights into the overall theme of micro frontends. This is unquestionably becoming a trending topic and good documentation is becoming easier to find over time.
Software Quality Attribute analysis documentation on micro frontends is another topic lacking documentation. Most of the micro frontend documentation is focusing on the actual implementation with some insights about the benefits and downsides. Usually, there is not a clear analysis of metrics to understand how good or bad a solution is quantitatively.
In terms of development and experimentation, due to several reasons such as the limited time available on the elaboration of the overall work or a single author without vast experience on web development, might end up as validation threats, as some of the results such as performance or testability might be slightly improved by focusing on following better code practices. Also, being able to develop a larger proof of concept with an increased number of features could as well impact the results.
Testing in a local environment and using open source or community licensed tools was a limitation found as well, mainly due to computational resource limitations on the local environment and some tools providing better features that are not available for free. Assessing the solution in an enterprise testing environment with licensed tools might help to provide more viable and reliable testing results, closer to a real production environment.

8.3 Achieved Goals

This document provides an insight on the current knowledge about micro frontends available on books and articles, such as techniques, patterns, frameworks, and their benefits or drawbacks translated into quality attributes. A proof of concept was developed using one of the latest client-side composition techniques for micro frontends, called Module Federation, where several use cases are tackled and there is an analysis of the solution based on the Maintainability, Performance Efficiency, Scalability, and Testability as initially planned. Due to limitations, a more intensive analysis was not possible in some of the Software Quality Attributes.

Nevertheless, the output of this document is an assessment of part of the benefits and possible caveats to pay attention to, also a good starting point when using of Module Federation and client-side composition.

8.4 Future Work

Over time, along with the increased usage of client-side techniques in an enterprise environment with large projects composed of numerous teams, it will be possible to better assess the real caveats on using them since the overall micro frontend topic is indicated mainly for large projects.

In parallel, several authors and teams are improving or creating new ways of delivering micro frontend solutions to increase productivity and allow for smoother migrations from monoliths to micro frontends on both codebase and team organization.

As future work, the author suggests ways to further develop the study on the topics assessed in this document:

- A new iteration of the work developed in this thesis, with new features added to the proof of concept and an attempt on studying different software quality attributes such as Security, for instance, being equally important and can provide enormous value on the topic.
- Further investigate themes such as dependency management or import maps, that should be able to be applied along with module federation as well.
- In an enterprise environment, migrate a monolith single-page application to a client-side composition micro frontend approach, and assess its benefits and caveats.

- Focusing on Module Federation, assessing this technique on a server-side composition approach.

8.5 Final remarks

The development of this thesis was shown very positive overall, due to the valuable knowledge acquired in web development, such as new approaches and techniques, and micro frontends in general, which is a very trending topic and of increased importance each day.

It was very satisfactory to learn more about architectural design as well, which is an area of interest and transversal on both frontend and backend development.

In retrospect, despite the demanding work needed to conclude this document and the limited experience in the web development area, it is very satisfactory to be able to reach the goals initially planned and to leave my comfort zone, by expanding my knowledge on different areas of software development as well as learning new technologies that certainly will be valuable for the future career path or personal fulfillment.

References

(Baldassarre et al.,2020)	Baldassarre, T., Lenarduzzi, V., Romani, S., Saarimäki, N. (2020) On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. *Information and Software Technology– Volume 128, December 2020, 106377*
(Belliveau, et al.,2002)	Belliveau, P., Griffin, A. and Somermeyer, S. (2002). The PDMA toolbook 1for new product development. *New York, N.Y.: John wiley & Sons, pp.2-21.*
(Bidelman, n.d.)	Bidelman, E. (n.d.). *Custom Elements v1: Reusable Web Components*. [online] Google Developers. Available at: https://developers.google.com/web/fundamentals/web-components/customelements#historysupporthttps://developers.google.com/web/fundamentals/web-components/customelements [Accessed 8 Dec. 2020].
(Blixt, 2013)	Blixt, A. (2013). *What is the technology stack behind the Spotify web client?*. [online] Quora. Available at: https://www.quora.com/What-is-the-technology-stack-behind-the-Spotify-web-client/answer/Andreas-Blixt [Accessed 6 Dec. 2020].
(Bvaughn, 2021)	Bvaughn-github (n.d.). *Github - react-error-boundary* [online] Available at: https://github.com/bvaughn/react-error-boundary#readme [Accessed 2 Jun. 2021].
(Calabrese et al., 2018)	Calabrese, J., Muñoz, R., Ariel, P., Silvia, E. (2017) Assistant for the Evaluation of Software Product Quality Characteristics Proposed by ISO/IEC 25010 Based on GQM-Defined Metrics. *Computer Science – CACIC 2017 (pp.164-175)*
(CodeTalks, 2017)	CodeTalks (2017). *code.talks 2017 - The Recipe For Scalable Frontends (Zalando)*. [image] Available at: https://www.youtube.com/watch?v=m32EdvitXy4&feature=youtu.be [Accessed 13 Dec. 2020].

(Colin, 2018)	Colin, J. (2018). *Fragments: limitations, solutions and our approach*. [online] Jobs.zalando.com. Available at: https://jobs.zalando.com/en/tech/blog/front-end-micro-services/?gh_src=4n3gxh1 [Accessed 17 Dec. 2020].
(Convective, 2018)	Convective (2018). *AngularJS End of Life Announced | Migration and Upgrade Strategies*. [online] Available at: https://www.convective.com/angularjs-end-of-life/ [Accessed 5 Feb. 2021].
(Dhiman, 2019)	Dhiman, R. (2019). *Micro Frontends Architecture*. [online] App.pluralsight.com. Available at: https://app.pluralsight.com/library/courses/micro-frontends-architecture/ [Accessed 24 Oct. 2020].
(digital.ai, 2020)	Digital.ai (2020). *14th annual State of Agile Report* [online] Available at: https://stateofagile.com/# [Accessed 20 Feb. 2021].
(docs.sonarqube.org, 2021)	docs.sonarqube.org (2021). *Metric Definitions* [online] Available at: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ [Accessed 22 Jun. 2021].
(Dodson, n.d.)	Dodson, R. (n.d.). *Custom Elements Everywhere*. [online] Custom-elements-everywhere.com. Available at: https://custom-elements-everywhere.com/ [Accessed 2 Jan. 2021].
(Doodle, 2019)	Doodle. (2019). *Doodle | The state of meetings report*. [online] Available at: https://meeting-report.com [Accessed 5 Feb. 2021].
(Facebook.com, 2010)	Facebook.com. (2010). *BigPipe: Pipelining web pages for high performance*. [online] Available at: https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/ [Accessed 17 Dec. 2020].
(freeCodeCamp.org, 2021)	freeCodeCamp.org (2021). *Build a Shopping Cart with React and TypeScript - Tutorial* [video] Available at: https://www.youtube.com/watch?v=sfmL6bGbiN8 [Accessed 27 May 2021].

(Gan et al. 2005)	Gan, Z., Wei, D. and Varadharajan, V. (2005). Evaluating the Performance and Scalability of Web Application systems. *Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05)*
(Geers, 2019)	Geers, M. (2019). *Michael Geers - Micro Frontends - The Nitty Gritty Details or Frontend, Backend, Happyend*. [online] Youtube.com. Available at: https://www.youtube.com/watch?v=wCHYILvM7kU [Accessed 24 Oct. 2020].
(Geers, 2020)	Geers, M. (2020). *Micro Frontends in Action*. Manning Publication.
(Geers, n.d.)	Geers, M. (n.d.). *Micro Frontends - extending the microservice idea to frontend development*. [online] Micro Frontends. Available at: https://micro-frontends.org/ [Accessed 11 Dec. 2020].
(Gidlund, 2016)	Gidlund, M. (2016). Measuring feature team characteristics of software development teams. *School of Computer Science and Engineering - Royal Institute of Technology.*
(GitHub-Piral, n.d)	GitHub-Piral. (n.d.). *smapiot/piral*. [online] Available at: https://github.com/smapiot/piral/blob/develop/docs/features.md [Accessed 17 Dec. 2020].
(Gouigoux and Tamzalit, 2017)	Gouigoux, J. and Tamzalit, D. (2017). From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp.62-65.
(Herrington, 2020a)	Herrington, J. (2020). *Introducing Federated Modules in Webpack 5.* [online] Youtube.com. Available at: https://www.youtube.com/watch?v=D3XYAx30CNc [Accessed 27 Dec. 2020].
(Herrington, 2020b)	Herrington J. (2020). *Micro-FEs Simplified* [online] dev.to Available at URL https://dev.to/jherr/micro-fes-simplified-5ac6 [Accessed 20 Dec. 2020].
(Herrington, 2020c)	Herrington, J. (2020). *Zack Jackson - Module Federation* [online] Youtube.com. Available at: https://www.youtube.com/watch?v=AU7dKWNfWiA [Accessed 19 Dec. 2020].

(ISO25000, 2020) ISO25000 (2020). *ISO/IEC 25010* [online] Available at: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 [Accessed 10 Feb. 2021].

(Jackson, 2019) Jackson, C. (2019). *Micro Frontends*. [online] martinfowler.com. Available at: https://martinfowler.com/articles/micro-frontends.html [Accessed 5 Dec. 2020].

(Jackson, 2020) Jackson Z. (2020). *Webpack 5 Federation. A Game-changer to Javascript architecture.* [online] indepth.dev Available at URL https://indepth.dev/posts/1173/webpack-5-module-federation-a-game-changer-in-javascript-architecture [Accessed 28 Dec. 2020].

(Johansson, 2015) Johansson, M. (2015). *How is JavaScript used within the Spotify desktop application? Is it packaged up and run locally only retrieving the assets as and when needed? What JavaScript VM is used?*. [online] Quora. Available at: https://www.quora.com/How-is-JavaScript-used-within-the-Spotify-desktop-application-Is-it-packaged-up-and-run-locally-only-retrieving-the-assets-as-and-when-needed-What-JavaScript-VM-is-used/answer/Mattias-Petter-Johansson [Accessed 7 Dec. 2020].

(k6.io, 2021a) K6.io (2021). *Testing guides – Running large tests* [online] Available at: https://k6.io/docs/testing-guides/running-large-tests/ [Accessed 22 Jun. 2021].

(k6.io, 2021b) K6.io (2021). *Metrics* [online] Available at: https://k6.io/docs/using-k6/metrics/ [Accessed 22 Jun. 2021].

(Kalagara, 2020) Kalagara, S. (2020) Cyclomatic Complexity in Software Development. *NCSMSD – 2020 Conference Proceedings*.

(Keikavousi, 2021) Keikavousi, M (2021). *Fake Store API* [online] Available at https://fakestoreapi.com/ [Accessed 28 May 2021].

(Koen, et al., 2014) Koen, P., Bertels, H. and Kleinschmidt, E. (2014). Managing the Front End of Innovation—Part I. *Research-Technology Management*, 57(2), pp.34-43.

(Kotte, 2017) Kotte, G. (2017). *Øredev 2017 - Gustaf Nilsson Kotte - Microservice Websites*. [video] Available at: https://www.youtube.com/watch?v=j2ynHColelw [Accessed 10 Dec. 2020].

(Kroiss, 2021) Kroiss, M. (2021) From Backend to Frontend - Case study on adopting Micro Frontends from a Single Page ERP Application monolith.

(Kropp, et al., 1998) Kropp, N., Koopman, P. and Siewiorek, D. (1998). Automated Robustness Testing of Off-the-Shelf Software Components. *FTCS-28, The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*.

(Larman and Vodde, 2009) Larman, C. and Vodde, B. (2009). *Scaling lean & agile development*. Upper Saddle River, NJ: Addison-Wesley, pp.150-192.

(Leithead and Eicholz, 2015) Leithead, T. and Eicholz, A. (2015). *Bringing componentization to the web: An overview of Web Components*. [online] Windows Blogs. Available at: https://blogs.windows.com/msedgedev/2015/07/14/bringing-componentization-to-the-web-an-overview-of-web-components/ [Accessed 18 Dec. 2020].

(Less.works, n.d.) Less.works. (n.d.). *Feature Teams - Large Scale Scrum (LeSS)*. [online] Available at: https://less.works/less/structure/feature-teams.html [Accessed 27 Dec. 2020].

(lighthouse, 2021) lighthouse-github (n.d.). *Github – lighthouse - Network Throttling* [online] Available at: https://github.com/GoogleChrome/lighthouse/blob/master/docs/throttling.md#devtools-lighthouse-panel-throttling [Accessed 28 Jun. 2021].

(Luigi-project.io, n.d.) Luigi-Project (n.d.). *Luigi - The Enterprise-Ready Micro Frontend Framework* [online] Available at: https://luigi-project.io/ [Accessed 30 Dec. 2020].

(Manfred, 2020) Manfred S. (2020). *Getting Out of Version-Mismatch-Hell with Module Federation* [online] Available at URL https://www.angulararchitects.io/aktuelles/getting-out-of-version-mismatch-hell-with-module-federation/ [Accessed 23 Dec. 2020].

(MDN, 2021) MDN (2021). *What is web performance?* [online] Available at: https://developer.mozilla.org/en-US/docs/Learn/Performance/What_is_web_performance [Accessed 19 Feb. 2021].

(Mezzalira, 2019) Mezzalira, L. (2019). *Micro Frontend Architecture - Luca Mezzalira, DAZN*. [video] Available at: https://www.youtube.com/watch?v=BuRB3djraeM [Accessed 18 Dec. 2020].

(Mezzalira, 2020)	Mezzalira, L (2020). *Micro-frontends in context.* [online] Available at: https://increment.com/frontend/micro-frontends-in-context/ [Accessed 20 Feb. 2021].
(Milanovic and Malek, 2004)	Milanovic, N. and Malek, M. (2004). Current solutions for Web service composition. *IEEE Internet Computing*, 8(6), pp.51-59.
(Mrowetz, 2020)	Mrowetz M. (2020). *Module Federation—Federated Application Architectures* [online] rangle.io Available at URL https://rangle.io/blog/module-federation-federated-application-architectures/ [Accessed 28 Dec. 2020].
(Newman, 2015)	Newman, S. (2015). *Pattern: Backends For Frontends*. [online] Samnewman.io. Available at: https://samnewman.io/patterns/architectural/bff/ [Accessed 11 Dec. 2020].
(Newman, 2019)	Newman, S. (2019). Monolith to Microservices - Evolutionary Patterns to Transform Your Monolith. *1st ed. Sebastopol, CA: O'Reilly Media, Inc., pp.98-104.*
(Nicola, 2019)	Nicola, S. (2019). Método de Análise Hierárquica. *Instituto Superior de Engenharia do Porto, 2019.*
(Nicola, et al., 2012)	Nicola, S., Ferreira, E. and Ferreira, J. (2012). A Novel Framework For Modeling Value For The Customer, An Essay On Negotiation. *International Journal of Information Technology & Decision Making, 11(03), pp.661-703*.
(Oh, Ahn and Kim, 2017)	Oh, J., Ahn, W. and Kim, T. (2017). Web app restructuring based on shadow DOMs to improve maintainability. *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS).*
(Osterwalder & Pigneur, 2003)	A.Osterwalder & Y.Pigneur, (2003). Modeling Value Propositions in E-Business. *Proceedings of the 5th International Conference on Electronic Commerce, ICEC 2003, Pittsburgh, Pennsylvania, USA.*
(Peffers, et al., 2006)	Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V. and Bragge, J. (2006). The design science research process: A model for producing and presenting information systems research. *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST 2006*, pp.84 - 92.

(Peffers, et al., 2007)	Peffers, K., Tuunanen, T., Rothenberger, M. A. and Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information System*, 24(3), pp.45 - 78.
(Peltonen, et al., 2020)	Peltonen, S., Mezzalira, L. and Taibi, D. (2020) Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology - 2020*
(Plotnicki, 2015)	Plotnicki, L. (2015). *BFF @ SoundCloud*. [online] ThoughtWorks. Available at: https://www.thoughtworks.com/insights/blog/bff-soundcloud [Accessed 21 Jan. 2021].
(Podvezko, 2009)	Podvezko, V. (2009). *Application of AHP technique*. Journal of Business Economics and Management, 10(2), pp.181-189.
(Porter, 1985)	Porter, M. (1985). Competitive advantage. *New York: Free Press, pp.36-48.*
(Pressman, 2014)	Pressman, R., (2014). Software Engineering: A Practitioner's Approach, 8th Ed, p.503-505*. Boston, Mass.: McGraw Hill.*
(ProfessionalQA, 2020)	ProfessionalQA (2020). *Software Testability* [online] Available at: https://www.professionalqa.com/software-testability [Accessed 20 Feb. 2021].
(PuzzleJS, 2020)	GitHub - PuzzleJS. (n.d.). *puzzle-js/puzzle-js*. [online] Available at: https://github.com/puzzle-js/puzzle-js [Accessed 14 Dec. 2020].
(ReactJS, 2021)	ReactJS (2021). *ReactJS Documentation – Error Boundaries* [online] Available at https://reactjs.org/docs/error-boundaries.html [Accessed 2 Jun. 2021].
(Richardson, n.d.)	Richardson, C. (2020). *Pattern: API Gateway / Backends for Frontends* [online] Available at: https://microservices.io/patterns/apigateway.html [Accessed 10 Feb. 2021].
(Rousos et al., 2021)	Rousos, M., Wagner, B., Torre, C. (2021). *The API gateway pattern versus the Direct client-to-microservice communication* [online] Available at: docs.microsoft.comUS/docs/Learn/Performance/What_is_web_performance [Accessed 19 Feb. 2021].
(Saaty, 1990)	Saaty, T. (1990). How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1), pp.9-26.

(Section.io, 2020)	Section.io (2021). *Scaling Horizontally vs. Scaling Vertically* [online] Available at: https://www.section.io/blog/scaling-horizontally-vs-vertically/ [Accessed 15 Feb. 2021].
(Senders, 2017)	Senders, P. (2017). *Front-end Microservices at Hello Fresh* [online] HelloTech Available at: https://engineering.hellofresh.com/front-end-microservices-at-hellofresh-23978a611b87 [accessed 29 Dec. 2020].
(Silva, 2021)	Silva, R. (2021) Github – *A micro frontend solution – analyzing quality attributes – source code*. Available at: https://github.com/ricsilva93/micro-frontends-solution-analyzing-quality-attributes [accessed 29 Jul. 2021].
(Simeonova, 2020)	Simeonova A. (2020). *Introduction to Luigi* [online] SAP – Developers Available at: https://developers.sap.com/tutorials/luigi-getting-started.html [accessed 30 Dec. 2020].
(Smartsheet, n.d.)	Smartsheet. (n.d.). *The Complete Guide to Value Chain Modeling | Smartsheet*. [online] Available at: https://www.smartsheet.com/value-chain-model [Accessed 3 Jan. 2021].
(Stenberg, 2018)	Stenberg, J. (2018). *Experiences Using Micro Frontends at IKEA*. [online] *InfoQ*. Available at: https://www.infoq.com/news/2018/08/experiences-micro-frontends/?utm_campaign=infoq_content&utm_source=twitter&utm_medium=feed&utm_term=architecture-design [Accessed 8 Dec. 2020].
(Steyer, 2019)	Steyer, M. (2019*). Enterprise Angular*. Leanpub.
(The Software House, 2020)	The Software House (2020*). State of Microservices 2020* [online] Available at: https://tsh.io/state-of-microservices/#ebook [Accessed 20 Feb. 2021].
(Thoughtworks, 2019)	Thoughtworks (2019). *Micro frontends | Technology Radar | ThoughtWorks.* [online] Available at: https://www.thoughtworks.com/radar/techniques/micro-frontends [Accessed 4 Nov. 2020].
(Thoughtworks, 2020)	Thoughtworks (2019). *Micro frontend anarchy | Technology Radar | ThoughtWorks.* [online] Available at: https://www.thoughtworks.com/radar/techniques/micro-frontend-anarchy [Accessed 19 Nov. 2020].

(Visual Studio Docs, 2021)	Visual Studio Docs (2020). *Code metrics - Maintainability index range and meaning* [online] Available at: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2019 [Accessed 20 Feb. 2021].
(Visual-paradigm.com, 2020)	Visual-paradigm.com. (2020). *Feature Team vs Component Team in Agile.* [online] Available at: https://www.visual-paradigm.com/scrum/feature-team-vs-component-team-in-agile/ [Accessed 7 Dec. 2020].
(Webcomponents.org, n.d.)	Webcomponents.org. (n.d.). webcomponents.org - *Discuss & share web components.* [online] Available at: https://www.webcomponents.org/specs [Accessed 12 Dec. 2020].
(webpack.js.org, 2021)	webpack.js.org (2021). *Documentation - Production* [online] Available at: https://webpack.js.org/guides/production/ [Accessed 25 Jun. 2021].
(Woodroof, 1997)	Woodruff, R. (1997). Customer value: The next source for competitive advantage. *Journal of the Academy of Marketing Science*, 25(2), pp.139-153.
(Yang et al., 2019)	Yang, C., Liu, C. and Su, Z. (2019). Research and Application of Micro Frontends*. IOP Conference Series: Materials Science and Engineering, 490, p.062082.*
(Zeithaml, 1988)	Zeithaml, V. (1988). Consumer Perceptions of Price, Quality, and Value: A Means-End Model and Synthesis of Evidence. *Journal of Marketing, 52(3), pp.2-22.*

Annex A – Solution Experimentation Results

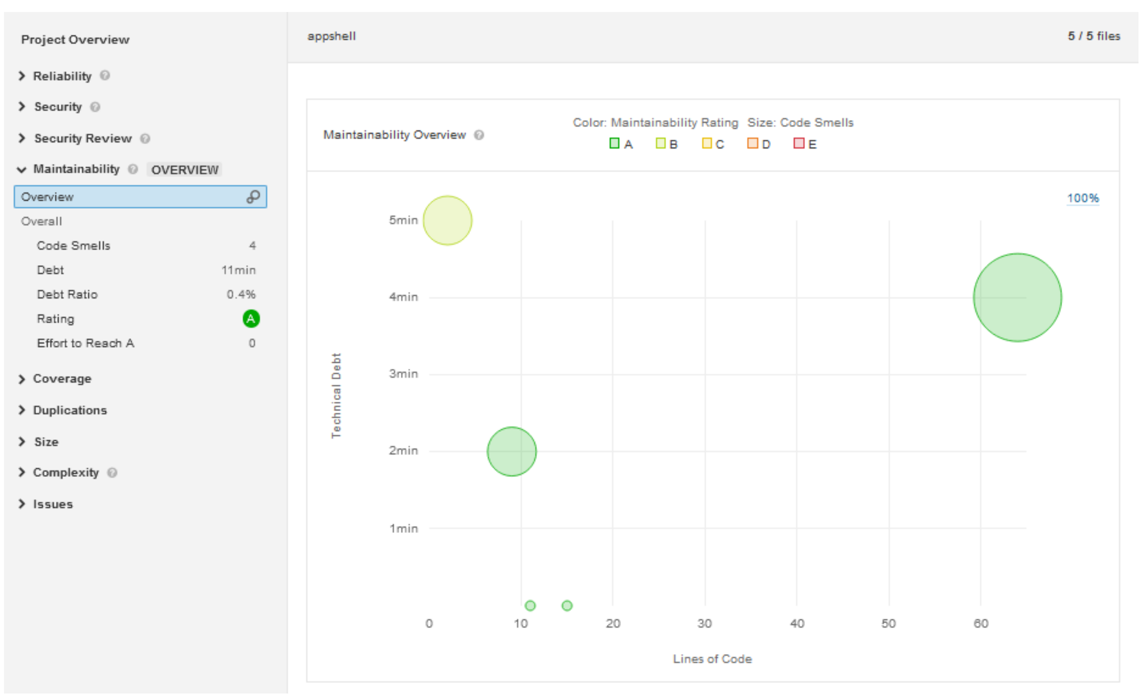


Figure 31 - Summary of Maintainability Rating for AppShell Micro Frontend.

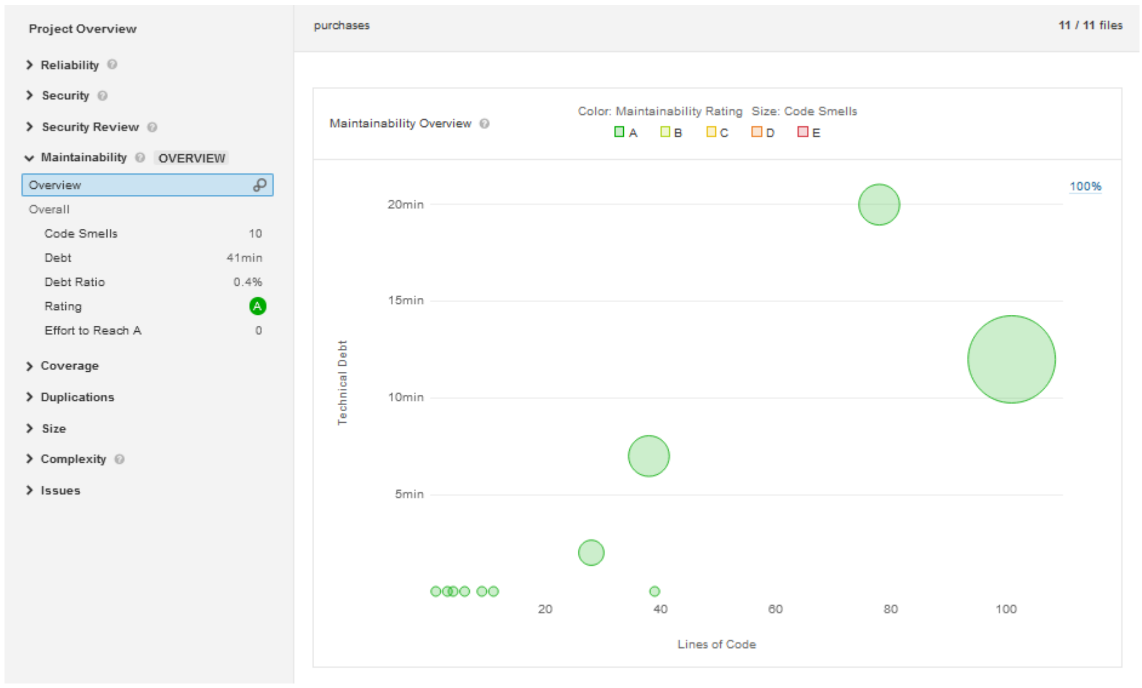


Figure 32 - Summary of Maintainability Rating for Purchases Micro Frontend.

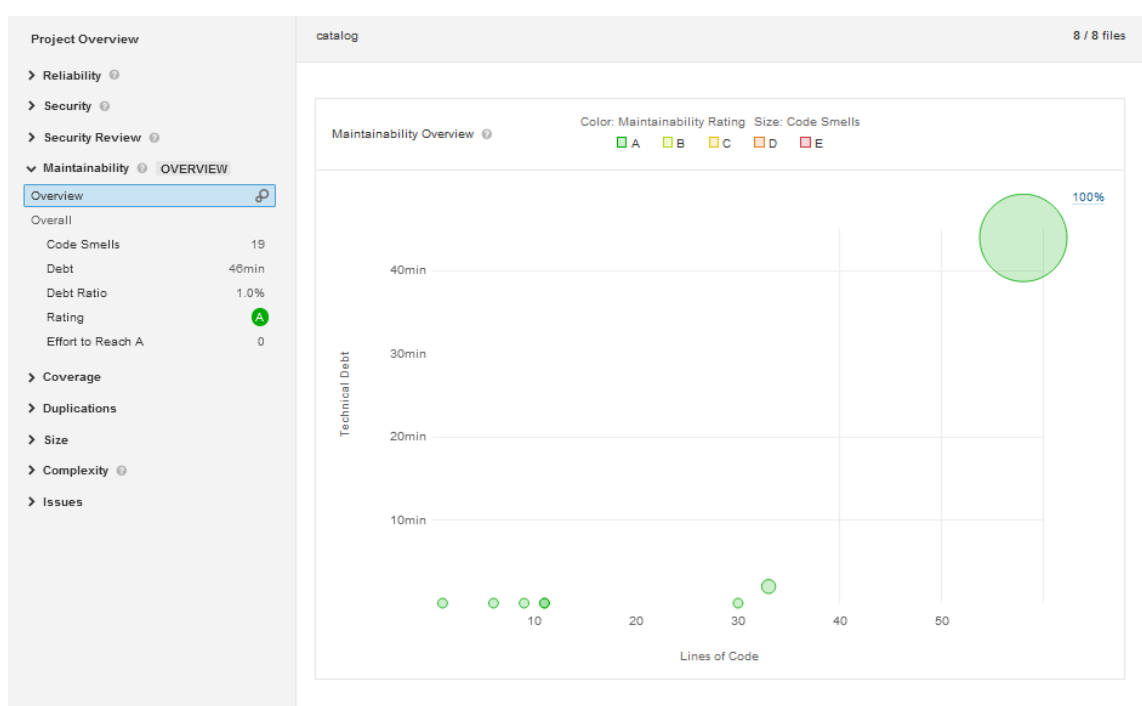


Figure 33 - Summary of Maintainability Rating for Catalog Micro Frontend.

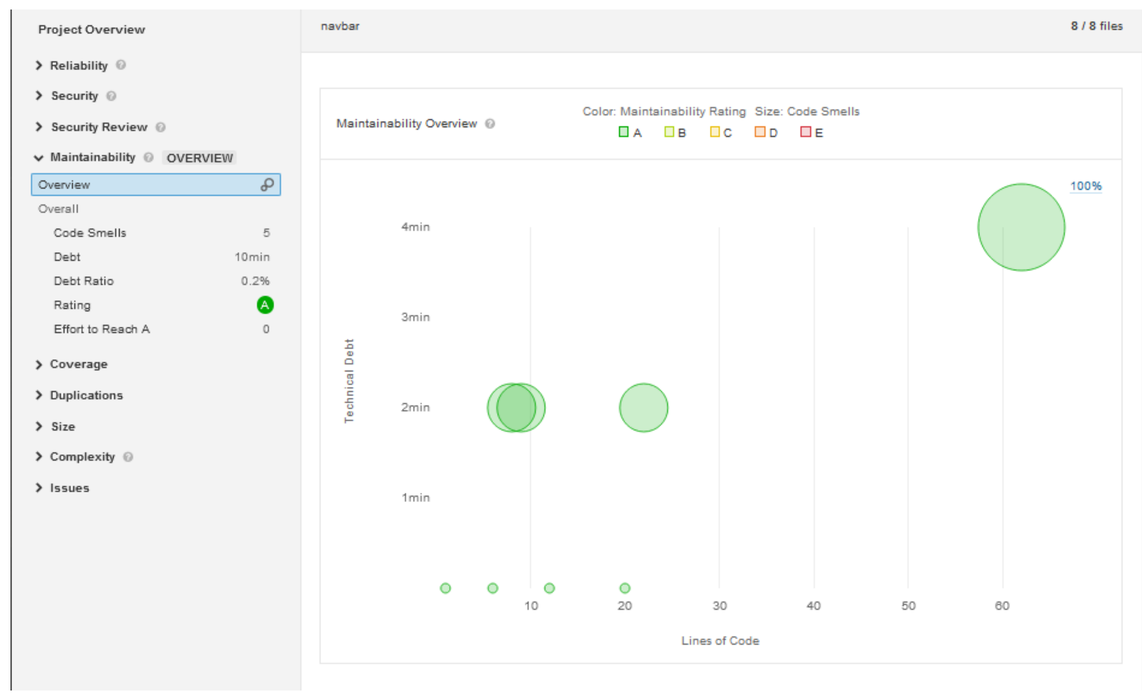


Figure 34 - Summary of Maintainability Rating for Navbar Micro Frontend.

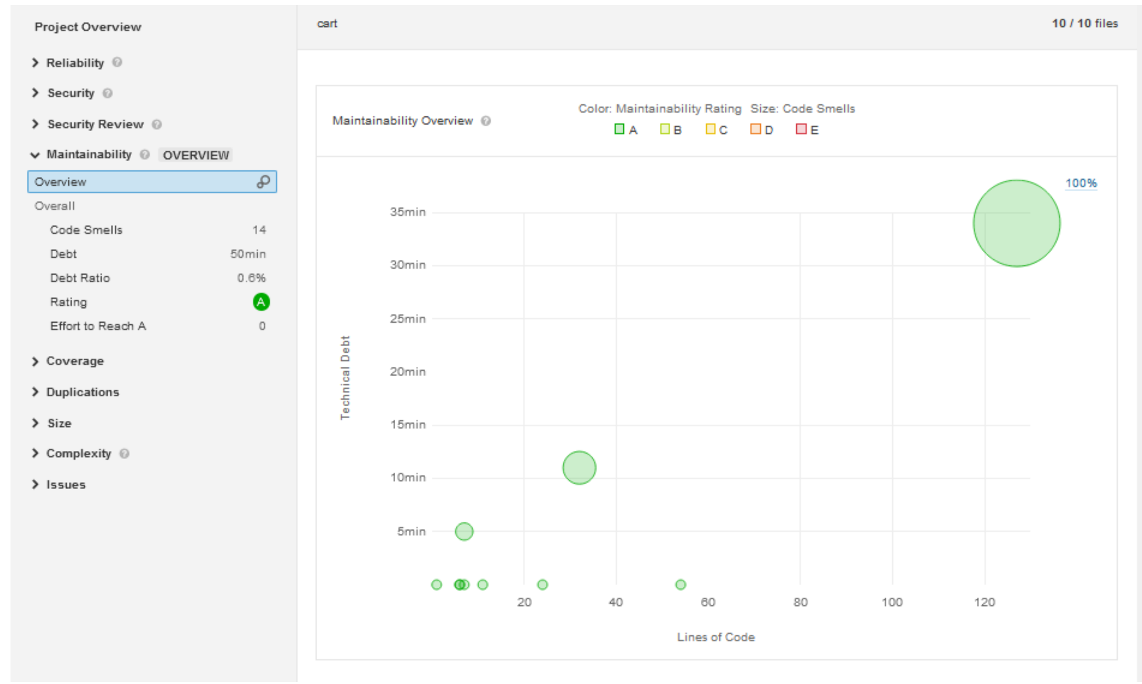


Figure 35 - Summary of Maintainability Rating for Cart Micro Frontend.

```
data_received..................: 22 MB  18 MB/s
data_sent......................: 3.8 kB 3.0 kB/s
group_duration.................: avg=275.21ms min=275.21ms med=275.21ms max=275.21ms p(90)=275.21ms p(95)=275.21ms
http_req_blocked...............: avg=108.45µs min=2.1µs    med=3.3µs    max=330.1µs  p(90)=270.34µs p(95)=300.22µs
http_req_connecting............: avg=73.4µs   min=0s       med=0s       max=237.6µs  p(90)=165.68µs p(95)=201.64µs
http_req_duration..............: avg=30.29ms  min=1.51ms   med=4.56ms   max=167.2ms  p(90)=72.42ms  p(95)=119.81ms
  { expected_response:true }...: avg=30.29ms  min=1.51ms   med=4.56ms   max=167.2ms  p(90)=72.42ms  p(95)=119.81ms
http_req_failed................: 0.00%  ✓ 0    ✗ 9
http_req_receiving.............: avg=23.44ms  min=67.7µs   med=133µs    max=144.88ms p(90)=60.9ms   p(95)=102.89ms
http_req_sending...............: avg=57.34µs  min=13.1µs   med=46µs     max=106.9µs  p(90)=99.78µs  p(95)=103.34µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=6.78ms   min=1.37ms   med=4.31ms   max=22.22ms  p(90)=13.76ms  p(95)=17.99ms
http_reqs......................: 9      7.046144/s
iteration_duration.............: avg=1.27s    min=1.27s    med=1.27s    max=1.27s    p(90)=1.27s    p(95)=1.27s
iterations.....................: 1      0.782905/s
vus............................: 1      min=1 max=1
vus_max........................: 1      min=1 max=1
```

Figure 36 - Scalability experiment with 1 virtual user and no scaling performed.

```
data_received..................: 13 GB  298 MB/s
data_sent......................: 2.1 MB 51 kB/s
group_duration.................: avg=1.45s    min=243.35ms med=1.6s    max=2.73s    p(90)=2.23s    p(95)=2.33s
http_req_blocked...............: avg=13.23µs  min=1.3µs    med=4µs     max=2.51ms   p(90)=7.5µs    p(95)=21.63µs
http_req_connecting............: avg=5.74µs   min=0s       med=0s      max=2.38ms   p(90)=0s       p(95)=0s
http_req_duration..............: avg=161.83ms min=943.4µs  med=9.39ms  max=1.77s    p(90)=397.24ms p(95)=1.21s
  { expected_response:true }...: avg=161.83ms min=943.4µs  med=9.39ms  max=1.77s    p(90)=397.24ms p(95)=1.21s
http_req_failed................: 0.00%  ✓ 0      ✗ 5067
http_req_receiving.............: avg=149.12ms min=13.8µs   med=96.3µs  max=1.71s    p(90)=375.5ms  p(95)=1.17s
http_req_sending...............: avg=33.08µs  min=7.7µs    med=23.4µs  max=9.71ms   p(90)=56.1µs   p(95)=71.03µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=12.67ms  min=870.2µs  med=8.35ms  max=117.1ms  p(90)=30.43ms  p(95)=41.31ms
http_reqs......................: 5067   119.993211/s
iteration_duration.............: avg=2.45s    min=1.24s    med=2.6s    max=3.73s    p(90)=3.23s    p(95)=3.33s
iterations.....................: 563    13.332579/s
vus............................: 16     min=6  max=50
vus_max........................: 50     min=50 max=50
```

Figure 37 - Scalability experiment with 50 virtual users and no scaling performed.

```
data_received..................: 17 GB  277 MB/s
data_sent......................: 2.8 MB 47 kB/s
group_duration.................: avg=26.14s   min=630.47ms med=29.92s   max=43.47s   p(90)=39s      p(95)=40.65s
http_req_blocked...............: avg=122.02µs min=1.1µs    med=7.1µs    max=13.29ms  p(90)=243.82µs p(95)=342.6µs
http_req_connecting............: avg=89.76µs  min=0s       med=0s       max=13.25ms  p(90)=168.92µs p(95)=253.54µs
http_req_duration..............: avg=2.9s     min=967.8µs  med=41.11ms  max=35.56s   p(90)=6.41s    p(95)=25.01s
  { expected_response:true }...: avg=2.9s     min=967.8µs  med=41.11ms  max=35.56s   p(90)=6.41s    p(95)=25.01s
http_req_failed................: 0.00%  ✓ 0      ✗ 6660
http_req_receiving.............: avg=2.8s     min=12.2µs   med=80.65µs  max=35.49s   p(90)=6.32s    p(95)=23.91s
http_req_sending...............: avg=274.6µs  min=6.3µs    med=43µs     max=899.62ms p(90)=96.01µs  p(95)=124.8µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=94.54ms  min=899.7µs  med=29.86ms  max=2.9s     p(90)=115.8ms  p(95)=230.44ms
http_reqs......................: 6660   111.31512/s
iteration_duration.............: avg=27.14s   min=1.63s    med=30.92s   max=44.47s   p(90)=40s      p(95)=41.65s
iterations.....................: 740    12.368347/s
vus............................: 157    min=21  max=500
vus_max........................: 500    min=500 max=500
```

Figure 38 - Scalability experiment with 500 virtual users and no scaling performed.

```
data_received..................: 20 GB   278 MB/s
data_sent......................: 2.2 MB  31 kB/s
group_duration.................: avg=30.07s    min=1.6s     med=20.85s    max=1m7s      p(90)=1m3s      p(95)=1m4s
http_req_blocked...............: avg=155.9µs   min=1.4µs    med=132.9µs   max=8.72ms    p(90)=273.24µs  p(95)=366.9µs
http_req_connecting............: avg=106.75µs  min=0s       med=76.1µs    max=8.58ms    p(90)=177.54µs  p(95)=262.83µs
http_req_duration..............: avg=8.56s     min=1.14ms   med=254.49ms  max=1m0s      p(90)=46.77s    p(95)=53.17s
  { expected_response:true }...: avg=6.62s     min=1.14ms   med=218.4ms   max=59.73s    p(90)=36.83s    p(95)=48.94s
http_req_failed................: 3.66%   ✓ 137     ✗ 3606
http_req_receiving.............: avg=8.26s     min=14.2µs   med=146.88ms  max=59.95s    p(90)=44.28s    p(95)=51.82s
http_req_sending...............: avg=369.06µs  min=8µs      med=61.5µs    max=835.48ms  p(90)=113.68µs  p(95)=142.47µs
http_req_tls_handshaking.......: avg=0s        min=0s       med=0s        max=0s        p(90)=0s        p(95)=0s
http_req_waiting...............: avg=300.81ms  min=1.04ms   med=26.15ms   max=3.21s     p(90)=1.08s     p(95)=2.56s
http_reqs......................: 3743    53.410098/s
iteration_duration.............: avg=30.95s    min=2.61s    med=21.13s    max=1m8s      p(90)=1m4s      p(95)=1m5s
iterations.....................: 94      1.341317/s
vus............................: 958     min=38    max=1000
vus_max........................: 1000    min=1000 max=1000
```

Figure 39 - Scalability experiment with 1.000 virtual users and no scaling performed.

```
data_received..................: 22 MB   17 MB/s
data_sent......................: 3.8 kB  2.9 kB/s
group_duration.................: avg=294.76ms min=294.76ms med=294.76ms max=294.76ms p(90)=294.76ms p(95)=294.76ms
http_req_blocked...............: avg=121.25µs min=1.8µs    med=3µs      max=378.7µs  p(90)=310.94µs p(95)=344.81µs
http_req_connecting............: avg=80.57µs  min=0s       med=0s       max=208.7µs  p(90)=191.34µs p(95)=200.01µs
http_req_duration..............: avg=32.48ms  min=1.66ms   med=5.37ms   max=177.93ms p(90)=76.17ms  p(95)=127.05ms
  { expected_response:true }...: avg=32.48ms  min=1.66ms   med=5.37ms   max=177.93ms p(90)=76.17ms  p(95)=127.05ms
http_req_failed................: 0.00%   ✓ 0      ✗ 9
http_req_receiving.............: avg=25.39ms  min=68.7µs   med=134.6µs  max=155.18ms p(90)=65.27ms  p(95)=110.23ms
http_req_sending...............: avg=40.74µs  min=14.2µs   med=40.9µs   max=79µs     p(90)=62.92µs  p(95)=70.96µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=7.04ms   min=1.51ms   med=5.21ms   max=22.69ms  p(90)=13.63ms  p(95)=18.16ms
http_reqs......................: 9       6.942255/s
iteration_duration.............: avg=1.29s    min=1.29s    med=1.29s    max=1.29s    p(90)=1.29s    p(95)=1.29s
iterations.....................: 1       0.771362/s
vus............................: 1       min=1 max=1
vus_max........................: 1       min=1 max=1
```

Figure 40 - Scalability experiment with 1 virtual user and scaling up to 3 replicas.

```
data_received..................: 13 GB   301 MB/s
data_sent......................: 2.2 MB  51 kB/s
group_duration.................: avg=1.43s     min=237.38ms med=1.49s    max=2.54s     p(90)=2.22s     p(95)=2.29s
http_req_blocked...............: avg=15.35µs   min=1.5µs    med=3.8µs    max=12.59ms   p(90)=7.2µs     p(95)=21.5µs
http_req_connecting............: avg=5.68µs    min=0s       med=0s       max=933.4µs   p(90)=0s        p(95)=0s
http_req_duration..............: avg=159.1ms   min=864.8µs  med=8.84ms   max=1.84s     p(90)=394.5ms   p(95)=1.08s
  { expected_response:true }...: avg=159.1ms   min=864.8µs  med=8.84ms   max=1.84s     p(90)=394.5ms   p(95)=1.08s
http_req_failed................: 0.00%   ✓ 0       ✗ 5094
http_req_receiving.............: avg=146.41ms  min=14.9µs   med=93.7µs   max=1.74s     p(90)=373.23ms  p(95)=1.04s
http_req_sending...............: avg=29.32µs   min=6.9µs    med=22µs     max=304.9µs   p(90)=55.26µs   p(95)=69.8µs
http_req_tls_handshaking.......: avg=0s        min=0s       med=0s       max=0s        p(90)=0s        p(95)=0s
http_req_waiting...............: avg=12.66ms   min=794µs    med=8.17ms   max=762.55ms  p(90)=29.28ms   p(95)=36.08ms
http_reqs......................: 5094    121.086058/s
iteration_duration.............: avg=2.43s     min=1.23s    med=2.49s    max=3.54s     p(90)=3.22s     p(95)=3.29s
iterations.....................: 566     13.454006/s
vus............................: 7       min=6  max=50
vus_max........................: 50      min=50 max=50
```

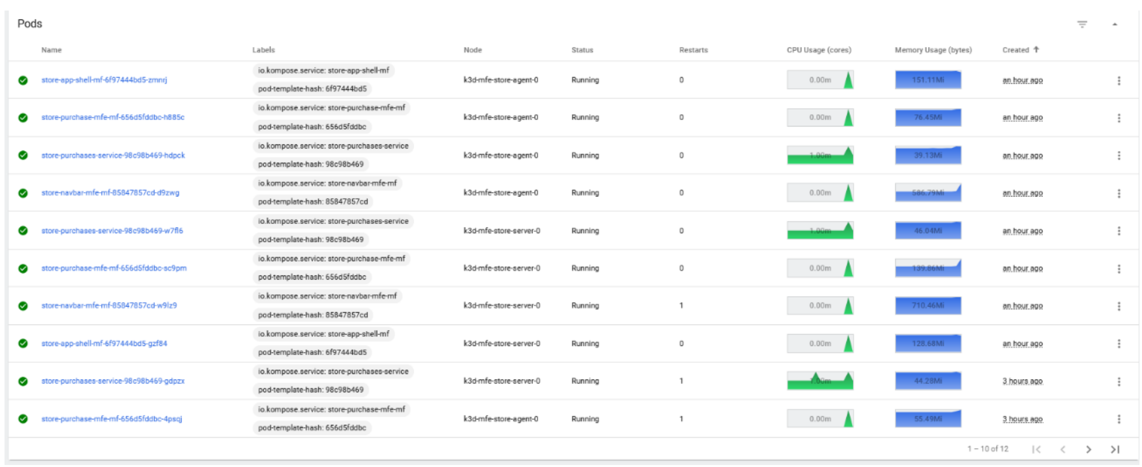
Figure 41 - Scalability experiment with 50 virtual users and scaling up to 3 replicas.

```
data_received..................: 17 GB   237 MB/s
data_sent......................: 2.8 MB  40 kB/s
group_duration.................: avg=26.19s    min=763.14ms med=30.9s    max=47.59s   p(90)=36.53s    p(95)=38.05s
http_req_blocked...............: avg=131.44µs  min=1.1µs    med=11.8µs   max=10.98ms  p(90)=241.99µs  p(95)=343µs
http_req_connecting............: avg=94.35µs   min=0s       med=0s       max=10.85ms  p(90)=162.46µs  p(95)=249.55µs
http_req_duration..............: avg=2.91s     min=981.4µs  med=33.66ms  max=42.81s   p(90)=7.56s     p(95)=23.38s
  { expected_response:true }...: avg=2.91s     min=981.4µs  med=33.66ms  max=42.81s   p(90)=7.56s     p(95)=23.38s
http_req_failed................: 0.00%   ✓ 0       ✗ 6672
http_req_receiving.............: avg=2.87s     min=13µs     med=81.8µs   max=42.74s   p(90)=7.52s     p(95)=23.29s
http_req_sending...............: avg=73.83µs   min=6.7µs    med=44.8µs   max=59.89ms  p(90)=96.3µs    p(95)=120.58µs
http_req_tls_handshaking.......: avg=0s        min=0s       med=0s       max=0s       p(90)=0s        p(95)=0s
http_req_waiting...............: avg=33.95ms   min=801.3µs  med=25.34ms  max=218.46ms p(90)=72.8ms    p(95)=95.44ms
http_reqs......................: 6672    95.310003/s
iteration_duration.............: avg=27.19s    min=1.76s    med=31.9s    max=48.59s   p(90)=37.53s    p(95)=39.05s
iterations.....................: 741     10.585239/s
vus............................: 1       min=1    max=500
vus_max........................: 500     min=500  max=500
```

Figure 42 - Scalability experiment with 500 virtual users and scaling up to 3 replicas.

```
data_received..................: 18 GB   263 MB/s
data_sent......................: 2.3 MB  34 kB/s
group_duration.................: avg=40.22s    min=1.3s     med=45.59s   max=1m4s     p(90)=1m1s      p(95)=1m2s
http_req_blocked...............: avg=189.65µs  min=1.2µs    med=114.2µs  max=10.79ms  p(90)=273.94µs  p(95)=519.43µs
http_req_connecting............: avg=147.35µs  min=0s       med=68.45µs  max=10.74ms  p(90)=191.22µs  p(95)=423.31µs
http_req_duration..............: avg=6.22s     min=813.4µs  med=206.42ms max=1m0s     p(90)=14.28s    p(95)=41.14s
  { expected_response:true }...: avg=6.21s     min=813.4µs  med=206.4ms  max=58.05s   p(90)=14.27s    p(95)=41.14s
http_req_failed................: 0.02%   ✓ 1       ✗ 4171
http_req_receiving.............: avg=6.12s     min=12.4µs   med=84.5µs   max=59.92s   p(90)=14.14s    p(95)=41.08s
http_req_sending...............: avg=226.81µs  min=7.6µs    med=51.2µs   max=169.26ms p(90)=100.59µs  p(95)=132.83µs
http_req_tls_handshaking.......: avg=0s        min=0s       med=0s       max=0s       p(90)=0s        p(95)=0s
http_req_waiting...............: avg=104.35ms  min=746.9µs  med=83.74ms  max=707.21ms p(90)=214.85ms  p(95)=259.55ms
http_reqs......................: 4172    59.550181/s
iteration_duration.............: avg=39.6s     min=2.3s     med=43.66s   max=1m4s     p(90)=1m1s      p(95)=1m2s
iterations.....................: 247     3.525622/s
vus............................: 847     min=38   max=1000
vus_max........................: 1000    min=1000 max=1000
```

Figure 43 - Scalability experiment with 1.000 virtual users and scaling up to 3 replicas.

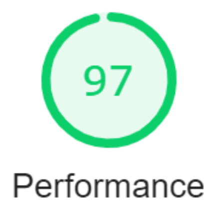


Pods

Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
store-app-shell-mf-6f97444bd5-zmnrj	io.kompose.service: store-app-shell-mf pod-template-hash: 6f97444bd5	k3d-mfe-store-agent-0	Running	0	0.00m	151.11Mi	an hour ago
store-purchase-mfe-mf-656d5fddbc-h885c	io.kompose.service: store-purchase-mfe-mf pod-template-hash: 656d5fddbc	k3d-mfe-store-agent-0	Running	0	0.00m	76.45Mi	an hour ago
store-purchases-service-98c98b469-hdpck	io.kompose.service: store-purchases-service pod-template-hash: 98c98b469	k3d-mfe-store-agent-0	Running	0	1.00m	39.13Mi	an hour ago
store-navbar-mfe-mf-85847857cd-d9zwg	io.kompose.service: store-navbar-mfe-mf pod-template-hash: 85847857cd	k3d-mfe-store-agent-0	Running	0	0.00m	586.79Mi	an hour ago
store-purchases-service-98c98b469-w7fl6	io.kompose.service: store-purchases-service pod-template-hash: 98c98b469	k3d-mfe-store-server-0	Running	0	1.00m	46.04Mi	an hour ago
store-purchase-mfe-mf-656d5fddbc-sc9pm	io.kompose.service: store-purchase-mfe-mf pod-template-hash: 656d5fddbc	k3d-mfe-store-server-0	Running	0	0.00m	139.86Mi	an hour ago
store-navbar-mfe-mf-85847857cd-w9lz9	io.kompose.service: store-navbar-mfe-mf pod-template-hash: 85847857cd	k3d-mfe-store-server-0	Running	1	0.00m	710.46Mi	an hour ago
store-app-shell-mf-6f97444bd5-gzf84	io.kompose.service: store-app-shell-mf pod-template-hash: 6f97444bd5	k3d-mfe-store-server-0	Running	0	0.00m	128.68Mi	an hour ago
store-purchases-service-98c98b469-gdpzx	io.kompose.service: store-purchases-service pod-template-hash: 98c98b469	k3d-mfe-store-server-0	Running	1	1.00m	44.28Mi	3 hours ago
store-purchase-mfe-mf-656d5fddbc-4psqj	io.kompose.service: store-purchase-mfe-mf pod-template-hash: 656d5fddbc	k3d-mfe-store-agent-0	Running	1	0.00m	55.49Mi	3 hours ago

1 – 10 of 12

Figure 44 - Kubernetes dashboard shows activity peak on each replica while load testing.



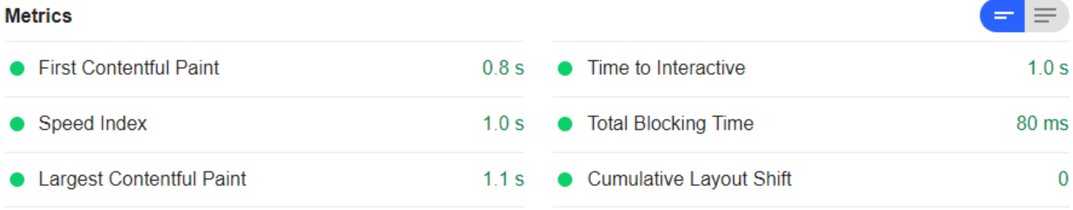


Figure 45 - Performance testing summary with only Simulated Throttling.

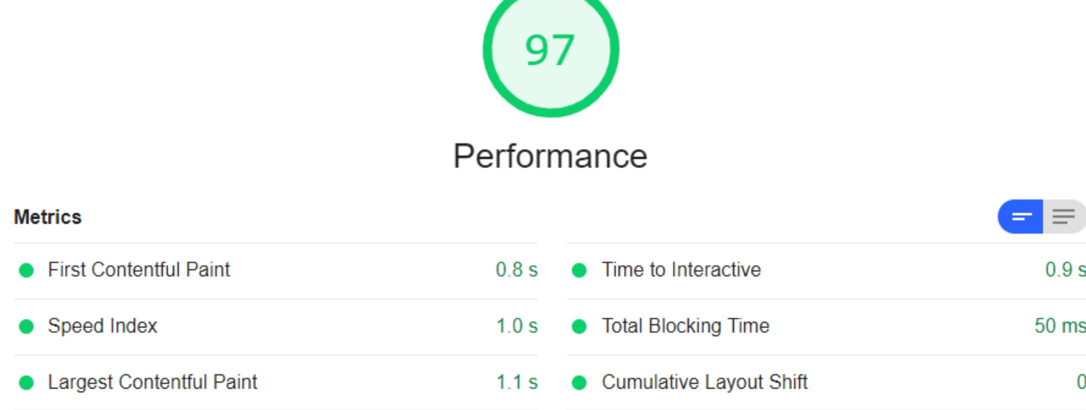
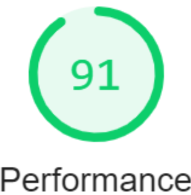


Figure 46 - Performance testing summary without constraints applied.



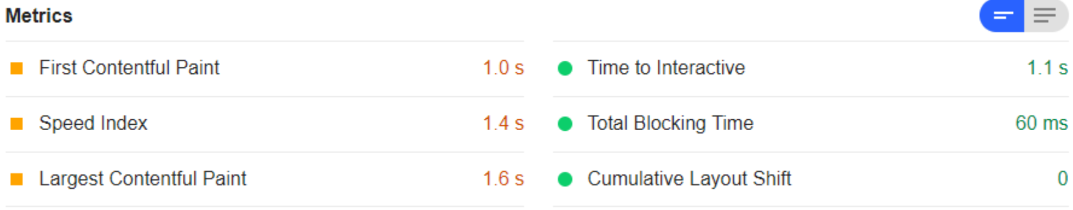


Figure 47 - Performance testing summary with only Clear Cache applied.

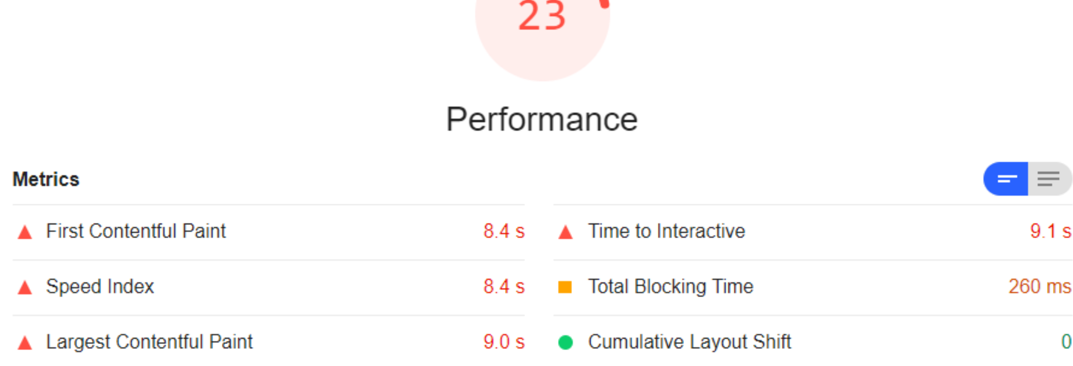


Figure 48 - Performance testing summary with Simulated Throttling and Clear Cache.

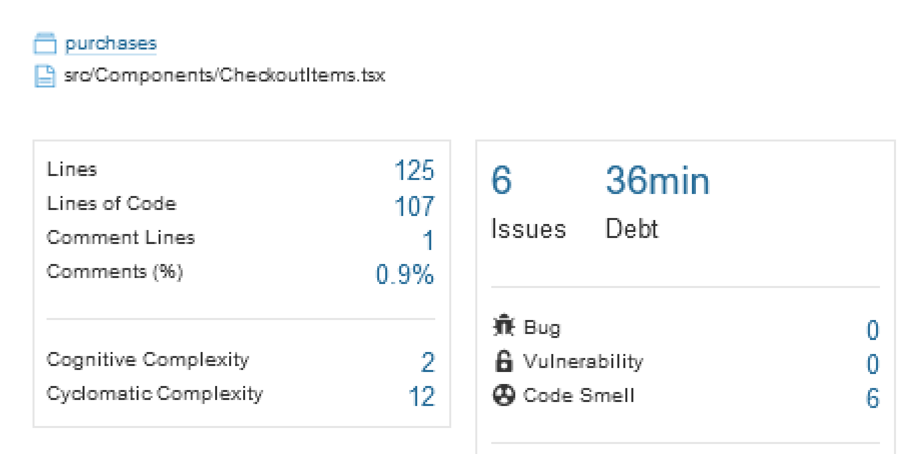


Figure 49 - Cyclomatic Complexity results for Purchases micro frontend.

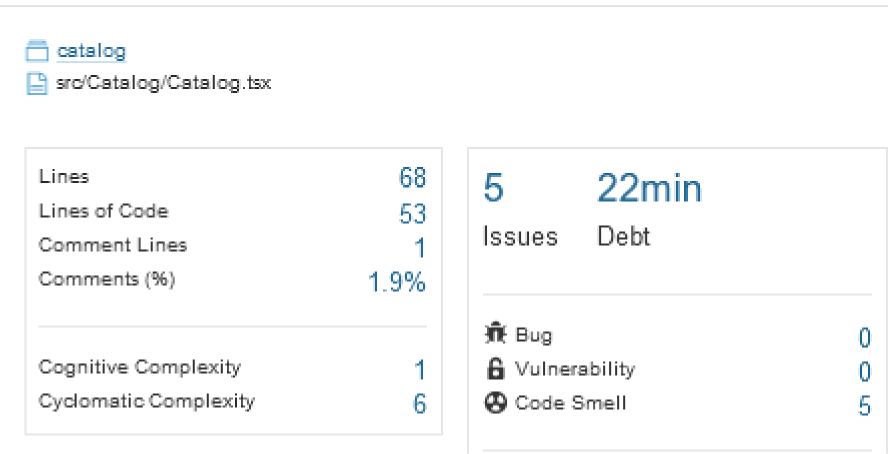


Figure 50 - Cyclomatic Complexity results for Catalog micro frontend.

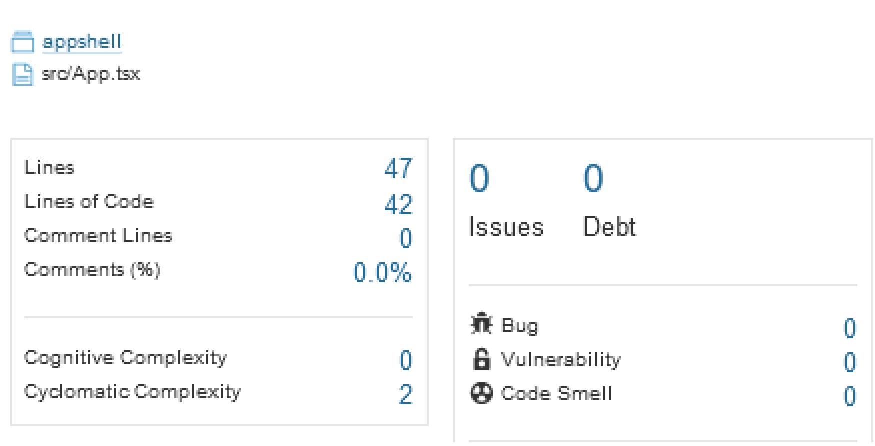


Figure 51 - Cyclomatic Complexity results for AppShell micro frontend.

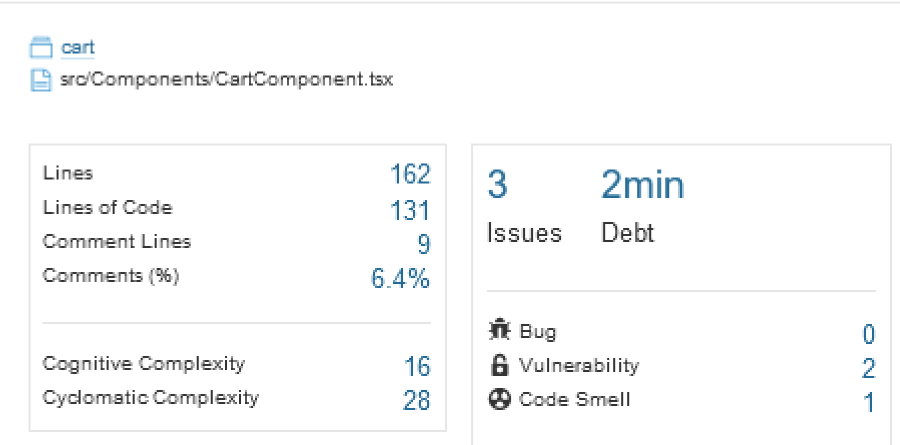


Figure 52 - Cyclomatic Complexity results for Cart micro frontend.

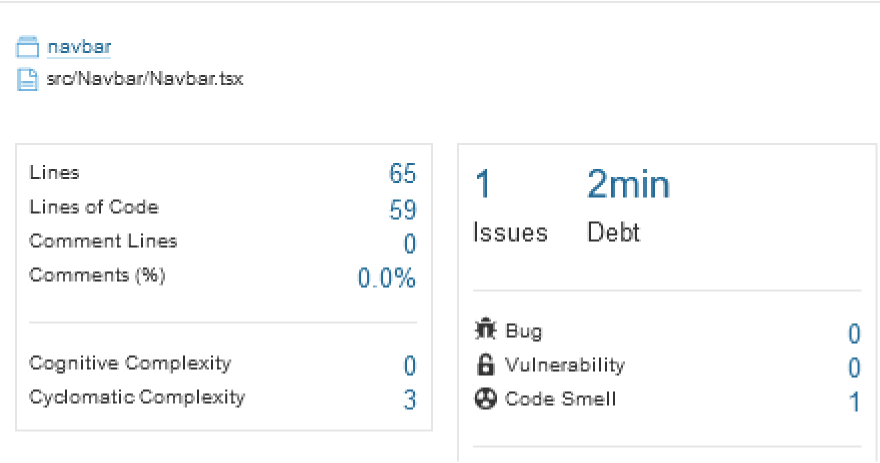


Figure 53 - Cyclomatic Complexity results for Navbar micro frontend.

Annex B – Solution Validation Results

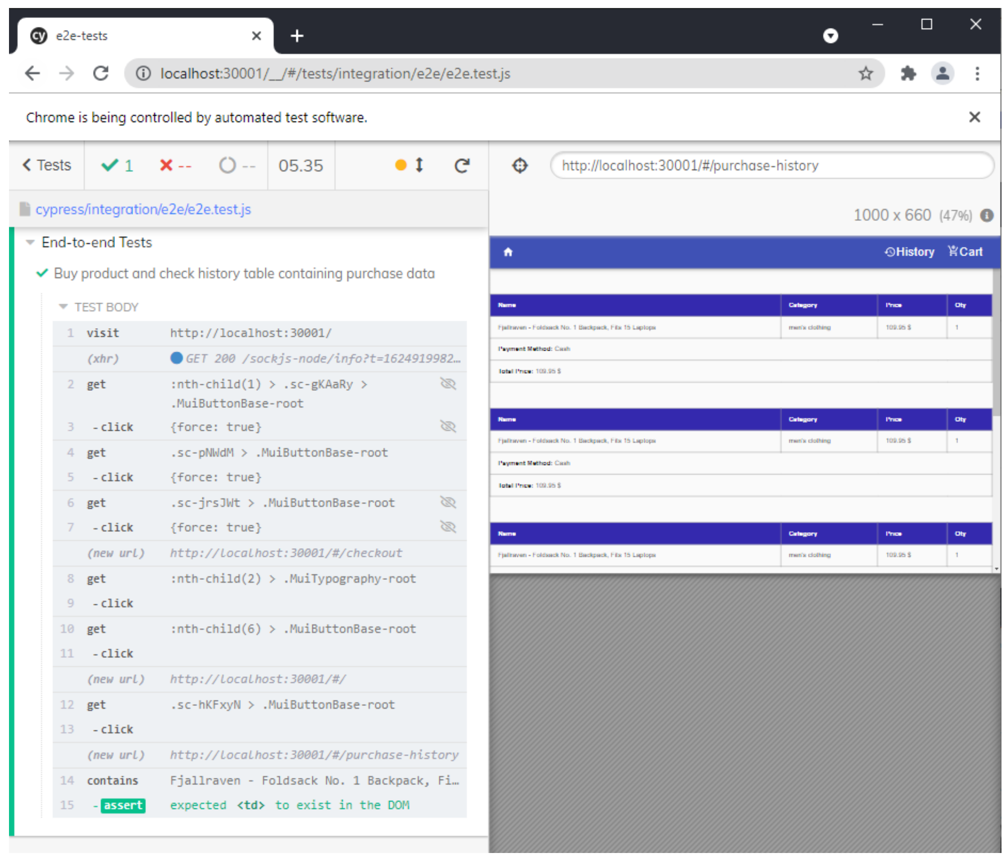


Figure 54 - “Buy a product” end-to-end test result, using cypress GUI.

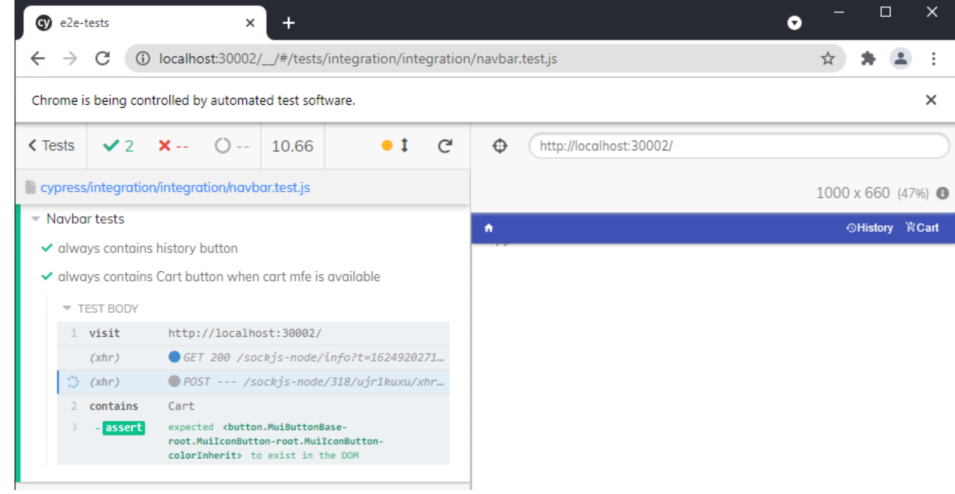


Figure 55 - Navbar and Cart Integration tests, using Cypress GUI.