INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Desenvolvimento Web Orientado a Micro Frontends

JOÃO PEDRO DA SILVA BASTOS
Julho de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Desenvolvimento *Web* Orientado a Micro *Frontends*

João Pedro da Silva Bastos

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

Orientador: Paulo Alexandre Gandra de Sousa

Porto, julho de 2020

Dedico este trabalho a todos aqueles que nunca deixaram de acreditar nas minhas capacidades e me encorajaram a não baixar os braços mesmo nos momentos de maior adversidade.

Resumo

Nos últimos anos, as necessidades de *software* têm vindo a aumentar e as aplicações *web* inegavelmente a ganhar terreno às aplicações *desktop*. Nestas aplicações, tem-se acentuado o nível de exigência quer em termos de experiência de utilização quer em termos de complexidade da lógica de negócio, o que levou a comunidade a unir esforços em prol de dar resposta a esta questão.

A arquitetura vulgarmente usada, a monolítica, que outrora conferia uma visão geral do projeto a toda a equipa de desenvolvimento e assegurava um desenvolvimento e lançamento rápido de funcionalidades, deixou de ser capaz de suprir estas solicitações.

Para mitigar o problema, as organizações têm procurado dividir os projetos de *software* em várias peças diferenciadas. Assim sendo, tornou-se prática comum organizar o *software* e até as equipas por capacidades técnicas, isto é, passou a reger-se por camadas horizontais, correspondentes ao *backend* e *frontend* das aplicações.

No entanto, dado que estas camadas continuavam a ser desenvolvidas na íntegra por uma única equipa, o processo de desenvolvimento era moroso e com o tempo a qualidade do código decrescia e o acoplamento aumentava, o que tornou a manutenibilidade insustentável.

O surgimento da arquitetura orientada a micro serviços, entrou em rutura com estas práticas, ao derrubar o *backend* monolítico, e atingiu uma popularidade de tal ordem, que abriu as portas para que o *frontend* monolítico seja desfeito, com o surgimento dos micro *frontends*.

Este conceito emerge como uma abordagem distinta de organização do *software* e das equipas, que visa transpor os princípios da arquitetura baseada em micro serviços para o âmbito do *frontend* com o intuito de criar uma *Single-Page Application*, construída sobre uma camada de micro serviços, que resulta da integração de funcionalidades pertencentes a equipas distintas.

Assim, propõe que as aplicações sejam estruturadas em camadas verticais, denominadas micro *frontends*, desenvolvidas de forma autónoma e na totalidade, desde a persistência de dados até à interface com o utilizador, por uma equipa multidisciplinar autónoma, dedicada e especializada numa determinada área de negócio.

Sendo a arquitetura de micro *frontends* recente, ainda não existe muita informação relativa ao impacto da sua adoção no processo de desenvolvimento nas organizações e à viabilidade de algumas técnicas de integração em cenários reais, pelo que o estudo desta abordagem visa atestar a sua viabilidade e eventuais mais-valias face a alternativas mais convencionais.

Palavras-chave: desenvolvimento *web*, micro *frontends*, arquitetura de *software*, organização de *software*.

Abstract

In the last few years, software needs have been rising and nowadays web applications are undeniably taking over the place occupied by desktop applications in the past.

In these applications, due to the remarkable increase in demands related with user experience as well as business logic complexity, the community have been working together in order to solve this matter.

The standard architecture, the monolithic one, which once gave an overview of the project to all team's developers and ensured a fast development and release of new features, is no longer a candidate empowered enough to face these demands.

To mitigate the problem, many organizations have been splitting *software* projects into several pieces of software. Therefore, software and team organization by technical skills became sort of a standard. Software's architecture and team's arrangement started to be governed by horizontal layers, corresponding to applications frontend and backend.

However, since these layers remained being entirely developed by a single team, the development process proved to be time consuming and over the time the code's quality decreased substantially, the coupling increased a lot and the single responsibility principle (SRP) simply vanished. As a result, maintenance became unsustainable.

The appearance of the micro services architecture, clash these practices, overthrowing the monolithic frontend and reached such an astonishing popularity that created an excellent opportunity to finish the frontend monolith through micro frontends.

This concept arrives as an alternative approach to organize software and teams, which aims to extend the principles of micro services to the frontend's world in order to create a Single Page Application, which sits on top of a micro service architecture, that consists in a composition of features owned by different teams.

Thus, it states that applications should be structured in vertical layers, called micro frontends, that are autonomously developed end-to-end, from the database to the user interface, by an dedicated and autonomous cross-functional team specialized in a specific business logic area.

Since the micro frontend's architecture itself is recent, there isn't enough information whose subject is the impacts that come from the micro frontends adoption on the *software* development process at organizations and the feasibility of some integration techniques in real scenarios. So, the study of this approach aims to check its feasibility as well as possible gains compared to more conventional architectural alternatives.

Keywords: web development, micro frontends, software architecture, software organization.

Agradecimentos

Aqui chegados, é tempo de agradecer a todos aqueles que de alguma forma me acompanharam no decorrer do meu percurso académico. Foram, sem dúvida, seis anos intensos, quer na vertente académica, quer profissional e pessoal, impregnados de altos e baixos, com muito sacrifício e numa azáfama constante, mas acima de tudo marcados por muita realização profissional, académica e pessoal.

Nesse sentido, agradeço ao Instituto Superior de Engenharia do Porto pelos conhecimentos transmitidos e pelo apoio prestado pelos docentes, em especial o Dr. Paulo Gandra de Sousa por me ter orientado ao longo do projeto e me ter incentivado a investir na produção do documento.

Além do contributo dos docentes, há que reconhecer que não seria possível concretizar este objetivo sem o apoio incondicional de alguns dos meus colegas de curso, em especial a Joana Santos e o João Tavares, que mesmo numa fase em que me encontrava com menor capacidade física e mental, em virtude de algumas vicissitudes, fizeram tudo o que estava ao seu alcance para que eu não deitasse a toalha ao chão, e deram um forte atributo na revisão deste documento.

Quero deixar também uma palavra de enorme agradecimento à Glintt por me ter aberto as portas para o mercado de trabalho, e me ter permitido expandir conhecimentos e privar de perto com pessoas extraordinárias, em ambos os projetos em que estive inserido.

Esta aprendizagem foi fundamental para que eu estivesse à altura do desafio profissional que foi, entretanto, colocado por parte da Blip. Foram, sem dúvida, 18 meses de aquisição de conhecimento técnico e boas práticas de desenvolvimento, acompanhados de um enorme espírito de companheirismo, em particular com a equipa “Jean Pierre”, que se mantém até hoje.

Ainda em termos profissionais, gostaria de agradecer à Critical Techworks por ter acreditado nas minhas capacidades e por ter vindo a proporcionar as condições para o sucesso e evolução e me ter apoiado desde o princípio na realização deste trabalho, em particular a equipa Pegasus, da qual faço parte, a Dina Lopes e o João Gonçalves.

Relativamente à elaboração deste documento, queria agradecer ainda a Michael Geers, autor do livro “*Micro Frontends in Action*” pelo apoio inexcedível no esclarecimento e debate de questões suscitadas ao longo do estudo, e ao Antero Oliveira e à Marta Teixeira pelo seu contributo na conceção, revisão e divulgação do questionário aplicado.

Por último, mas não menos importante, queria agradecer aos meus pais, irmã e amigos mais próximos por terem feito tudo o que estava ao seu alcance para me encorajarem a dar o melhor de mim, à Dra. Cláudia Pimentel por me ter ensinado a encarar a vida com outros olhos e a todos aqueles que com simples gestos me derem alento mesmo nas horas mais difíceis.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Blocos de Código

Glossário

Atomic *Design* metodologia para a criação de sistemas de *design* que pensa a interface como o resultado de uma hierarquia de componentes: átomos (elementos básicos), moléculas (grupos de átomos), organismos, *templates* e páginas (Frost, 2013).

Cache Busting é um processo que permite assegurar a atualização de ficheiros estáticos armazenados em cache, sempre que sejam atualizados. Para tal, é atribuído um identificador ao ficheiro (seja usando versionamento, *timestamp*, *hash*, etc.) para informar o *browser* de que uma nova versão está disponível, invalidando o ficheiro atualmente em cache e desencadeando um pedido ao servidor de origem para obter a versão atualizada (KeyCDN, 2018).

Código JavaScript Isomórfico/Universal A universalidade deste código escrito em JavaScript prende-se com "poder ser executado no cliente e no servidor" (Robbins, 2014), isto é, em praticamente todos os ambientes que suportam JavaScript, incluindo *browsers* e Node.js (M. Jackson, 2015). O isomorfismo refere-se ao facto de a técnica de integração/composição de páginas poder ocorrer no cliente ou no servidor (Schneider, 2015).

Community of Practice refere-se a um grupo de pessoas que possuem um interesse comum num domínio técnico ou de negócio específico. Estes grupos colaboram e reúnem-se regularmente para partilhar e expandir conhecimentos do conhecimento do grupo acerca desse domínio (Webber, 2019).

Content Delivery Network refere-se a um grupo de servidores distribuídos geograficamente cuja cooperação se traduz na entrega de conteúdo mais rapidamente. Para tal, armazenam este conteúdo em cache (Cloudfare, 2016), o que permite colmatar problemas de *performance* decorrentes do carregamento local de recursos.

Continuous Delivery é uma prática que se assumindo como uma extensão da *Continuous Integration*, visa garantir que podem ser enviadas novas alterações para o cliente de forma rápida e segura, implicando para tal a automatização não somente dos testes como também do processo de lançamento (Atlassian, sem data). Na *Continuous Delivery*, o processo de implantação está à distância de um clique num botão, o que permite à equipa decidir com que frequência são efetuadas as *releases* (Pittet, sem data).

Continuous Deployment é uma prática que não só automatiza o processo de *release*, como prescinde de qualquer intervenção humana no mesmo. Assim, qualquer alteração que passe em todas as etapas da *pipeline* de produção é lançada para os clientes (Pittet, sem data).

Continuous Integration é uma prática que requer que os programadores integrem o seu código com o desenvolvido por outros regularmente, através do *merge* das alterações introduzidas no *branch* principal (Pittet, sem data).

DOM Diffing é uma técnica fornecida por algumas *frameworks* de JavaScript que permite que a aplicação reaja rapidamente e de forma mais eficiente a alterações ao seu estado ou propriedades, ao re-renderizar apenas o DOM que foi de facto afetado por essas alterações, que é identificado por via de um algoritmo (React, sem data-c).

Downstream Num contexto de interação entre dois serviços A e B, em que B depende da informação fornecida por A, podemos afirmar que B está *downstream* de A, isto é, que o input de B corresponde à informação obtida de A (Hombergs, 2018).

ECMAScript Especificação de linguagem de programação baseada em *scripts*, padronizada pela ECMA International, com o intuito de definir *standard*s para o uso de JS (ECMA International, 2019).

ES Module é um *standard* do ECMAScript relativo a módulos JS (MDN contributors, 2019d).

Hydration é um processo *client-side* durante o qual o HTML estático renderizado no servidor é transformado em DOM dinâmico, de modo à aplicação conseguir reagir rapidamente a alterações efetuados sobre os dados ocorridas do lado do cliente (Gordon, 2018).

Latência é uma medida que quantifica o tempo decorrido entre o envio dos dados e sua chegada ao destino (Duffy, 2019).

Lei de Conway afirma que as organizações estão limitadas a produzir aplicações cujo *design* reflete as suas estruturas de comunicação, o que geralmente acarreta dívida técnica (ThoughtWorks, 2015).

Manobra Inversa de Conway recomenda a evolução da estrutura da equipa e da própria organização para promover a arquitetura desejada (ThoughtWorks, 2015).

Minimum Viable Product é a menor versão de um novo produto que traz valor para os utilizadores. É a representação dos objetivos a curto prazo mais prioritários para o negócio (P. V. da Silva et al., 2019).

Progressive Enhancement é uma filosofia centrada em apresentar o essencial do conteúdo e funcionalidades de uma aplicação *web* tão cedo quanto possível no cliente (*browser*) enquanto os recursos que se possibilitam uma maior interação com o utilizador ainda se encontram a ser carregados (MDN contributors, 2019b).

Progressive Web App são aplicações *web* cujo *desenvolvimento* sendo baseado nas potencialidades oferecidas pelas aplicações *web* e nativas permite a criação de *sites* multiplataforma rápidos, integrados, fiáveis e dinâmicos (Posnick, 2018).

Regra das Duas Pizzas é uma regra da autoria de Jeff Bezos, CEO da Amazon, que defende numa aplicação distribuída a dimensão das equipas não deve exceder os doze elementos, número de pessoas é possível alimentar com duas pizzas familiares americanas (Newman, 2015).

Reverse-Proxy é um serviço de proxy que ao receber um pedido do cliente o encaminha para um ou mais servidores, obtém a resposta e posteriormente devolve a resposta do servidor ao cliente (Linuxwize, 2019).

Rich Internet Application é uma aplicação capaz de resolver os problemas relacionados com usabilidade e a engenharia de *software* verificados nas aplicações *web* tradicionais. Uma RIA consegue oferecer uma *User Interface* dinâmica, rica em conteúdo e com um nível de interatividade equiparável a uma aplicação *desktop* (Bali et al., 2012; Farrell & Nezlek, 2007)

Self-Contained System abordagem que propõe a separação de funcionalidades em sistemas independentes, fazendo com que o sistema lógico resulte da colaboração de vários sistemas de *software* mais pequenos, que comunicam entre si de forma assíncrona. Promove o desacoplamento e demove a partilha de lógica (Wolff, 2016).

Service-Oriented Architecture é um estilo arquitetural que estrutura um sistema como um conjunto de serviços (à semelhança da arquitetura de MSs). As aplicações SOA tipicamente usam tecnologias pesadas como o protocolo SOAP e integram os serviços através de um *smart pipe* que contém lógica de negócio e processamento de mensagens - ESB (*Enterprise Service Bus*). Já na arquitetura de MSs os micro serviços comunicam através de *dumb pipes*, usando message brokers ou protocolos leves como REST. Por outro lado, as aplicações SOA possuem um modelo de dados e DBs partilhadas, ao contrários dos MSs (Richardson, 2018b).

Service Worker é um *script* que o *browser* executa em segundo plano, que contempla funcionalidades que não dependem de interação com o utilizador. Atualmente, permite gerir *push notifications*, sincronização em segundo plano e intercetar e tratar pedidos de rede cujas respostas pode armazenar em cache (Gaunt, 2019).

Testes A/B são testes que permitem analisar o impacto de uma alteração tendo por base uma amostra de utilizadores, sem controlar ou considerar fatores que não o que se encontra a ser testado (Naso, 2018).

Transclusão Consiste na inclusão num documento de conteúdo proveniente de recurso ou documento externo, por exemplo proveniente de outro serviço. A inclusão ser efetuada de forma imperativa ou declarativa (usando diretivas SSI, ESI ou CSI), do lado do servidor ou do cliente (Kotte, 2017)

Throughtput Corresponde ao nº de unidades de output produzidas pela organização/equipa por unidade de tempo (Grealou, 2015). É uma métrica de análise comparativa da eficácia de um processo/operação, ao determinar quão rapidamente os produtos são desenvolvidos. No contexto *agile*, corresponde ao nº de unidades de trabalho (*user stories*, *tickets*, etc.) num dado período (p.e, uma *sprint*), permitindo tirar conclusões sobre o ritmo de entrega (Drayton, 2018).

Upstream Num contexto de interação entre dois serviços A e B, em que B depende da informação fornecida por A, podemos afirmar que A está *upstream* de B, isto é, que o informação proveniente de A servirá de output a B (Hombergs, 2018).

Lista de Acrónimos

API	*Application Programing Interface*
BE	*Backend*
BFF	*Backend for Frontend*
BLL	*Business Logic Layer*
CD	*Continuous Delivery/Deployment*
CDN	*Content Delivery Network*
CI	*Continuous Integration*
CoP	*Community of Practice*
CSI	*Client Side Includes*
CSR	*Client-Side Rendering*
DAL	*Data Access Layer*
DDD	*Domain-Driven Design*
DOM	*Document Object Model*
E2E	*End-to-End*
ES	*ECMAScript*
ESI	*Edge Side Includes*
FE	*Frontend*
FFE	*Fuzzy Front End*
GCP	*Google Cloud Platform*
HOQ	*House of Quality*
JS	*JavaScript*
MFE	Micro *Frontend*
MS	Micro *Service*
MVP	*Minimum Viable Product*

NCD	*New Concept Development*
OC	*OpenComponents*
OO	*Object-Oriented*
POC	Prova de Conceito
PWA	*Progressive Web Application*
QFD	*Quality Function Deployment*
RIA	*Rich Internet Application*
SCS	*Self-Contained Systems*
SGBD	Sistema de Gestão de Base de Dados
SEO	*Search Engine Otimization*
SOA	*Service-Oriented Architecture*
SPA	*Single Page Application*
SSI	*Server Side Includes*
SSR	*Server-Side Rendering*
TTFB	*Time To First Byte*
UC	*Use Case*
VA	*Value Analysis*
WC	*Web Component*

1 Introdução

Este capítulo começa por apresentar o contexto e o problema que motiva este estudo, a par dos objetivos delineados.

Seguidamente, é introduzida a metodologia de trabalho adotada, em termos de recolha e tratamento da informação.

Por fim, é dada a conhecer a estrutura preconizada para este documento, de modo a agilizar a navegação do mesmo por parte do leitor.

1.1 Contexto

Nas últimas décadas, deu-se uma autêntica revolução tecnológica, sobretudo após a invenção do computador e o aparecimento da internet. Desde então, tem-se assistido a uma evolução e proliferação dos dispositivos tecnológicos sem precedentes, bem como a uma universalização no acesso à Internet (Samsung, 2015).

E se outrora os sistemas computacionais ocupavam um armazém e os telefones estavam dependentes de uma linha telefónica, acabariam por respetivamente dar lugar a *PCs* e numa primeira fase ao telemóvel, que posteriormente foi sucedido pelo *smartphone* que se tornou um autêntico computador de bolso. A par destes dispositivos surgiram gadgets tecnológicos como tablets, quadros interativos, dispositivos de realidade aumentada, *smart TVs, Internet of Things* (IoT) e mais recentemente *wearables* como os *smartwatches*, que se tornaram parte integrante do nosso quotidiano (Bankai, 2017).

Ora, esta diversificação e massificação de dispositivos tecnológicos tem-se traduzido num aumento exponencial das necessidades de *software* em todo o mundo e Portugal não ficou à margem desta tendência, dado que estudos conduzidos pela InvestPorto verificaram entre 2013 e 2018 um aumento de 71% no número de projetos de investimento direto estrangeiro no setor na região norte do país (Castro, 2019).

Assim, nos dias de hoje, num contexto de concorrência feroz, as organizações que prescindam ou que decidam tardiamente dar o salto tecnológico, podem incorrer em desvantagens competitivas.

No entanto, se por um lado esta demanda de *software*, com particular destaque para o desenvolvimento *web*, se traduz numa oportunidade de negócio para as organizações do setor, por outro, acarreta desafios para as equipas de desenvolvimento, que se prendem com diversos fatores.

Desde logo, é imperativo que o *software* desenvolvido seja capaz de suprir os requisitos não funcionais, nomeadamente a heterogeneidade de dispositivos (cada um com as suas especificidades em termos de dimensão do ecrã, sistema operativo, etc.) e o tráfego potencialmente elevado (devido ao número de acessos e solicitações), a par de requisitos funcionais, intimamente ligados à lógica de negócio do cliente, cuja complexidade tende gradualmente a intensificar-se.

Ora, o facto de muitas vezes se pretender que o *software* seja lançado tão cedo quanto possível para obter o *feedback* dos utilizadores, pode fazer com que a metodologia e processo de desenvolvimento de *software* não seja devidamente pensada, não só do ponto de vista de análise e desenho arquitetural das aplicações (descurando-se a pesquisa, experimentação e discussão de abordagens, ferramentas e tecnologias a atuar, bem como boas-práticas e aspetos essenciais como a manutenibilidade e a escalabilidade) mas também numa vertente de organização das equipas (que poderia traduzir-se num maior *throughtput*).

A par disto, de forma quase ininterrupta são colocadas à disposição dos programadores novas abordagens, metodologias, ferramentas e tecnologias, que sendo objeto de estudo e experimentação poderiam revelar-se alternativas viáveis a outras soluções mais convencionais e quiçá mais-valias na obtenção de aplicações mais dinâmicas, modulares, resilientes e escaláveis, caraterísticas que certamente se refletiriam numa maior longevidade das mesmas.

1.2 Problema

Num contexto de aplicações *web*, assente no modelo cliente-servidor, as práticas de desenvolvimento mais convencionais passam ora por dividir as aplicações nas camadas de *frontend* e *backend*, entregando a sua implementação a duas equipas distintas, ora por seguir um modelo monolítico, isto é, por serem desenvolvidas na íntegra por uma mesma equipa, o que acarreta em ambos os contextos uma série de problemas (Geers, 2018; Mezzalira, 2019a; Richardson, 2018b).

Enveredando pela vertente monolítica, estando uma única equipa incumbida do processo de desenvolvimento, este tende a ser moroso e a extensão do *software* poderá traduzir-se em dificuldades acrescidas, uma vez que descurar aspetos como a separação de responsabilidades entre componentes poderá repercutir-se em comportamentos inesperados.

Já caso o desenvolvimento de *backend* e *frontend* esteja a cargo de equipas distintas, é geralmente adotada uma de duas abordagens: ambas as aplicações *frontend* e *backend* são monolíticas ou apenas a aplicação *frontend* é monolítica e o *backend* é orientado a micro serviços.

No primeiro cenário, se a aplicação cliente for desenvolvida à *posteriori*, pode acabar sobrecarregada com lógica de negócio, que deveria ter sido acautelada do lado do servidor.

Caso sejam desenvolvidas em simultâneo, poderá haver a tentação de no *backend* se criarem *endpoints* de forma desmesurada para que a obtenção de dados por parte do *frontend* seja obtida através de um número reduzido de pedidos ao *backend*, que acaba por agregar toda a informação de que a aplicação cliente necessita. Ora, este caminho não só atenta contra os princípios do REST, como também se reflete num elevado acoplamento entre aplicações.

Já no último cenário, mesmo com o *backend* a adotar uma arquitetura orientada a micro serviços, acaba por não se tirar o real proveito desta abordagem, uma vez que o lançamento de uma funcionalidade depende da conclusão da implementação quer do serviço quer da interface do utilizador, o que exige um esforço acrescido de comunicação entre equipas e impossibilita lançamentos com frequência.

Além disso, a implementação destas aplicações, por vezes, pressupõe a satisfação de uma vasta panóplia de requisitos dentro de um tempo de execução limitado, o que inviabiliza que uma única equipa se possa encarregar de todo o processo de desenvolvimento.

1.3 Objetivos propostos

Tratando-se este trabalho de um estudo, visa essencialmente a pesquisa, seleção e análise de informação com vista a clarificar como adotar uma abordagem orientada a micro *frontends* (MFEs) no processo de desenvolvimento de *software web* e a determinar em que situações esta abordagem poderá revelar-se particularmente vantajosa, por oposição a outras alternativas. Neste sentido, os objetivos principais consistem em:

1. Recolher e estudar informação relativa a práticas de desenvolvimento de *web*, dando particular ênfase a micro *frontends*;

2. Determinar em que contextos a adoção de micro *frontends* poderá ser particularmente vantajosa;

3. Comparar padrões de integração de micro *frontends*, identificando as principais vantagens e desvantagens, bem como os custos inerentes;

4. Implementar provas de conceito usando os padrões de integração selecionados.

1.4 Metodologia preconizada

Um caminho possível para a resolução do problema passa por atribuir a implementação de diferentes funcionalidades a equipas distintas.

Nesse sentido, em 2016 surge o conceito de micro *frontends*, com o intuito de transpor os pressupostos da arquitetura orientada a micro serviços para o mundo do *frontend*, o que se traduz numa mudança de paradigma, ao propor precisamente que a implementação *end-to-end* de cada funcionalidade esteja a cargo de equipas independentes.

A adoção deste estilo arquitetural traduz-se na obtenção de diferentes módulos aplicacionais entregáveis, sendo necessário recorrer a ferramentas que permitam a sua integração numa única aplicação.

Sendo o propósito deste projeto o estudo do impacto da adoção de micro *frontends* no processo de desenvolvimento *web*, numa primeira fase será efetuada a recolha e tratamento de informação relacionada com essa temática, proveniente de múltiplas fontes, nomeadamente:

- obras literárias no âmbito da arquitetura de *software*, com particular enfoque para o livro *"Micro Frontends in Action"* da autoria de Michael Geers e para a versão de rascunho do livro *"Building Micro-Frontends"* da autoria de Luca Mezzalira;
- apresentações em conferências por parte de oradores que dominam o conceito;
- vídeos provenientes de canais de Youtube sugeridos na documentação oficial das ferramentas a utilizar na prova de conceito;
- artigos científicos vocacionados para o estudo de micro *frontends*[1], micro serviços, migração de monolíticos para micro serviços e sistemas autocontidos;
- artigos publicados em diversos *sites* e blogs, etc.

Mobilizando o conhecimento adquirido do estudo e investigação, na segunda fase será implementada uma prova de conceito na qual será desenvolvida uma aplicação *e-commerce* aplicando a arquitetura de micro *frontends*. Esta prova de conceito terá um conjunto de versões seguindo diferentes metodologias de integração e recorrendo a mecanismos distintos de partilha de lógica e de comunicação entre micro *frontends*, com vista a aferir as vantagens e desvantagens inerentes.

[1] Aplicando como *query string* "micro *frontends*", obteve-se um número irrisório de resultados, mais concretamente um único artigo intitulado "Research and Application of Micro *frontends*" (Yang et al., 2019)

1.5 Estrutura do documento

O presente documento encontra-se subdividido em oito capítulos, respetivamente Introdução, Contexto, Análise de Valor, Micro *Frontends*, Integração e Comunicação de Micro *Frontends*, Prova de Conceito, Experiências e Avaliação e Conclusão.

Assim, a introdução assume-se como o fio condutor para o restante documento, dado que apresenta o contexto no se insere o estudo, o problema ao qual se pretende dar resposta, os objetivos a atingir e a abordagem preconizada.

No capítulo de contexto, é efetuada uma contextualização teórica relativa ao mundo do desenvolvimento *web* e às principais abordagens arquiteturais passíveis de serem adotadas em aplicações *web* e a evolução desde a arquitetura monolítica até à orientada a micro *frontends.*

No capítulo dedicado à análise de valor, são introduzidos uma série de conceitos e modelos teóricos, com vista à posterior identificação e análise da oportunidade, geração e seleção de ideias, com base no método de avaliação hierárquico.

No capítulo intitulado micro *frontends* é dissecado o conceito, com o intuito de apresentar as suas motivações e princípios norteadores, que estratégias oferece para mitigar um conjunto de desafios frequentes no desenvolvimento *web*, bem como são abordados aspetos como a gestão de equipas e responsabilidades, a experiência de desenvolvimento local e serão introduzidas formas de testar a integração e estratégias de migração de um monolítico para MFEs.

No capítulo de integração e comunicação de micro *frontends*, como o próprio nome indica serão abordadas as principais técnicas e bibliotecas / *frameworks* que permitem a integração de MFEs, bem como os mecanismos existentes para estabelecer a comunicação entre MFEs.

Dado que é através destas técnicas que é possível colocar de pé uma aplicação orientada a micro *frontends*, neste capítulo serão igualmente apresentados vários casos da aplicação da abordagem em contexto empresarial, com vista a determinar as vantagens e limitações identificadas pelas organizações.

No que concerne o capítulo destinado à prova de conceito, começa-se por contextualizar a aplicação a desenvolver, para posteriormente proceder à análise e *design*, onde será apresentada a arquitetura idealizada para cada técnica de integração contemplada, implementação e avaliação.

No capítulo de experiências e avaliação, são formuladas as hipóteses a verificar, os indicadores e fontes de informação, os métodos de avaliação e serão avaliadas as experiências realizadas e retiradas conclusões, não só da recolha e tratamento de informação, mas também da prova de conceito, questionário e casos práticos de aplicação de micro *frontends*

Por fim, no capítulo de conclusões são descritos os objetivos atingidos, as limitações e desafios encontrados e são feitas algumas considerações relativas a trabalho futuro e à apreciação global.

2 Contexto

Este capítulo tem como objetivo:

- Enquadrar teoricamente os principais conceitos do desenvolvimento *web* relacionados com o tema em estudo;
- Descrever a evolução arquitetural das aplicações que motivou o surgimento da arquitetura orientada a micro *frontends*.

2.1 Enquadramento Teórico

Esta secção destina-se a fornecer algum enquadramento teórico relativamente à terminologia presente no desenvolvimento de aplicações *web*, que poderá revelar-se útil para uma melhor compreensão do documento.

2.1.1 Modelo Cliente-Servidor

O modelo Cliente-Servidor pressupõe a divisão da aplicação entre o lado do cliente, que normalmente é executado no *browser*, e o lado do servidor.

O servidor está incumbido de gerir grande parte das operações e armazenar todos os dados, enquanto o cliente efetua pedidos para obter acesso a esses dados. Note-se que apesar disso também pode ser processada informação do lado do cliente (Subramanian et al., 2017)

Os conceitos *Backend* (BE) e *Frontend* (FE), parte integrante das aplicações *web,* podem assumir-se como servidor e cliente respetivamente.

O *Backend* corresponde ao lado de servidor no modelo Cliente-Servidor e é responsável pelo acesso e gestão dos dados, em conformidade com o princípio *Separation of Concerns*.

Envolve o uso de linguagens de *scripting* como JS, Ruby, PHP, etc., administração de bases de dados, mecanismos de processamento de dados, de autenticação e autorização, etc. (OpenLearn, 2019)

Quanto ao FE corresponde exclusivamente ao lado manipulado pelo cliente, isto é, à camada de apresentação (*Presentation Layer*), que solicita a informação a apresentar ao servidor.

2.1.2 Aplicações *Web* Tradicionais vs. *Single-Page Applications*

As aplicações *web* vieram para ficar, pois contrariamente às aplicações *desktop*, são mais intuitivas e facilmente atualizáveis e adaptáveis à heterogeneidade dos dispositivos (Neoteric, 2016).

Existem dois principais padrões de *design* para aplicações *web* passíveis de ser seguidos: o mais tradicional, denominado *Multi-Page Application* (MPA), e *Single-Page Application* (SPA) (Marcano et al., 2019).

Antes de proceder à implementação propriamente dita, deve seguir-se uma abordagem *content-first* ou *mobile-first*, que defende que uma aplicação deve ser desenhada a pensar primeiramente na sua utilização em dispositivos móveis (dado que possuem mais restrições, nomeadamente em termos de dimensão de ecrã (UXPin, 2015)), definindo qual o conteúdo a ser apresentado, e determinando o que o utilizador mais valoriza.

É precisamente o tipo de conteúdo a apresentar que vai ter um papel preponderante na escolha do padrão a seguir.

No que concerne as MPAs, grande parte da lógica aplicacional acontece do lado do servidor, dado que sempre que se verifica qualquer interação do utilizador com a aplicação, por exemplo com o intuito de apresentar dados ou submeter dados, é despoletado um pedido do cliente ao servidor de uma nova página para renderizar no *browser*.

Esta página quando é gerada e enviada para o cliente, provoca o recarregamento da aplicação para ser apresentada. Este processo requerendo algum tempo para ser concluído, pode ter impacto na *User Experience*.

As MPAs são mais complexas, apresentando diversos níveis de interface com o utilizador devido ao maior volume de conteúdo.

Embora o AJAX permita recarregar apenas parte dos componentes da aplicação, o que reduz a quantidade de informação a transferir, torna o processo de desenvolvimento complexo e moroso e cria um elevado acoplamento entre *backend e frontend*.

Assim, MPAs são a opção indicada em situações em que:

- a interface a apresentar ao utilizador praticamente não varia em função da manipulação do estado do lado do cliente;
- se pretenda fornecer um mapa visual da aplicação ao utilizador;
- a equipa não esteja familiarizada com a aplicação;
- a aplicação possa ser executada em *browsers* que não possuam suporte para a linguagem JavaScript.

Além disso, as MPAs permitem uma melhor *Search Engine Otimization* (SEO), pois como cada página tem o próprio URL é mais facilmente indexada pelos motores de pesquisa.

Relativamente às SPAs, não requerem recarregamento pois correspondem a uma única página, para a qual o conteúdo a apresentar é carregado dinamicamente usando JavaScript (o que pode dificultar o SEO).

SPAs são, regra geral, mais rápidas que MPAs, uma vez que executam a lógica de renderização do lado do *browser* e que contrariamente ao que acontece nas MPAs, após o carregamento inicial da página, na qual são transferidos a maioria dos recursos (HTML, CSS e JavaScript - JS), apenas ocorrem transações de dados entre cliente e servidor (Macquin, 2018).

Deste forma, o cliente evita sucessivos round-trips ao servidor para carregar lógica aplicacional adicional e as *views* são renderizadas instantaneamente durante o ciclo de vida da aplicação, o que melhora a *User Experience* (UX) de tal forma que simula de forma bastante realista a interação com uma aplicação nativa (Mezzalira, 2020).

SPAS permitem adotar uma de duas abordagens: *Thin Client / Fat Server* ou *Fat Client / Thin Serve*r. Estas designações indicam onde se concentra a maior parte do esforço computacional, sendo mais frequente em aplicações multiplataforma que a escolha recaia sobre a primeira opção (Mezzalira, 2020).

É que não havendo um esforço de renderização do lado do servidor, o desenvolvimento é mais rápido e o *debug* mais simples, além de reduzir o tempo de desenvolvimento de aplicações móveis que venham a necessitar dos dados fornecidos pelas APIs, uma vez que o código do *Backend* pode ser reutilizado.

Além disso, as SPAs permitem que após efetuar um pedido, a informação obtida possa ser guardada em cache, de modo, a que a aplicação continue a funcionar mesmo que *offline*.

No entanto, comparativamente à abordagem tradicional, SPAs são menos seguras, pois são mais suscetíveis a ataques de *Cross-Site Scripting* (XSS), isto é, intrusões que advém da injeção de *scripts* maliciosos do lado do cliente (Macquin, 2018).

2.1.3 Aplicações Isomórficas

Aplicações isomórficas/universais consistem em aplicações *web* nas quais o código entre servidor e cliente é partilhado e pode ser executado em ambos os contextos.

O servidor pode obter a informação a apresentar e pré-renderizar a página enquanto no cliente, devido à pré-renderização ocorrer no servidor, se verifica um número significativamente menor de *round-trips* (uma vez que não há necessidade de carregar ficheiros adicionais) e o *browser* vai interpretar uma página estática com praticamente tudo, o que vai permite melhorar o tempo de resposta e de interação com o utilizado (Mezzalira, 2020).

A quantidade de código partilhado depende da abordagem adotada: abordagem híbrida ou *server-side rendering* e gestão de estado no servidor.

Na abordagem híbrida, parte da página é renderizada no servidor de forma a proporcionar menor tempo de carregamento inicial. Já os ficheiros JavaScript adicionais que permitem obter os benefícios do isomorfismo e estabelecer um alto nível de interatividade, transformando uma página *web* estática numa SPA, são *lazy loaded*.

Além disso, esta opção permite dosear a quantidade de código que é partilhada no *backend* com base nos requisitos do projeto, podendo se optar por apenas renderizar as *views* incluindo somente o CSS e JS mínimos para devolver um esqueleto HTML carregado rapidamente pelo *browser*, que pode assim dar *feedback* mais cedo ao utilizador ou então por entregar as tarefas de renderização e de integração ao servidor perante cenários de reduzida interatividade.

Alternativamente pode partir-se a aplicação em múltiplas SPAs sendo a primeira *view* renderizada pelo servidor e a partir daí delegar nos JS adicionais a gestão do comportamento, modelos e *routing* de cada SPA.

Já na outra abordagem, mais convencional, a renderização e estado é processada apenas do lado do servidor, fornecendo a página HTML a ser apresentada no *browser*.

As aplicações isomórficas permitem ainda integrar facilmente plataformas de testes A/B, que permitem isolar e aferir de forma fiável o impacto de uma *Progressive Web App* (PWA).

Estes testes são tipicamente conduzidos no lado do servidor, dado que no servidor os *developers* podem testar a pré-renderização do que pretendem ver apresentado do lado do cliente.

Apesar destas mais-valias, as aplicações isomórficas podem sofrer problemas de escalabilidade perante tráfego elevado, devido à sobrecarga do servidor, algo que, apesar de tudo, é minimizável implementando uma estratégia adequada de cache (Mezzalira, 2020).

2.1.4 *Server-Side Rendering* vs. *Client-Side Rendering*

2.1.4.1 Server-Side Rendering

O *Server-Side Rendering* (SSR) corresponde ao processo de renderizar do lado do servidor todo o HTML de uma página, que inclui os principais recursos (estilos, imagens e JavaScript) de que uma aplicação necessita, evitando *round-trips* adicionais (Miller & Osmani, 2019).

O HTML gerado é posteriormente enviado para o cliente a fim de ser apresentado no ecrã (ver Figura 1).

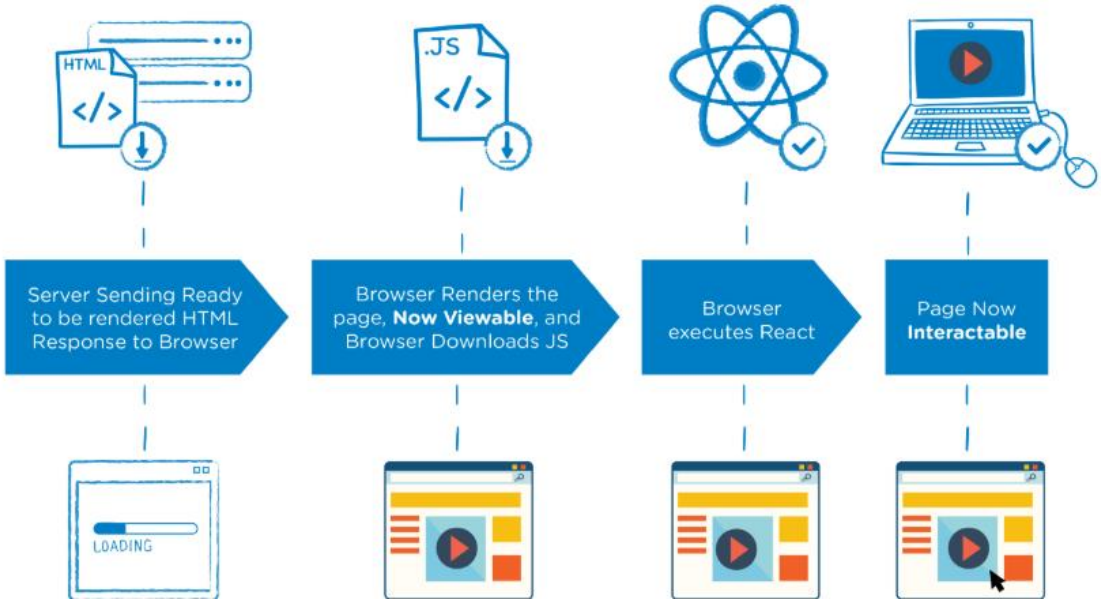


Figura 1 - *Server-Side Rendering* (Grigoryan 2017)

O SSR permite que um *site* possua um tempo de primeiro carregamento substancialmente menor, já que o *browser* ao analisar o HTML já não necessita de efetuar a transferência dos recursos necessários, e que seja totalmente indexável para efeitos de SEO (Rocha, 2018).

O *browser* fica incumbido somente de ao receber a resposta do servidor apresentar a página ao utilizador e paralelamente efetuar o *download* do JS necessário para que seja capaz de executar o código que torna a aplicação interativa (por exemplo, o *download* do React, no caso de o cliente utilizar esta *framework*, que depois de carregado construirá o Virtual DOM e efetuará o atribuição dos eventos aos respetivos elementos) (Grigoryan, 2017).

No entanto, esta prática de renderização pode revelar-se uma péssima escolha num cenário de elevada interatividade com o utilizador, uma vez que cada interação do utilizador com a aplicação exigirá a realização de um novo pedido ao servidor para obter o HTML atualizado.

2.1.4.2 *Client-Side Rendering*

Num contexto de *Client-Side Rendering* (CSR) o processo de renderização ocorre diretamente no *browser* usando JavaScript e toda a lógica de *fetching* de dados e *routing* é tratada no cliente (Miller & Osmani, 2019).

Após o carregamento de página *web*, enquanto não forem localizados e transferidos todos os recursos referenciados no seu ficheiro HTML (sejam eles CSS, JS ou imagens), não é dado nenhum *feedback* ao utilizador, permanecendo a página em branco (Burkholder, 2018) (ver Figura 2).

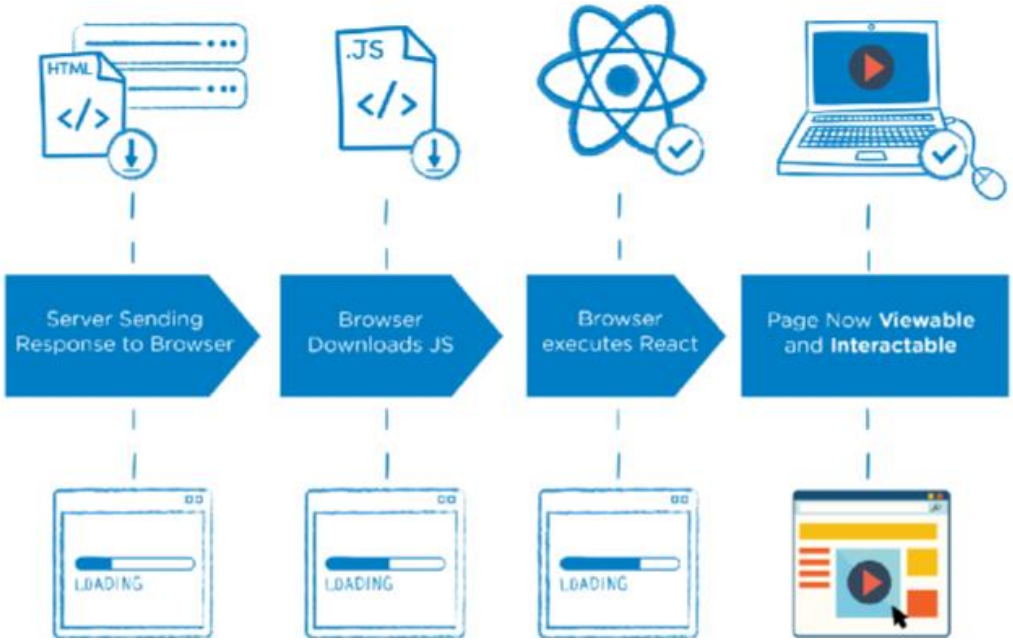


Figura 2 – *Client-Side Rendering* (Grigoryan 2017)

Como o tempo de carregamento inicial é substancialmente maior, a indexação por parte dos motores de pesquisa torna-se mais difícil.

Apesar disso, concluído o carregamento inicial a aplicação reage de forma quase imediatamente aos *inputs* do utilizador já que não está dependente de *round-trips* ao servidor.

Com o intuito de desmistificar a distinção entre SSR e CSR, Adam Zerner propõe a seguinte metáfora "*With server-side rendering, whenever you want to see a new web page, you have to go out and get it, this is analogous to you driving over to the supermarket every time you want to eat. With client-side rendering, you go to the supermarket once and spend 45 minutes walking around buying a bunch of food for the month. Then, whenever you want to eat, you just open the fridge.*" (Zerner, 2017)

Daqui se infere que, apesar da demora no carregamento inicial, o CSR tem a vantagem de reagir mais rapidamente às alterações promovidas pelo utilizador, devido a não estar dependente de sucessivos acessos ao servidor para que estas sejam efetivadas no HTML.

Daí que seja a opção acertada quando se pretende o desenvolvimento de uma aplicação dinâmica e interativa.

Contudo, para verdadeiramente usufruir das potencialidades de ambos os mundos pode-se optar por usar SSR para o primeiro carregamento da página e CSR para os subsequentes (Zerner, 2017).

2.1.5 Arquitetura Baseada em Camadas

A arquitetura baseada em camadas estabelece que os componentes sejam organizados em camadas horizontais, desempenhando cada uma um papel específico na aplicação, em conformidade com o que advoga o Princípio da Responsabilidade Única (Martin, 2000).

Embora, este padrão arquitetural não especifique o número e tipo de camadas, é frequente as aplicações serem decompostas em três camadas tidas como *standard*: camada de apresentação (*Presentation Layer*), camada de lógica de negócio (*Business Logic Layer - BLL*) e camada de acesso a dados (*Data Access Layer - DAL*) (Richards, 2015).

No que concerne as responsabilidades expectáveis de cada camada:

- a camada de apresentação está incumbida de fornecer uma interface que permita ao utilizador interagir com o sistema, apresentando a informação num determinado formato, sem se preocupar com a forma como os dados são obtidos;

- a camada de lógica de negócio, por sua vez, não necessita ter em consideração a origem dos dados ou o formato de apresentação dos mesmos, apenas de obter os dados e desempenhar operações e lógica de negócio sobre eles para depois transmiti-los à camada de apresentação;

- a camada de acesso a dados está incumbida de gerir o acesso à informação.

O acesso a cada camada é por norma fechado, podendo apenas ser efetuado pela camada colocada imediatamente acima, ainda que em situações excecionais o acesso possa ser aberto para permitir que os pedidos passem através dela para aceder à camada imediatamente inferior (Richards, 2015).

Num contexto de *Domain Driven Design* (DDD), uma vez que deve ser assegurada uma separação clara entre o domínio e o estado da aplicação, é comum a BLL dar lugar às camadas de Domínio e de Aplicação (como ilustrado na Figura 3) e a responsabilidade de aceder e persistir dados ser delegada na camada de infraestrutura (Avram & Marinescu, 2007).

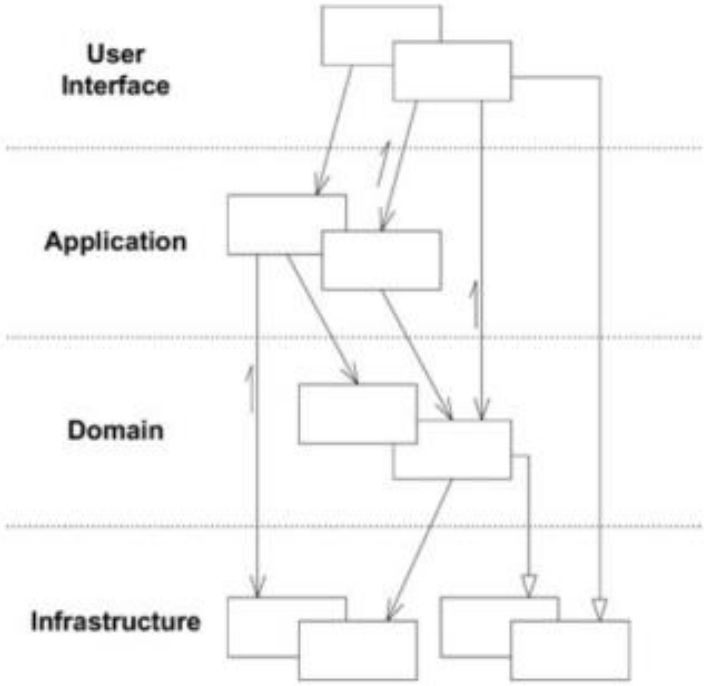


Figura 3 - Arquitetura em Camadas Segundo o DDD (Evans, 2003)

Neste caso, a camada de domínio fica encarregue da gestão da informação e estado dos objetos de domínio enquanto a camada de aplicação, não contendo lógica de negócio, fica somente responsável pela atividade e estado da aplicação.

2.1.6 *Domain Driven Design*

O desenho de *software* é uma arte e como tal, não pode ser ensinada ou aprendida através de fórmulas ou teoremas como se de uma ciência exata se tratasse. (Avram & Marinescu, 2007).

Nesse sentido, não existe um único caminho para converter as necessidades do mundo real num pedaço de código que as satisfaça, uma vez que existe uma panóplia de técnicas e metodologias passíveis de ser integradas no processo de desenvolvimento e um produto de *software* resulta sempre da simbiose entre as metodologias adotadas e o toque pessoal dos intervenientes.

Todos os programas de *software* estão relacionados com alguma atividade ou ideia que suscite interesse nos respetivos utilizadores (Evans, 2003).

Apesar de para o compilador o *software* desenvolvido não ir além de um conjunto sequencial de linhas de código (instruções), este não deve ser o foco das inquietações, sob pena de se ficar com uma visão demasiado redutora do processo de desenvolvimento de uma aplicação no seu todo.

Primeiramente, há que proceder ao levantamento de requisitos, e posteriormente à análise e ao *design*, fase na qual antes de se chegar a uma decisão final podem ocorrer refinamentos e a construção e experimentação de modelos.

Os processos de negócio ou problemas do mundo real incluídos no âmbito de um *software* correspondem ao seu domínio (Avram & Marinescu, 2007).

Ora, para construir um *software* com valor e qualidade, um amplo conhecimento do seu domínio é condição *sine qua non*, já que apenas dessa forma será possível tirar verdadeiro proveito da sua concretização.

Perante um elevado volume de informação e/ou lógica de negócio demasiado complexa de percecionar e gerir, é fundamental proceder à modelação do domínio.

A modelação traduz-se na obtenção de modelos que permitem mapear os conceitos de domínio resultantes da recolha da informação essencial junto dos especialistas do domínio em artefactos de *software*, para que o *software* seja capaz de refletir o domínio, ao incorporar os seus conceitos e elementos principais e estabelecer as respetivas relações (Avram & Marinescu, 2007).

O DDD propõe, portanto, que o *design* seja encetado com a definição do domínio e criação de uma abstração do mesmo, para que este possa ser repercutido no código.

O modelo de domínio consiste numa "abstração seletiva e rigorosamente organizada do conhecimento do domínio" (Evans, 2003), isto é, partindo do domínio como um todo seleciona apenas a informação relevante para a negócio, organiza-a, sistematiza-a, divide-a e agrupa-a em módulos lógicos (Avram & Marinescu, 2007).

Esta modularidade traduz-se numa maior manutenibilidade e assegura a separação de responsabilidades.

O modelo torna-se a peça-chave do desenho de um *software*, ainda que para se tirar o real proveito do mesmo seja imprescindível definir uma estratégia que permita a especialistas de domínio, *designers* e programadores comunicar o modelo sem ambiguidade, nomeadamente através do recurso a diagramas, casos de uso, esquemas, etc., e da definição de uma linguagem comum usada na discussão de problemáticas inerentes ao domínio (denominada linguagem ubíqua).

2.1.6.1 Domínio e Subdomínios

O domínio é referente a aspetos do mundo real de uma solução (por exemplo, a monitorização do tráfego aéreo de uma determinada região), dando conta dos requisitos e critérios de aceitação do sistema a ser implementado.

Para que o domínio seja convenientemente definido, deve existir uma troca de conhecimento entre equipa de desenvolvimento, arquitetos de *software* e especialistas de domínio, na qual à medida que são inquiridos os especialistas, são revelados conceitos de negócio que, apesar de surgirem de forma não tratada e desorganizada, são essenciais para a compreensão do domínio.

Após o correto processamento da informação, chegar-se-á a um esboço da visão do domínio, o modelo do domínio, que não sendo uma visão completa detém os conceitos aglutinadores do domínio (Avram & Marinescu, 2007), que permitem definir a estrutura de um *software*.

Num determinado momento os especialistas de *software* poderão querer construir um protótipo, com vista a testar o funcionamento do modelo definido e eventualmente reajustá-lo sempre que a experimentação, comunicação e *feedback* assim o exijam.

O modelo torna-se o lugar onde o conhecimento dos especialistas de *software* e do domínio convergem e mostra como negócio, *design* e *software* podem unir esforços em prol de uma melhor solução.

E se um *design* bem concebido é sinónimo um desenvolvimento mais ágil, o *feedback* obtido do processo de desenvolvimento vai certamente contribuir para melhorar o *design* (Avram & Marinescu, 2007).

Um domínio extenso pode ter um modelo igualmente extenso mesmo após ter sido submetido a refinamentos e terem sido criadas abstrações.

Assim, para tornar viável a concretização deste modelo em código, Evans propõe encetar um processo que designa por destilação (Evans, 2003).

Num sentido mais literal, destilação consiste no processo de separações das substâncias que compõem uma mistura (Avram & Marinescu, 2007). Ora, transpondo o conceito para o *design* aplicacional, permite decompor o domínio em vários subdomínios.

Os subdomínios consistem em partes específicas do domínio, que dão resposta a problemas mais pequenos, possuem limites e responsabilidades bem definidas e que tipicamente espelham a própria estrutura organizacional - Lei de Conway (Vernon, 2015), nas quais a respetiva equipa define uma linguagem específica (ubíqua) para comunicar sem ambiguidade.

Por exemplo, o domínio automóvel pode ser decomposto nos subdomínios de logística, pesquisa e desenvolvimento, vendas, produção e marketing, que podem comunicar entre si (Norelus, 2019).

Cada subdomínio pode conter os seus próprios subdomínios e em função da importância da missão que desempenha no domínio pode ser classificado como domínio principal (*Core*), subdomínio genérico ou subdomínio de suporte, como ilustrado na Tabela 1.

Tabela 1 - Tipos de Subdomínios do DDD (Oliver, 2009)

TIPO DE SUBDOMÍNIO	CARATERÍSTICAS
DOMÍNIO PRINCIPAL (*CORE DOMAIN*)	Consiste na razão principal de existência da aplicação e a sua definição depende impreterivelmente da perspetiva/prisma de como é analisado o domínio. Englobando os conceitos essenciais ao domínio, que permitem acrescentar valor ao negócio e conferir vantagem competitiva à organização, devem ser alocados os recursos mais qualificados da equipa na sua implementação.
SUBDOMÍNIO GENÉRICO (*GENERIC SUBDOMAIN*)	Corresponde a partes do sistema que apesar de facilitarem o negócio, não podem ser consideradas nucleares e para as quais há uma elevada probabilidade de já existirem soluções chave-na-mão passíveis de ser aplicadas, modelos ou linhas orientadoras para a sua implementação.
SUBDOMÍNIO DE SUPORTE (*SUPPORTING SUBDOMAIN*)	São módulos que atuam como funções de suporte ao domínio principal. Por exemplo, no contexto de uma loja *online*, o catálogo de produtos é considerado *core* e os subdomínios de encomendas ou entregas correspondem a subdomínios de suporte.

2.1.6.2 Model-Driven Design

Após o processo de modelação se ter traduzido na obtenção de um modelo capaz de refletir o domínio, tem lugar a fase de implementação deste modelo no código.

Ao analisar o modelo construído, a equipa de desenvolvimento pode concluir que parte dos conceitos ou relações descritas não pode ser mapeada de forma inequívoca através do código.

Ora, o *design* de uma fração do sistema de *software* deve refletir tão fielmente quanto possível o que está definido no modelo para evitar ambiguidades, pelo que o código se torna uma expressão do modelo. Assim sendo, uma alteração no código pode acarretar uma potencial modificação no modelo.

O *Model-Driven Design* recorre a um conjunto de padrões, que visam apresentar os elementos principais da modelação de objetos e *design* de *software* segundo uma abordagem DDD, que Evans categoriza em *design* tático e *design* estratégico.

Segundo Evans os padrões *Entity, Value Object, Service, Aggregate, Factory* e *Repository* constituem o *design* tático do DDD. Já a linguagem ubíqua, os *Context Maps* e os *Bounded Contexts* são parte integrante do *design* estratégico.

A Figura 4 apresenta estes padrões e estabelece os relacionamentos entre eles.

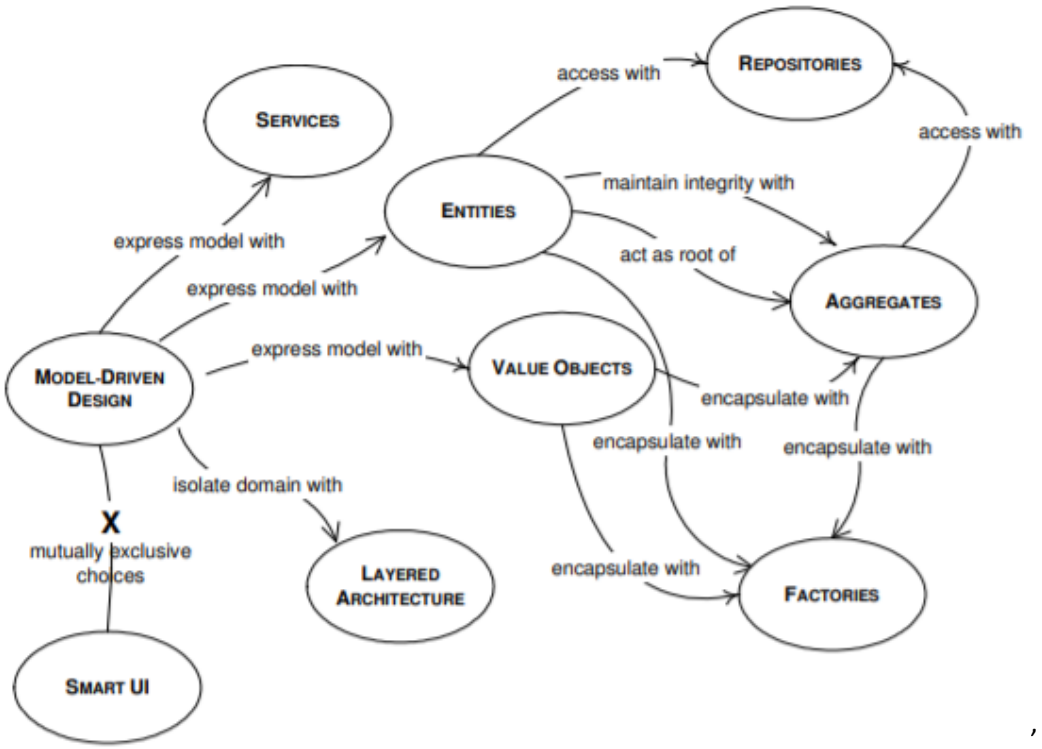


Figura 4 - Padrões do *Model-Driven Design* (Avram & Marinescu, 2007)

Numa arquitetura de MFEs, como o DDD surge essencialmente para a identificação dos diferentes MFEs, apenas o *design* estratégico será relevante no contexto deste documento.

2.1.6.3 DDD e Arquitetura em Camadas

Por norma, o domínio corresponde apenas a uma pequena fração de um *software* aplicacional, que coabita com código respeitante à infraestrutura, acesso à base de dados e à interface com o utilizador.

Numa abordagem OO, é frequente que UI, DAL e outras operações de suporte constem diretamente nos objetos de negócio, misturando-se código relacionado com o negócio com outras camadas (Avram & Marinescu, 2007).

Contudo, esta promiscuidade de código, atenta contra o princípio de separação de responsabilidades, fecha a porta à adição de novas funcionalidades e dificulta não só o esforço de compreensão como também o de manutenção.

A solução passa pela adoção de uma arquitetura em camadas, na qual o código relacionado com o domínio não está nem disseminado por toda a aplicação nem contaminado por operações que pouco ou nada têm haver com o negócio, uma vez que se encontra contido numa única camada, isolado do código relativo à UI, lógica aplicacional, infraestrutura e interação com a DB. Desta forma, a camada de domínio pode focar-se apenas em expressar o modelo de domínio, o que permite que o modelo evolua para ser rico e claro o suficiente para capturar o conhecimento essencial do negócio e colocá-lo em prática.

2.1.6.4 *Bounded Context*

Apurado o domínio da solução, é necessário limitar concetualmente o contexto em que os modelos que o compõem são aplicáveis, através da definição de *bounded contexts*.

Segundo Vaughn Vernon, um *bounded context* pode ser considerado uma delimitação linguística, uma vez que cada um possui a sua própria linguagem ubíqua (Vernon, 2016).

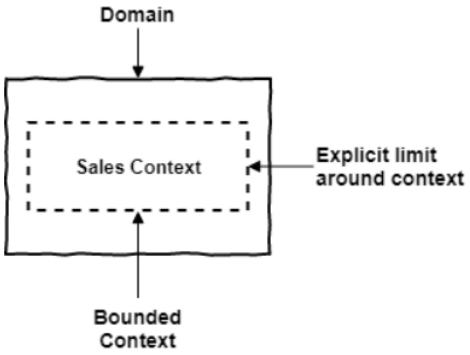


Figura 5 - *Bounded Context* (TheDomainDrivenDesign.IO, 2019)

Sendo frequente a confusão entre este conceito e subdomínio, para realmente se entender a diferença há que começar por distinguir domínio de modelo de domínio.

De acordo com Vernon, "os subdomínios vivem no espaço do problema enquanto os *bounded contexts* habitam o espaço da solução" (Vernon, 2015), isto é, os subdomínios correspondem a partes do domínio (ver secção 2.1.6.1) enquanto os *bounded contexts* estabelecem o contexto no qual os modelos definidos no processo de modelação ajudam a resolver o problema (Basic, 2018).

À medida que o modelo evolui, é necessário criar relações entre os *bounded contexts*, através de *context maps* (TheDomainDrivenDesign.IO, 2019).

Um *bounded context* é um artefacto de *software* que se assume como um meio de excelência para a definição do *scope* ou âmbito da funcionalidade de um micro serviço/micro *frontend*.

2.1.6.5 Linguagem Ubíqua

Sendo fundamental a cooperação entre especialistas de *software* e de domínio na definição do modelo do domínio, esta pode ser dificultada por barreiras linguísticas entre peritos em *software* e de domínio.

Os especialistas em *software* estão formatados para pensar e expressar-se usando conceitos como classes, métodos, algoritmos, padrões e tendendo a estabelecer uma relação direta entre um conceito do mundo real e um artefacto de *software*, procuram aferir que classes de objetos criar e quais as relações a criar, numa perspetiva puramente orientada a objetos.

Já os peritos em domínio não possuindo quaisquer noções de programação, tendem a adotar um jargão caraterístico do negócio na sua comunicação.

Para superar esta diferença no estilo de comunicação, o DDD propõe a criação de uma linguagem comum baseada no modelo (uma vez que o modelo é o lugar onde o *software* vai de encontro ao domínio), denominada linguagem ubíqua, que deve estar patente quer nas formas de comunicação usadas pela equipa (discurso, diagramas, etc.), quer no código implementado (Avram & Marinescu, 2007).

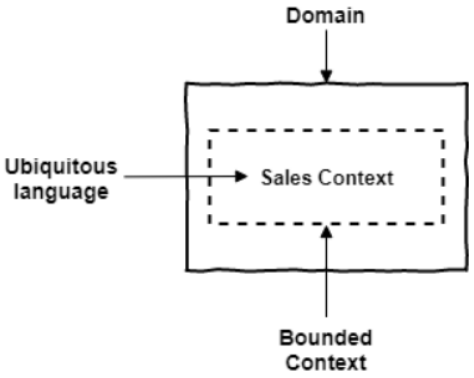


Figura 6 - Linguagem Ubíqua (TheDomainDrivenDesign.IO, 2019)

Esta linguagem inclui uma série de termos-chave que permitem a definição isenta de ambiguidades do domínio e do *design*, que deve evoluir à medida que o entendimento do domínio por parte da equipa aumenta. (Evans, 2003).

2.1.6.6 *Context Maps*

A necessidade de elaborar *context maps* prende-se com facilitar a compreensão do projeto no seu todo, ao traduzir as relações entre diferentes *bounded contexts* e assim permitir a construção mais correta do modelo de domínio.

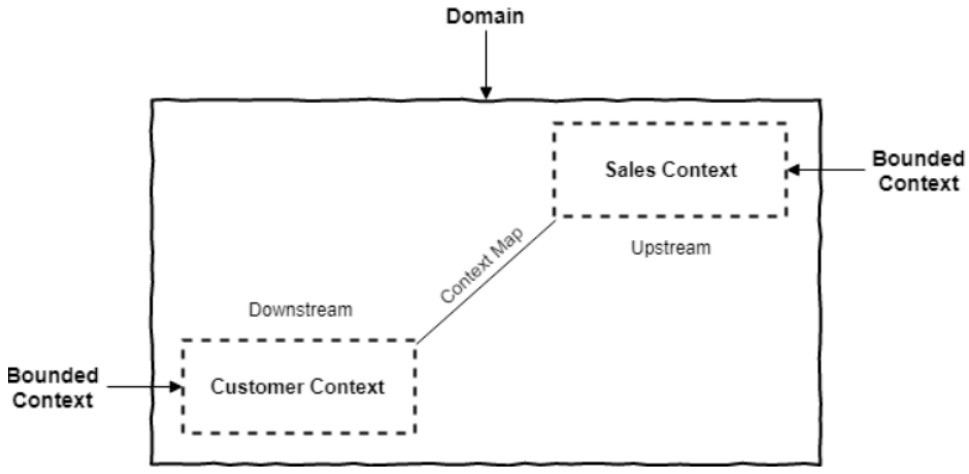


Figura 7 - *Context Map* (TheDomainDrivenDesign.IO, 2019)

2.1.7 *User-Centered Design*

O design centrado no utilizador, como o próprio nome indica, defende a modelação das equipas e sistemas em torno das necessidades dos utilizadores.

Representa uma alternativa ao DDD de identificação de micro *frontends*, particularmente útil em sistemas legados (Mezzalira, 2019b).

Recorrendo quer a técnicas como "*Design Thinking"* ou "Jobs To Be Done" (Geers, 2020a), quer a ferramentas como o Google Analytics é possível com base nas motivações ou comportamentos dos utilizadores identificar mais facilmente candidatos *a* micro *frontend*s (Mezzalira, 2019b).

2.2 Evolução Arquitetural até à Arquitetura de Micro *Frontends*

Convencionalmente as aplicações *web* apresentam uma arquitetural ora puramente monolítica, em que o desenvolvimento *end-to-end* de toda a aplicação está a cargo de uma única equipa, ora subdividida em BE e FE, estando estas camadas (que possuem ambas um cariz monolítico) ao cuidado de diferentes equipas.

No entanto, o surgimento da arquitetura orientada a micro serviços, uma especialização da SOA, entrou em rutura com estas práticas, derrubou o BE monolítico e com o crescimento que se tem verificado na sua adoção abriu portas para que o FE monolítico possa potencialmente em alguns casos ter os dias contados, com a entrada em ação da arquitetura de micro *frontend*s.

Posto isto, nesta secção serão abordadas estas arquiteturas numa perspetiva de evolução desde a arquitetura monolítica até à arquitetura de micro *frontends*.

2.2.1 Arquitetura Monolítica

Uma aplicação monolítica caracteriza-se por numa única *codebase* congregar múltiplos serviços, que comunicam com sistemas externos ou consumidores, através de diferentes interfaces como *Web* Services, páginas HTML ou REST API (Villamizar et al., 2017)

A abordagem monolítica é considerada uma boa opção no arranque de um projeto aplicacional, dado que o facto de existir uma única *codebase* de pequena dimensão, faz com que a arquitetura e código existente sejam amplamente conhecidas pela equipa, o que indubitavelmente agiliza o desenvolvimento, implantação e escalabilidade de uma aplicação.

A implantação de um único artefacto para ambiente de teste ou produção é simples. Contudo, qualquer alteração num único componente, exige que toda a aplicação seja reimplantada, o que dificulta o CD, faz com que os ciclos de lançamento sejam mais longos e o processo de desenvolvimento se torne mais lento (Richardson, 2015d).

Relativamente à escalabilidade, numa arquitetura monolítica restringe-se à adição de novas instâncias contendo o artefacto. Os componentes da aplicação que necessitam de mais recursos escalam juntamente com os que poderiam cumprir a sua função usando menos recursos, o que se traduz numa gestão ineficiente de recursos, que comporta custos adicionais para a organização.

Quanto à informação, é persistida numa única base de dados, o que permite que as transações sejam processadas mais facilmente visto que a maior parte dos SGBD asseguram a atomicidade, consistência, isolamento e durabilidade (ACID) das transações.

Uma aplicação monolítica pode apresentar diversos tipos de dados, que dadas as suas especificidades deveriam idealmente ser armazenados ora numa base de dados NoSQL, ora numa relacional. Contudo, seguindo esta abordagem a equipa de desenvolvimento necessita de escolher forçosamente uma única tipologia e usá-la para todos os tipos de dados.

Se é verdade que a maioria das aplicações no início parece simples, à medida que crescem a complexidade aumenta. Um caminho comum para lidar com a complexidade numa aplicação que apresenta uma arquitetura monolítica consiste em dividir a aplicação em diferentes camadas (Fowler, 2002) (ver secção 2.1.5).

Todavia, com o crescimento da aplicação e da organização, descurando a arquitetura e/ou a qualidade do código, o acoplamento torna-se de tal forma elevado que a adição/modificação de funcionalidades se torna complexa e o ciclo de desenvolvimento mais lento, uma vez que é cada vez mais difícil encontrar o local correto para aplicar essas alterações e ter uma real perceção dos comportamentos despoletados (Richardson, 2015d).

Este cenário poderia ser evitado, mantendo uma codificação modular em cada camada, efetuando um *refactoring* contínuo do código para o manter limpo e através de uma boa cobertura de testes. Contudo, como não há limites claros entre módulos em aplicações monolíticas, a modularidade fica comprometida (Kalske, 2017).

Para precaver eventuais problemas em ambiente de produção é necessário reforçar os testes de regressão, o que naturalmente atrasa o processo de lançamento de novas funcionalidades, já que o tempo necessário para efetuar a *build* e executar testes na *pipeline* da CI tende a aumentar.

Por outro lado, a possibilidade de várias pessoas estarem a submeter código em simultâneo pode dificultar a descoberta do *commit* por detrás de uma *build* falhada (Kalske, 2017). Daí que a adoção de *Continuous Deployment* seja complexa.

2.2.2 Arquitetura Orientada a Micro Serviços

Para dar resposta aos desafios do monolítico, as empresas têm vindo a migrar as soluções para micro serviços.

A arquitetura baseada em micro serviços estrutura uma aplicação BE como um conjunto de pequenos serviços com baixo acoplamento entre si (Richardson, 2015c) e com um contexto, interface e dependências bem definidos e em conformidade com os princípios do *Domain-Driven Design* (DDD) e *DevOps*.

Cada micro serviço estando focado numa única tarefa de negócio [New15a] respeita o "*Single Responsability Principle*" e possuindo uma interface e dependências claras (relativamente a outros micro serviços ou recursos externos), permite um desenvolvimento modular (Norelus, 2019).

Relativamente ao domínio, atualmente o ambiente de negócio é complexo e competitivo, dando pouco aso a erros. Para mitigar o risco, é essencial adotar uma abordagem de *design* de *software* preparada para lidar com domínios complexos, nomeadamente o DDD.

As equipas tornam-se autónomas e passam a ser responsáveis pela maioria das decisões e pela implementação de um conjunto de micro serviços no âmbito do seu domínio.

A modularidade alcançada pelos micro serviços, agiliza o processo de desenvolvimento ao isolar alterações efetuadas num módulo dos restantes (Baldwin & Clark, 2000), que são facilmente contidas devido aos limites claros existentes entre cada um dos micro serviços (Richardson, 2015d). Essa definição clara de limites determina que a comunicação entre si ocorra apenas quando estritamente necessário e por via das interfaces que expõem, o que permite alcançar elevada coesão e baixo acoplamento.

No entanto, estabelecer limites claros entre módulos com interfaces REST, pode conduzir a problemas de *performance* se a granularidade dos serviços for demasiado fina (Richards, 2016). Se um determinado UC de negócio necessitar de efetuar chamadas que percorram vários serviços, o tempo de execução das chamadas vai aumentar drasticamente, pelo que a solução pode passar por repensar a granularidade, nomeadamente através da fusão de diferentes serviços (desde que não comprometa os pressupostos da arquitetura de micro serviços) ou usar *message-queues*.

Em determinadas situações, apesar de os programadores serem instruídos no sentido de cumprir o princípio DRY - "*Don't Repeat Yourself*" (Hunt, 2000), que promove a reutilização de código, pode fazer sentido alguma duplicação (Newman, 2015), uma vez que num panorama em que vários micro serviços partilhem o mesmo código, se for efetuado uma alteração a um serviço, como se verifica um elevado acoplamento todos os serviços serão afetados.

Tratando-se o baixo acoplamento, alta coesão, princípio da responsabilidade única e modularidade de princípios a respeitar em qualquer arquitetura, o facto de numa abordagem

baseada em micro serviços existir uma perceção clara de quem é o serviço responsável pela implementação de uma determinada tarefa de negócio e de os serviços corresponderem a pequenas unidades de código, permite que estas qualidades permaneçam bem patentes no código ao longo do processo de desenvolvimento e consequentemente que os ciclos de desenvolvimento sejam mais curtos (Newman, 2015).

Existindo um elevado número de serviços, pequenos e isolados, a adoção de CI é fundamental (Balalaie et al., 2016) e apesar de o número de *pipelines* de CI ser superior, como são mais verificadas mais rapidamente, obtêm-se ciclos de lançamento e *feedback* mais rápidos.

Esta particularidade, por um lado, possibilita uma resposta célere às necessidades do utilizador e, por outro, permite às organizações que ao distinguirem prontamente as funcionalidades que vão dar resposta aos problemas do utilizador das que são acessórias (Olsson et al., 2012), possam definir prioridades na implementação e obter vantagens competitivas.

A arquitetura de micro serviços permite ainda uma escalabilidade mais granular e flexível comparativamente à monolítica (Newman, 2015), já vez que a natureza dos micro serviços permite escalar um único serviço, verticalmente e horizontalmente.

A escalabilidade horizontal consiste em aumentar o número de instâncias de um serviço em função da sua exigência em termos de recursos (Toffetti et al., 2015). Assim, podem ser criadas mais instâncias em serviços que exijam mais recursos, ou reduzidas se se verificar o contrário.

Quanto à escalabilidade vertical não é tão elástica como a horizontal, visto que o aumento ou redução da capacidade numa instância existente requer que esta fique inoperacional durante um determinado período (Ranchal et al., 2015).

Uma das principais vantagens de uma escalabilidade de baixa granularidade prende-se com a possibilidade de distribuir os serviços críticos por vários *hosts*, servidores físicos e *data centers*, e, assim minimizar os períodos de inoperacionalidade (Newman, 2015).

Uma arquitetura deste tipo abre espaço à inovação ao possibilitar a adoção de diferentes tecnologias e linguagens na implementação dos vários serviços (Newman, 2015), selecionando as mais adequadas para dar resposta aos desafios do negócio, ainda que, por vezes, existam algumas restrições neste processo de seleção que necessitam ser tidas em conta.

No que toca à interação com bases de dados, os micro serviços podem socorrer-se de diferentes SGBD, relacionais ou *NoSQL*, que melhor se adequem à natureza dos dados.

No entanto, a implementação de transações de negócio que afete vários serviços não só não é direta, como deve ser evitada (Richardson, 2015b). Tal deve-se ao facto de que, segundo o teorema de CAP, num sistema distribuído, perante uma falha ser impossível garantir a consistência e disponibilidade em simultâneo (Gilbert & Lynch, 2012).

Quanto à testabilidade, dado que os micro serviços possuem uma extensão reduzida e estão bem delimitados, são facilmente testáveis e a necessidade de recorrer a testes de regressão é manifestamente reduzida, uma vez que quaisquer alterações são autocontidas (Kalske, 2017).

Relativamente à dimensão dos micro serviços, não existindo um *standard* claro (Lewis & Fowler, 2014), devem ser pequenos o suficiente para que uma única equipa possa estar responsável pelo serviço na sua totalidade (Newman, 2015). No entanto, usando a Regra das Duas Pizzas como referencial, uma equipa não deve conter mais do 10 ou 12 elementos.

Apesar das mais-valias passíveis de ser obtidas com os micro serviços, a sua implementação acarreta complexidade e nem sempre é a opção arquitetural mais adequada, pelo que a sua adoção deve ser devidamente estudada.

Além disso, a adoção de micro serviços por si só não permite obter quaisquer ganhos de escalabilidade no FE, algo que no entender de Luca Mezzalira, se deve à tendência para que grande parte da lógica aplicacional se concentre no servidor e à menor interatividade das aplicações comparativamente com as exigências atuais dos utilizadores.

Por outro lado, o *Frontend* Monolítico não permite lançar MVPs mais rapidamente.

2.2.3 Arquitetura de Micro *Frontends*

Como já referido anteriormente, as aplicações *web* baseiam-se no modelo cliente-servidor, adotando uma arquitetura em camadas, que, grosso modo, as organiza em três camadas principais: *Frontend*, *Backend* e Base de Dados.

No arranque de novos projetos aplicacionais sendo o objetivo primordial obter *feedback* tão rapidamente quanto possível para colmatar falhas numa fase precoce e obter vantagem competitiva, é fundamental a adoção de uma abordagem arquitetural que permita agilizar o processo de desenvolvimento e entregar MVPs ao cliente num curto espaço de tempo.

Enquanto o projeto possui uma extensão limitada de requisitos e a lógica de negócio se mantém simples de apreender, é frequente a adoção de uma arquitetura monolítica, da qual resulta um único artefacto de *software* passível de ser implantado. Neste cenário, *Backend* e *Frontend* sendo desenvolvidos pela mesma equipa (de grande dimensão) e estão contidos no mesmo repositório (ainda que em pastas separadas), o que coloca lado a lado lógica de interface com o utilizador e lógica de negócio.

No entanto, com o aumento da complexidade da lógica de negócio e da extensão do código, o acoplamento eleva-se de tal ordem que a qualidade do código caindo drasticamente, faz com que qualquer alteração e/ou adição de funcionalidades se torne insustentável, pois para precaver comportamentos inesperados é necessário submeter toda a aplicação a testes exaustivos antes de proceder à novas *releases*. Além disso, a escalabilidade torna-se ineficiente e a manutenção demasiado árdua.

Numa tentativa de minorar o esforço de manutenção e acelerar o lançamento para cliente, as empresas começaram a promover uma abordagem de desenvolvimento autónomo de FE e BE, por equipas distintas.

Quer a abordagem puramente monolítica (ver diagrama à esquerda na Figura 8), quer a abordagem *Front-Back*, isto é, *Frontend-Backend* (ver diagrama ao centro na Figura 8), caracterizam-se pela existência de uma única base de dados. A diferença reside no facto de na primeira, o desenvolvimento ser levado a cabo por uma única equipa, por oposição à última em que este se encontra repartido por duas equipas, uma especializada em FE e outra em BE.

Esta abordagem FE-BE representa uma melhoria significativa ao introduzir não só um isolamento claro entre lógica de domínio e lógica de apresentação, mas também a possibilidade de expor o código desenvolvido no *Backend* através de *APIs* com as quais os diversos tipos de cliente podem comunicar.

Esta comunicação pode ocorrer diretamente ou via *API Gateways*, *Backend For Frontend* (BFF), GraphQL ou qualquer outra forma de *middleware*.

Contudo, na sequência do alargamento do número de funcionalidades, começam a registar-se problemas de escalabilidade no BE inerentes ao facto de um único sistema disponibilizar serviços consumidos por diversos tipos de *frontends* (com necessidades distintas).

Com vista a solucionar os constrangimentos no BE, começa-se por partir o monolítico em múltiplos micro serviços e por criar uma infraestrutura responsável não só pela sua integração, mas também por centralizar o acesso aos serviços por parte das aplicações-cliente.

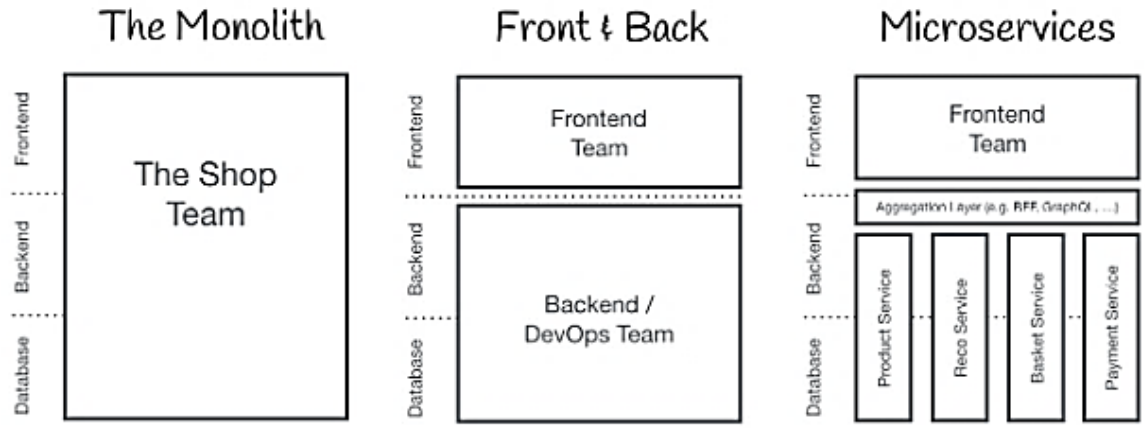


Figura 8 - Evolução da Arquitetura de Aplicações *Web* (Geers, 2018)

Estes micro serviços detém a sua própria base de dados (Richardson, 2015b) e podem ser desenvolvidos usando *stacks* tecnológicas distintas por equipas independentes e especializadas no contexto de domínio no qual se inserem (ver diagrama à direita na Figura 8).

No entanto, persiste a existência de uma aplicação puramente monolítica do lado do FE, *Frontend* Monolítico (Figura 9), que à semelhança do que se verifica num cenário de BE

monolítico, apresentará problemas de escalabilidade, dificultará a extensão e manutenção e exigirá um enorme esforço de comunicação entre PO, equipa de BE e de FE que introduzirá entropia no processo de desenvolvimento.

Assim, para tirar o máximo partido das potencialidades que os micro serviços têm para oferecer, mais concretamente, para se obter um desenvolvimento mais fluído e um ciclo de *feedback* contínuo, que pressupõe uma estreita colaboração entre *developers*, *testers* e/ou utilizadores, é necessário adotar transpor esta filosofia para o FE.

Só assim é possível alcançar um lançamento *end-to-end* que vá de encontro ao princípio "*Release early, release often and listen to your customers*" (Raymond, 1999), que constitui uma máxima em metodologias ágeis como o SCRUM (Lynch, 2019).

Nesse contexto, é proposto em 2016 pela ThoughtWorks o conceito de micro *frontends*, fortemente inspirado em abordagens como *Self-Contained Systems* (SCS) e *Atomic Design*, que se propõe fornecer um conjunto de "técnicas, estratégias e receitas para a construção de uma aplicação *web* moderna por várias equipas que podem enviar funcionalidades de forma independentes" (Geers, 2018).

Este conceito reinventa por completo a arquitetura em camadas, visto que a disposição horizontal baseada nas capacidades técnicas dá lugar à vertical, na qual cada camada abrange o conjunto de funcionalidades inseridas num determinado *bounded context* resultante não só do processo de modelação do DDD mas também (num cenário de conversão de uma aplicação já existente para MFEs) da análise do comportamento dos utilizadores ao utilizar a aplicação – *User-Centered Design* (Mezzalira, 2019b) (Figura 9).

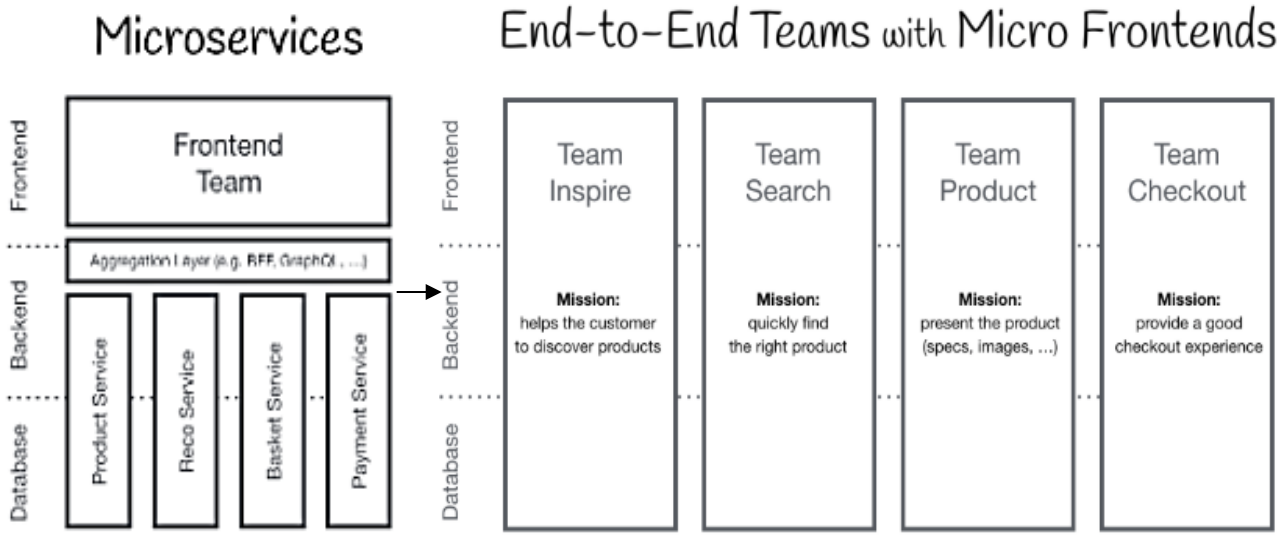


Figura 9 – Evolução da Arquitetura de Micro Serviços para Micro *Frontends* (Geers, 2018)

Daí que Luca Mezzalira defina micro *frontends* como "uma representação técnica de um subdomínio de negócio que fornece fronteiras bem definidas com contratos claros e na qual deve ser evitada a partilha de lógica entre subdomínios" (Mezzalira, 2018).

Cam Jackson, por sua vez, define micro *frontends* como um "estilo arquitetural no qual aplicações FE entregáveis de forma independente são compostas numa maior" (C. Jackson, 2019), isto é, perceciona uma aplicação *web* baseada em MFEs como o resultado da composição de funcionalidades que são mantidas por equipas independentes e especializadas numa área concreta do negócio (Geers, 2018).

Estas equipas multidisciplinares compostas por *designers*, *product owner (PO)* e outros especialistas de domínio, *developers* e *testers* estão incumbidas do processo de desenvolvimento *end-to-end* das funcionalidades, isto é, estão responsáveis por tudo o que seja necessário fazer para entregar valor aos clientes.

Como cada funcionalidade é implementada de forma autónoma, modular e bem delimitada, é possível alcançar uma evolução independente das aplicações (Dörnenburg, 2019), já que podem ser encetadas alterações num dado serviço sem necessidade de coordenação, o que se repercute num aumento do *throughput* e na redução do tempo do ciclo de *feedback*, além de que a alteração das tecnologias usadas na sua implementação deixa de estar dependente da reescrita integral do código da aplicação *web*.

Quanto à implantação, à semelhança do que acontece com os micro serviços, é efetuada de forma independente em cada micro *frontend*, que detém a sua própria *pipeline* de CI onde o código é compilado, submetido a testes e em caso de sucesso implantado para produção. É nesse ambiente que é feita a sua composição numa única aplicação (C. Jackson, 2019).

No que concerne a composição ou integração de micro *frontends* é efetuada numa aplicação contentor recorrendo a diferentes técnicas aplicadas no servidor ou no cliente (*browser*), que serão abordadas na secção 5.1 deste documento.

Assim, as ideias principais a reter são as seguintes (Geers, 2018; Yang et al., 2019):

- Cada equipa pode desenvolver de forma independente o seu micro *frontend*, definindo a respetiva *stack* tecnológica;
- Os componentes desenvolvidos devem ser autocontidos e assegurar um baixo acoplamento e elevado isolamento;
- Deve-se optar pela comunicação entre MFEs usando eventos do *browser* sempre que possível;
- Devem ser discutidas estratégias que permitam acautelar eventuais colisões ou problemas decorrentes de lógica partilhada;
- Deve-se assegurar a construção de *sites* resilientes e implementados de acordo com o *Progressive Enhancement*;
- Cada aplicação MFE tem a sua própria *pipeline* de CI/CD.

3 Análise de Valor

Sendo a Análise de Valor passível de ser aplicada a processos de desenvolvimento, este capítulo destina-se a:

- descrever esse processo começando por introduzir o conceito;
- no âmbito das fases de Orientação e Análise Funcional, selecionar o produto que será desenvolvido e identificar as funcionalidades mais importantes a implementar.

3.1 Definição de Análise de Valor

Segundo a *Society of American Value Engineers* (SAVE), o conceito de Análise de Valor (VA) corresponde a "um processo sistemático, formal e organizado de análise e avaliação com o intuito de melhorar produtos, serviços e ou processos".

Este processo visa "potenciar o valor para cliente ao relacionar os benefícios com os correspondentes custos associados, de forma a investir o dinheiro onde fizer a diferença" (Fallon, 1980), ou seja, destina-se a aumentar o valor de um produto, serviço ou serviço para o cliente pelo menor custo possível, uma vez que ao identificar e eliminar funcionalidades que não representam um real valor e cuja implementação traz custos associados, permite oferecer ao cliente um produto com qualidade superior e desenvolvido de forma mais eficiente (Rich & Holweg, 2000).

A abordagem VA, enquanto processo, compreende cinco etapas: Orientação, Análise Funcional, Criação de Alternativas, Análise e Avaliação de Alternativas e Implementação (Rich & Holweg, 2000), apresentadas na Figura 10.

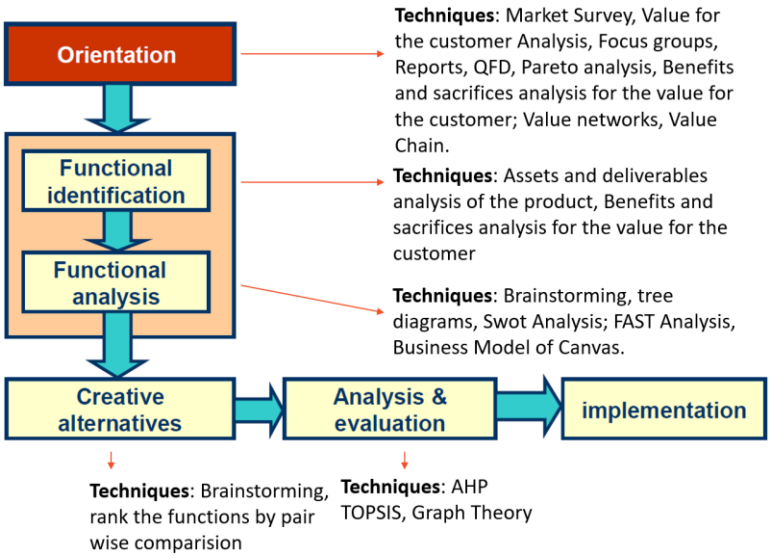


Figura 10 - Etapas da Análise de Valor (Rich e Holweg 2000)

3.2 Orientação

A primeira fase do processo de VA corresponde à fase de orientação, destinada à formação da equipa de VA e seleção do produto, serviço ou processo que será objeto de análise.

Apesar de a VA poder ser aplicada a praticamente qualquer produto, serviço ou processo existem certos critérios e atributos comerciais que podem potenciar o valor do produto (Rich & Holweg, 2000).

Nas últimas décadas, têm-se assistido a uma aposta forte na inovação por parte das organizações, já que a transformação de ideias em produtos, serviços ou processos novos ou melhorados, lhes pode conferir vantagem competitiva (Teza et al., 2015).

3.2.1 *Fuzzy Front End*

Nesse sentido, Koen et al., propuseram um processo de inovação dividido em 3 fases: *Fuzzy Front End* (FFE*)*, *New Product Development* (NPD) e comercialização do produto, como indicado na Figura 11.

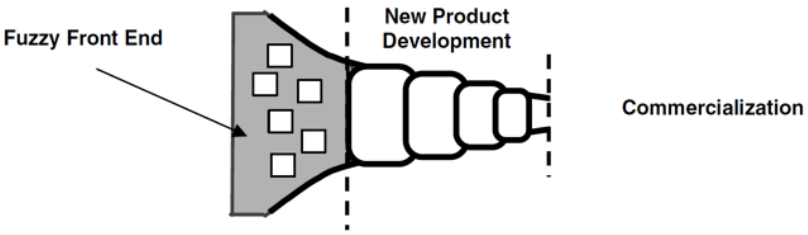


Figura 11 - Processo de Inovação (P. A. Koen et al., 2002)

A primeira parte, denominada *Fuzzy Front End* (FFE), é tida como a mais relevante em termos de oportunidades para melhorar o processo de inovação e é definida pelas atividades que antecedem o formal e bem definido *New Product Development* (NPD), regra geral, menos estruturadas e mais difíceis de prever. Da realização destas atividades, resulta o desenvolvimento de um conceito partindo de uma oportunidade identificada previamente, assumindo um NCD um papel preponderante nesta matéria (Smith & Reinertsen, 1991).

No decurso do FEE não há previsões da data de comercialização e as expetativas em termos de lucros são algo incertas e especulativas e o trabalho é sobretudo experimental.

Já no NPD o progresso traduz-se no cumprimento das tarefas previstas para a *milestone*, há uma noção mais tangível da data de lançamento, as previsões tornam-se sucessivamente mais aproximadas e realistas à medida que se aproxima o momento da comercialização, o trabalho é mais metódico e levado a cabo por equipas dedicadas a produtos ou processos.

Por fim, surge a fase de comercialização, na qual o produto novo ou submetido a melhorias é comercializado.

3.2.2 *New Concept Development Model*

O *New Concept Development Model* (*NCD Model*) providencia uma nomenclatura comum, que permite evitar erros de interpretação e ou compreensão, uma visão holística, isto é, integral, dos componentes principais do FFE (P. Koen, 2001).

Este modelo é constituído por três áreas distintas, mais especificamente o Motor (*Engine*), a Roda e o Aro que a delimita, conforme pode ser observado na Figura 12.

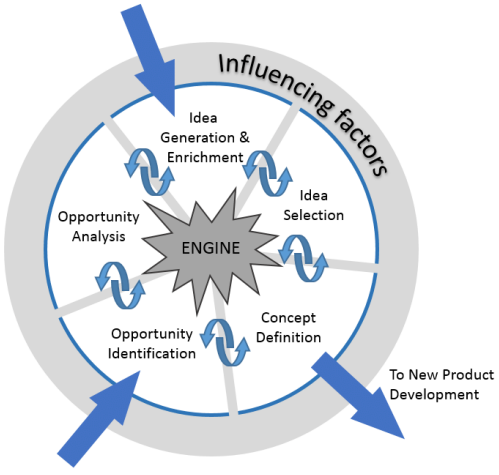


Figura 12 - Modelo NCD (P. Koen, 2001)

O motor representa a energia necessária para colocar o processo de FFE em andamento, que advém da liderança, cultura e estratégia de negócio da organização.

A roda, por sua vez, subdivide-se nos cinco elementos de atividade: Identificação da Oportunidade, Análise da Oportunidade, Geração e Enriquecimento de Ideias, Seleção da Ideias e finalmente Conceito e Desenvolvimento Tecnológico. Visto que o mote deste documento consiste no estudo de uma técnica de desenvolvimento de *software*, definida *à priori*, as etapas de Geração e Enriquecimento e Seleção de Ideias não sendo aplicáveis, não serão abordadas.

O aro corresponde aos fatores de influência, que compreendem fatores internos intimamente ligados às capacidades de uma organização, a par de fatores da envolvente externa, mais especificamente de natureza política, económica, tecnológica, cultural e ou legal.

O facto de o esquema ser circular pretende enfatizar a interação constante entre os cinco elementos.

As setas apontadas para o esquema representam os potenciais pontos de partidas no NCD, que pode começar pela identificação da oportunidade ou pela geração de ideia.

Contudo, assumindo-se que o maior ganho será obtido de necessidades ainda não satisfeitas no mercado, preferencialmente um projeto inovador deve começar pela identificação de oportunidades e somente após a análise das mesmas se proceder à formulação de ideias que permitam tirar partido das mesmas.

Já a seta com origem no esquema faz alusão ao momento em que estando definido o conceito do produto, serviço ou processo pode seguir para a etapa seguinte do processo de inovação.

3.2.3 Identificação de Oportunidades

A identificação de oportunidades, partindo de uma análise ao mercado, mais especificamente do ambiente transacional (que engloba clientes, fornecedores e concorrência direta) e contextual (contexto político, económico, social e tecnológico), visa identificar as oportunidades que vão de encontro à missão, visão estratégica e valores da organização.

Esta identificação pode ser mais facilmente concebida recorrendo a técnicas formais definidas pela organização (alinhadas com os fatores de influência) e outras técnicas menos formais, como simples discussões em grupo ou até a mera descoberta de necessidades do cliente às quais ainda não foi dada resposta (P. A. Koen et al., 2002).

Nas últimas décadas têm-se assistido a uma viragem relativamente aos modelos adotados pelas organizações no processo de desenvolvimento de *software*.

Se outrora o modelo em cascata permitia suprir as necessidades em praticamente todos os casos, num contexto de globalização como o atual em que a concorrência entre as organizações é cada vez mais feroz e perante uma cada vez maior complexidade de negócio e exigência dos consumidores, estas têm que se procurar reinventar no sentido de se diferenciarem.

A adoção de modelos ágeis como o SCRUM e o Kanban, tem assumido um papel preponderante naquilo que a essa reinvenção diz respeito, visto que de acordo com diversos estudos permitem reduzir custos, aumentar a produtividade, qualidade e satisfação dos vários *players* (Petrova, 2019).

Por oposição ao modelo em cascata, estas metodologias suportando-se num desenvolvimento iterativo e incremental, oferecem um fluxo contínuo de *feedback* (Cohen & Costa, 2012), que se traduz na obtenção de soluções de *software* mais alinhadas com os requisitos e necessidades do cliente. Daí que se tenham tornado as preferidas pela indústria.

Estando a maioria dos encargos relacionados com *software*, cerca de 67% (Alija, 2017), relacionados com manutenção (como comprova a Figura 13), é fundamental complementar e reforçar o contributo dado pelas metodologias ágeis nessa reinvenção através da adoção de técnicas e práticas de desenvolvimento que permitam minimizar esse custo.

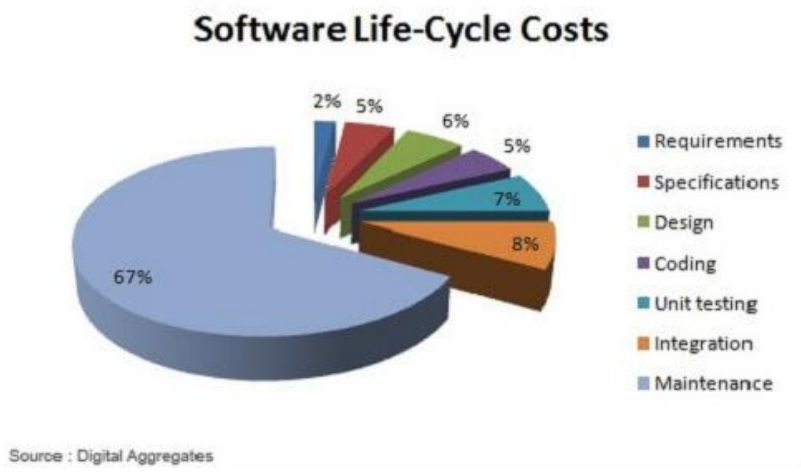


Figura 13 - Custos do Ciclo de Vida de um *Software* (Björklund, 2019)

Conforme referido na seção 2.2.3 deste documento, a adoção de micro *frontends* pode revelar-se uma mais-valia, já que propondo o desenvolvimento de peças de *software* modulares, com responsabilidades e limites bem definidos, altamente coesas e com baixo acoplamento entre si, facilmente escaláveis, extensíveis e mantidas, passíveis de evoluírem de forma independente e desenvolvidas por equipas especializadas no subdomínio de negócio que visam servir, pode permitir a agilização, eficiência e eficácia no processo de desenvolvimento *web* (Geers, 2020a).

No entanto, sendo um conceito recente, a literatura e na comunidade científica não dispõem de uma quantidade significativa de informação relativa ao real impacto da sua adoção no desenvolvimento *web* face a outras abordagens mais convencionais, pelo que o estudo desta técnica ou estilo arquitetural no desenvolvimento se assume como uma oportunidade.

3.2.4 Análise de Oportunidades

Definida a oportunidade, é tempo de analisá-la e avaliá-la no sentido de dados os impactos decorrentes da sua implementação, perceber se poderá, de facto, assumir-se como uma mais-valia. (P. A. Koen et al., 2002)

Para tal, é frequente recorrer-se a estudos de mercado, experimentações científicas e dados estatísticos.

Tendo em conta que existe pouca informação relativa às repercussões da adoção de uma abordagem a micro *frontends* no processo de desenvolvimento *web* nas organizações e esta radica da arquitetura de micro serviços, que tendo sido proposta anteriormente, tem sido objeto de diversos estudos, obras literárias e artigos a partir dos quais é possível extrair informação, é pertinente a sua análise.

Como referido na seção 2.2.2, destinada a explicar os pressupostos da arquitetura orientada a micro serviços, esta surge para procurar colmatar os problemas decorrentes do desenvolvimento monolítico do BE, ao providenciar uma aplicação segmentada em vários micro serviços, inseridos nos vários contextos de domínio identificados e delimitados de forma clara (de acordo com os pressupostos do DDD enunciados na seção 2.1.6), facilmente extensíveis e escaláveis e que podem ser enviados para o cliente de forma independente e mais célere, o que se reflete num encurtamento do *time to market* , conferindo assim uma vantagem competitiva.

Segundo um estudo conduzido pela NGINX em 2015 (cujo gráfico está apresentado na Figura 14) , estima-se que 68% das organizações já usem ou estejam a investigar esta arquitetura, uma utilização com maior relevo nas organizações de média e pequena dimensão, 50 e 44% respetivamente, ainda que nas de maior dimensão já se registe um valor na ordem dos 36%.

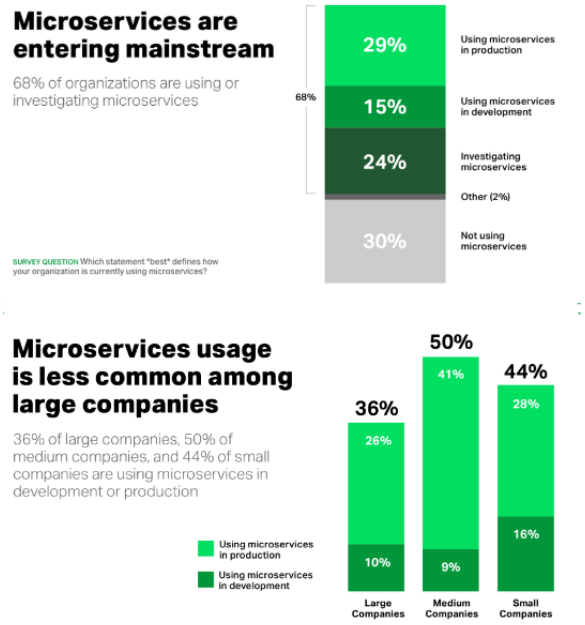


Figura 14 - Utilização de Micro serviços (NGINX 2015)

Em sentido convergente aponta um inquérito realizado pela LeanIX em 2017, no qual se constata que à data cerca de 80% das organizações inquiridas utilizavam micro serviços (um aumento de 12% em apenas dois anos). Nestas organizações, assistiu-se a um aumento bastante acentuado na frequência de lançamento de novas versões após a introdução desta arquitetura, com 17% das empresas a indicar a realização de várias *releases* semanais e 71% a ponderar mesmo reforçar a sua presença (Neoteric, 2017).

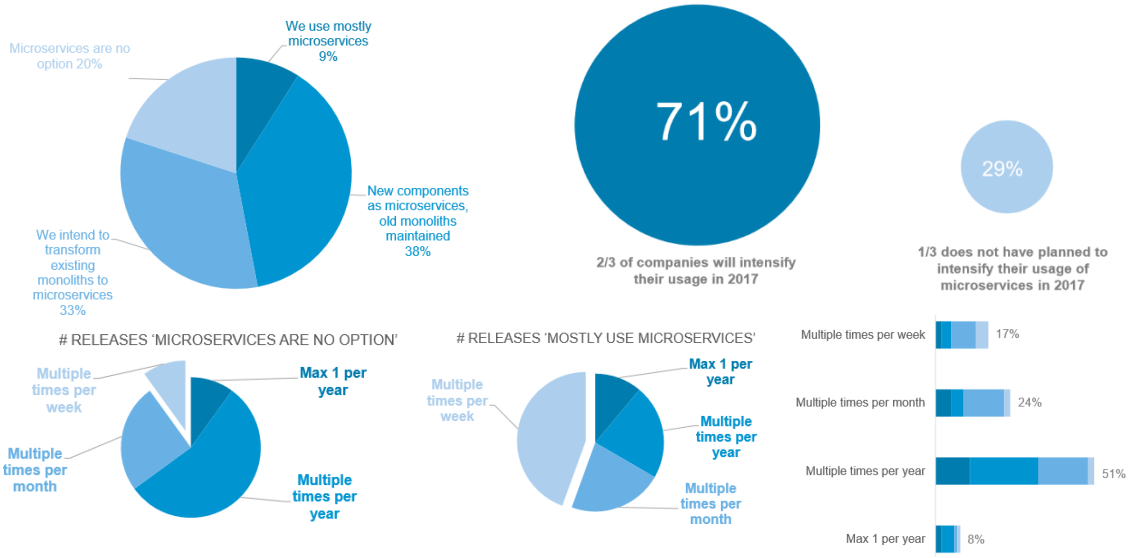


Figura 15 – Utilização de Micro Serviços, Periodicidade de *Releases* (com e sem microserviços) e Espetativa de Reforço da Sua Utilização – adaptado de (LeanIX GmbH, 2017)

Estes dados são altamente encorajadores visto que apesar da conversão do monolítico em micro serviços no BE, por si só já comportar benefícios, o exponente máximo do potencial da adoção desta arquitetura obtém-se quando se segue uma abordagem similar no FE, isto é, quando o FE monolítico dá lugar a micro *frontends* (consultar seção 2.2.3).

Desde que em novembro de 2016 foi introduzido pela ThoughtWorks (C. Jackson, 2019), o conceito de micro *frontends*, tem suscitado o interesse da comunidade, tendo servindo de mote a diversas apresentações em conferências de renome e a obras literárias, algo igualmente evidenciado no gráfico de tendências de pesquisa online obtido da ferramenta Google Trends, apresentado na Figura 16.

Figura 16 – Gráfico relativa às tendências de pesquisa no Google por "Micro *frontends*" de setembro de 2016 até à atualidade (Google, 2020)

Da análise do gráfico constata-se que desde novembro de 2016, tem-se registado um aumento no número de pesquisas por "Micro *frontends*", que apesar das flutuações se mantém alto, na ordem dos 60% (Google, 2020a).

3.2.5 Definição do Conceito

A reta final do modelo NCD "envolve o desenvolvimento de pelo menos um caso de negócio baseado em estimativas de potencial de mercado, necessidades de clientes, requisitos de investimento, avaliações de concorrentes, seleção da abordagem arquitetural e definição da *stack* tecnológica e risco geral de projeto" (P. A. Koen et al., 2002).

Consistindo a premissa deste documento no estudo da adoção de micro *frontends* no processo de desenvolvimento *web*, numa primeira fase vai-se proceder à recolha e tratamento de informação relativa a esta temática, proveniente de uma panóplia de fontes, desde apresentações em conferências a artigos publicados *online* por autores com conhecimentos sólidos do conceito passando por obras literárias , com particular destaque para o livro *"Micro frontends in Action"* da autoria de Michael Geers, que apesar de ainda se encontrar a ser redigido, já possui alguns capítulos disponíveis para consulta *online*.

Concluída esta fase, é tempo de selecionar as estratégias de integração de micro *frontends*, que serão alvo de experimentação e avaliação na prova de conceito a desenvolver.

Relativamente a esta prova de conceito, que será explanada posteriormente, consistirá no desenvolvimento de uma aplicação de *e-commerce* simplificada, subdividida em vários micro *frontends* correspondentes aos "contextos de domínio" apurados na fase de análise e *design*.

Com o intuito de ser requerida a comunicação entre diferentes módulos aplicacionais e a gestão de lógica comum, a aplicação deverá contemplar as seguintes funcionalidades: autenticação de utilizadores, navegação entre produtos, consulta de informação detalhada de um produto, adição de produtos ao carrinho, etc.

No que concerne a implementação da prova de conceito, estará subdivida em duas fases: primeiramente deverá ser efetuada o desenvolvimento dos vários micro *frontends*, para posteriormente serem criadas variantes da aplicação, nas quais deverão ser integrados e comunicarem entre si aplicando as estratégias de integração selecionadas.

Esta prova de conceito servirá essencialmente para aplicar o conhecimento mobilizado da fase de pesquisa e tratamento da informação e para perceber em que as situações o uso de micro *frontends* é a melhor opção e qual o padrão de integração mais indicado num determinado contexto.

3.2.6 Fatores de Influência

Estando o desenvolvimento de um produto, serviço ou processo inserido num determinado ambiente está sujeito a fatores de índole interna e/ou externa (políticos, sociais, científicos, etc.).

A adoção de uma abordagem orientada a micro *frontends* no processo de desenvolvimento por parte de uma organização é, desde logo, influenciada por fatores internos, entre os quais:

- a experiência na aplicação de metodologias ágeis como SCRUM, que asseguram um desenvolvimento iterativo e incremental e promovem um fluxo ininterrupto de *feedback*;

- conhecimentos de DDD para que a análise de domínio e desenho inicial (e refinamentos eventualmente necessários no decorrer da implementação) seja encetado de forma mais eficaz (devido ao know-how dos especialistas de domínio e dos especialistas de *design* e *software*) e eficiente (com uma maior brevidade);

- conhecimentos de ferramentas de automação como Jenkins, capazes de possibilitar a CI, algo fundamental no teste e integração num cenário de micro serviços e consequentemente micro *frontends*;

- a disponibilidade de recursos humanos necessária para viabilizar uma organização de equipas por funcionalidade e não por capacidades técnicas, que assegure que uma determinada equipa tem a seu cargo apenas e só a responsabilidade do desenvolvimento *end-to-end* de um determinado contexto de negócio.

Além disso, a envolvente externa também pode ter um papel a dizer uma vez que a experimentação e eventual inclusão de determinadas ferramentas e/ou bibliotecas que têm vindo a ser colocadas à disposição da comunidade, podem permitir que a integração dos diferentes micro *frontends* na prova de conceito seja mais facilmente executada.

3.3 Análise Funcional

A fase de análise visa analisar e sistematizar as funcionalidades do produto a desenvolver (Rich & Holweg, 2000). Para tal, será usado o método *Quality Function Deployment* (QFD).

3.3.1 *Quality Function Deployment*

O *Quality Function Deployment* é um método que permite analisar os requisitos de qualidade mais relevantes para o cliente (Ficalora & Cohen).

Visto que os clientes preferem os produtos que melhor atendem as suas necessidades, as equipas responsáveis pela análise de valor devem selecionar as funcionalidades que permitam dar resposta a esse desígnio (Hauser et al., 2010), socorrendo-se de ferramentas como a *House of Quality* (HOQ).

A *House of Quality* (Clausing & Hauser, 1988), converte os requisitos do cliente (*WHATs*) em características de engenharia (*HOWs*) (Barutçu, 2006). A sua construção envolve as seguintes etapas (Hauser et al. 2010):

1. Identificação das principais necessidades do cliente (WHATs), que serão listadas do lado esquerdo da matriz;

2. Determinação da importância/peso de cada necessidade junto do cliente, com vista a determinar aquelas que valoriza mais e menos, que deverá preceder a necessidade propriamente dita;

3. Elicitação das medidas de desempenho ou requisitos de *design* (HOWs), colocadas no topo da casa;

4. Identificação dos concorrentes e realização de uma análise comparativa entre eles (não aplicável tendo em conta o objetivo deste documento);

5. Determinação, no centro da casa, das relações entre *HOWs* e *WHATs*, isto é, de como cada medida de desempenho afeta o cumprimento de uma determinada necessidade;

6. Indicação das interações entre diferentes medidas de desempenho, no teto triangular da casa;

7. Determinação dos objetivos de *design*, que deverão ser colocados no fundo da casa.

No contexto do trabalho a desenvolver, a pertinência da construção da *HOQ* prende-se com determinar as principais necessidades dos clientes de lojas *online*, pois sendo esta a área de negócio da prova de conceito a desenvolver, permitirá identificar os requisitos mais importantes para estes.

Relativamente aos *WHATs* foram elencados os seguintes:

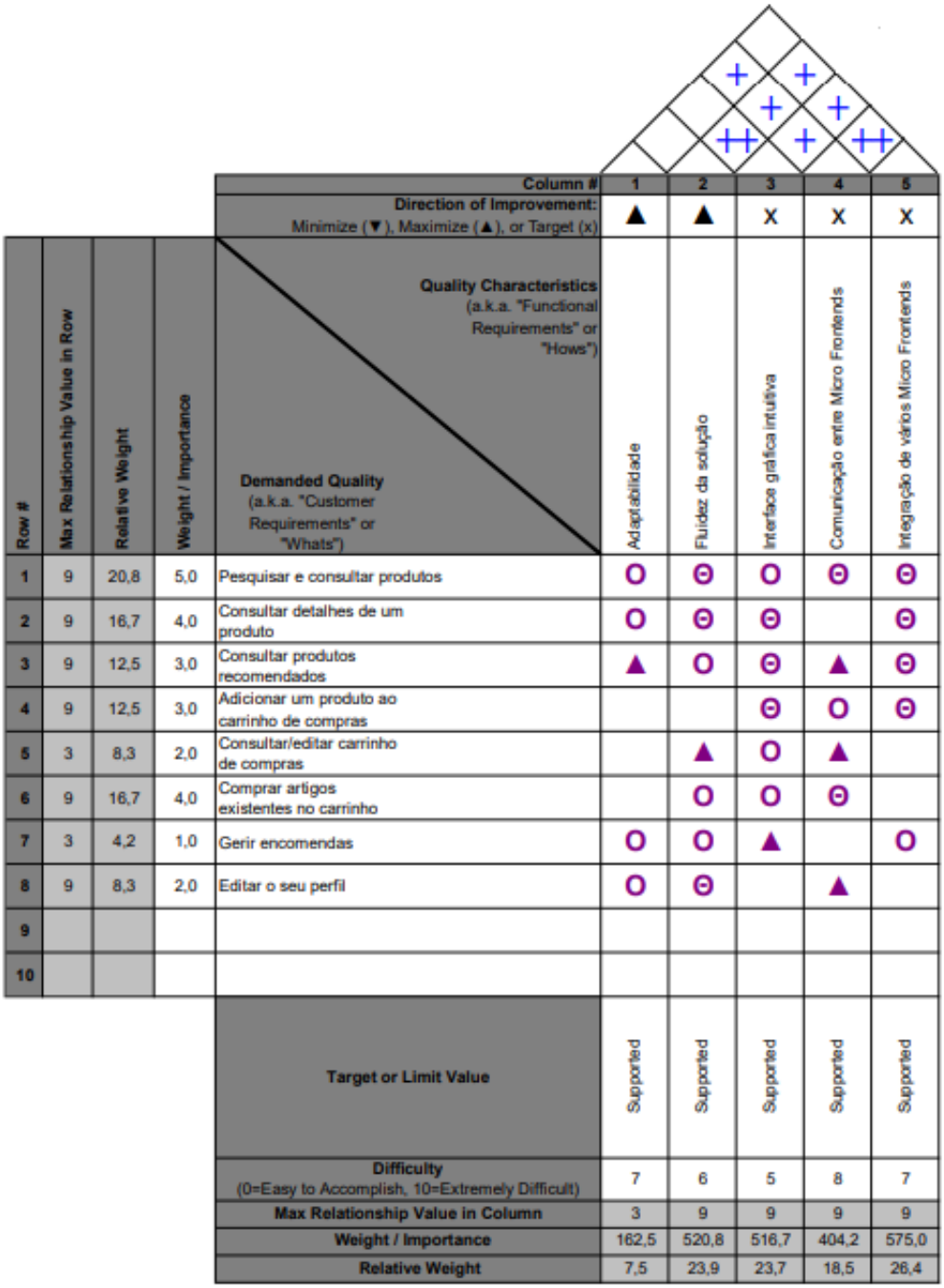
- Pesquisar e consultar produto;
- Consultar informação mais detalhada de um produto;
- Consultar produtos recomendados;
- Adicionar produto ao carrinho de compras;
- Consultar/editar artigos existentes no carrinho de compras;

- Comprar artigos existentes no carrinho de compras;
- Gerir encomendas;
- Editar o perfil do cliente (dados pessoais, morada de entrega e de faturação e métodos de pagamento).

No que concerne os *HOWs* foram definidos os seguintes:

- Adaptabilidade, isto é, a capacidade de se adicionarem ou modificarem funcionalidades do *software*;
- Fluidez da solução, que deve ser reforçada com vista a decrescer os tempos de carregamento da aplicação e de efetivação das alterações de estado da aplicação;
- UI intuitiva, ou seja, a implementação de uma interface gráfica coerente e simples de utilizar;
- Comunicação entre micro *frontends*, motivada pela existência de lógica aplicacional comum e pela possibilidade de dada a ocorrência de alterações de estado no âmbito de um micro *frontend* todos os micro *frontends* que estejam a observar estas alterações deverem ser notificados;
- Integração dos vários micro *frontends*, condição *sine qua non* para que os diversos módulos aplicacionais sejam compostos a fim de ser possível disponibilizar ao cliente uma aplicação que em todo se assemelhe a uma SPA convencional.

Com base nestas necessidades do cliente e caraterísticas de qualidade, foi construída uma casa de qualidade relativa à prova de conceito a desenvolver na segunda fase deste projeto, apresentada na Figura 17.



Row #	Max Relationship Value in Row	Relative Weight	Weight / Importance	Demanded Quality (a.k.a. "Customer Requirements" or "Whats") / Quality Characteristics (a.k.a. "Functional Requirements" or "Hows")	1 Adaptabilidade	2 Fluidez da solução	3 Interface gráfica intuitiva	4 Comunicação entre Micro Frontends	5 Integração de vários Micro Frontends
				Direction of Improvement: Minimize (▼), Maximize (▲), or Target (x)	▲	▲	x	x	x
1	9	20,8	5,0	Pesquisar e consultar produtos	O	Θ	O	Θ	Θ
2	9	16,7	4,0	Consultar detalhes de um produto	O	Θ	Θ		Θ
3	9	12,5	3,0	Consultar produtos recomendados	▲	O	Θ	▲	Θ
4	9	12,5	3,0	Adicionar um produto ao carrinho de compras			Θ	O	Θ
5	3	8,3	2,0	Consultar/editar carrinho de compras		▲	O	▲	
6	9	16,7	4,0	Comprar artigos existentes no carrinho		O	O	Θ	
7	3	4,2	1,0	Gerir encomendas	O	O	▲		O
8	9	8,3	2,0	Editar o seu perfil	O	Θ		▲	
9									
10									
				Target or Limit Value	Supported	Supported	Supported	Supported	Supported
				Difficulty (0=Easy to Accomplish, 10=Extremely Difficult)	7	6	5	8	7
				Max Relationship Value in Column	3	9	9	9	9
				Weight / Importance	162,5	520,8	516,7	404,2	575,0
				Relative Weight	7,5	23,9	23,7	18,5	26,4

Figura 17 - Casa de qualidade da prova de conceito

Da análise da figura depreende-se que a característica de qualidade mais direcionada aos requisitos do cliente é a integração dos vários micro *frontends*, visto que é a que tem maior peso relativo. Tal deve-se ao facto de apesar de o processo de desenvolvimento *web* em análise ser baseado em micro *frontends*, a aplicação colocada ao dispor do cliente resultar da composição dos vários módulos aplicacionais, sendo para tal crucial a sua integração.

4 Micro *Frontends*

Este capítulo tem como objetivo:

- Explicitar as motivações que levaram ao surgimento desta abordagem;
- Dissecar os princípios norteadores de micro *frontends*;
- Explanar as ferramentas oferecidas por esta abordagem para mitigar desafios comuns no processo de desenvolvimento *web*;
- Abordar o impacto da adoção de micro *frontends* na estrutura organizacional e na explicitação de responsabilidades das equipas;
- Descrever como efetuar o desenvolvimento local num contexto de micro *frontends*;
- Apresentar o tipo de testes automáticos a aplicar num contexto de micro *frontends*;
- Analisar as principais estratégias de migração de monolítico para micro *frontends*.

4.1 Motivações

A arquitetura de micro *frontends* surgiu da necessidade de colmatar as limitações inerentes a uma abordagem monolítica e de potenciar os benefícios alcançados com a adoção de micro serviços na camada de *backend*.

Regra geral, no arranque de um projeto de software, todos os *developers* detém uma visão holística de tudo o que pretende do ponto de vista funcional. Desta forma, as funcionalidades são desenvolvidas rapidamente, implantadas sobre a forma de um único artefacto e eventuais dúvidas ou discussões são endereçadas diretamente com quem de direito.

No entanto, o alargamento do âmbito do projeto e a entrada de novos elementos na equipa, impossibilitam que cada *developer* conheça todas as especificidades do projeto, podendo formar-se silos de conhecimento, e aumentam substancialmente a complexidade do projeto. Daí, que eventuais alterações ou adições num dado ponto da *codebase* possam despoletar efeitos inesperados noutras partes.

Por outro lado, o facto de existirem várias equipas a trabalhar numa aplicação monolítica faz com que o alinhamento seja alcançado por via de reuniões formais, o que implica uma sobrecarga de comunicação e coordenação (Geers, 2020a).

Ao nível da *codebase* assiste-se a um aumento desenfreado da extensão, complexidade, acoplamento e degradação da qualidade do código produzido.

Além disso, verifica-se uma tendência para a pré-otimização e o excesso de engenharia, o que dificulta a alteração ou adição de funcionalidades e a manutenção, mina a escalabilidade e causa uma derrapagem nos ciclos de desenvolvimento e lançamento de novas funcionalidades (Geers, 2020a; Mezzalira, 2020).

Relativamente à persistência, numa abordagem monolítica a cada *release* é necessário efetuar o *deployment* de toda a base de dados, o que naturalmente aumenta a probabilidade de ocorrência de erros e diminui a testabilidade.

Quanto à dinâmica da equipa constata-se que em certos momentos, a adição de novos elementos não se repercute num aumento da produtividade.

Esta particularidade foi analisada por Frederick Brooks na obra "*The Mythical Man-Month*" (Brooks, 1995).

Citando Brooks, "*adding more people lenghters, not shorters the schedule, because software construction involves a great deal of communication*", isto é, denota-se particularmente em projetos de *software* que a adição de novos elementos à equipa não agiliza o processo de desenvolvimento.

É que se por um lado é necessário alocar tempo para que os novos elementos se consigam inteirar das tecnologias, práticas e processos, por outro verifica-se uma maior necessidade de comunicação, que se reflete numa quebra da produtividade e derrapagem dos tempos de execução (Brooks, 1995).

Na sua perspetiva, estes atrasos podem ser motivados por previsões demasiado otimistas dos programadores, sobretudo por parte dos mais inexperientes, e eventuais falhas/inconsistências só serem detetadas durante a implementação, daí que a experimentação seja essencial.

Numa tentativa de mitigar estas limitações, é frequente dividir os projetos em várias parcelas. Os projetos de *software* não são exceção e, por isso, tornou-se prática comum segmentar o *software* e estruturar as equipas por capacidade técnica, isto é, promover uma divisão por camadas horizontais, na qual o *frontend* é entregue a uma equipa e o *backend* a uma ou mais.

Do lado do *backend* têm-se verificado que a arquitetura de micro serviços tem vindo a ganhar terreno. Ainda assim, apesar dos ganhos inerentes à sua adoção, não permite ainda efetuar o *deploy* de uma funcionalidade de forma independente.

Ora, o conceito de micro *frontends* veio introduzir uma revolução do ponto de vista arquitetural e organizacional do *software* e das equipas assente na divisão de um *software* em camadas verticais construídas E2E por uma equipa multidisciplinar autónoma e especializada.

Tecnicamente falando, possuindo a abordagem de micro *frontends* um cunho indelével da arquitetura de micro serviços, assegura-se a flexibilidade, autocontenção, delimitação e encapsulamento entre MFEs. Todavia, introduz-se alguma complexidade decorrente da implementação de uma infraestrutura automatizada, que contempla mecanismos de monitorização, *tracing* e *logging,* que asseguram a integridade e *performance* da aplicação, e que permite a CI e a CD.

Lamentavelmente, constata-se que tipicamente a arquitetura das soluções de *software* não tem em consideração o ambiente e o contexto organizacional no qual o projeto será desenvolvido (Geers, 2020a).

Neste contexto, a Lei de Conway, advoga que as estruturas de comunicação de uma organização são refletidas nos sistemas que criam, isto é, que um sistema deve ser desenhado em conformidade com a estrutura organizacional.

Assim, uma estrutura desadequada pode trazer pesadas implicações aos projetos de *software* (Torre, 2017).

Contudo, eventuais constrangimentos podem ser mitigados aplicando a Manobra Inversa de Conway, que defende que equipas e organizações devem ser estruturadas de acordo com a arquitetura desejada e não serem uma consequência da mesma (ThoughtWorks, 2015).

4.2 Princípios Norteadores

Sendo a abordagem de micro *frontends* baseada nos pressupostos dos micro serviços, os princípios que norteiam esta abordagem são bastante similares, ainda que contenham algumas idiossincrasias ligadas à camada de *frontend* (Mezzalira, 2020).

Assim sendo, de entre os princípios que regem esta abordagem, destacam-se a:

1. Modelação em torno de domínios de negócio;
2. Cultura de automação;
3. Abstração de detalhes de implementação;
4. Descentralização e autonomia versus centralização;
5. *Deployments* independentes;
6. Isolamento de falhas;
7. Elevada observabilidade.

Seguindo estes princípios criam-se as condições para obter uma solução em que cada equipa fica responsável por um domínio de negócio, o que permite a entrega de diferentes partes de projetos a equipas distintas e a redução dos ciclos de desenvolvimento e *release*.

Simultaneamente, introduzindo os MFEs alguma complexidade, em especial em termos de infraestrutura, é necessário determinar quando e em que projetos aplicar esta abordagem.

4.2.1 Modelação em torno de domínios de negócio

No arranque dos projetos deve-se investir tempo no processo de modelação do domínio de negócio segundo o DDD, uma vez que cada módulo aplicacional deve refletir o que faz a empresa e como se organiza e o *design* arquitetural deve basear-se em domínios e subdomínios, nos quais são promovidas linguagens ubíquas para facilitar a comunicação.

Esta modelação deve ser levada a cabo por indivíduos com um considerável conhecimento do negócio, para possibilitar um mapeamento mais fidedigno do modelo de domínio no código e uma mais clara delimitação e definição de responsabilidades dos vários micro *frontends*.

4.2.2 Cultura de automação

À semelhança do que acontece com os micro serviços, num contexto de micro *frontends* é fundamental garantir a robustez das *pipelines* de CI e CD.

Detendo as equipas competências e responsabilidades E2E, devem investir tempo no seu *design* e implementação, dado que é condição *sine qua non* para viabilizar o *deployment* independente de micro *frontends* autónomos (Mezzalira, 2020) e quiçá para diferentes ambientes.

4.2.3 Abstração de detalhes de implementação

Abstrair os detalhes de implementação e estabelecer e cumprir contratos firmados *à priori* entre equipas é essencial para que os diferentes micro *frontends* consigam comunicar entre si.

Nesta matéria, o DDD defende o encapsulamento dos componentes dentro de um subdomínio, expondo apenas as interfaces estritamente necessárias para a comunicação e integração (Vernon, 2016).

Assim, cada equipa deverá capaz de proceder a modificações nos seus componentes sem que isso impacte outras equipas, exceto se o contrato for colocado em causa.

Desta forma, reduzem-se as necessidades de comunicação e de coordenação e cada equipa pode desenvolver ao seu próprio ritmo e sem dependências externas.

4.2.4 Descentralização e Autonomia versus Centralização

A centralização e o *"one size fits all"* deram lugar à descentralização e delegação das decisões nas equipas, que sendo especializadas estão em melhor posição para o fazer (Geers, 2020a).

A descentralização traduz-se num reforço das competências das equipas que, passando a ser responsáveis pela tomada de decisão, podem reagir mais prontamente a eventuais impedimentos, bloqueios ou outro tipo de problemas.

Apesar de serem autónomas, as decisões das equipas não devem atentar contra as diretrizes e orientações da organização.

A par disso, deve-se fomentar uma cultura partilhada, quer de valores, quer de conhecimento entre equipas.

4.2.5 *Deploy* de forma independente

Por oposição ao *deployment* de um único artefacto, caraterístico da abordagem monolítica, o uso de micro *frontends* permite às equipas efetuarem o *deploy* de artefactos independentes, com o ritmo desejado e sem depender de terceiros para seguir para produção.

Ocorrendo com maior frequência, por norma, os *deploys* abarcam menos alterações, o que minora o risco de erros em produção, sobretudo se se recorrer a técnicas de gestão de *release* como o *Blue-Green Deployment* ou *Canary Releases* (Mezzalira, 2020).

4.2.5.1 *Blue-Green Deployment*

Esta técnica assenta na ideia de existirem duas versões idênticas do ambiente de produção designadas azul e verde, em execução simultânea (Fowler, 2010).

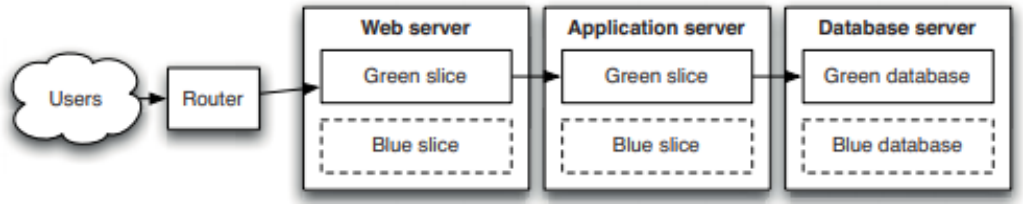


Figura 18 - *Blue-Green Deployment* (Humble & Farley, 2010)

No exemplo apresentado na Figura 18, os *users* do sistema são encaminhados para o ambiente verde, que atua atualmente como produção. Nesse caso, quando se efetuar uma *release* o *deployment* é efetuado para o ambiente azul, sem que isso impacte a operação do ambiente verde (Humble & Farley, 2010).

Durante a execução da *pipeline*, a versão no ambiente azul é submetida a testes, entre os quais *smoke tests*. A passagem dos testes indica que reúne condições para seguir para produção, bastando alterar a configuração do router para apontar para o ambiente azul, que nesse momento se torna o ambiente de produção.

De qualquer forma, se ocorrer alguma anomalia no ambiente azul, é possível a qualquer momento apontar produção de volta para o ambiente verde, o que reduz o risco de indisponibilidade decorrente de problemas em produção.

Apesar dos benefícios inerentes à sua aplicação, a técnica *Blue-Green Deployment* exige um cuidado acrescido com a gestão de versões de bases de dados, uma vez que perante alterações ao seu *schema*, a troca de ambientes na configuração pode ser demorada e complexa (Humble & Farley, 2010).

4.2.5.2 *Canary Releasing*

Por norma, existe apenas uma versão do *software* em produção num dado momento. Apesar de facilitar a gestão da infraestrutura, reduz a testabilidade, em particular dos testes de usabilidade, o que aumenta a probabilidade de resvalarem erros para produção.

Mesmo com ciclos de desenvolvimento já curtos, as equipas podem beneficiar de *feedback* mais rápido. Além disso, em ambientes de produção extremamente extensos, é praticamente impossível criar um ambiente com capacidade significativa de testes (Humble & Farley, 2010).

Perante estes desafios, a técnica de *Canary Releasing*, oferece a possibilidade de lançar uma nova versão da aplicação apenas para um subconjunto dos servidores de produção, para os quais são encaminhados grupos de utilizadores (ver figura abaixo), a fim de obter *feedback* e

identificar eventuais problemas com a nova versão mais rapidamente e sem afetar a maioria dos utilizadores. Daí, que seja uma forma de reduzir o risco de lançamento de uma nova versão.

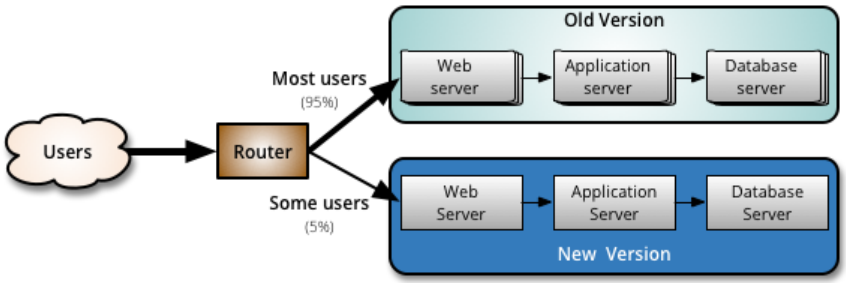


Figura 19 - *Canary Releasing* (Sato, 2014)

À semelhança do que acontecia com o *blue-green*, inicialmente o *deployment* deve ser efetuado para um conjunto de servidores que ainda não recebam encaminhamento de utilizadores.

Estes servidores são posteriormente submetidos a *smoke tests* e eventualmente a testes de capacidade, para que finalmente possam começar a encaminhar os utilizadores selecionados para a nova versão (Humble & Farley, 2010).

Podem existir múltiplas versões da aplicação em simultâneo em ambiente de produção, para as quais diferentes grupos de utilizadores são encaminhados. Contudo, é desaconselhável existirem mais do que duas, dado o elevado custo de suportar múltiplas versões comporta.

Esta técnica oferece vantagens como facilitar o *rollback* de versões e ser um meio preferencial para a aplicação de testes A/B e de testes de carga.

Os testes de carga permitem aferir se a aplicação cumpre os requisitos de capacidade, ao irem aumentando gradualmente a carga e o número de utilizadores encaminhados e analisando tempos de resposta e outras métricas de *performance* complexa (Humble & Farley, 2010).

Apesar dos benefícios da sua adoção, esta técnica não é apropriada a todo o tipo de aplicações, sendo difícil de aplicar em *software desktop*, e impõe restrições nos *upgrades* a bases de dados e outros recursos partilhados que necessitam de funcionar com todas as versões da aplicação em produção (algo contornável enveredando pela arquitetura *shared-nothing*).

4.2.6 Isolamento de falhas

A indisponibilidade de um serviço não deve comprometer a integridade e disponibilidade do resto do sistema, podendo o carácter autocontido dos MFEs dar um contributo nesse aspeto.

Em SPAs o isolamento de falhas não é problemático dada a sua arquitetura. Contudo, em situações em que a integração de micro *frontends* ocorra em tempo real é imprescindível apostar em mecanismos de tolerância à falha (Mezzalira, 2020).

Um dos pontos a favor da arquitetura monolítica reside na maior facilidade em observar um único sistema unificado do que um fragmentado em várias subaplicações.

Numa arquitetura distribuída, orientada a micro serviços e/ou micro *frontends*, na qual podem ocorrer falhas em qualquer ponto, a observabilidade é crucial para detetar e resolver rapidamente os problemas que possam surgir (Mezzalira, 2020).

Ora, a observabilidade é alcançada com a existência de mecanismos de *logging*, *tracing* e monitorização, cuja implementação representa alguma complexidade.

4.3 Desafios Comuns no Desenvolvimento *Web*

Aquando do desenvolvimento de aplicações *web*, os *developers* são, regra geral, confrontados com desafios relacionados com a *performance*, a consistência visual das aplicações e em alguns casos com colisões de estilos e *scripts*, que serão objeto de estudo ao longo desta secção.

4.3.1 Carregamento de *Assets*

As aplicações requerem, por norma, a importação de recursos estáticos como imagens, ficheiros de estilo e *scripts*.

Podendo estes recursos atingir um tamanho considerável, a sua transferência e carregamento irão inevitavelmente impactar os tempos de carregamentos das aplicações.

Aliás, esta é uma situação que tende a agudizar-se quando existem dependências duplicadas.

4.3.1.1 Estratégias de Carregamento de *Assets*

A integração de *assets* numa página pode ser efetuada usando as *tags* de *link* e *script*. Quanto ao carregamento de módulos pode ser efetuado usando RequireJS, CommonJS ou ES Modules, sendo que os últimos são um standard suportado pelos principais *browsers*, que não requerem a importação de bibliotecas adicionais (Geers, 2020a).

Assumindo que cada equipa que desenvolva fragmentos gera um ficheiro JS e um ficheiro CSS, cabe à equipa detentora da página que vai incluir os fragmentos referenciar estes *assets*.

A referenciação pode ser efetuada diretamente, por via de *redirect, include, inlining* ou baseada em *frameworks* de integração como o Tailor ou o Podium.

Relativamente à referência direta, pode ser feita através de um *import* no topo do ficheiro de código fonte, salvo se o *routing* for gerido através da *Application Shell*. Neste caso, como existe um único documento HTML, tipicamente a responsabilidade de carregar os *assets* para todos os micro *frontends* recai sobre a *App Shell*, que devem ser incluídos e carregados *à priori*. Todavia, é possível carregar os *assets* apenas quando o utilizador deles necessite (*Lazy Loading*).

Para melhorar a *performance*, importação dos ficheiros de estilos deve realizar-se no <head> do documento HTML e os *scripts* no <body>, para que a renderização seja mais rápida e os *assets* estáticos com *header* de cache com validade de um ano sejam enviados em separado, para evitar que o *browser* descarregue o mesmo ficheiro repetidamente (Geers, 2020a).

Apesar do cache *busting* ser uma estratégia válida de invalidação de cache, é desadequada para um contexto de configuração de micro *frontends* distribuídos, dado que cria demasiado acoplamento entre as equipas (que se veem obrigadas a atualizar os caminhos para os *assets* sempre que for gerada uma nova *hash*) e mina a capacidade de *deploy* independente.

Quanto à referenciação via redireccionamento, faz com o URL que permite referenciar os *assets* na aplicação integradora se mantenha inalterado, cabendo à equipa responsável pelo fragmento e respetivos *assets* efetuar o redireccionamento para a versão mais recente.

Esta alternativa garante o desacoplamento entre equipas, contudo não permite efetuar cache ao recurso inicial que retorna o redireccionamento, que requer a realização de um pedido adicional para validar se aponta para o mesmo *asset*, nem garantir a sincronização entre *assets* e as diferentes *builds*.

No que concerne a referenciação via *include*, ocorre no servidor e a principal vantagem face à referenciação por redireccionamento prende-se com o decréscimo de latência alcançado graças à comunicação entre servidores ser mais rápida. Usando ESI ou SSI é possível substituir a referenciação via <script> ou <link> por diretivas.

As equipas necessitam de fornecer os *endpoints* onde registam os *scripts* e os estilos, que são parte do contrato entre equipas. Como a *markup* que chega ao *browser* já contém o *include* resolvido, o *browser* pode proceder de imediato ao download do *asset*.

Esta abordagem fornece um baixo acoplamento, já que as equipas podem alterar os URLs dos *assets* sem notificar as outras equipas, que a torna a solução preferencial em termos de *performance*. Todavia, persistem os problemas de sincronização.

Relativamente à referenciação *inline*, como o próprio nome indicia, é efetuado na *markup* do próprio fragmento. Embora permita assegurar a sincronização de versões, pode incluir de forma redundante estilos e *scripts* e funciona apenas com integração *server-side*.

Já a referenciação baseada em bibliotecas de micro *frontends*, como o Podium e o Tailor, não permite evitar problemas de sincronização. Além disso, o contrato estabelecido entre equipas não cria acoplamento (Geers, 2020a).

4.3.1.2 Granularidade do *Bundle*

Não existe uma regra formal que defina a granularidade dos *assets*, podendo existir um único *bundle*, um por micro *frontend*, equipa ou até mesmo um por página ou fragmento.

Optar por um *bundle* que inclua os *assets* de todas as equipas introduz um elevado nível de acoplamento, dado que alguém necessita desenvolver e manter o serviço e os *deployments* têm de ser sincronizados entre o serviço de *assets* e as aplicações para garantir que a *markup* corresponde aos recursos entregues.

Num *bundle* centralizado pode seguir bastante código que não é utilizado e há uma forte probabilidade de invalidação de toda a cache, que se repercute na necessidade de descarregar novamente todo o *bundle*. Ainda assim, é de salientar o facto de permitir eliminar código redundante (Geers, 2020a).

Possuindo um *bundle* por equipa, é possível reduzir o tamanho do *bundle* e o *over-fetching*, apesar de poder impactar negativamente a reutilização entre páginas. No entanto, se uma equipa fornecer um fragmento que requeira bastante CSS que é usado por uma única página, deve ser criado um *bundle* isolado.

Já a criação de um *bundle* de JS e CSS por página ou fragmento, assegura que apenas se descarregam os *assets* necessários, mas, dependendo da quantidade de fragmentos incluídos numa página, o número de *assets* a descarregar pode permanecer elevado.

4.3.1.3 Carregamento *On-Demand*

As equipas podem adotar técnicas como o *code-splitting* no seu *bundle* para reduzir o tamanho do *bundle* descarregado inicialmente e o tempo de carregamento das aplicações, uma vez que algumas partes do código e estilo só serão carregadas quando necessário (*Lazy Loading*).

4.3.2 Performance

Tipicamente, verifica-se alguma desconfiança e resistência dos *developers* aquando da adesão pela primeira vez à arquitetura de micro *frontends*, por recearem que comprometa a *performance*, que é um dos requisitos a ter em conta desde o momento em que se inicia um projeto de *software* (Mezzalira, 2020).

Tipicamente, os projetos de micro *frontends* superam em termos de *performance* os monolíticos que vieram substituir, respondendo mais rapidamente, enviando menos código para o *browser* e alcançando melhores tempos de carregamento (Geers, 2020a).

4.3.2.1 Métricas de *Performance*

O significado de boa *performance* é subjetivo, oscilando em função dos casos de uso, dos requisitos de *performance* identificados e das métricas definidas por cada equipa.

As métricas de *performance* devem manter-se dentro dos limites definidos (*performance budget*), limites esses que sendo excedidos devem implicar a interrupção do desenvolvimento até que determine a causa e se decida a ação a tomar.

O *Performance Budget* de uma métrica pode ser dividido por todos os micro *frontends*, estabelecendo-se um valor limite para cada MFE e eventualmente para a página que os contém, ou então ser atribuído à página no seu todo, sendo neste contexto o detentor da página responsável por garantir que o limite não é superado e quando tal acontece tomar medidas (Geers, 2020a).

Efetuar o *debug* de um sistema distribuído é uma tarefa complexa, pelo que a monitorização é fundamental para identificar em tempo real desvios no *budget* e determinar a sua origem. Contudo a própria monitorização de código executado no *browser* pode ser complicada, dado que as aplicações de todas as equipas partilham recursos.

Possuir uma visão centralizada dos *deployments* das várias equipas pode ajudar a determinar a alteração no sistema que provocou um desvio.

O isolamento do problema é uma técnica popular de *debugging*, já que comentando, desativando ou bloqueando partes da aplicação é mais fácil determinar a sua origem.

O conceito de micro *frontends*, pela sua natureza, providencia benefícios como o *Code Splitting*, a otimização da resposta aos casos de uso por parte das equipas, que são especializadas no subdomínio que engloba estes UCs e a maior facilidade em encetar alterações.

4.3.2.2 Gestão de Bibliotecas Usadas em Múltiplos Micro *Frontends*

A gestão de bibliotecas iguais (em tipo e versão) usadas por diferentes equipas deve evitar o download do respetivo código duas ou mais vezes, pois como diria Pavlov, "é ineficiente e devemos evitá-lo".

Como vem sendo reiterado ao longo do documento, o facto de se poder usar diferentes *frameworks* em equipas distintas, não é sinónimo de que deva acontecer.

Partilhando a mesma *framework*, as equipas podem-se entreajudar e os *developers* podem trocar de equipa mais facilmente. Contudo, deve ser possível que uma nova equipa possa escolher uma nova tecnologia se se justificar. Daí que a integração deva ocorrer de forma agnóstica da tecnologia.

O *bundler* JavaScript de cada equipa cria um ficheiro de *assets* otimizado, que é autocontido e que inclui todas as dependências de que necessita nos seus componentes e páginas.

Ora, tem-se assistido a uma maior preocupação em adotar bibliotecas ou *frameworks* pequenas em detrimento de opções como o Angular, que apesar de serem bastante completas, aumentam substancialmente o tamanho do *bundle*.

Desse modo, é possível reduzir a sobrecarga inerente a carregá-la várias vezes, sobretudo quando a percentagem de código do *vendor* no *bundle* é elevada e em páginas compostas por um número considerável de fragmentos provenientes de diferentes equipas (Geers, 2020a).

Num cenário em que várias equipas usem a mesma versão de uma *framework*, uma otimização pode passar por importá-la no *bundle* geral da aplicação e excluí-la dos *bundles* individuais para que não seja descarregada mais do que uma vez. (Geers, 2020a).

Utilizando o Webpack como *bundler*, para que esta biblioteca seja marcada como disponível globalmente deve ser incluída nos "*externals*".

Estes artefactos centralizados podem ser mantidos por uma equipa de plataformas (caso exista) ou por uma equipa de produto, que se voluntarie. Esta equipa é responsável por informar as restantes sempre que surja uma nova versão para que testem o *software* contra ela.

Contudo, perante alterações substanciais, que exijam a reestruturação de parte do *software*, só quando todas as equipas tiverem procedido às mesmas é que se pode proceder à atualização da versão do *bundle*, para não perturbar o normal funcionamento do *site*.

Esta situação, em que o *deployment* de um MFE requer o *deployment* de outros, *lock-step deployment* (Devs, 2015), viola o princípio da capacidade de evolução independente de um MFE.

Para que tal não aconteça, deve-se apostar numa abordagem baseada em versionamento que, dando suporte a mais do que uma versão da mesma *framework*, permita que cada equipa proceda à atualização ao seu próprio ritmo (Geers, 2020a).

Esta abordagem pode ser posta em prática usando o plugin D11 do Webpack, *ES modules* nativos ou *import maps* (Geers, 2020b).

Note-se que num contexto de micro *frontends* apenas se deve centralizar código de *vendor* genérico, já que partilhar lógica de negócio pode introduzir acoplamento e reduzir a autonomia.

4.3.3 Consistência de Interface Gráfica entre Micro *Frontends*

O facto de os micro *frontends* serem desenvolvidos de forma independente por equipas distintas pode traduzir-se num risco acrescido de inconsistência visual, algo que, no entanto, pode ser minimizado através dos contratos firmados entre as equipas e/ou da existência de um *design system* ou *styleguide*.

4.3.3.1 Conceito de *Design System*

Tipicamente as organizações criam um *design system*, ou pelo menos um *styleguide*, com o intuito de uniformizar visualmente as suas soluções de *software*.

Um *design system* contém *design tokens* (fontes, cores, ícones, etc.), componentes visuais reutilizáveis (botões, *inputs* e outros elementos de formulários, etc.) e um conjunto detalhado de regras que definem como devem ser utilizados (Geers, 2020b).

O *design system* deve ser visto como um produto que serve outros produtos.

Apesar de não criar valor por si só, oferece consistência e uma linguagem partilhada que melhora a comunicação entre equipas e diminui o risco de más interpretações e inconformidades.

Além disso, agiliza o processo de desenvolvimento e o seu valor aumenta com a escalabilidade (Geers, 2020b).

Sendo a sua criação e complexa e dispendiosa, antes de proceder à sua implementação as equipas devem verificar se existe alguma estratégia de *branding* da organização que deva ser seguida pelos produtos de *software*.

Em caso afirmativo, deve ser criado um *design system* personalizado, que possibilite a incorporação de componentes únicos do domínio de negócio, caso contrário a adoção de bibliotecas como o Bootstrap ou Material Design da Google é suficiente, devendo a escolha da biblioteca ser baseada não só na aparência, mas também na forma como é integrada (Geers, 2020b).

Um *design system* deve ser percecionado como uma parte de infraestrutura em contínua evolução, pois apesar de grande parte ser produzido numa fase inicial, deve regularmente ser alvo de melhorias, com a introdução de funcionalidades que permitam dar resposta a novos casos de uso e às solicitações das equipas.

Resultando os seus componentes e regras de *design* de discussões entre UX, *developers* e POs das equipas, o *design system* constitui a única fonte de verdade em matéria de *design*.

4.3.3.2 Desenvolvimento de um *Design System*

Deve haver uma estreita colaboração entre a equipa de UX e as equipas de produto ao longo do processo de desenvolvimento.

Para evitar que ocorram perdas de produtividade decorrentes de parte da equipa estar sem trabalho em *backlog* ou da implementação de soluções provisórias devido à inexistência de *wireframes* e posterior reimplementação, a equipa de *design* alocada a um projeto de *software* deve iniciar o seu trabalho com um avanço de algumas semanas (Geers, 2020a).

Todas as funcionalidades visíveis produzidas pela equipa de produto dependem do *design system*, pelo que caso seja preciso adicionar um ícone ou um componente apesar de haver a tentação por parte da equipa de produto de o fazer diretamente no seu micro *frontend*, isso não deve acontecer.

O *design system* pode tornar-se um ponto de estrangulamento quando as solicitações das equipas de produto excedem a capacidade de resposta da equipa central, podendo as equipas de produto ver-se obrigadas a adiar o lançamento de funcionalidades enquanto aguardam a ação da equipa responsável pelo *design system*.

Para colmatar esta limitação o *design system* pode ser desenvolvido de forma federada. Este modelo de desenvolvimento respeita os princípios de micro *frontends* e pressupõe que os *developers* e *designers* de cada equipa possam contribuir, existindo uma equipa responsável por rever os *merge* requests abertos pelas equipas de produto, no sentido de assegurar a qualidade do código produzido e a consistência.

Relativamente à integração dos *Design Design* pode ocorrer em *runtime* ou por via de *packages* versionados. No primeiro caso, o *deployment* é efetuado diretamente para produção e todas as equipas de produto devem usar a versão mais recente, o que atenta contra a autonomia. No último, cada equipa pode decidir quando adotar a versão mais recente (Geers, 2020a).

4.3.3.3 *Design System* Local

A partilha de componentes entre equipas comporta custos, uma vez que requerem um nível elevado de qualidade.

Assim sendo, a biblioteca partilhada de componentes pode incluir componentes mais ou menos complexos, desde botões e ícones até tabelas paginàveis e ordenáveis, desde que contenham apenas e só lógica de interfaces.

Quanto a componentes embebidos de lógica de domínio, não devem atravessar os limites do micro *frontend* onde estão definidos, sob pena de se criar demasiado acoplamento entre aplicações (C. Jackson, 2019).

Contudo, podem ser colocados numa biblioteca local de componentes, passível de ser usada por toda a equipa (Geers, 2020a).

4.3.4 Colisões de Estilos

O CSS, por si, não vem com sistemas de módulos, *namespacing* ou encapsulamento.

Num contexto de micro *frontends* se equipas diferentes definirem regras para um mesmo seletor podem ocorrer colisões (C. Jackson, 2019), evitáveis recorrendo a:

- convenções de nome rígidas como o *Block Element Modifier* (BEM) (Strukchinsky & Starkov, 2016);
- pré-processadores de CSS como o SASS, nos qual o estilo de um componente pode ser definido dentro do bloco de regras do respetivo seletor, que se torna assim uma espécie de *namespacing*;

- CSS Modules ou bibliotecas como CSS-In-JS, que garantem que o CSS é aplicado diretamente nos elementos pretendidos;

- Usando *Shadow* DOM, que oferece isolamento de estilos (Bidelman, 2019; Perrott, 2020).

4.4 Equipas e Responsabilidades

A adoção de micro *frontends* além de benefícios técnicos comporta benefícios de cariz organizacional, uma vez que a responsabilidade pelo desenvolvimento de diferentes funcionalidades pertence a equipas independentes e especializadas, que detêm autonomia para tomar decisões.

Aliás, são precisamente as mais-valias que advém desta orgânica de equipas, que levam muitas empresas a decidir adotar esta abordagem de desenvolvimento.

Tal como vincado no modelo organizacional da Critical Techworks, uma organização que promove a multidisciplinaridade das equipas, devem-se proporcionar os meios para assegurar a sua motivação, agilizar processos e potenciar a produtividade (P. V. da Silva et al., 2019).

4.4.1 Alinhamento de sistemas e equipas

Um dos principais benefícios da adoção da arquitetura de micro *frontends* é a explicitação das responsabilidades de cada equipa.

Assim, nesta secção serão analisados aspetos como os limites entre equipas, a sua composição a partilha de conhecimento, a responsabilidade pelas *cross-cutting concerns* e o nível de diversidade tecnológica que pode verificar-se entre de equipa para equipa.

4.4.1.1 Limites entre equipas

Seguindo à risca a lei de *Conway*, se o desenvolvimento de um produto for entregue a uma única equipa, há uma forte probabilidade de que a aplicação enverede por uma arquitetura monolítica, enquanto se o mesmo produto for entregue a um conjunto de equipas, de que a solução obtida seja mais modular (Geers, 2020a).

Sobre esta temática, Coplien e Harrison na obra "*Organizational Patterns of Agile Software Developmen*t", afirmam "que se as partes que compõem a organização não refletirem as partes essenciais do produto, (…) o projeto estará em problemas" (Coplien & Harrison, 2004), o que transposto para uma arquitetura de micro *frontends*, significa que os limites da equipas devem estar alinhados com os limites das camadas verticais que formam o produto.

Estes limites podem ser identificados aplicando DDD, *User-Centered Design* ou após a análise da estrutura de páginas do projeto atual.

Da aplicação do DDD, esmiuçada na seção 2.1.6 deste documento, resulta uma série de *bounded contexts*, podendo cada um tornar-se uma aplicação micro *frontend* desenvolvida por uma equipa dedicada.

Relativamente ao *User-Centered Design*, permite definir os limites entre MFEs (e implicitamente entre equipas) com base naquilo no utilizador mais valoriza e nas suas necessidades (Babich, 2019).

No que concerne à definição de limites com base na estrutura de páginas do projeto atual, consiste no levantamento de todo o tipo de páginas e o seu agrupamento com base na intuição de um grupo experiente de colaboradores (Geers, 2020a).

Tal como no *User-Centered Design* a ferramenta Google Analytics pode ser usada, neste caso para validar os grupos estabelecidos. Contudo esta opção não é a ideal visto que as páginas podem ter mais do que um objetivo (o que atenta contra o SRP), mas serve de mote para discussões mais aprofundadas.

4.4.1.2 Composição das equipas

Em função da sua composição, as equipas de micro *frontends* podem ser classificadas como "apenas de *frontend*", "*full-stack*" ou com "plena autonomia" (Geers, 2020a).

As "equipas apenas de *frontend*", são formadas unicamente por *frontend developers*. Assim, as micro *frontends* destas equipas são construídos sobre o *backend* desenvolvido por outras equipas, cuja arquitetura pode ser monolítica ou orientada a micro serviços, podendo neste último caso ser desenvolvido um BFF para estabelecer a comunicação entre cada MFE e os serviços de que necessita (ver secção 5.4.4).

Esta tipologia de equipa já permite escalar o desenvolvimento e facilitar a migração de uma aplicação. Contudo, requer um enorme esforço de comunicação e coordenação.

Quanto às "equipas *full-stack*", incluem *developers* que no seu conjunto reúnem competências E2E, o que se traduz num menor esforço de coordenação e possibilita o desenvolvimento e *deployment* independente de MVPs mais rapidamente.

Relativamente às "equipas com plena autonomia", além de *developers*, incluem *designers*, especialistas de domínio e *stakeholders*. Ora, colaboradores com estes perfis tipicamente trabalham em departamentos que nada têm que ver com o desenvolvimento de *software* e limitam-se a especificar requisitos.

Colocar lado-a-lado equipa de desenvolvimento e especialistas de negócio permite formular e validar ideias mais rapidamente e uma obter vantagens competitivas decorrentes de uma adaptabilidade mais célere ao mercado (Geers, 2020a).

Porém esta mudança de paradigma é complexa e se é mais ou menos consensual em *startups*, o mesmo não pode afirmar-se relativamente a organizações de grande dimensão, nas quais poderá enfrentar focos de resistência e a conversão da estrutura organizacional será morosa.

4.4.2 Partilha de Conhecimento entre Equipas

Num panorama em que uma aplicação está dividida em camadas verticais entregues a equipas distintas, apesar da otimização da comunicação estabelecida entre os intervenientes do desenvolvimento de micro *frontend* paira no ar o risco de se formarem silos de conhecimentos.

Consequentemente, a não ser que existam na organização canais de partilha de conhecimento entre equipas, sempre que uma equipa for confrontada com um desafio novo para si, mas já superado ou pelo menos enfrentado por outras, ver-se-á obrigada a "reinventar a roda", desperdiçando-se o *know-how* existente.

Assim, o conhecimento pode ser disseminado através da formação de *Communities of Practice* (CoP), isto é, de grupos de pessoas que possuem funções similares nas respetivas equipas, da aposta na formação e evolução contínua das equipas, da apresentação por parte de uma equipa do trabalho realizado às restantes equipas (Geers, 2020a) que contribuem para o desenvolvimento do produto e da documentação rigorosa do *software* produzido por parte das equipas responsáveis.

4.4.3 Gestão de *Cross-Cutting Concerns*

Autenticação, monitorização, *logging* e *tracing*, são apenas alguns exemplos de *cross-cutting concerns* que sendo transversais devem ser tratadas numa infraestrutura centralizada para que as equipas de produto se foquem em produzir valor.

Apesar de se poder recorrer a soluções SaaS, que ao serem fornecidas por um serviço externo não introduzem acoplamento entre as equipas, nem sempre é possível devido à falta de suporte de alguma destas responsabilidades ou ao custo associado ao serviço.

Nesse caso, a infraestrutura terá de ser idealizada e construída pela equipa de desenvolvimento, podendo esta tarefa ficar a cargo de uma ou mais equipas de produto ou até de uma equipa dedicada, ainda que esta última opção deva ser devidamente ponderada, sob pena de poder atentar contra a arquitetura de MFE e de constituir um ponto único de falha.

Relativamente à gestão de *cross-cutting concerns* que não dependam de serviços ou bibliotecas, como a internacionalização, pode ser efetuada através do estabelecimento de um contrato entre as equipas que estipule como deve ser efetuada a implementação (Geers, 2020a).

4.4.4 Nível de Diversidade Tecnológica

Como frisado inúmeras vezes ao longo deste documento, num contexto de micro *frontends* as equipas têm liberdade para definir a *stack* tecnológica.

Todavia, com vista a não sobrecarregar o *bundle*, facilitar a permuta entre elementos de equipas distintas e tirar maior proveito do *know-how* existente, pode recorrer-se a estratégias que permitam orientar o desenvolvimento a rumar em determinada direção, com vista a uniformizar tanto quanto possível as tecnologias utilizadas. Para tal, Geers sugere a criação de uma *toolbox* e de um projeto-base (Geers, 2020a).

Relativamente à *toolbox*, consiste numa espécie de guia que contém várias sugestões de *frameworks*, bibliotecas, ferramentas de teste, etc., passíveis de ser incluídas pelas equipas na respetiva *stack* tecnológica (Geers, 2020a).

Apesar de ser definida a nível organizacional, a *toolbox* pode a qualquer momento incluir novas tecnologias por sugestão das equipas.

Quanto ao projeto-base, como o próprio nome indica, é um projeto que inclui todos os aspetos técnicos que uma aplicação em micro *frontends* necessita, mais especificamente a estrutura de diretórios, configuração de ferramentas de teste, regras de *lint*, comunicação com APIs, configuração da ferramenta de compilação, etc. (Geers, 2020a).

Note-se que este projeto-base não deve ser encarado como algo de carácter obrigatório, mas como uma orientação, cabendo às equipas decidir se devem ou não o utilizar.

4.5 Experiência de Desenvolvimento Local

Se executar e desenvolver localmente sobre uma aplicação monolítica é algo de trivial, uma vez que existindo um único repositório, basta cloná-lo para que se tenha localmente tudo o que é necessário para executar a aplicação, o mesmo não se pode dizer acerca de uma aplicação constituída por vários módulos.

Neste tipo de aplicações, cada equipa detém o seu próprio repositório e pode utilizar tecnologias e configurações distintas.

Apesar de tecnicamente ser possível a um *developer* possuir cópias locais dos repositórios das outras equipas, atualizadas através de *pulls* periódicos, isso obrigá-lo-ia a conhecer o funcionamento interno dos outros micro *frontends* e poderia impedi-lo de executar o seu micro *frontend* devido a conflitos de versões ou *bugs* noutras partes da aplicação (Geers, 2020a).

O foco do *developer* deverá incidir sobre a *codebase* da sua equipa, sendo fundamental para tal que o modo de execução da aplicação possa variar de ambiente de desenvolvimento para produção.

Em ambiente de desenvolvimento, para se isolarem falhas ou erros e se desenvolver de forma mais fluída e robusta, pode-se adotar uma das seguintes estratégias:

- construir *mocks* dos fragmentos de outras equipas incluídos quer sejam renderizados no cliente ou no servidor (nesse caso, deve seguir-se uma abordagem idêntica à descrita na seção 5.1.5). A interface dos fragmentos deve ser documentada pelas equipas responsáveis, para que essa documentação possa servir de base à criação dos *mocks*;

- cada equipa possuir uma página que serve como *playground*, no qual os fragmentos que desenvolve são apresentados e podem ser testadas interações, como os mecanismos de comunicação, por via de *toggles*;

- caso o *mock* ou o isolamento não forem suficientes, pode-se apontar o caminho dos fragmentos de outras equipas incorporados na página para a sua versão atualmente em produção.

4.6 Aplicação de Testes Automáticos

Em matéria de testes não se verificam diferenças significativas entre *frontends* monolíticos e a arquitetura de micro *frontends*, dado que cada micro *frontend* deve estar coberto por testes unitários que permitam validar quer a lógica de renderização, quer a lógica interna de negócio (Geers, 2020a).

A maior diferença reside nos testes de integração/E2E dos micro *frontends* na aplicação-contentor.

Estes testes, sendo mais lentos e envolvendo mais recursos, devem limitar-se a validar que a página é montada corretamente, podendo ser efetuados usando ferramentas de testes como o Selenium ou Cypress (C. Jackson, 2019).

Relativamente ao nível de integração destes testes, geralmente é suficiente verificar se as versões *mock* dos fragmentos estão presentes na página, contudo há raras exceções em que têm de ser executados contra um ambiente idêntico a produção.

Em termos de responsabilidades, cada equipa pode validar a presença dos seus fragmentos nas páginas que integram.

A título exemplificativo, a equipa de carrinho pode testar que o seu botão de adicionar está presente na página de produto, mantida por outra equipa (Geers, 2020a).

4.7 Estratégias de Migração de Monolítico para Micro *Frontends*

A migração de um projeto aplicacional não trivial é, regra geral, um processo complexo, algo moroso e que comporta custos elevados, que pode não gerar valor no imediato.

Existindo um rol diverso de possibilidades de operacionalizar a migração de um *software* esmiuçados na literatura e em artigos publicados pela comunidade, este documento incidirá sobre as mais ajustadas às especificidades da arquitetura de micro *frontends* (Geers, 2020a).

Nesse contexto, emergem três principais abordagens de migração de um monolítico para micro *frontends* que serão objeto de análise: *Big Bang, Slice-by-Slice* e *Frontend-First*.

4.7.1.1 Big Bang

O Big-Bang consiste na migração de um sistema legado do zero, uma abordagem que representa um risco elevado já que funcionalidades não documentadas devidamente podem ser perdidas e a entrega de valor diminui substancialmente porque as novas funcionalidades podem necessitar de ser implementadas em ambos os sistemas enquanto a migração não for concluída e o sistema monolítico for descontinuado (Dehghani, 2018).

Relativamente a esta abordagem Martin Fowler é perentório ao afirmar que "se reescrevermos um monolítico usando o Big-Bang, a única garantia que temos é a de que enfrentaremos isso mesmo, um *Big- Bang*" (Newman, 2019).

4.7.1.2 *Slice-by-slice*

À medida que são adicionadas novas funcionalidades a uma aplicação monolítica, a sua complexidade pode aumentar substancialmente, fazendo com que seja mais difícil a sua manutenção e a introdução de novas funcionalidades (Mendes, 2018).

Assim, num contexto aplicacional complexo promover uma migração gradual para o novo sistema reduz o risco ao manter o sistema antigo para processar as funcionalidades que ainda não foram migradas enquanto o processo não for concluído.

Perante este cenário, à primeira vista seria expetável que os clientes soubessem a cada momento em que sistema estão localizadas as diferentes funcionalidades e que fossem atualizados sempre que alguma localização fosse alterada.

No entanto, este padrão permite contornar essa questão ao criar uma mecanismo de integração no *frontend* que atua como uma fachada responsável por intercetar os pedidos para o sistema legado e por encaminhar esses pedidos para o sistema correspondente com base na rota (sendo encaminho para o sistema novo se a funcionalidade já tiver sido migrada, caso contrário segue para o antigo) (Buck et al., 2019) a fim de obter os fragmentos ou páginas e efetuar a sua composição, conforme ilustrado na Figura 20.

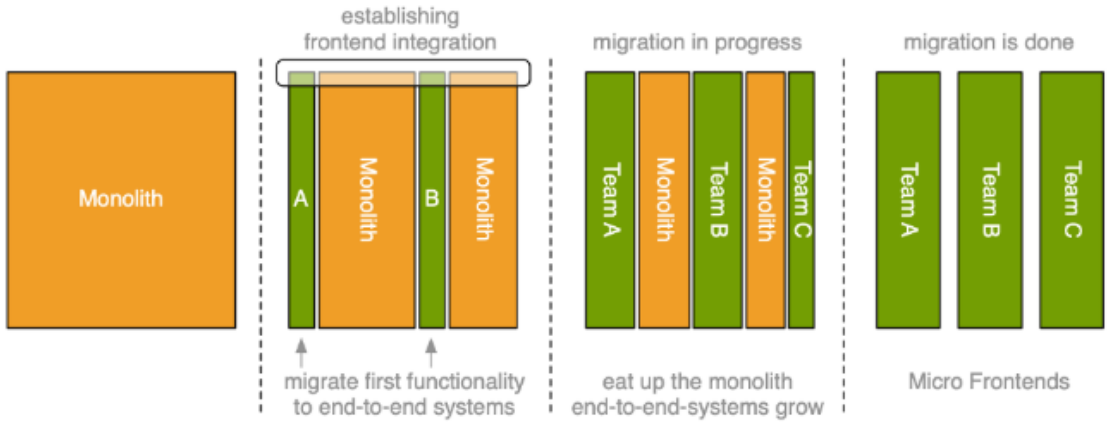


Figura 20 – Migração camada a camada de monolítico para *micro frontends* (Geers, 2020).

À medida que a migração de funcionalidades se desenrola, cada vez mais pedidos são encaminhados para o novo sistema até que o sistema legado acaba "estrangulado", isto é, é descontinuado e nesse momento a fachada deve ser removida (ver Figura 21).

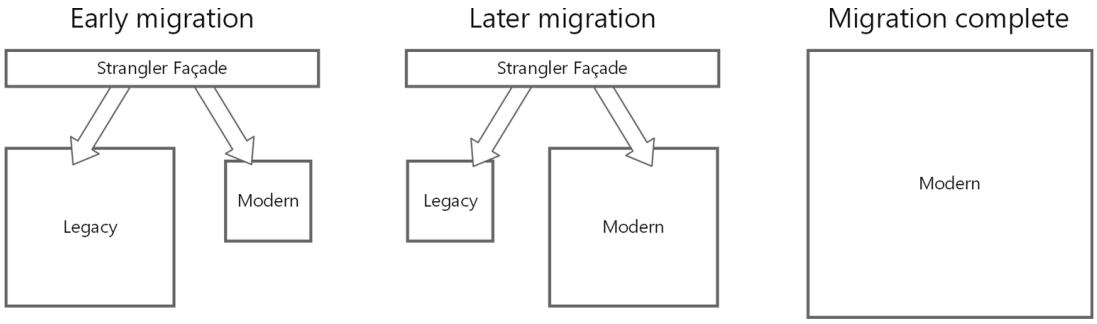


Figura 21 - Migração de um monolítico usando o padrão *Strangler* (Buck et al., 2019)

Apesar de requerer mais tempo, para estrangular o monolítico, esta abordagem possibilita manter a entrega contínua de valor (Diaz, 2019b) e não só monitorizar mais minuciosamente o progresso uma vez que são efetuadas *releases* com frequência, mas também identificar e descartar funcionalidades desnecessárias no sistema durante a migração.

Comparativamente a uma aplicação construída de raiz usando *micro frontends*, esta abordagem requer mais preparação e só deve ser levada a cabo quando se possua um conhecimento holístico do sistema existente. Além disso, requer um enorme esforço de coordenação, uma vez que ambos os sistemas vão coexistir no decurso da migração.

Apesar das funcionalidades migradas não necessitarem de ser removidas, é necessário pelo menos adaptar a UI do monolítico para que se assemelhe à dos novos *micro frontends*. Nessa matéria, podendo ocorrer colisões de estilos, o uso de *Web Components* e *Shadom DOM* pode ser uma mais-valia, apesar de existirem outras alternativas (Geers, 2020a).

Visto que o código produzido hoje é o *software* legado de amanhã, devem-se criar as bases para agilizar no futuro a migração desse *software* usando este padrão (se aplicável) (Fowler, 2004).

4.7.1.3 *Frontend First*

Apesar desta estratégia de migração se assemelhar à *slice-by-slice* por recorrer ao padrão Strangler, não mistura o código do novo *frontend* com o do antigo, o que por si só já agiliza o processo especialmente quando a migração é acompanhada de um *redesign* (Geers, 2020a).

O *Frontend First* compreende duas fases, sendo que se começa por quebrar o monolítico do lado do *frontend* e posteriormente o *backend*.

Antes de se dar início à migração propriamente dita, definem-se os limites e responsabilidades das equipas, que vão desenvolver uma parte específica do *frontend*.

Assim, na primeira fase são desenvolvidos os novos *frontends*, que vão abrir caminho para a divisão do monolítico em camadas verticais *end-to-end* e que serão posteriormente integrados aplicando uma das técnicas abordadas na secção 5. Relativamente aos dados a apresentar são obtidos do monolítico por via de APIs criadas para o efeito (ver Figura 22).

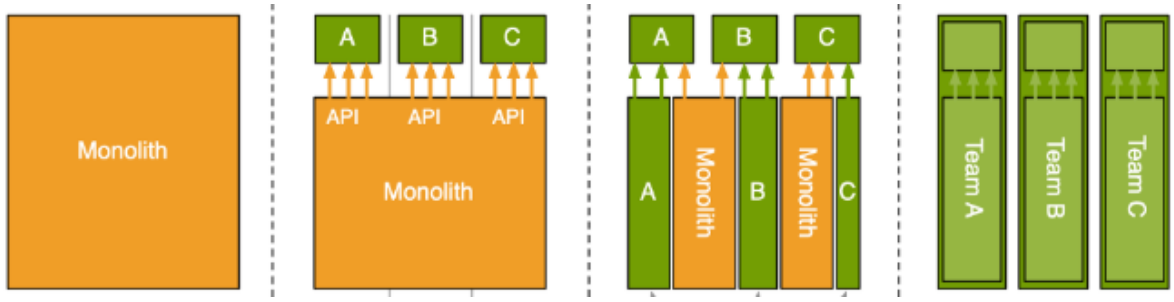


Figura 22 – Migração de monolítico usando *Frontend first* (Geers, 2020a)

Estando os novos micro *frontends* e a API operacionais na sua plenitude, passa-se à segunda fase, na qual o monolítico do *backend* será substituído por um conjunto de serviços independentes. Nesse sentido, as APIs implementadas na fase anterior servem de referência em termos de limites, cabendo a cada equipa criar a aplicação *backend* que irá substituir a API criada para o novo *frontend* comunicar com o monolítico. Para tal é aplicado o padrão de estrangulamento para substituir gradualmente estas APIs até que o monolítico se torne dispensável.

Posto isto, a abordagem *Frontend First* tem a vantagem de não misturar o código do monolítico com o dos novos micro *frontends*, que abarcando pouca lógica de negócio ou com pouco complexidade permite entregar resultados de forma rápida.

Todavia, em termos de volume de trabalho ou esforço requerido para a sua concretização podem verificar-se assimetrias entre *frontend* e *backend* (colmatáveis ao encorajar as equipas a trabalhar de forma multifuncional) e na perceção do progresso da migração, que é mais claro durante a primeira fase do que quando o foco incide sobre o *backend* (Geers, 2020a).

5 Integração e Comunicação de Micro *Frontends*

Este capítulo incide sobre aspetos mais técnicos da abordagem de Micro *Frontends* e apresenta alguns casos da sua aplicação em contexto real.

Assim sendo, tem como objetivo:

- Esmiuçar as principais técnicas e bibliotecas de integração de micro *frontends;*
- Analisar comparativamente as diferentes técnicas e bibliotecas de micro *frontends* numa tabela-síntese;
- Analisar mecanismos de comunicação passíveis de ser mobilizados na interação entre diferentes camadas dos micro *frontends*;
- Apresentar casos de adoção de micro *frontends* em contexto empresarial.

5.1 Técnicas de Integração

Dada a natureza autocontida dos módulos aplicacionais correspondentes aos vários MFEs, cujo propósito é através do desenvolvimento independente e especializado responder às necessidades inerentes aos respetivos contextos de negócio, para que no final seja apresentada ao utilizador uma aplicação completamente funcional, é necessário que se proceda à sua integração/orquestração.

Este processo de orquestração pode ocorrer em dois momentos distintos: em tempo de compilação ou em tempo de execução, sendo que neste último pode dar-se do lado do cliente, do servidor ou até mesmo resultar da combinação de ambos os cenários com vista a potenciar a *performance* (Nagy, 2019).

5.1.1 *Build-Time Integration*

A integração em tempo de compilação é uma técnica que propõe a publicação de cada MFE como um pacote que é depois incluído como dependência pela aplicação integradora.

Apesar de à primeira vista poder fazer sentido produzir um único artefacto JavaScript implantável, atualizar um único micro *frontend* pode revelar-se um processo algo sinuoso (C. Jackson, 2019).

Isto deve-se em grande medida ao facto de a aplicação integradora na prática consistir num monolítico. Daí que sempre que se proceda a uma alteração num MFE, seja necessário publicar no servidor local de pacotes uma nova versão, atualizar a referência no monolítico, recompilar todos os MFEs e efetuar uma nova *release* da aplicação integradora (Sowiński, 2019).

Esta técnica provoca inevitavelmente atrasos significativos no processo de desenvolvimento e introduz um elevado acoplamento no processo de *release*, o que coloca em causa os princípios que a arquitetura de MFEs advoga.

Perante isto, é perentório afirmar-se que se deve enveredar por uma técnica de integração que ocorra em tempo de execução (seja *Server-Side* ou *Client-Side*).

Tabela 2 - Vantagens e Desvantagens da *Build-Time Integration*

VANTAGENS	**DESVANTAGENS**
Evita duplicação de código	Atraso o processo de desenvolvimento
	Não permite *releases* independentes

5.1.2 Client-Side Integration

As técnicas *de Server-Side Integration* são imprescindíveis para *sites* em que o carregamento rápido é uma preocupação de primeira-linha. Todavia, em algumas aplicações o tempo de carregamento não é tão relevante e a ênfase recai sobretudo em reagir prontamente aos *inputs* do utilizador. Nessa matéria, ficando a *Server-Side Integration* aquém do esperado, abre-se espaço para a *Client-Side Rendering*.

A CSR, que permite que a estrutura do HTML seja produzida e atualizada diretamente no *browser*, tem vindo a popularizar-se sobretudo com o aparecimento de *frameworks* como Vue.js, Angular e React.

5.1.2.1 Integração usando *Links*

A integração por links determina que cada equipa detenha o seu próprio diretório de ficheiros, no qual constam ficheiros HTML e *assets* estáticos como ficheiros de estilo, *scripts* e imagens requeridos pelas aplicações.

As funcionalidades desenvolvidas no âmbito destas aplicações são renderizadas em páginas HTML independentes.

Suponha-se que existem dois serviços a integrar, mantidos por equipas autónomas e possuindo cada um a sua própria base de dados, em conformidade com o padrão *Database por Service* (Richardson, 2015b).

Neste cenário hipotético, apresentado na Figura 23, o serviço 1 é responsável pelos detalhes de um sapato (incluindo um botão para comprar) e o serviço 2 pelo carrinho de compras.

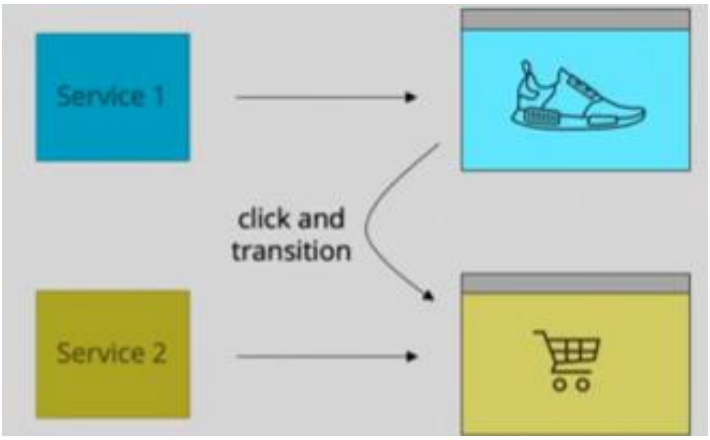


Figura 23 - Integração usando Links (Dörnenburg, 2019)

Enquanto o serviço 1 armazena informação relativa ao produto, o serviço 2 detém parte dessa informação, mais especificamente, o preço, o que se traduz inevitavelmente em alguma duplicação da informação (Dörnenburg, 2019).

Adicionados nos projetos de ambas as equipas os recursos das páginas, verifica-se nos ficheiros CSS (divididos nas seções de estilos globais, de página e de conteúdo da página) alguma replicação de código. Assim sendo, se se pretender promover uma alteração global nos estilos, por exemplo da fonte ambas as equipas têm de editar os seus ficheiros.

Num cenário em que exista uma grande diversidade de sapatos a apresentar, apesar de o HTML ser muito similar entre eles e, portanto, facilmente parametrizável usando outra abordagem de integração, é necessário a criação de um ficheiro HTML por sapato.

A duplicação, no entanto, não é problemática já que segundo os pressupostos da arquitetura de MFE é aceitável não cumprir escrupulosamente o princípio “DRY” se a salvaguarda do isolamento e baixo acoplamento entre os MFE assim o exigir.

A integração entre MFEs é obtida através de navegação entre *links*. Neste caso, o utilizador ao aceder à página de detalhes do sapato (serviço 1) e clicar no botão de compra, é encaminhado para a página do carrinho de compras (serviço 2).

Posto isto, a tabela abaixo sintetiza as vantagens e desvantagens inerentes a esta alternativa.

Tabela 3 - Vantagens e Desvantagens da Integração por Links

VANTAGENS	DESVANTAGENS
Baixo Acoplamento	Muita redundância e sobrecarga
Alta robustez, já que a falha de um MFE não compromete a disponibilidade de toda a aplicação	UI extremamente limitada

Daqui se depreende que a integração por *links* só deverá ser aplicada em cenários em que não seja necessário num MFE incorporar informação ou funcionalidades de outros MFEs, o que atualmente corresponde a uma percentagem muito residual das aplicações *web*.

5.1.2.2 IFrame

Dada a sua natureza, os *iframes* permitem a construção de uma página a partir da incorporação de subpáginas independentes (C. Jackson, 2019).

Para tal, basta adicionar ao HTML da página principal uma *tag* <iframe> especificando como "*src*" a localização absoluta do MFE a carregar.

Este elemento HTML possui uma altura e largura predefinidas e nesta matéria se a largura pode ser facilmente gerida dinamicamente, o mesmo não acontece com a altura que tendo que assumir um valor fixo, requer uma coordenação entre as equipas sempre que se pretenda proceder a uma a alteração do componente a apresentar na *iframe* (Geers, 2020a).

Quanto às vantagens e desvantagens desta técnica, encontram-se sintetizadas na tabela abaixo.

Tabela 4 - Vantagens e Desvantagens da Integração por *Iframes*

VANTAGENS	DESVANTAGENS
Baixo Acoplamento	Dificuldades de acessibilidade
Elevado isolamento	Dificuldades de SEO
Elevada robustez	Restrições de *layout*

Este baixo acoplamento deve-se ao carácter puramente autocontido das *iframes*, que permite um desenvolvimento independente por parte das equipas.

Já o elevado isolamento e robustez prende-se respetivamente com o facto de se poder prescindir de convenções ou *namespacing* no JS e CSS para evitar colisões e com perante uma situação de falha ou indisponibilidade de um MFE, a página que o integra através da *iframe* continuar disponível.

No entanto, esta peculiaridade das *iframes* criarem um contexto interno, possui um impacto negativo quer em termos de *performance*, quer nas vertentes de SEO e acessibilidade, uma vez que a criação de contextos viola a semântica da página. A estas desvantagens acresce as restrições de *layout* inerentes a este elemento HTML, que requerendo uma coordenação entre as equipas para ser contornadas criam acoplamento no *layout* (Geers, 2020a).

O Spotify é um exemplo de organização que já adotou esta abordagem na sua aplicação *desktop* (Mezzalira, 2019a).

5.1.2.3 AJAX

As técnicas de integração via *Iframes* ou *Links* apesar de oferecerem um irrefutável isolamento, sobretudo devido aos MFE serem desenvolvidos e integrados como páginas separadas, deixam bastante a desejar no que toca à redundância de código, acessibilidade, e SEO e usabilidade. Abrindo-se espaço para a aposta numa integração mais profunda, o AJAX pode ser a solução.

Contrariamente às técnicas de *Client-Side Integration* suprarreferidas, através da transclusão do lado do cliente, *Client-Side Includes* (CSI), o AJAX permite que as funcionalidades desenvolvidas pelas equipas sejam devolvidas sob a forma de fragmentos HTML que podem ser facilmente integrados no DOM de uma página que deles necessite (Kotte, 2017).

Tendo em conta o cenário hipotético da loja online de sapatos mencionado na seção 5.1.2.1, usando AJAX o carrinho de compras pode ser renderizado como um fragmento HTML integrável na página de detalhes do produto, conforme descrito na Figura 24.

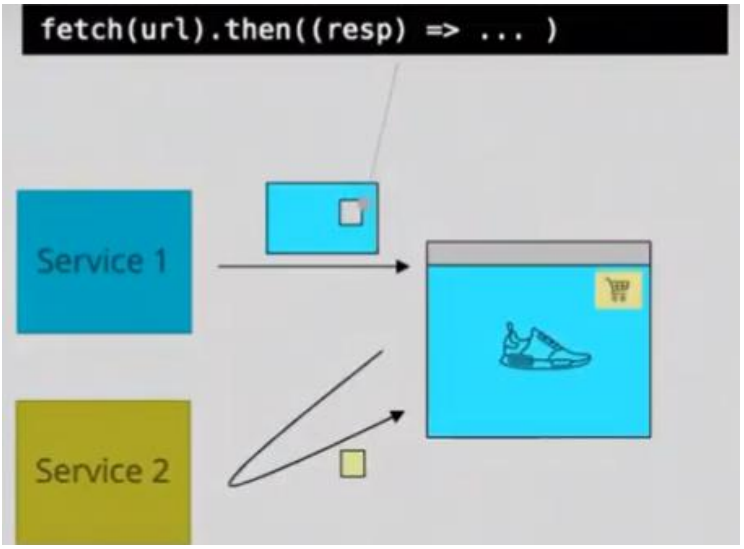


Figura 24 - Integração usando AJAX (Dörnenburg, 2019)

Supondo que as equipas responsáveis pelo desenvolvimento desta loja tinham inicialmente enveredado por uma integração via *iframes*, a reconversão é simples, consistindo apenas na remoção das *tags* <html> e <body> dos ficheiros HTML e na criação de um ficheiro CSS que vai alojar as regras de estilização do fragmento.

Efetuadas estas alterações, a equipa responsável pelo serviço 1 já pode, através de um pedido AJAX, carregar o fragmento correspondente ao carrinho de compras disponibilizado pelo serviço 2 e injetá-lo no seu DOM.

Este pedido AJAX é efetuado através de um *script*, que será incluído no HTML da página de produto, ao adicionarmos um elemento <script> cujo atributo "*src*" aponta para a localização do ficheiro JS. Neste *script* é efetuado um pedido para obter o fragmento HTML, que depois de obtido com sucesso é então injetado no nó especificado da página.

Apesar desta técnica fornecer bastante flexibilidade em termos de integração, dando liberdade a cada *bounded context* (conceito abordado na secção 2.1.6.4) para decidir como certas partes da página são renderizadas, também é possível elencar algumas limitações (Millett & Tune, 2015).

Desde logo, ao não assegurar por defeito uma autocontenção do CSS e do JS dos MFE que integra, o que pode refletir-se em colisões de estilos e de variáveis. Todavia, estas colisões podem ser contornadas ou minimizadas através da definição de convenções.

Além disso, implica que o *browser* suporte JavaScript, pelo que não se seguindo os pressupostos do *Progressive Enhancement*, se falhar o carregamento do JS a integração cai por terra e a aplicação fica indisponível.

No entanto, os princípios desta filosofia podem ser respeitados desde que o carregamento dos fragmentos HTML e dos respetivos recursos não dependa de JS e este ser utilizado meramente para melhorar a UX, algo que será esmiuçado mais à frente.

Estando o desenvolvimento dos vários serviços ao cuidado de equipas distintas, não existindo uma infraestrutura centralizada, estes por norma são expostos através de domínios diferentes, o que se faz com que os URLs da loja sejam demasiado extensos e eventualmente confusos aos olhos dos utilizadores (Geers, 2020a).

Este problema pode ser solucionado através da criação de um servidor *web* partilhado, que atuando como *middleware* centraliza os pedidos (Richardson, 2015a) provenientes do *browser* e destinados aos MFE a integrar e analisando o caminho do URL verifica se há correspondência com alguma regra definida na tabela de encaminhamento. Em caso afirmativo, encaminha os pedidos para a respetiva aplicação.

Além disso, permite centralizar questões como a monitorização e a autenticação. O NGINX é uma tecnologia capaz de servir os propósitos deste servidor (NGINX, 2015).

Num contexto de adoção da arquitetura de MFEs em que as equipas sendo autónomas estão incumbidas pela gestão das aplicações *end-to-end*, numa ótica de baixo acoplamento, a existência de um único serviço centralizado se não for devidamente acautelada em termos de responsabilidades pode atentar contra esse desígnio. Todavia, como se comprovará posteriormente neste documento, é possível resolver a questão, sem entrar em rota de colisão com os pressupostos desta abordagem.

Posto isto, as vantagens e desvantagens inerentes a esta abordagem encontram-se compiladas na Tabela 5.

Tabela 5 - Vantagens e Desvantagens da Integração via AJAX

VANTAGENS	**DESVANTAGENS**
Não existem limitações de *layout*	Carregamento é assíncrono
Possibilita a adoção do Progressive Enhancement	Não fornece um isolamento predefinido
Preserva a semântica da página, assegurando a acessibilidade e potenciando o SEO	Qualquer ação despoletada pelo utilizador vai requerer a realização de um novo pedido ao servidor para gerar uma versão atualizada do fragmento HTML

Daqui podemos inferir que a integração por AJAX é uma opção a considerar sobretudo quando a renderização ocorre do lado do servidor, oferecendo robustez e fácil implementação.

Contudo, é desaconselhada em cenários em que a elevada interatividade da aplicação seja um requisito, visto que os sucessivos *round-trips* ao servidor vão impactar o tempo de resposta, ainda que esse impacto possa ser minimizado recorrendo a um servidor *web* partilhado, cache, ou efetuando a renderização do lado do cliente (Geers, 2020a).

5.1.2.4 *Web Components*

Numa arquitetura de MFE, as UIs das diferentes equipas devem ser autocontidas e capazes de evoluir de forma independente.

A integração dos diferentes MFE, deve ser agnóstica da tecnologia e não ser baseada numa única *framework*, sob pena de uma alteração nessa *framework* impactar todo o FE (Geers, 2020a).

Os *Web Components* visam permitir o desenvolvimento de componentes personalizados autocontidos, que funcionem em qualquer lugar e com qualquer *framework*.

Assim, potenciam a modularização de aplicações complexas e ou de grande dimensão (Ast & Gaedke, 2017) e fornecem um melhor encapsulamento e interoperabilidade (Martínez-Ortiz et al., 2016) entre as bibliotecas ou *frameworks* presentes na *stack* tecnológica dos MFE detidos pelas várias equipas.

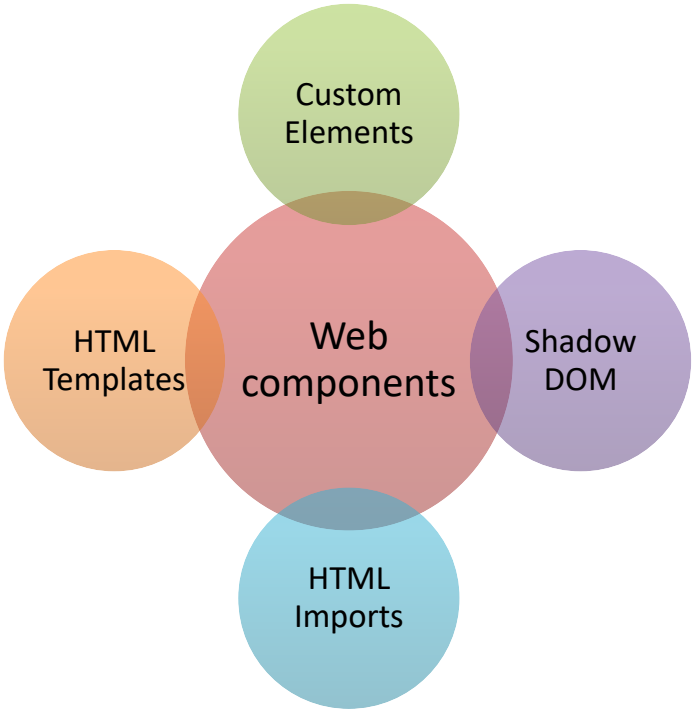


Figura 25 - Padrões dos *Web Components*

Com base na figura, podemos constatar que o conceito de *Web Components* assenta em quatro padrões: *Custom Elements*, *Shadow* DOM, *HTML Templates* e *HTML Imports*[2] (Bidelman, 2019).

Relativamente aos *Custom Elements*, possibilitam o desenvolvimento de novos elementos, que não constam da especificação HTML, através de uma *tag* HTML associada.

No que concerne o *Shadow* DOM, permite isolar uma subárvore do DOM do resto da página, evitando colisões de estilo entre os MFEs, já que o seu contexto/*scope* é definido com o nome do respetivo *Custom Element* – *Scoped* CSS.

Apesar dos benefícios que oferece, o *Shadow* DOM não é suportado nativamente em *browsers* mais antigos, requer JavaScript para funcionar e ser definido, dificulta a partilha de estilos comuns entre *Shadow* DOM distintos e não funciona com Bootstrap ou outras bibliotecas globais de estilização.

[2] Os HTML Imports têm vindo a ser substituídos por ES Modules. Assim, assegura-se a compatibilidade dos módulos ES com os módulos HTML e vice-versa (Cooney, 2016)

Dissecados os conceitos fundamentais que sustentam esta abordagem, as vantagens e desvantagens estão sintetizadas na Tabela 6.

Tabela 6 - Vantagens e Desvantagens da Integração via *Web* Components

VANTAGENS	**DESVANTAGENS**
É uma estratégia de integração amplamente adotada	Requer a execução de JavaScript no cliente para funcionar
Através dos *Custom Elements* e *Shadow* DOM permite que os *Web Components* sejam autocontidos e detendo um isolamento semelhante ao do obtido com *iframes*	Não permite alcançar um *Progressive Enhancement*
É agnóstico da tecnologia adotada nos vários micro *frontends*	O *Shadow* DOM não é suportado nativamente por *browsers* mais antigos
Server-Side Rendering do Angular >= 2 permite um SEO efetivo e boa usabilidade	Caso se tratem de *web* componentes nativos e não conseguirem ser renderizados, também não conseguem ser indexados pelo SEO (Jorgé, 2016)

Posto isto, constata-se que a *Client-Side Integration* assente em *Web Components* é uma opção sólida no desenvolvimento de aplicações interativas e nas quais os componentes visuais desenvolvidos pelas várias equipas devem ser integrados numa única página (SPA).

Contudo, isso não significa que toda a aplicação precise enveredar pela *Client-Side Rendering*, podendo verificar-se uma combinação de integração *Server-Side* e *Client-Side*.

5.1.3 *Server-Side Integration*

A *Server-Side Integration* é uma técnica que já se encontra plenamente enraizada no processo de desenvolvimento de FE. Consiste na renderização do HTML do lado de servidor a partir de múltiplos *templates* ou fragmentos (C. Jackson, 2019), cuja integração ao acontecer antes de chegar ao *browser*, permite um carregamento mais rápido da página.

Outro benefício reside no facto de não depender da execução de JavaScript do lado do cliente, possibilitando um desenvolvimento consonante com os princípios do *Progressive Enhancement*.

Partindo de um ficheiro HTML que contém os elementos comuns da página (que no exemplo da Figura 26 corresponde à página de detalhe do produto), integra conteúdo específico proveniente dos fragmentos HTML das aplicações/MFE a integrar.

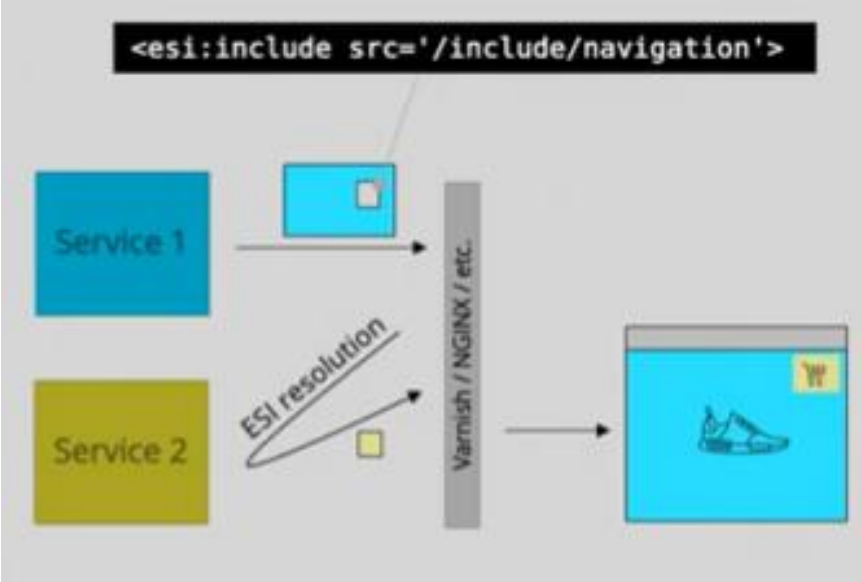


Figura 26 - *Server-Side Integration* usando ESI (Dörnenburg, 2019)

Esta integração é concretizada num servidor *web* centralizado, colocado entre o *browser* e os diversos módulos aplicacionais (Geers, 2020a), recorrendo a diretivas como o *Server-Side-Includes* e o *Edge-Side Includes*, pelo que o primeiro passo é configurar este servidor.

5.1.3.1 *Server Side Includes*

O *Server-Side Includes* (SSI) corresponde a uma linguagem de marcação que fornece um conjunto de diretivas que permitem adicionar conteúdo dinâmico a documentos ou páginas HTML (Apache, sem data) - transclusão, do lado do servidor, suportado por servidores *web* como o NGINX, Apache (Isaac, 2014) e IIS (Anderson, 2016).

Implementando o SSI numa página que integre fragmentos HTML, a primeira resposta de HTML irá conter as referências para todos os recursos necessários, que podem ser carregados mais prontamente e em paralelo.

Deste modo, o *browser* é capaz de montar a estrutura da página e apresentar algo ao utilizador mais cedo.

Quanto à implementação do SSI propriamente dita, há que começar por habilitar no ficheiro de configuração do servidor o suporte ao SSI para seguidamente se adicionarem as diretivas, estruturadas como ilustrado na linha de código apresentada abaixo.

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

Código 1 - Diretiva SSI (Geers, 2020a)

Aquando da análise do HTML pelo servidor, estas diretivas serão substituídas pelo conteúdo proveniente do URL especificado.

Relativamente às potencialidades que o SSI oferece, destacam-se as seguintes:

- melhores tempos de carregamento face a outras técnicas de integração como o AJAX, que no caso ilustrado na Figura 27 chega a ser menor em 40%;

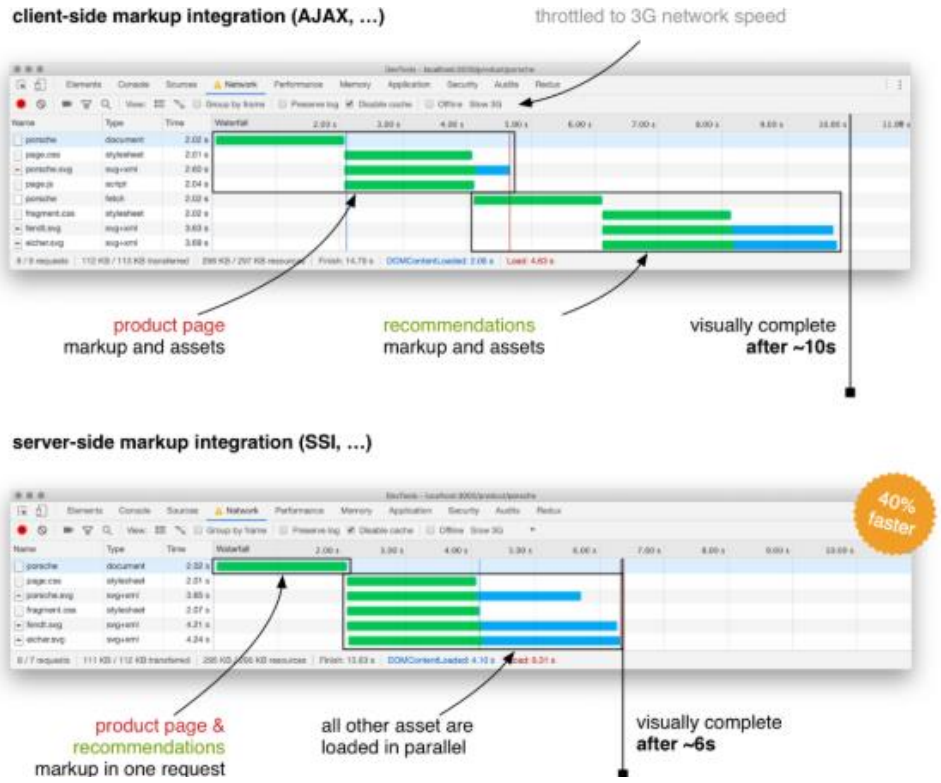


Figura 27 - Tempo de carregamento de uma página usando AJAX e SSI (Geers, 2020a)

- possibilidade de lidar com fragmentos não fiáveis, ou seja, de impedir atrasos na *Server-Side Integration* motivados pela inclusão de fragmentos HTML cujo carregamento é demorado ou falha, já que somente após o processamento de todas as diretivas SSI é que o servidor responde ao *browser* enviando o HTML já com os fragmentos integrados. Para tal, o NGINX permite especificar tempos máximos de resposta ou *timeouts* e conteúdos alternativos ou *fallback content*, isto é, de conteúdo a ser renderizado quando falha a inclusão de um dado fragmento;

- carregamento em paralelo dos vários fragmentos, designando-se o tempo necessário para a montagem do HTML completo e o seu envio para o *browser* de *Time To First Byte* (TTFB);

- carregamento de *nested fragments*, isto é, *includes* dentro de *includes*, ainda que o seu uso deva ser devidamente ponderado;

- num contexto de integração em páginas com alguma complexidade, devendo-se usar SSI para o carregamento dos fragmentos mais importantes, os restantes podem ser carregados apenas quando necessário – *lazy loading*.

A decisão da solução de *Server-Side Integration* a adotar deve recair sobre qual a forma de carregamento de fragmentos HTML que melhor se ajusta aos requisitos, pois refletindo-se no TTFB, por conseguinte, dita o tempo de carregamento da aplicação *web*.

Como mencionado anteriormente, o NGINX só envia o HTML depois de concluída a inclusão dos vários fragmentos.

Além do envio integral, também é possível optar por abordagens em que o envio é realizado parcialmente (usando ESI em algumas versões do Varnish) ou então através de *streaming*, ocorrendo neste caso em simultâneo o envio para o *browser* das primeiras partes do *template* e o carregamento dos fragmentos (Podium e Zalando).

5.1.3.2 Edge Side Includes

O *Edge Side Includes* (ESI) é uma linguagem de marcação concebida pela Akamai e a Oracle, que à semelhança da SSI permite através de diretivas a transclusão do lado do servidor (Akamai, sem data).

Na integração baseada em ESI a o NGINX é substituído pelo Varnish [3] e a sintaxe das diretivas muda, assumindo a estrutura apresentada na linha de código abaixo.

```
<esi:include src="/url/to/include">
```

Código 2 - Diretiva ESI (Geers, 2020)

À semelhança do SSI, o ESI possibilita a definição de *timeouts* e a especificação de conteúdo alternativo, no entanto, difere na forma de carregamento dos fragmentos a integrar, visto que na versão comercial do Varnish permite o envio parcial.

Sendo o NGINX e o Varnish bastante similares enquanto servidores de *reverse proxy*, a diferença reside no facto de o Varnish oferecer um vasto leque de comandos para gerir a cache, apesar de o NGINX também poder ser usado como servidor de cache (Gao, 2015).

Além disso, o NGINX é um *software open-source* (OSS) enquanto para ter acesso a todo o potencial que Vanish pode oferecer é necessário adquirir a versão comercial (Gao, 2015).

O *site* da IKEA é um exemplo de utilização de ESI (juntamente com *Client Side Includes*, que será abordada posteriormente) (Kotte, 2017; Mezzalira, 2019a).

Daqui se infere, que num *site* em que se verifique um elevado tráfego e se apresente conteúdo pesado, a cache poderá ser um fator determinante na melhoria de *performance*, pelo que a escolha deve recair sobre o Varnish. Caso contrário, o NGINX é mais do que suficiente.

[3] Note-se que existem outros servidores que suportam ESI (Gao, 2015; Geers, 2020a).

5.1.4 *Single-Page Application* Unificada

Depois de nas subseções anteriores (2.3.1 a 2.3.3), o foco ter incidido sobre técnicas de integração das *UIs* desenvolvidas pelas várias equipas numa única *view*, esta secção abordará a integração ao nível das páginas.

A integração por *links*, referida anteriormente, apesar de ter como propósito a integração de páginas, não permite responder minimamente às necessidades das aplicações *web* atuais.

Por outro lado, na integração usando AJAX foi referida a possibilidade de colocar um servidor, entre *browser* e aplicações das várias equipas, responsável por encaminhar os pedidos recebidos de acordo com a sua rota à equipa correspondente, isto é, que atua como *router*.

Contudo, atualmente a maioria das *frameworks* JavaScript conta com uma solução de *routing* dedicada, através da qual é possível navegar entre diferentes páginas de uma aplicação sem que isso despolete um *full refresh* e tenha de se proceder ao pedido e processamento de um novo documento HTML. O *browser* apenas necessita de renderizar novamente as partes da página que sofreram alterações. Nesse sentido, optar por uma transição de página no *browser* é a solução mais apelativa em termos de UX.

Num cenário de aplicação monolítica baseada em *links* é efetuado um *full refresh* sempre que se navega entre páginas. Já num contexto de várias SPAs que comunicam entre si via *links* e que internamente possuem o seu próprio router, constata-se a realização de *full refresh* sempre que o utilizador cruza as fronteiras das SPAs e de simples transições quando tal não acontece.

No entanto, através da introdução de uma aplicação centralizada é possível erradicar os *full refresh* e assegurar uma UI mais consistente e apelativa ao utilizador assente em transições suaves. Surge assim o conceito de *App Shell*.

5.1.4.1 *App Shell*

A arquitetura *Application Shell* permite a criação de *Progressive Web Apps* (PWAs), aplicações *web* desenvolvidas adotando um conjunto de tecnologias específicas e padrões que permitem tirar proveito das funcionalidades oferecidas pelas aplicações *web* e nativas (MDN contributors, 2020). Em SPAs repletas de JavaScript, a adoção desta abordagem é decisiva na melhoria dos tempos de carregamento.

A *app shell* compreende o conjunto de recursos (HTML, CSS e JS) estritamente necessário para apresentar uma primeira versão da UI. O facto destes recursos serem armazenados em cache (através de um *service worker* que interceta os pedidos recebidos e guarda as respostas desejadas) permite garantir uma resposta instantânea e confiável ao utilizador, uma vez que nos acessos seguintes não voltarão a ser transferidos da rede, necessitando a aplicação apenas de transferir o conteúdo dinâmico para cada página ou MFE (Osmani, 2019).

Numa arquitetura de MFEs, esta *app shell* atua como "aplicação-contentor" dos vários MFEs, já que estando sempre à escuta de alterações do URL é responsável por intersetar e analisar os pedidos recebidos e renderizar o conteúdo pretendido em função da rota.

Visto que esta "aplicação-contentor" constitui um pedaço de código partilhado, deve procurar manter-se a sua simplicidade, ainda que possa contemplar responsabilidades transversais como autenticação ou monitorização, e acima de tudo evitar que seja contaminada por lógica de negócio que extravase de algum dos *bounded contexts* que em si confluem.

Sendo um dos princípios norteadores da arquitetura de MFEs a definição clara de responsabilidades, é necessário definir quem estará incumbido pela implementação e gestão da *app shell* a par das preocupações transversais a desenvolver.

Nessa vertente, Geers avança duas alternativas:

- em organizações de maior dimensão, este papel deverá ser desempenhado por equipas dedicadas à infraestrutura, numa ótica similar às "*Infrastructure Squads*" do Spotify (Fernandes, 2017a);

- nos restantes casos, visto que estas tarefas não têm um valor direto para o cliente, poderão ser distribuídas pelas equipas multidisciplinares incumbidas das aplicações dos vários MFEs.

Neste contexto, por exemplo, a equipa responsável pela aplicação MFE de produto pode ficar incumbida da *app shell* e sempre que uma das restantes equipas necessite de efetuar alguma alteração na *app shell* basta fazê-lo e abrir um *merge request (MR)*. Este *MR* será depois analisado e eventualmente aceite pela equipa de produto, na qualidade de responsável pela *app shell* (M. Geers, comunicação pessoal, 19 de Fevereiro de 2020).

Posto isto, espera-se que num contexto de MFEs, a *app shell* forneça o documento HTML partilhado, mapeie os URLs para as páginas das equipas, renderize a página correspondente à rota recebida e que atualize o URL apresentado na barra de endereços em conformidade (Geers, 2020a).

A *app shell* e as equipas firmam um contrato no qual se definem os contornos da sua relação.

Este contrato estabelece que as equipas se comprometem a publicar a lista de URLs das suas páginas para que outras equipas possam aceder mesmo sem conhecerem o nome do respetivo componente, uma vez que a *app shell* é que será responsável pelo *routing* deste pedido. Este *routing* pode possuir um ou dois níveis.

No primeiro caso como as rotas de todas as páginas desenvolvidos pelas várias equipas se encontram especificadas no ficheiro HTML centralizado, é necessário que a *app shell* conheça todos os URLs, o que cria um acoplamento tal entre as equipas e a *app shell* que faz com que

sempre que uma só equipa pretenda manipular os seus URLs seja necessário ajustar e reimplantar a *app shell* (Geers, 2020a).

Ora, usando um *routing* em dois níveis, as rotas internas de uma equipa, isto é, das páginas que desenvolve, passam a ser especificadas num componente mantido por essa mesma equipa e que atua como router interno para os *endpoints* específicos das suas páginas.

Neste cenário, a tabela de encaminhamento da *app shell* é mais concisa e ao receber um pedido analisa o URL e encaminha-o para o router interno da equipa responsável.

Desta forma, o acoplamento verificado entre *app shell* e as equipas no que toca às rotas definidas, é desfeito sempre que a alteração seja interna à equipa, todavia, persiste quando muda a titularidade sobre uma página ou componente (algo muito raro).

Posto isto, esta técnica de integração pode fazer sentido quando o utilizador necessita de alternar frequentemente entre interfaces detidas por equipas distintas e em aplicações nas quais fornecer uma elevada interatividade é mais importante do que o tempo inicial de carregamento da página ou seja necessário centralizar preocupações como autenticação e monitorização (Geers, 2020a).

Apesar das vantagens em termos de UX e tratamento centralizado de preocupações transversais aos vários MFEs, a *app shell* também levanta desafios, nomeadamente quanto à comunicação entre MFEs, constituir um ponto único de falha e dificultar a deteção de erros, etc.

5.1.5 *Universal Composition*

O amplo suporte da renderização universal pelas principais *frameworks* veio agilizar o desenvolvimento de aplicações executadas quer no servidor, quer no *browser*, uma vez que existindo uma única *codebase* se evita a duplicação de código.

A composição universal consiste na combinação de técnicas *client-side* e *server-side*.

Conforme evidenciado na Figura 28, primeiramente é construído o esqueleto da *markup* dos vários micro *frontends* no servidor através de técnicas *server-side* como SSI, ESI ou Podium, que efetuam a integração destes blocos no documento HTML enviado para o *browser*. Do lado do cliente, o HTML estático de cada MFE é transformado em DOM dinâmico (*hydration*), para que doravante a aplicação interaja com o utilizador inteiramente no *browser*.

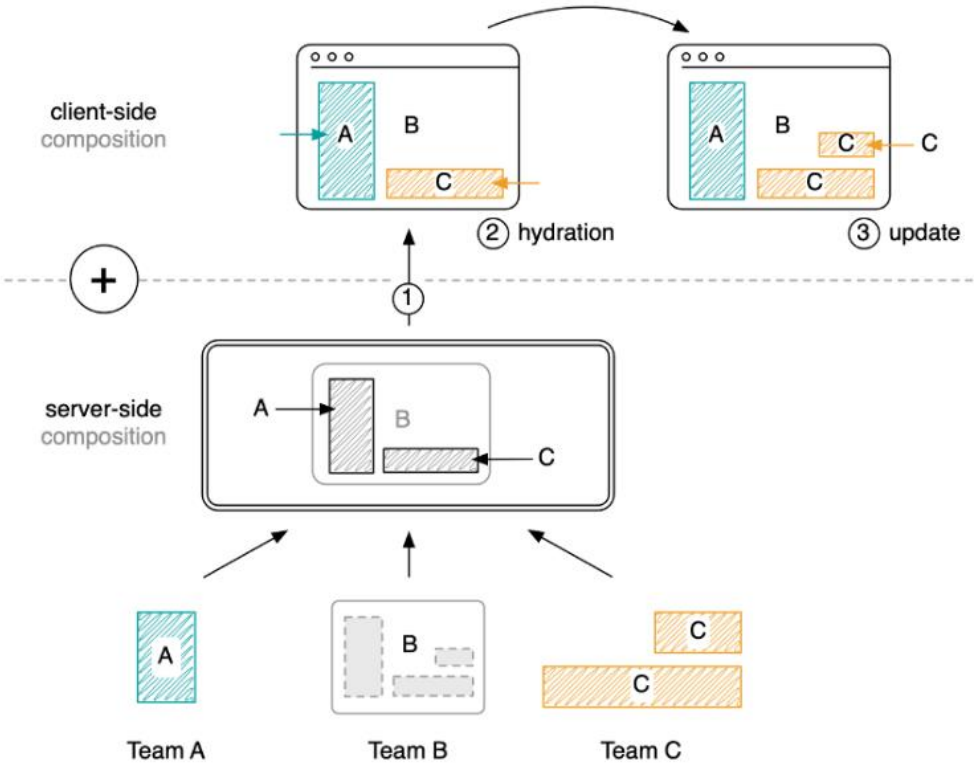


Figura 28 – *Universal Composition* (Geers, 2020a)

Esta técnica permite contornar problemas como demora, falha ou bloqueio no carregamento de JavaScript do lado do cliente, JavaScript esse que é imprescindível para a integração dos diversos micro *frontends* seguindo uma técnica puramente *client-side*, e diminuir substancialmente o tempo de carregamento inicial, ao renderizar instantaneamente apenas o estritamente necessário para que a aplicação fique funcional (*progressive enhancement),* sem que isso comprometa a reatividade e interatividade da aplicação no *browser* (Geers, 2020a).

Suponhamos que numa aplicação *e-commerce* na página de detalhe de um produto existe um botão de adicionar ao carrinho, que se trata de um *web component.* Analisando o HTML recebido pelo *browser*, constata-se a existência de um *custom element*, inicialmente vazio (Código 3), cujo *template* só será apresentado no ecrã após ser carregado pelo JavaScript do lado do cliente (Código 4).

```
<checkout-buy sku="fendt"></checkout-
```

Código 3 – Presença de um *custom element* no HTML recebido pelo *browser* (Geers, 2020a)

```
<checkout-buy sku="fendt">
        <button type="button">buy for 54 $</button>
</checkout-buy>
```

Código 4 – Conteúdo renderizado após o carregamento do JS no cliente (Geers, 2020a)

Por predefinição, os *web components* não fornecem qualquer *standard* para a SSR, que, no entanto, pode ser ocorrer por via de diretivas SSI ou ESI enviadas como *children* destes *Custom Elements*, conforme apresentado no código abaixo.

```
<checkout-buy sku="fendt">
        <!--#include virtual="/checkout/fragment/buy/fendt" -->
</checkout-buy>
```

Código 5 – Passagem de diretiva SSI como *child* do *Custom Element* (Geers, 2020a)

Neste caso, um servidor NGINX ou similar fará a substituição da diretiva ESI pela *markup* do botão, carregada do *endpoint* especificado, que será adicionada ao documento HTML enviado ao *browser*.

Visto que se estão a combinar duas técnicas de integração, cabe à equipa responsável pelo micro *frontend* fornecer quer a definição do *Custom Element*, quer o *endpoint* que devolve a *markup* que deve ser renderizada no servidor.

Relativamente à *tag* associada ao *Custom Element*, para evitar colisões deve ser definida com base em convenções de nomes, sendo boa prática usar como prefixo o nome da equipa.

Desta forma, o *browser* poderá apresentar o botão de imediato. Enquanto o carregamento de JavaScript não estiver concluído, por omissão, não executaria qualquer ação.

Contudo, aplicando os princípios do *Progressive Enhancement*, pode-se alterar a *markup* renderizada no servidor colocando o botão dentro de um elemento *form*, que possibilite a execução de uma ação similar à despoletada assim que o carregamento for concluído e se proceda ao *hydrate* (no qual é validada a *markup* e efetuado o *bind* dos eventos para possibilitar a interação) (Geers, 2020a).

Apesar de acarretar um trabalho adicional, o desenvolvimento em conformidade com o *Progressive Enhancement* torna as aplicações mais robustas e resilientes.

Alternativamente a usar SSI e *Web Components* para aplicar esta técnica, poder-se-ia:

- recorrer a bibliotecas com capacidades de renderização *server-side* para dinamicamente gerar a *markup* no servidor e após o carregamento de JavaScript no *browser* estar concluído efetuar o *hydrate* dessa *markup* (Geers, 2020a)*;*
- utilizar outras técnicas de *Server-Side Integration;*
- adotar a Ara Framework, uma biblioteca de *Universal Rendering*, esmiuçada na seção 5.2.4.1.

Teoricamente também seria possível criar uma *Single-Page Application* Unificada Universal, adotando uma abordagem similar à seguida na renderização também para o *routing*, contudo, poderia introduzir um nível adicional de complexidade e à data de realização deste documento não foi encontrado nenhum exemplo de implementação.

Note-se que apesar de uma equipa poder querer implementar *Universal Rendering*, nem sempre se justifica. Numa situação em que existam duas aplicações cujos fragmentos não tenham de comunicar e/ou não sejam interativos, a SSR é suficiente, que pode a qualquer momento e por decisão da equipa ser complementada com CSR (Geers, 2020a).

5.2 Bibliotecas/*Frameworks* de Integração

Esta secção destina-se a apresentar as bibliotecas e *frameworks* à disposição das equipas de desenvolvimento, que permitem facilitar a implementação de algumas das técnicas de integração de MFEs descritas na secção 5.1. Assim sendo, em função do local onde ocorra a integração podem ser categorizadas em *client-side*, *server-side* e *universal composition*.

5.2.1 *Client-Side Integration*

Relativamente às bibliotecas/*framework* de integração *client-side*, existem várias opções, entre as quais o projeto Luigi, que procedem à composição dos MFEs no *browser*.

5.2.1.1 Luigi

O Luigi é uma *framework* desenvolvida pela SAP que permite a equipas distribuídas criarem de forma agnóstica de tecnologia aplicações modulares, extensíveis, escaláveis e que fornecem uma interface unificada ao utilizador (SAP, sem data-b).

Lançado originalmente com o propósito de produzir aplicações empresariais, modulares e facilmente integráveis com o SPA, o Luigi adota como técnica de orquestração as *iframes*.

Podendo ser estas aplicações classificadas como soluções B2B, aspetos como SEO e largura de banda não são particularmente relevantes e considerando a gestão eficiente de memória que as *iframes* oferecem, resultante do facto da memória ser libertada aquando da remoção da *iframe* do DOM, a adoção desta técnica de integração neste contexto pode revelar-se uma mais-valia (Mezzalira, 2020).

O Luigi é composto pela aplicação Luigi Core e pelas bibliotecas Luigi Client, que estabelecem a comunicação entre a aplicação e os micro *frontends* que contêm por via da API *postMessage*, disponibilizada nativamente pelos *browsers* (*SAP/luigi*, 2018/2020).

Para permitir uma comunicação fluída, é possível configurar definições como *routing*, navegação, autorização e elementos da UI.

5.2.1.2 Outras *frameworks*

Existem outras *frameworks* capazes de integrar SPAs numa única aplicação, como o FrintJS e o Nut. No entanto, não existindo muita informação sobre as mesmas, sobretudo em termos da sua aplicação em contexto real, não serão abordadas (FrintJS, sem data; Nut, sem data).

5.2.2 *Server-Side Integration*

Relativamente às bibliotecas/*framework* de integração *server-side* de MFEs, existem uma série de alternativas, entre as quais o Tailor e o Podium.

5.2.2.1 Tailor

A implementação de *Server-Side Integration* usando ESI e SSI, mesmo que otimizada por via da paralelização da transferência de recursos e de *lazy loading*, revela-se ineficaz perante as exigências dos *sites* modernos, em especial as aplicações ricas em conteúdo (Facebook, 2010).

Além disso, estas otimizações não permitem dar uma resposta verdadeiramente eficaz ao estrangulamento provocado pela execução sequencial do carregamento dos fragmentos no servidor e da transferência dos recursos no *browser*.

Com vista a reduzir a latência em toda a extensão, deve-se procurar paralelizar o processo de carregamento e integração dos fragmentos no servidor e o carregamento de recursos e renderização da página no *browser*.

Alicerçada nesta premissa e parcialmente inspirado no BigPipe do Facebook (Shanmugam, 2016), surge a biblioteca Tailor da Zalando, parte integrante de um vasto rol de ferramentas publicadas no âmbito do denominado Project Mosaic que esta plataforma de *e-commerce* desenvolveu aquando do seu processo de conversão de uma arquitetura monolítica em MFEs (Zalando, sem data).

O Tailor é uma biblioteca de Node.js que usa *streams* para compor uma página *web* a partir de serviços de fragmentos (*zalando/tailor*, 2016/2020), que durante a análise do HTML das páginas ao detetar elementos HTML semelhantes ao apresentado abaixo, solicita o respetivo conteúdo, que depois de obtido vai ser adicionado ao HTML, substituindo esta *tag*.

```
<fragment src="/path/to/fragment">
```

Código 6 - Elemento HTML *Fragment* da biblioteca Tailor (*zalando/tailor*, 2016/2020)

Tal como sucede com o ESI e SSI também o Tailor permite definir *timeout* e conteúdo alternativo. No entanto, contrariamente ao SSI o *timeout* não é definido por *upstream* mas de forma declarativa em cada fragmento.

Quanto ao TTFB, ao possibilitar o processamento simultâneo no *browser* e no servidor, o Tailor permite alcançar bons resultados.

No que concerne o tratamento de recursos, o Tailor permite que na resposta a um pedido ao *endpoint* de um dado fragmento, além do HTML seja possível especificar estilos e *scripts* associados, ainda que não se consiga ter um verdadeiro controlo quanto à forma como são adicionados ao HTML (Geers, 2020a). Posto isto, na Tabela 7 estão elencadas as principais vantagens e desvantagens do Tailor.

Tabela 7 - Vantagens e Desvantagens da Biblioteca Tailor

VANTAGENS	DESVANTAGENS
Permite a pré-renderização (*Server-Side Rendering*), o que tem um impacto positivo no SEO e tempo de carregamento	Não permite controlar como CSS e JS são adicionados ao HTML
Não requer uma infraestrutura intermédia de integração, já que esta ocorre no contexto da aplicação Node.js	Pressupõe que os módulos de JS sejam carregados de forma assíncrona.
Menor TTFB, devido à paralelização de carregamentos e *streaming*	Não permite isolamento por omissão
Tolerante a falhas	

5.2.2.2 Podium

O Podium é uma biblioteca de integração baseada em Node.js, concebida pela Finn.no, uma plataforma de classificados norueguesa (Finn, 2018).

Fortemente influenciada pela biblioteca Tailor, o Podium introduz os conceitos de *podlet* e *layout*, que correspondem a versões melhoradas do fragmento e página, respetivamente.

O conceito aglutinador do Podium é o *podlet manifest*, que determina que por cada *podlet* exista um ficheiro JSON que detenha as informações apresentadas na Figura 29.

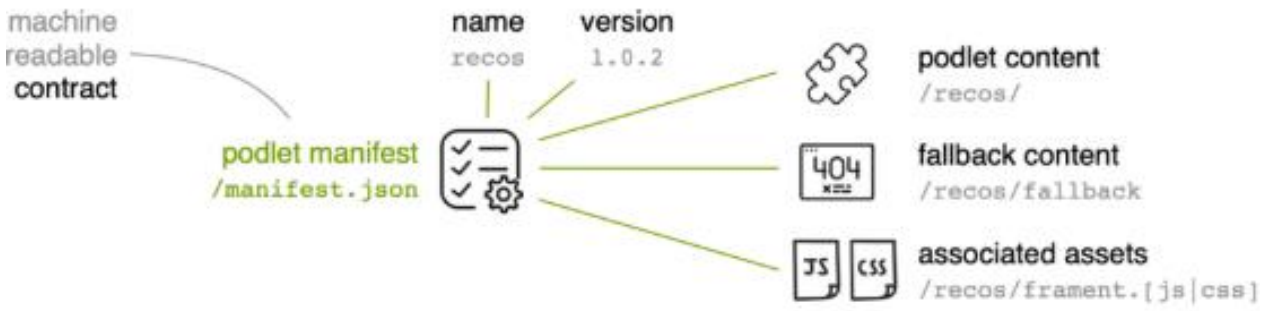


Figura 29 - *Podlet Manifest* (Geers, 2020a)

A figura faz ainda alusão à atuação do *podlet manifest* como um contrato estabelecido entre o detentor do *podlet* e o seu integrador. Este manifesto é disponibilizado sob a forma do ficheiro manifest.json através de um *endpoint* específico.

Arquiteturalmente falando, o Podium abrange duas bibliotecas: *layout* e *podlet*.

Relativamente à biblioteca *layout*, visa implementar no servidor tudo o que for necessário para este ser capaz de obter os conteúdos dos *podlets* a integrar. Para tal, analisa os manifest.json dos vários *podlets* afim de extrair os meta-dados requeridos na sua integração, sendo esta análise efetuada uma só vez por *podlet*. Além disso, o *layout* também está incumbido da implementação de cache (Podium, 2019b).

Quanto à biblioteca *podlet* é usada pela equipas responsáveis pelo desenvolvimento dos *podlets*, gerando um ficheiro manifest.json por *podlet* (Podium, 2019a).

Com o intuito de evitar problemas decorrentes da inclusão de fragmentos pouco fiáveis, tal como nas abordagens de *Server-Side Integration* referidas anteriormente, também no Podium é possível definir *timeouts* e *fallback content* (Podium, 2019c).

Relativamente ao *fallback content* é especificado pela equipa detentora do *podlet* e é guardado em cache pela equipa que o pretende integrar (Geers, 2020a).

A aplicação desta biblioteca de integração tem como principais vantagens:

- Permite a pré-renderização (*Server-Side Rendering*), o que tem um impacto positivo no SEO e tempo de carregamento;
- Não requer uma infraestrutura intermédia de integração, já que esta ocorre no contexto da aplicação Node.js;
- Menor TTFB, devido à paralelização de carregamentos e *streaming*;
- Através do *podlet manifest* é simples gerir os *assets* de que o *podlet* necessita, bem como o *fallback content*;
- Permite a utilização de qualquer motor de *templates* de Node.js;
- Tolerante a falhas

Contudo, o Podium, por omissão, não permite alcance o isolamento.

5.2.2.3 OpenComponents

OpenComponents (OC) consiste numa *framework* orientada a MFE, criada pela empresa OpenTable em 2014 e usada por organizações como a SkyScanner e a própria OpenTable (*OpenComponents*, sem data).

Assenta essencialmente em dois conceitos: componentes (*components*) e consumidores (*consumers*) (*OpenComponents*, sem data; *Opencomponents/Oc*, sem data).

Relativamente aos *components*, consistem em pequenos módulos de código isomórfico, compostos por HTML, JS e CSS, que podem conter alguma lógica que possibilite a uma aplicação *server-side* em Node.js renderizar uma *view* que resulte da composição destes *components*. Em termos de sintaxe, assemelha-se à apresentada no código abaixo.

```
<oc-component href="/path/to/fragment">
        Optionally some fallback text or HTML here
</oc-component>
```

Código 7 – Elemento HTML Component da OpenComponents

Como se evidencia na figura é possível adicionar *fallback content*.

Quanto aos *consumers*, correspondem a *sites* ou micro *sites* [4], que necessitam destes *components* para renderizar parte do conteúdo das suas páginas.

Em termos estruturais esta *framework* é composta por três partes: *cli* (que possibilita aos programadores criar, testar e publicar componentes), *library* (local onde os componentes são publicados, alojando também os recursos estáticos de que dependem) e *registry* (REST API usada para consumir e publicar componentes) (*Opencomponents/Oc*, sem data).

Um componente quando é criado é armazenado no *registry* e ao ser usado numa *view* é renderizado por via de SSR. Assim, apesar de ser necessária alguma coordenação, a necessidade de comunicação é mais diminuta já que em qualquer momento, basta que o *developer* aceda ao *registry* para conseguir incluir o componente pretendido (Herrington, 2019).

Note-se que quando um componente é publicado inclui o respetivo BE e FE (Mezzalira, 2019a).

Tabela 8 - Vantagens e Desvantagens da Biblioteca OpenComponents

VANTAGENS	**DESVANTAGENS**
Permite a pré-renderização (*Server-Side Rendering*), o que tem um impacto positivo no SEO e tempo de carregamento	Não permite controlar como CSS e JS são adicionados ao HTML
Menor TTFB, devido à paralelização de carregamentos e *streaming*	
Tolerante a falhas	

[4] A OpenTable define micro *sites* como "um pequeno conjunto de páginas *web* responsáveis pelo tratamento de uma parte específica da lógica de domínio" (Figus, 2015)

5.2.2.4 Outras *frameworks*

Existem ainda outras opções de *frameworks* que permitem *Server-Side Integration* sem requerer a criação de uma infraestrutura entre *browser* e aplicações/MFEs, entre as quais o Puzzle.js (*Puzzle-Js/Puzzle-Js*, sem data). No entanto, sendo bastante recentes e não existindo testemunhos relativos à sua adoção em contexto empresarial, não serão abordadas.

5.2.3 *Single-Page Application* Unificada usando Single-SPA

Existem várias bibliotecas/*frameworks* que permitem aplicar a técnica de *Single-Page Application* Unificada, entre as quais o Single-SPA e o Piral (que não possuindo casos de utilização em contexto empresarial, não será abordado).

A single-spa é uma *framework* desenvolvida pela Canopy, ainda em fase de desenvolvimento, que permite integrar múltiplos MFEs numa única aplicação FE, MFEs esses que podem ser desenvolvidos usando *frameworks* distintas. Assim, permite a utilização de diferentes *frameworks* numa mesma página sem necessitar de fazer *refresh* à página (Denning, 2020a).

Este carácter agnóstico de tecnologia, dá autonomia às equipas na definição da sua *stack* tecnológica e permite a adoção de novas tecnologias sem o fantasma da reescrita total a pairar sobre a aplicação. Além disso, a single-spa oferece a possibilidade de *lazy loading* e de implantar os MFEs de forma independente.

A organização define micro *frontend* como "uma seção da UI, que pode ser constituída por múltiplos componentes, a cargo de uma equipa independente, a quem cabe definir a *framework* a utilizar, ainda que se promova a adoção de uma única *framework* em toda a aplicação", que partilham com os micro serviços a possibilidade das *builds* e *deployments* serem realizados de forma independente.

O single-spa contempla três categorias distintas de micro *frontends*: *applications*, *parcels* e *utility* modules, cuja finalidade está descrita na tabela abaixo.

Tabela 9 – Tipos de Micro *Frontends* no Single-SPA (Denning, McMurdie, & Wilson, 2020)

TIPO	FINALIDADE
APPLICATION	Renderizam componentes para um conjunto de rotas específicas
PARCEL	Renderizam componentes que não controlam quaisquer rotas
UTILITY MODULE	Exportam lógica de JavaScript e CSS partilhada

É encorajado o uso de *applications* em detrimento de *parcels*, visto que tipicamente não havendo partilha de estado da UI na transição entre rotas, a gestão de estado é mais linear.

Já a criação de *utility modules* é indicada para bibliotecas de componentes/guias de estilos e *cross-cutting concerns* (Denning, McMurdie, & Fedosov, 2020).

Apesar de constituir uma arquitetura avançada e inovadora, à semelhança da *app Shell*, a sua utilização não implica ter de começar tudo do zero.

Esta *framework* é já uma realidade em empresas como a Scania, a OpenMRS (um sistema de registos médicos OSS) e até em tecnológicas da nossa praça como o Glintt (A. Mota, comunicação pessoal, 1 de Julho de 2020).

5.2.3.1 Configuração da Single-SPA

A *root config* da single-spa engloba um ficheiro HTML, que é partilhado por todas as *applications* do single-spa, e um ficheiro JavaScript no qual se começa por registar as várias *applications*, através da função *registerApplication* para de seguida carregá-las, ainda que não as inicialize ou adicione ao DOM, ao invocar o método *start*.

5.2.3.2 Application

Criada a aplicação é necessário proceder ao seu registo para que seja possível utilizá-la.

Tipicamente, o registo é efetuado no ficheiro global de configuração da single-spa usando a função *registerApplication* que recebe o nome, a aplicação, uma rota ou *array* de rotas e um objeto opcional contendo propriedades personalizadas (Denning, 2020d)

```
singleSpa.registerApplication({
  name: 'myApp',
  app: () => import('src/myApp/main2.js'),
  activeWhen: '/myApp2',
  customProps: {
    some: 'value',
  },
});
```

Figura 30 – Registo de *application* no single-SPA (Denning, 2020d)

Uma aplicação registada pode responder a eventos de *routing* e é responsável pela sua inicialização, *mount* e *unmount* no DOM.

A principal diferença entre um SPA convencional e as aplicações registadas no single-spa reside no facto destas aplicações poderem coexistir na SPA que as contém e não possuírem uma página HTML dedicada.

No contexto da biblioteca single-spa a SPA detém um conjunto de aplicações registadas. Estas detêm o seu próprio *client-side routing* e podem ser implementadas recorrendo a *frameworks*, que vão desde o React até ao Dojo (Denning, 2020f).

Durante a execução de uma página da single-spa, as aplicações registadas são carregadas (inicialmente ou *lazy loaded*), inicializadas, adicionadas ao DOM, removidas e mesmo descarregadas.

As aplicações registadas renderizam o seu próprio HTML e gerem o seu estado autonomamente enquanto o seu conteúdo se encontrar no DOM.

Em matéria de configuração, estas aplicações regem-se pelo que está definido no ficheiro global, ainda que aspetos como *timeouts* possam ser redefinidos por aplicação e/ou por *hook* (Denning, 2020d).

5.2.3.3 *Parcel*

Uma parcel é um componente agnóstico de *framework*, pensado para ser partilhado entre *applications* implementadas em *frameworks* distintas.

Já quando um componente é utilizado por *applications* desenvolvidas com a mesma *framework* deve-se privilegiar o uso de componentes implementados com essa *framework* em detrimento de *parcels*, visto serem mais facilmente interoperáveis (Denning & McMurdie, 2020).

5.2.3.4 *Utility Module*

Um *utility module* visa a partilha de lógica comum, desde lógica para a realização de pedidos HTTP até *cross-cutting concerns* como a autenticação e autorização (Denning, McMurdie, & Hawkins, 2020)

Um *utility module* permite implementar e exportar lógica que pode ser posteriormente importada pelas *applications*.

5.2.3.5 Configuração Recomendada

Em termos de configuração, a Canopy aconselha que se usem ES Modules e *import maps* com vista a agilizar a gestão de bibliotecas partilhadas, a partilha de código e o *lazy loading* das aplicações, que podem ser desenvolvidas e *deployed* de forma independente.

Relativamente aos módulos, é recomendável que quer as aplicações registadas, quer as dependências partilhadas de grande dimensão (nomeadamente *frameworks* de JS) sejam registadas como módulos JS resolvidos pelo *browser* e tudo o resto como módulos carregados em tempo de compilação (Denning, McMurdie, & Fedosov, 2020).

No que concerne estes últimos, existe uma técnica baseada em Webpack denominada *Module Federation*, destinada à partilha de módulos em tempo de compilação.

É precisamente o Webpack a ferramenta de compilação que é sugerida, por tratar-se de um *standard* de *bundling* de JavaScript (Denning, McMurdie, & Fedosov, 2020).

5.2.3.6 *Module Federation*

A *Module Federation* é uma técnica que permite efetuar o *bundle* das dependências comuns e específicas de cada micro *frontend*, que são descarregadas uma única vez (Herrington, 2020).

Esta técnica permite definir como serão carregados os micro *frontends*, sendo a utilização de *import maps* complementada com SystemJS, uma opção igualmente válida.

Ainda possa ser conjugada com o uso de *import maps*, recomenda-se a adoção de uma única abordagem para gestão de dependências, sendo que a Canopy recomenda o uso de *import maps*, já que o impacto em termos de *performance* é mais reduzido (Denning, McMurdie, & Fedosov, 2020).

5.2.3.7 SystemJS

O SystemJS é uma espécie de *polyfill* para *import maps* e ES Modules, que permite carregar múltiplos módulos sem a necessidade de efetuar vários pedidos (Denning, McMurdie, & Fedosov, 2020).

Fornece uma API que contempla métodos para visualizar todos os módulos JS disponíveis no *browser*, para devolver o URL de um módulo específico, etc. (Denning, 2020b).

5.2.3.8 *Import Maps*

Os *import maps* são uma especificação do *browser* que permite criar um *bundle* de dependências.

Podem ser internos ou externos, caso o JSON que contém os vários imports seja especificado dentro de um elemento *script* no ficheiro HTML da *root config* (conforme apresentado no código abaixo) ou num ficheiro externo, respetivamente (Denning, 2019a).

```
<script type="systemjs-importmap">
    {
        "imports": {
            "single-spa": "https://cdn.jsdelivr.net/npm/single-
spa/lib/system/single-spa.min.js"
        }
    }
</script>
```

Código 8 – Exemplo de *import map*

5.2.3.9 Deployment

No contexto do single-spa, o *deployment* independente de um determinado micro *frontend*, envolve apenas o *deploy* de um ficheiro JS que contém o código mais recente e a atualização no *import map* do respetivo URL, que passa a apontar para esse ficheiro que, regra geral, está alojado num servidor de CDN (Denning, McMurdie, & Fedosov, 2020).

Para agilizar o desenvolvimento e lançamento de aplicações detentoras de micro *frontends* baseadas no single-spa, a Canopy propõe a utilização do projeto import-map-deployer.

O import-map-deployer é um serviço HTTP de *backend* executado num Docker container de Node, que expõe API's REST para visualizar e modificar os *import maps* dos ambientes especificados na configuração (Denning, 2019c).

Esta ferramenta surgiu com o propósito de assegurar a realização de *deployments* concorrentes de forma segura, isto é, que ocorrendo em paralelo não comprometam a integridade do sistema (Denning, 2020h).

Para tirar partido do import-map-deployer é necessário em primeiro lugar efetuar o *deployment* do Docker container para algum tipo de *registry* e de seguida proceder ao *upload* da versão inicial do ficheiro import-map.json, que não contém quaisquer entradas (Denning, McMurdie, & Fedosov, 2020).

Existindo uma vasta panóplia de *cloud providers*, entre os quais, Azure, AWS e Google Cloud Platform, optou-se por esta última opção, conforme sugerido no tutorial disponibilizado na documentação oficial do single-spa (Denning, 2019c).

5.2.3.10 Single-SPA Layout

O single-spa-layout é um package que pode ser adicionado ao single-spa que atua como um *layout engine*, fornecendo uma API de *routing* que controla o primeiro nível de *routing*, *applications* e componentes.

Facilita a orquestração de *applications*, o carregamento de UIs quando as *applications* são descarregadas, a definição de rotas *default* para *Not Found* e página 404 e futuramente a transição entre rotas.

O *layout engine* é responsável pela geração de configuração de registo e por estar à escuta de eventos de *routing* para garantir que os elementos no DOM sejam dispostos da forma correta antes do *mount* das *applications* do single-spa (Denning, 2020c).

5.2.4 Universal Composition

A composição isomórfica pressupõe que a composição do esqueleto da aplicação ocorra no servidor, delegando a restante integração no *browser* (Geers, 2020a).

Da pesquisa efetuada, a única *framework* de *universal composition* encontrada foi a Ara Framework (Diaz, 2019a; Mezzalira, 2020; Perez, sem data).

5.2.4.1 Ara Framework

É uma *framework* construída sobre o Hypernova da Aibnb que permite a utilização de qualquer biblioteca de JavaScript para desenvolver as *views* do Nova e a renderização universal ou apenas *client-side* dessas *views*, independentemente da tecnologia usada na aplicação que vai consumir as *views* (Diaz, 2019a).

Relativamente à experiência de desenvolvimento, esta *framework* coloca à disposição um CLI através do qual é possível criar serviços, executá-los, etc.

No que concerne o termo Nova, designa a arquitetura de base de integração de micro *frontends*.

Quanto à renderização de componentes pode ocorrer de forma universal ou somente do lado do cliente.

A arquitetura Nova com renderização universal, cujo diagrama pode ser consultado na Figura 58 colocada nos anexos, inclui quatro componentes:

- Nova *Bindings* (permite ao hypernova renderizar *views* em JavaScript usando qualquer biblioteca para o desenvolvimento de UIs);

- Nova *Directive* – renderiza um *placeholder* com a informação necessária para permitir ao Nova Proxy efetuar o pedido das *views* dos micro *frontends* e definir em que parte da página deve ser apresentado;

- Nova *Proxy* e *Middleware* – é um servidor de reverse *proxy* que usa diretivas SSI. É responsável efetuar o parse do HTML gerado pela Nova *Directive* para efetuar o pedido das *views* do MFE ao Nova *Cluster* ou diretamente aos MFEs pretendidos;

- Nova Cluster – é um agregador de micro *frontends* que permite aos consumidores das *views* proceder ao pedido das mesmas sem necessitar de saber qual o MFE responsável por cada *view*.

A arquitetura Nova assente em *Client-Side Rendering*, por sua vez, não é mais do que uma aplicação hospedeira (SPA) que contém referências para os *bundles* JavaScript de cada micro *frontend*. Estes *bundles* são apenas carregados caso se aceda à rota correspondente ou sempre que sejam adicionados ao DOM da página (*lazy loading*).

Conforme atesta a Figura 59 contida no Anexo A, SPA usa a Nova Bridge para renderizar um *placeholder* na página após se emitir um evento do tipo “NovaMount”. O ponto de entrada do micro *frontend* (no *browser*) ao ser notificado despoleta a renderização e o “*mount*” da respetiva *view*.

5.2.4.1.1 Hypernova

O Hypernova é um serviço que permite a SSR de *views* desenvolvidas em JavaScript, algo que na perspetiva da Airbnb oferece uma melhor *User* e *Developer Experience* quando comparado à CSR (Perez, sem data).

No que toca à *User Experience*, o conteúdo é disponibilizado mais rapidamente, a página é acessível mesmo quando o JS falha ou está desativado e os motores de pesquisa conseguem indexá-la mais facilmente.

Quanto à *Developer Experience* sai reforçada, dado que o carácter isomórfico dos componentes renderizados, evita duplicar a escrita do HTML no servidor e no cliente.

Para tal, o utilizador começar por efetuar o pedido da página ao servidor.

O servidor reúne toda a informação de que necessita para renderizar a página e usa o cliente Hypernova para submeter um pedido HTTP para o servidor Hypernova.

O servidor Hypernova converte as *views* numa *string* HTML e devolve-a para o cliente (Airbnb, sem data).

O servidor responde à aplicação cliente enviando o HTML e o JavaScript estritamente necessário, sendo os recursos adicionais carregados no *browser*, em conformidade com o *Progressive Enhancement* (Airbnb, sem data).

Em termos arquiteturais, o hypernova é constituído por:

- hypernova/server – serviço que aceita dados através de pedidos HTTP e devolve HTML;
- hypernova – componente universal que converte a *view* produzida pelos *developers* numa estrutura HTML passível de ser renderizada no servidor. No *browser* inicializa a marcação renderizada no servidor e executa-a;
- hypernova-<tipoDeCliente> – permite ao cliente efetuar *queries* ao Hypernova e especificar conteúdo alternativo renderizado no cliente perante uma falha. Têm suporte a clientes implementados usando *frameworks* como Angular e React.

5.3 Análise Comparativa de Abordagens de Integração

Apresentadas as diversas técnicas de integração de MFEs consideradas no âmbito deste documento, com vista a consolidar o conhecimento adquirido, nesta secção será efetuada, através da Tabela 10 e da Figura 31, uma análise comparativa das mesmas tendo em conta indicadores intimamente ligados ao desenvolvimento e comportamento aplicacional.

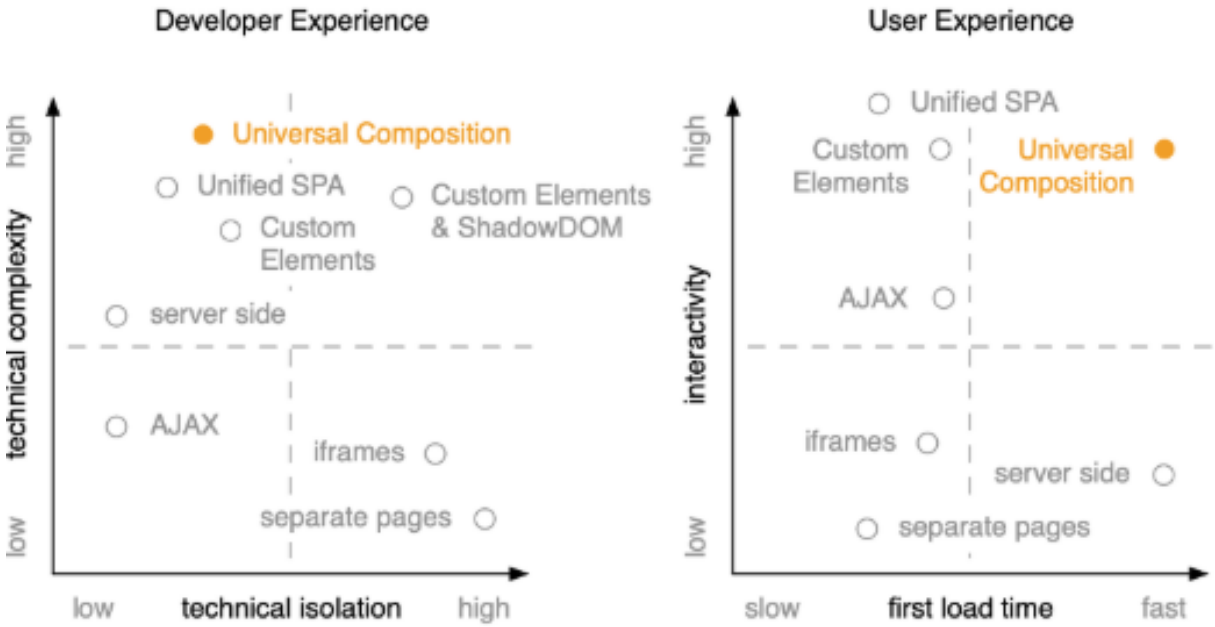


Figura 31 – Comparação das Principais Técnicas de Integração de MFEs (Geers, 2020a)

Com base na análise da tabela da página seguinte e dos gráficos acima pode-se afirmar que:

- em aplicações bastante interativas e reativas, a técnica mais apropriada é a SPA Unificada (seja usando *App Shell* ou *frameworks* como Single-SPA);
- em aplicações em que é dada maior ênfase ao tempo de carregamento, acessibilidade e SEO, deve-se optar pela integração *Server-Side*;
- em aplicações bastante interativas e em que seja necessário garantir um tempo de carregamento inicial baixo e acautelar a possibilidade de falha, demora ou bloqueio de JavaScript, deve ser equacionada a possibilidade de usar *Universal Composition*;
- em aplicações onde quer o tempo de carregamento, quer a interatividade são importantes, a solução pode passar pela integração *Client-Side* complementada com pré-renderização no servidor, por exemplo, através da adoção de AJAX com NGINX ou *Web Components* com pré-renderização no servidor.

Tabela 10 - Análise Comparativa das Técnicas de Integração de Micro *Frontends*

TÉCNICA / CARACTERÍSTICA	BUILD TIME	CLIENT-SIDE				SERVER-SIDE					SPA UNIFICADA		UNIVERSAL COMPOSING	
		Links	IFrame	AJAX[5]	WC	SSI	ESI	Tailor	Podium	OC	App Shell	Single-SPA	WC + SSI	Ara Framework
Baixo Acoplamento	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Isolamento	n.a.	X	X		X[6]									
Acessibilidade	n.a.			X	X	X	X	X	X	X	X	X	X	X
SEO	n.a.	X		X	X[7]	X	X	X	X	X	X	X	X	X
Robustez [8]		X	X	X	X	X	X	X	X	X	X	X	X	X
Restrições de *Layout*	n.a.	X	X											
UI Interativa e Reativa				X[9]	X						X	X	X	X
Progressive Enhancement	n.a.	n.a.	n.a.	X		X	X	X	X	X	X	X[10]	X	X
Lazy Loading		n.a.	n.a.	X	X	[11]	[11]	?	?	?	X	X	X	X
Releases Independentes		X	X	X	X	X	X	X	X	X	X	X	X	X

[5] Usando um servidor *web* partilhado.
[6] Usando *Shadow* DOM
[7] Por norma, quando num *Web Component* que opera nativamente no *browser* falha a renderização não é indexado para efeitos de SEO. No entanto, usando as capacidades de *pré-rendering* do React e Angular esta questão é contornada (Jorgé, 2016).
[8] Prende-se com a aplicação não ficar indisponível devido a falhas de carregamento de algum dos MFEs a integrar. Não tem em consideração a indisponibilidade decorrente de problemas na própria aplicação integradora (quer seja SSI, ESI ou App Shell).
[9] Em termos de interatividade o AJAX está situado entre *Server-Side Rendering* e *Client-Side Rendering*
[10] Atualmente não tem suporte a SSR mas a Canopy prevê que tal venha a acontecer brevemente.
[11] É apenas possível se se complementar a adoção de ESI/SSI com AJAX

5.4 Mecanismos de Comunicação

Apesar de em teoria os micro *frontends* não deverem precisar de comunicar dado o seu carácter autocontido e autossuficiente, é frequente existir a necessidade de aceder a dados provenientes de outros micro *frontends* ou transversais à aplicação, nomeadamente relacionados com *cross-cutting concerns* como a Autenticação, como o *token*.

Para tal, é imperioso recorrer a mecanismos de comunicação, que detetando alterações sobre os dados ou *inputs* do utilizador na UI, notifiquem sempre que possível os micro *frontends* que subscreveram estas ações.

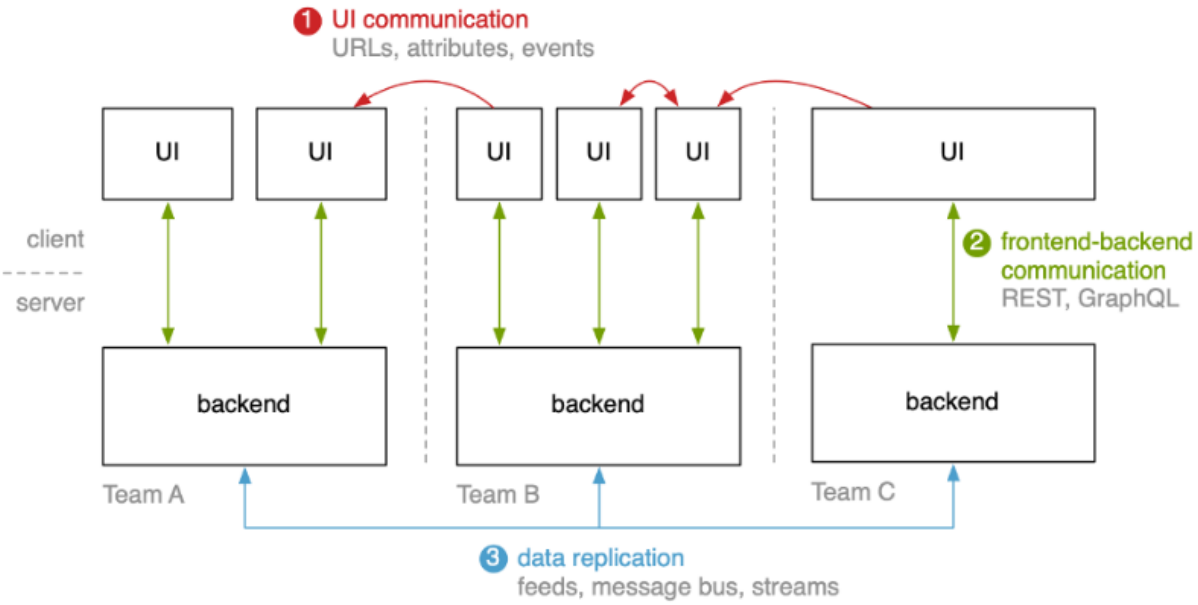


Figura 32 - Mecanismos de Comunicação numa Arquitetura de Micro *Frontends* (Geers, 2020a)

Como ilustrado na figura, a comunicação pode estabelecer-se entre UIs, *frontend* e *backend* ou entre diferentes serviços de *backend*.

5.4.1 Comunicação entre UIs

Ao nível do *frontend*, quando existem vários micro *frontends* na mesma página gerir a User Interface e a comunicação entre si sem que se apercebam da existência dos demais é uma tarefa árdua, mas imprescindível para manter baixo acoplamento e salvaguardar a capacidade de evolução independente (Geers, 2020a).

O grau de dificuldade na comunicação entre UIs vai depender da complexidade da própria aplicação. Pode ser estabelecida através do simples envio de informação via URL ou alternativamente mobilizar mecanismos mais avançados.

Quando as aplicações concentram múltiplos casos de uso na mesma página, um link deixa de ser suficiente e é tempo de equacionar a aplicação de um ou mais desses mecanismos (Geers, 2020a).

Num cenário em que uma página agregue vários fragmentos, a comunicação pode ocorrer nos seguintes sentidos: página para fragmento, fragmento para página e entre fragmentos.

Com base numa aplicação *e-commerce* similar à da POC, nas subsecções seguintes serão apresentados casos de uso que servirão de mote à análise de cada um destes tipos de comunicação entre componentes visuais.

5.4.1.1 Página para fragmento (*Parent-Child*)

Suponha-se que na página de detalhe de um produto (desenvolvida pela equipa de catálogo de produto) existe um botão através do qual é possível adicioná-lo ao carrinho de compras (mantido pela equipa de carrinho de compras).

Necessitando o botão apenas do id do produto, detido pela página, para adicionar o item ao carrinho, esta informação pode ser passada da página para o fragmento por via de um atributo criado para o efeito neste componente visual.

Para propagar alterações de estado de uma página para um fragmento, *frameworks* como React ou Angular implementam o padrão "*One-Way Binding*", despoletando a re-renderização perante alterações aos atributos ou *props*.

Já seguindo uma abordagem mais purista, na qual se desenvolvem *Web Components* usando Vanilla JS, é necessário do lado do componente especificar os atributos a observar e dentro do método *attributeChangedCallback* executar a função responsável pela renderização usando os valores atuais dos atributos (Geers, 2020a).

5.4.1.2 Fragmento para página (*Child-Parent*)

Com base no exemplo da secção anterior, suponha que ao adicionar um produto ao carrinho se pretende apresentar uma indicação visual na imagem principal do produto. Ora, não devendo a equipa de carrinho efetuar alterações que estravassem as suas competências, sob pena de criar acoplamento, a página deverá ser notificada aquando da adição bem-sucedida do item ao carrinho para poder apresentar uma indicação visual de sucesso (desenvolvida pela equipa de catálogo de produtos).

A comunicação entre fragmento e página pode ser obtida usando a API de *Custom Events*, suportada pelos principais *browsers* (MDN contributors, 2019c).

Assim, para que a página seja notificada o fragmento deve instanciar um *Custom Event* com o tipo acordado entre equipa de catálogo de produto e equipa de carrinho e efetuar o seu *dispatch* invocando o método *dispatchEvent*.

```
this.querySelector("button").addEventListener("click", () => {
    ...
    const event = new CustomEvent("checkout:item_added");
    this.dispatchEvent(event);
});
```

Código 9– Instanciação e *dispatch* de um *Custom Event* (Geers, 2020a)

A página, que se encontra à escuta de eventos desse tipo, é notificada e a indicação visual é apresentada.

```
buyButton.addEventListener("checkout:item_added", e => {
  product.classList.add("decide_product--confirm");
});
```

Código 10 – *Listener* de um *Custom Event* (Geers, 2020)

O uso de *Custom Events* permite criar eventos com nomes que refletem a linguagem de domínio, sem que os fragmentos precisem de conhecer as páginas que os contêm, têm amplo suporte e são facilmente submetidos a *debug*.

5.4.1.3 Fragmento para fragmento

Com base no exemplo da secção anterior, suponha que na página existe um *widget* que apresenta os itens atualmente no carrinho, notificado e atualizado sempre que o utilizador clicar no botão para adicionar um produto ao carrinho.

O *widget* de carrinho não recebe nenhum atributo nem emite quaisquer eventos e assim que é adicionado ao DOM, renderiza a lista obtida a partir de uma chamada ao *backend*.

O botão de adicionar pode notificar o *widget* de carrinho por via de comunicação direta, orquestração através do *parent* ou event-bus/*broadcasting*.

No *browser*, qualquer fragmento tem acesso à árvore de DOM completa, na qual podem procurar pelo fragmento com quem pretendem comunicar e dessa forma estabelecer a comunicação. No entanto, deve descartar-se a referenciação direta, já que introduz demasiado acoplamento.

Quanto à orquestração, é efetuada pela página que contém os fragmentos. Neste caso, a página de produto atuaria como intermediária, ficando à escuta do evento despoletado pelo clique no botão adicionar e acionando o evento de atualização do carrinho. Esta é uma solução limpa, contudo quaisquer alterações na comunicação requerem que as duas equipas adaptem o seu *software*.

No que concerne o uso de event-bus ou da *Broadcast Channel API*, introduzem um canal global de comunicação.

Os fragmentos podem publicar eventos para este canal, que podem ser subscritos por outros fragmentos. Este mecanismo, assente no padrão *Publish/Subscribe* e passível de ser implementado usando *Custom Events*, reduz o acoplamento.

Relativamente aos *Custom Events*, permitem além de especificar o tipo de evento, enviar um *payload* personalizado através da *key* "*detail*".

Além do envio do *payload*, é possível colocar o valor da key "*bubbles*" a true (por defeito, a false) para que o evento seja disparado primeiro no elemento ao qual está atrelado e depois se propague e seja despoletado nos ancestrais desde o mais próximo até ao mais distante até chegar à *window* (*Bubbling and Capturing*, 2020; MDN contributors, 2019c; M. Silva, 2014). A principal diferença de invocar *element.dispatchEvent* em detrimento de *window.dispatchEvent* consiste na manutenção da origem do evento e a possibilidade de intercetá-lo e parar a propagação (Geers, 2020a).

Quanto à *Broadcast Channel API*, cria um *message bus* que atravessa *iframes*, separadores e até janelas, no qual quaisquer fragmentos podem receber as mensagens enviadas para esse canal (*Broadcast Channel API*, sem data).

Usando *iframes* como técnica de integração de micro *frontends*, como cada iframe possui o seu próprio contexto e, por conseguinte, o seu próprio objeto window, não é possível usar *Custom Events* (Mezzalira, 2020). Neste contexto, uma opção pode passar por utilizar o método *postMessage*, um mecanismo idealizado para a comunicação segura entre documentos ou componentes existentes em diferentes domínios (MDN contributors, 2019a).

A função *postMessage*, parte integrante da *Broadcast Channel* API, cria e faz o *dispatch* de um evento do tipo "message enviando a informação especificada no primeiro parâmetro, destinado ao domínio de destino passado no segundo parâmetro (*Iframes and the postMessage Method*, sem data).

5.4.1.4 Conclusões

A comunicação entre UIs deve cingir-se ao estritamente necessário e baseada em standards como *Custom Events*, *Broadcast Channel* API ou numa implementação customizada do padrão *Publish/Subscribe*, que pode ser colocada num módulo partilhado e importado pelas equipas em tempo de execução.

Em relação aos *payloads* passados através dos eventos, devem manter-se tão simples quanto possível enquanto *view-models* e objetos de domínio não devem cruzar as fronteiras da equipa.

Relativamente ao uso de eventos ou *broadcasting*, deve ter-se em conta que os MFEs não conseguem aceder à informação enviada até que concluam a sua inicialização (Geers, 2020a).

5.4.2 Informação de Contexto e Autenticação

5.4.2.1 Passagem de informação de contexto para os micro *frontends*

Cada micro *frontend* dá resposta às necessidades específicas do seu domínio. Contudo, numa aplicação com alguma complexidade, estes requerem alguma informação de contexto, na qual se inclui o idioma, moeda, país, estado de autenticação (utilizador autenticado ou não), ambiente de execução, etc.

A forma como a informação de contexto é fornecida aos micro *frontends* dependerá de como obtêm o acesso à informação e de quem é responsável pela informação de contexto.

Quando a renderização ocorre do lado do servidor, tipicamente a informação de contexto é obtida de *headers* HTTP ou de cookies, configuráveis através de um proxy ou serviço de composição.

Já numa aplicação predominantemente *client-side*, uma opção passa por fornecer uma API global de JavaScript através da qual cada equipa pode aceder à informação. Se a orquestração dos micro *frontends* for efetuada por via de uma *Application Shell*, é comum que integre o tratamento da informação de contexto.

Em termos de responsabilidades, poderá estar a cargo de uma equipa de plataformas dedicada (se existir) ou num contexto distribuído, nas mãos de uma das equipas com MFEs a seu cargo ou existindo uma *App Shell* com uma equipa dedicada caberá a essa equipa (Geers, 2020a).

5.4.2.2 Autenticação

A autenticação é uma *cross-cutting concern* cujo desenvolvimento e manutenção pode estar nas mãos de uma equipa dedicada ou de uma equipa de Produto. A equipa responsável será doravante o fornecedor de autenticação para todas as outras equipas, disponibilizando a página de login e o fragmento que permite redirecionar um utilizador não autenticado. A autenticação em si deve basear-se em standards como OAuth ou JWT (Geers, 2020a).

Após a autenticação ser bem-sucedida é necessário armazenar numa *web storage* (*session* ou *local storage*) ou em cookies o *token*, que se torna acessível a todos os micro *frontends* presentes no mesmo domínio. O uso de *query string* também é possível, mas é pouco seguro, particularmente na transmissão de dados sensíveis (Mezzalira, 2020).

5.4.3 Gestão de Estado

Tipicamente, a gestão de estado deve estar a cargo dos componentes. No entanto, usando bibliotecas de gestão de estado como Redux, cada MFE pode possuir o seu próprio estado local.

Quanto à reutilização de estado entre MFEs, deve ser evitada sob pena de criar demasiado acoplamento e dificultar eventuais alterações (Geers, 2020a).

5.4.4 Comunicação entre *Frontend* e *Backend*

Cada micro *frontend* deve limitar-se a comunicar com os serviços de *backend* desenvolvidos pela sua equipa e em nenhuma circunstância com *endpoints* mantidos por outras equipas para não criar acoplamento nem dependência entre equipas.

No entanto, num cenário em que existam equipas separadas a trabalhar de forma independente no *frontend* e no *backend* (apesar de a arquitetura de micro *frontends* pressupor o desenvolvimento E2E das aplicações pelas equipas), seguindo respetivamente uma arquitetura de micro *frontends* e de micro serviços, a comunicação entre estas camadas pode ser estabelecida com recurso ao padrão *Backend for Frontend* (BFF) (C. Jackson, 2019) ou então através da criação de uma camada intermédia e/ou de integração (*middleware)*.

Apesar de o propósito original do BFF consistir na criação de *backends* à medida das necessidades de cada canal de acesso ao *frontend* (*web*, *mobile*, etc.), o padrão pode ser estendido para que também possa significar um *backend* para cada micro *frontend*.

Estes BFFs podem deter a sua própria lógica de negócio e base de dados ou funcionarem como agregadores de *downstream services*.

5.4.5 Comunicação entre Serviços de *Backend*

Num cenário em que uma equipa necessite de aceder a informação armazenada na base de dados de outra equipa, apesar de poder obter essa informação por via de chamadas direta à API da equipa detentora da base de dados, essa ação violaria os pressupostos de autonomia da arquitetura de micro *frontends*, dado que criaria um elevado acoplamento e falhando a API, os micro *frontends* de ambas as equipas ficariam indisponíveis (Geers, 2020a).

Contudo, esta situação é contornável apostando na replicação de dados ou na composição de UIs.

No que concerne a replicação de dados, a equipa detentora da informação disponibiliza uma interface que pode ser usada para replicar periodicamente em background a informação necessária, através de um mecanismo de *feed*. Nesse caso, se a aplicação da equipa detentora dos dados falhar, os restantes micro *frontends* continuam a funcionar, uma vez que detêm uma cópia da informação nas suas bases de dados.

Quanto à composição de UIs, consiste na disponibilização por parte da equipa responsável pelos dados de micro *frontends* facilmente integráveis, nos quais é apresentada a informação diretamente para o utilizador (Geers, 2020a).

5.5 Organizações que Adotaram Micro *Frontends*

Apesar de o conceito ser relativamente recente, os micro *frontends* fazem parte da realidade de um vasto leque de organizações, ainda que na maioria dos casos não seja utilizada esta nomenclatura para designar a abordagem de desenvolvimento de *software* adotada.

5.5.1 HelloFresh

A HelloFresh é um serviço digital que vende caixas que contêm os ingredientes necessários para confecionar uma determinada receita culinária.

A organização define micro *frontend* como um "um micro serviço de *frontend* que está à escuta de pedidos num ou mais *endpoints*, aos quais responde enviando uma página HTML juntamente com os links para os recursos existentes no seu *scope*", ressalvando que este não deve possuir dependências externas críticas com vista a poder ser integrado de forma agnóstica de tecnologia (Senders, 2017).

5.5.1.1 Abordagem Preconizada

A arquitetura proposta é bastante inspirada no caminho traçado pela Zalando, englobando os seguintes conceitos:

- Fragmento *(florp):* consiste num serviço que se encontra em execução num ou mais *endpoints* disponibilizados pelo servidor de entrada, nos quais fornece um rol de SPAs orquestradas por URL;
- Partícula *(glorp): partículas* são aplicações que sendo transversais a toda a plataforma necessitam de ser carregadas sincronamente e renderizadas no servidor. Ex: *header* e *footer*.
- *Tag*: pequenas aplicações que são despoletadas assincronamente na página, por exemplo na sequência de um clique, que não estando necessariamente vinculadas a uma rota, podem ser carregadas de forma diferida. Dessa forma, não impacta no carregamento da página. Ex: modal.

O fluxo de execução pode ser explicado da seguinte forma: quando um cliente envia um pedido para obter uma das páginas disponibilizadas, caso não se encontre armazenada em cache ao nível da CDN uma cópia da mesma, a CDN vai direcionar o pedido para o servidor de entrada que depois faz o encaminhamento para o devido fragmento, que renderiza e devolve uma página HTML após solicitar e integrar as partículas contidas (Senders, 2017).

Esta estratégia permite aceder aos servidores finais apenas para obter conteúdo que não se encontre em cache e confere às equipas de desenvolvimento autonomia e autoridade para definir as respetivas *stacks* tecnológicas.

5.5.1.2 Impactos da adoção

No que concerne os impactos da abordagem preconizada, a HelloFresh reconhece que devido à curva de aprendizagem inerente ao desenvolvimento E2E de um projeto, o ritmo de desenvolvimento inicial foi mais lento, mas que acelerou à medida que a confiança e conhecimento dos *developers* denotavam um aumento significativo (Senders, 2017).

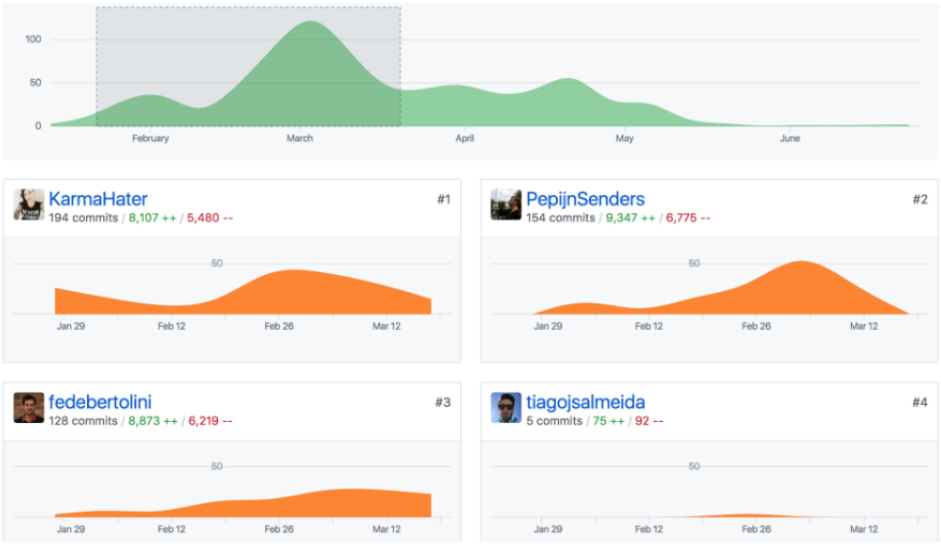


Figura 33 - Ritmo de desenvolvimento na HelloFresh no 1º trimestre de 2017 (Senders, 2017)

Além disso, a possibilidade de adoção de diferentes tecnologias entre projetos sem que isso comprometa o normal funcionamento do *website* (que correspondia a um dos objetivos traçados inicialmente) foi assegurada, as responsabilidades passaram a ser delimitadas de forma mais clara, a plataforma tornou-se mais modular e mais facilmente mantida e a deteção e correção de erros e inconsistências foi agilizada (Senders, 2017).

5.5.2 AllegroTech

A AllegroTech é uma empresa polaca detentora de uma plataforma de comércio eletrónico e de leilões que, de acordo com a Alexa, empresa norte-americana dedicada à análise de tráfego *web*, ocupa atualmente no seu ranking, que mede o tráfego e grau de *engagement*, a 312ª posição a nível global e a 4ª na Polónia. (Alexa, 2020b).

Perante a complexidade do seu domínio de negócio, o elevado número de acessos ao *site* e as dificuldades e ineficiência em termos de escalabilidade, a Allegro decidiu migrar de uma solução puramente monolítica (desenvolvida em PHP) para uma arquitetura de *Frontend* assente num *Backend* orientado a micro serviços (mais propício à escalabilidade), tendo chegado a duas abordagens: o *Frontend* Monolítico e Abordagem “Frankenstein”.

Quanto ao *Frontend* Monolítico, apesar de fornecer uma UX unificada, fica aquém em termos de escalabilidade e num contexto em que existam várias equipas a trabalhar sobre a mesma *codebase* pode impactar o desenvolvimento.

Já a abordagem Frankenstein, uma designação da autoria da Allegro, advoga a divisão da aplicação em vários módulos, desenvolvidos E2E por equipas independentes e sem qualquer tipo de partilha (Gałek et al., 2016).

Estes módulos são constituídos não só por lógica de *backend*, mas também por fragmentos de página HTML (*header*, carrinho, barra de pesquisa, etc.), disponibilizados através de *endpoints* como parte do seu *frontend*, que são depois integrados nas páginas usando ESI.

Apesar desta abordagem escalar sem problemas, pode resultar em inconsistência entre as interfaces gráficas, daí a designação Frankenstein.

Ora, havendo todo um espetro de possibilidades entre o Frankenstein e o Monolítico, o *site* da Allegro atualmente segue uma abordagem mais aproximada à Frankenstein e encontra-se a trabalhar numa nova solução, denominada OpBox, que assume um carácter mais próximo do monolítico.

No que concerne a abordagem atual, a integração entre módulos é efetuada no servidor, por transclusão, mais especificamente usando *tags* ESI que são depois processadas pelo Varnish.

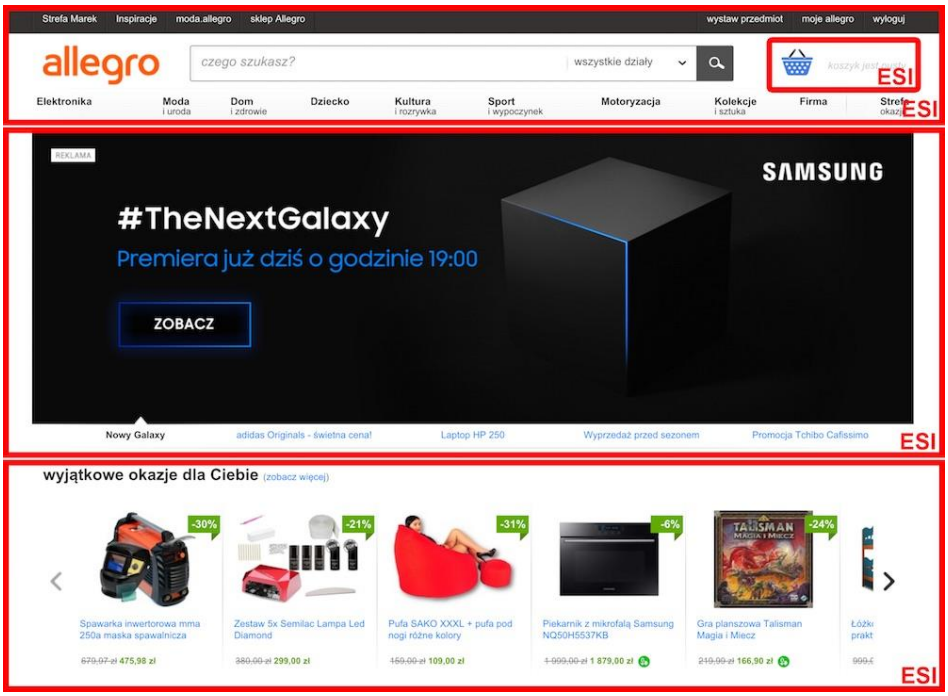


Figura 34 - Identificação dos fragmentos que constituem o *site* da Allegro (Gałek et al., 2016)

Se, por um lado, o Varnish permite melhorar a *performance* geral, já que os seus servidores se encontram expostos aos utilizadores e fazem cache de todos os pedidos de conteúdo estático, por outro, dado que seguindo esta abordagem cada fragmento possui o seu conjunto de

recursos verifica-se bastante duplicação e conflitos de versões, o que a torna difícil de manter e dificulta o esforço de consistência de UI.

Ciente destas adversidades, a Allegro tem concentrado esforços no desenvolvimento do projeto OpBox, alicerçado no conceito de *box*.

Uma *box* consiste num componente reutilizável, cujos dados podem ser obtidos a partir de REST/JSON, que pode ser renderizado condicionalmente. Uma box pode incluir outras boxes, sendo a página final montada a partir de um conjunto de boxes.

Ainda no âmbito deste projeto, encontram-se em fase de desenvolvimento a criação de um *Event Bus* para possibilitar a interação entre Boxes, bem como de um mecanismo de gestão de dependências e de recursos.

5.5.3 Spotify

A Spotify usa micro *frontends* na sua aplicação desktop, recorrendo a *iframes* para integrar componentes mantidos por diferentes equipas na mesma *view*.

A comunicação entre *iframes* é efetuada através de um *event bus*, o que permite dissociar de tal forma as diferentes partes da aplicação que conseguem comunicar sem saber quem está à escuta de uma determinada mensagem ou evento (Mezzalira, 2019a).

Ao inspecionar os ficheiros da aplicação *desktop*, constatamos que existe uma pasta denominada "Apps", na qual estão contidos ficheiros que possuindo a extensão .spa, facilmente se infere tratarem-se de artefactos que correspondem a *Single Page Applications*.

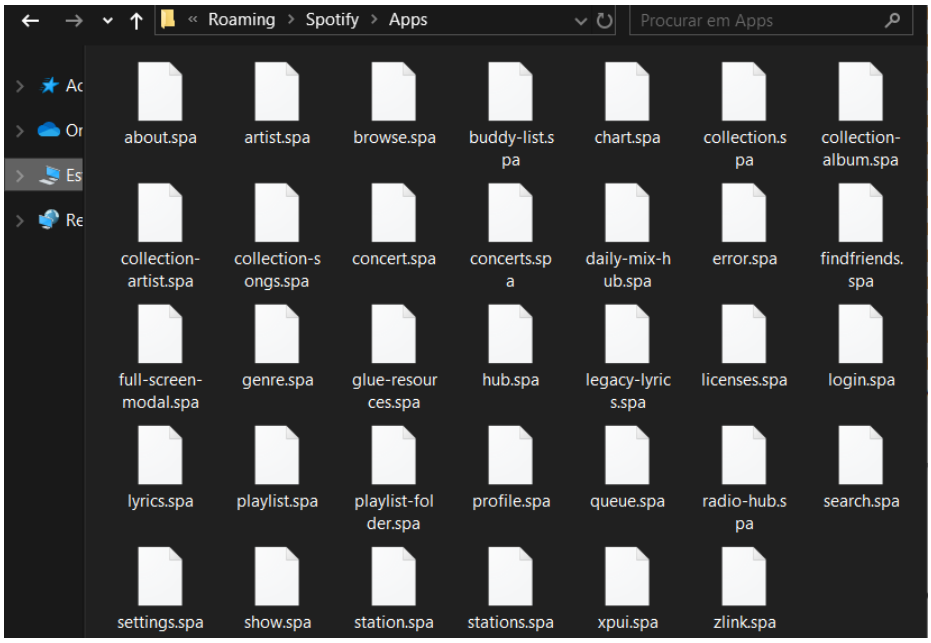


Figura 35 – Pasta contendo os micro *frontends* da aplicação *desktop* do Spotify

Ao descompactar um destes artefactos, a própria estrutura evidencia tratar-se de uma aplicação *web*, visto ser constituído por um ficheiro HTML, múltiplos ficheiros CSS, o manifest.json e um *bundle* JavaScript minificado.

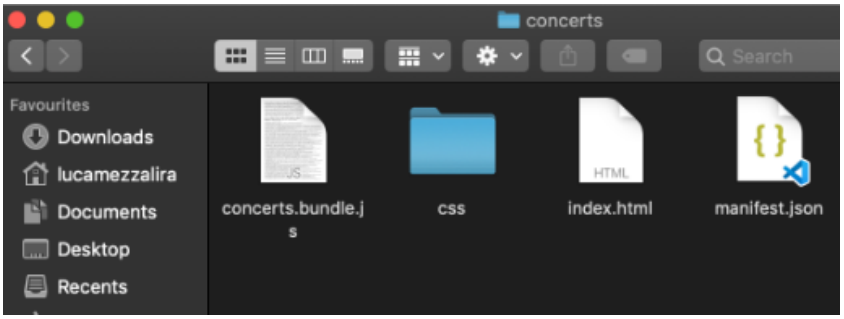


Figura 36 – Conteúdo de um artefacto de micro *frontend* do Spotify (Mezzalira, 2020)

Todos estes ficheiros eram depois carregados dentro das *iframes* para compor a UI final da aplicação. Desta forma, diferentes equipas podiam desenvolver iterativamente e efetuar *releases* de forma isolada.

O *web player* do Spotify foi lançado em 2012 com o objetivo de complementar a experiência em equipamentos desktop, seguindo a mesma abordagem arquitetural da aplicação desktop. Além disso, sendo o próprio código bastante similar, as equipas podiam produzir uma nova funcionalidade e disponibilizá-la facilmente na aplicação e no *web player* (Pérez, 2019).

Contudo, começou a tornar-se evidente que a *performance* do *web player* ficava muito aquém do desejável, sobretudo o tempo de carregamento. A razão prendia-se com a necessidade de transferir múltiplos recursos sempre que o utilizador navegava entre páginas, o que irremediavelmente deteriorava a *User Experience*. Por outro lado, esta arquitetura de vistas isoladas dificultava a manutenção (Pérez, 2019).

Regra geral, a Spotify opta por tentar melhorar os sistemas existentes de forma iterativa ao invés de os substituir por novos. Contudo, com o progressivo aumento de exigências dos utilizadores face às aplicações *web*, estas limitações agudizaram-se e o desfecho era inevitável.

Assim, em 2016 após investigar a viabilidade de atualizar o *web player*, *view* a *view*, e de paralelamente desenvolver um novo protótipo seguindo uma arquitetura similar à da aplicação para *Smart* TV, a organização deliberou descontinuar esta abordagem baseada em *iframes* e desenvolver um novo *web player* de raiz. Para a decisão contribuíram aspetos como a complexidade inerente ao sistema responsável por fazer chegar o código às *views*, o facto do *web player* se basear em bibliotecas obsoletas e a demora na execução dos testes às alterações (Pérez, 2019).

A nova aplicação segue uma arquitetura de SPA e usa a *Web* API do Spotify, que combina o acesso a vários micro serviços para criar uma interface unificada na qual se manipulam os dados do Spotify.

Para não incorrer nos mesmos erros do passado e decidir quais as funcionalidades a contemplar no novo *web player*, o *web player* existente foi submetido a testes A/B. Foram removidas certas funcionalidades para alguns utilizadores e medido o impacto na *customer engagement*, com o intuito de identificar as funcionalidades que sendo mais procuradas e apreciadas vão traduzir-se numa maior criação de valor e cuja implementação deve, portanto, ser priorizada (Pérez, 2019).

Com base nas ilações retiradas, ao longo dos meses seguintes foi desenvolvido de forma iterativa e incremental um MVP, submetido a testes de usabilidade e afinado com base no *feedback* obtido.

Assim que a equipa entendeu que o novo *web player* estava preparado lançou uma *canary release* e efetuou uma análise comparativa da *performance* com o *web player* existente, constatando que em todas as métricas o novo *web player* superou o antigo.

Em termos de *stack* tecnológica, o novo *player* foi implementado usando React e Redux, o que facilitou a partilha de componentes entre *views*, a testabilidade e *debugging*. Apesar dos componentes não poderem ser reaproveitados para outros clientes Spotify, a adoção de tecnologias *open-source* com amplo suporte da comunidade em detrimento das bibliotecas do Spotify potenciou não só o lançamento de novas funcionalidades, como também a integração de novos colaboradores e o número de contribuições por parte dos *web developers* da organização (Pérez, 2019).

Além disso, a conversão tecnológica do *web player* fez com que a empresa promovesse o desenvolvimento de *pipelines* de CI/CD, de modo a agilizar o lançamento de novas versões, e usufruísse das potencialidades oferecidas por diversas tecnologias.

Em termos de cultura e metodologia de trabalho, a Spotify é completamente Agile, tendo, no entanto, adotado uma espécie de SCRUM adaptado, no qual os convencionais papéis, artefactos e eventos são opcionais, uma vez que se "valorizam mais princípios do que práticas e processos específicos". Daí, que na Spotify as designações de *Scrum Master* e *Scrum Team* tenham sido substituídas por Agile Coach e Squad, respetivamente (Fernandes, 2017a).

Uma *squad* é uma equipa multidisciplinar, constituída, regra geral, por menos de 8 elementos, responsável pelo desenvolvimento E2E e que cooperam para atingir os objetivos de curto prazo, mantendo-se sempre alinhados com a sua missão e estratégia de produto.

As *squads* têm autonomia para decidir o que desenvolver, como desenvolver e como vão trabalhar em conjunto enquanto desenvolvem (Fernandes, 2017a).

Para a organização "a autonomia aliada ao alinhamento (com os princípios da organização), potencia a motivação, a qualidade e permite *releases* mais rápidas".

Além disso, a Spotify privilegia a polinização cruzada, isto é, uma estandardização que emana da perceção de que uma ferramenta utilizada e testada por múltiplas *squads* pode ser utilizada por outras *squads*, face a uma estandardização baseada em diretrizes formais. Em termos de

metodologias, existe revisão de código entre pares, o que promove a colaboração entre colaboradores e a troca de conhecimento.

Alicerçada na premissa "*Think It, Build It, Ship It, Tweak It*!", o foco da organização incide sobre a evolução das *squads* e dos produtos desenvolvidos (Fernandes, 2017b).

Relativamente à estrutura organizacional, dado que a Spotify possui um conjunto alargado de *squads*, estas são agrupadas em *tribes*, que consistem num conjunto de *squads* focadas na entrega de um produto com qualidade e geridas pelo mesmo manager. As *tribes*, por sua vez, englobam grupos de colaboradores que assumem as mesmas competências, denominados *chapters*, por exemplo, *quality assurance*, *web development*, etc. e *guilds* (Fernandes, 2017b), que correspondem a CoPs.

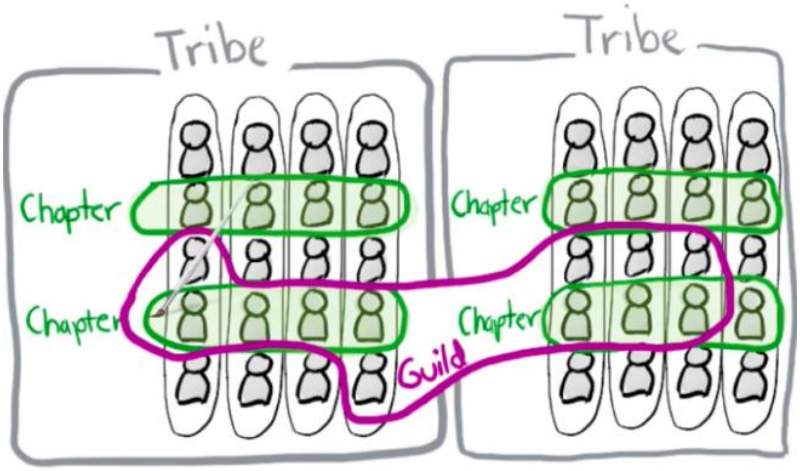


Figura 37 – Modelo organizacional da Spotify (Kniberg, 2014)

A Spotify denota que este modelo organizacional cria um sentimento de coletividade e pertença nos colaboradores (Kniberg, 2014).

No que concerne o processo de *release*, com o mote de efetuar *releases* contidas cedo e com frequência e investir na automação da infraestrutura para possibilitar a *Continuous Delivery* e assim agilizar o processo, a Spotify criou condições para a realização de *releases* desacopladas, ao dividir o *website* em múltiplas secções, para as quais cada *squad* pode efetuar *releases* de forma autónoma.

Estas *releases* são enviadas periodicamente, podendo conter além de funcionalidades completas, outras que apesar de ainda não se encontrarem concluídas (e que seguem ocultadas - *Feature Toggles*), podem permitir a exposição de problemas de integração numa fase precoce (Kniberg, 2014).

5.5.4 SAP

A SAP é uma empresa de *software* germânica, cujo produto principal, o SAP ERP, é líder de mercado à escala global em soluções integradas de gestão empresarial (Pang, 2019).

Dada a extensão e abrangência deste *software*, contempla diversos módulos relativos a áreas de negócio específicas, desde Vendas e Distribuição, Recursos Humanos, Gestão Financeira até Planeamento de Produção, entre outros (Verma, 2020).

A organização encontra-se há praticamente cinco décadas em atividade e dada a competitividade feroz que se regista sobretudo no segmento dos ERP e CRM, no qual estão presentes organizações de renome como a Microsoft e a Oracle, o SAP tem procurado reinventar-se.

Tendo a SAP traçado como meta para esta década tornar-se líder na computação em *cloud* e em redes de negócio de comércio eletrónico, um desígnio que tem procurado alcançar não só por via da aquisição e incorporação de outras organizações (SAP, 2011), e por extensão do *know-how* trazido para dentro de portas, mas também através da investigação e experimentação encetada internamente.

É precisamente neste âmbito, que a organização lançou o Luigi Project, uma *framework* de micro *frontends* que possibilita a criação de uma interface gráfica unificada numa aplicação *Web* distribuída e o desenvolvimento de micro *frontends* e a sua integração e comunicação com a aplicação (SAP, sem data-a).

Enquanto parte integrante do projeto Luigi, Philipp Pracht, arquiteto e *Product Owner* na SAP, numa entrevista concedida a Luca Mezzalira, realizada no âmbito da redação da obra *"Building Micro-Frontends"*, narra como tem sido a experiência.
Destaca como principal benefício o impulso da eficiência para qualquer projeto de desenvolvimento de UIs em larga escala por parte de várias equipas. Conta que depois que implementaram a arquitetura de micro *frontends* (não designada dessa forma) pediram às equipas de micro serviços para assumirem a responsabilidade pelas UIs correspondentes e surpreendentemente tudo correu bem, dado que, após uma breve introdução técnica às técnicas necessárias para a implementação, todas as equipas foram capazes de desenvolver e efetuar *releases* de forma independente (Mezzalira, 2020).

No que concerne as dificuldades considera que não se prenderam com questões técnicas, mas sim com infindáveis e pouco produtivas discussões, motivadas pela resistência em aderir aos princípios de micro *frontends* ou em simplesmente aceitar a autonomia e independência das equipas.

Além disso, alerta para o facto de que somente com uma estrutura organizacional ajustada, que possibilite que equipas dedicadas possam inteirar-se do desenvolvimento E2E e lançamento de funcionalidades, é possível adotar com sucesso esta arquitetura (Mezzalira, 2020).

5.5.5 Airbnb

Em 2017, a plataforma de aluguer de alojamentos locais registava mais de 75 milhões de pesquisas diárias, o que tornava a página de pesquisa a mais visitada de todo o *site*.

Na sequência da introdução de funcionalidades como Experiências e Lugares, a Airbnb aproveitou para repensar a experiência de pesquisa com o intuito de a tornar mais fluída, intuitiva e ajustada ao tipo de utilizador. Para tal, era imperioso acabar com a abordagem página-à-página e reestruturar o *frontend*, dado que se verificava um acoplamento tão elevado que qualquer modificação poderia despoletar comportamentos inesperados (Neary, 2017).

Relativamente ao processo de migração, envolveu duas fases: a migração dos dados passíveis de ser manipulados por via de APIs (informação específica do negócio e do utilizador) e posteriormente dos dados transversais a toda a aplicação que não se enquadram na condição de recursos disponibilizados por uma API, nomeadamente configurações, ferramentas de internacionalização e de localização, etc.

Quanto à migração dos pontos de entrada de dados, Neary admite que a maior dificuldade não residiu na sua conversão em *endpoints* de API, mas em reeducar todas as equipas que interagem com o fluxo de reservas de hóspedes para passarem a enviar os dados para os componentes renderizados do lado do servidor através da API.

No que concerne a informação que pela sua natureza não pode ser manipulada através de APIs, a Airbnb desenvolveu ao longo de tempo ferramentas para as suportar, no entanto, os mecanismos para as entregar ao *frontend* não eram muito fiáveis.

Havia ainda o receio de que a utilização de React no servidor impactasse negativamente a *performance* e de que as traduções disponibilizadas no cliente não fossem disponibilizadas devidamente no servidor.

Além disso, existiam na *codebase* chamadas a funções legadas que apesar de inicialmente serem do conhecimento de toda a equipa de desenvolvimento e permitirem dar resposta a questões transversais sem replicação de código, progressivamente tornaram-se complexas e a sua lógica interna do desconhecimento de parte substancial da equipa que, entretanto, se expandiu.

Esta expansão e a inevitável segregação em múltiplas equipas de desenvolvimento fez com que cada equipa possuísse os seus próprios mecanismos de carregamento de configuração, desenvolvidos em função das suas necessidades específicas.

5.5.5.1 Abordagem

Neste panorama de estagnação da sua plataforma, a Airbnb decidiu apostar na convergência, desenvolvendo um mecanismo comum de inicialização de dados e começando a migrar gradualmente as aplicações/páginas desenvolvidas em Rails para uma nova solução implementada usando React e Hypernova.

Este mecanismo consiste no uso de um "Higher-Order Component (HOC)", isto é, uma função que recebendo um componente o transforma noutro componente, introduzindo-lhe lógica adicional (React, sem data-a). Os HOCS recebem um objeto que contém os dados para a inicialização das ferramentas de suporte necessárias à renderização quer do lado do servidor, quer do lado do cliente. Desta forma, o uso de HOCs permite a reutilização de lógica, e neste caso centralizar a lógica de inicialização.

Do lado do *frontend*, ao invés de carregar toda a *Single Page Application* de uma só vez, a abordagem da Airbnb promove o *code-splitting* e o *lazy-loading*,

Relativamente à abordagem em si, consiste em renderizar, usando o Hypernova, os componentes escritos em React no servidor, fornecendo o mínimo de JavaScript necessário para a tornar interativa no *browser* e procedendo ao download dos restantes recursos assim que o *browser* esteja inativo (Neary, 2017).

Do lado do Rails, existe um controlador para todas as rotas disponibilizadas através da SPA e cada ação é responsável por efetuar qualquer pedido à API despoletado aquando da navegação entre rotas no cliente, enviando depois os dados obtidos juntamente com a configuração para o Hypernova, onde esta será inicializada, o que permitiu reduzir drasticamente o tamanho da *codebase*, bem como obter transições mais suaves e rápidas.

Se antes do React integrar a sua *stack* tecnológica a Airbnb renderizava uma página inteira de uma vez, atualmente através de um componente assíncrono divide as respetivas secções em *bundles* carregados hierarquicamente depois do "*mount*", isto é, após serem adicionadas ao DOM (React, sem data-b).

Esta solução permitiu não só decrementar o tempo de carregamento, uma vez que a aplicação pode ir carregando o *bundle* em *background*, mas também melhorar a *performance* ao importar *scripts* ou estilos usados num componente específico apenas se este for realmente adicionado ao DOM. Os *bundles* obtidos podem então ser carregados de antemão em background e armazenados em cache pelo *browser*.

Outro aspeto endereçado pela equipa de desenvolvimento foi a acessibilidade, tendo-se procedido à reformulação da biblioteca interna de componentes para que este requisito não funcional fosse devidamente acautelado na UI.

No que concerne a gestão de estado da aplicação, segundo Neary a Airbnb utiliza Redux apenas para lidar com os dados relativos à API e com dados globais como autenticação e outras configurações. Já para interações com o utilizador de mais baixo-nível, como assinalar *checkboxes* ou editar o conteúdo de uma caixa de texto, a gestão de estado é efetuada ao nível do componente para não comprometer o tempo de resposta.

5.5.6 Upwork

A Upwork é uma das maiores plataformas de *freelancing* à escala global. Está sediada nos Estados Unidos, é cotada em bolsa e de acordo com a Alexa ocupava a 443ª posição no *ranking* de tráfego (Alexa, 2020a).

Ao final de 10 anos de atividade, decidiu promover a modernização do *frontend* da aplicação pouco depois de no *backend* ter procedido a uma conversão arquitetural do monolítico em micro serviços. Com esta conversão a Upwork pretendia adequar o modelo de domínio ao conhecimento do negócio, contornar as limitações inerentes à arquitetura monolítica, entre as quais modernizar a *stack* tecnológica algo obsoleta, e permitir que a aplicação fosse capaz de acompanhar a evolução do negócio.

5.5.6.1 Abordagem

No que concerne o *backend*, a abordagem arquitetural seguida pela Upwork, denominada Agora, apesar de enquadrável numa arquitetura de micro serviços, a organização prefere classificá-la como uma "arquitetura de serviços com fina granularidade" para enfatizar que o foco não incide sobre o tamanho dos serviços, mas em garantir o SRP.

O Agora fornece os instrumentos necessários para que cada equipa possa efetuar várias *releases* diárias dos artefactos para o ambiente pretendido, sem impactar ou depender da ação de outras equipas e com baixo risco de indisponibilidade associado uma vez que o *deployment* é efetuado mobilizando técnicas como *Blue/Green* e *Canary Releases* (Karamanlakis, 2017).

A organização constatou que várias das razões que motivaram a adoção de micro serviços eram aplicáveis no *frontend*, nomeadamente a possibilidade de as equipas desenvolverem e efetuarem *releases* ao seu ritmo, a capacidade de integrar mais facilmente novas tecnologias, bem como dar resposta a requisitos como a monitorização, logging, métricas e infraestrutura de integração.

Todavia, a cisão do monolítico *Frontend* apresentava desafios específicos. Quanto ao *design* do *Frontend*, era essencial produzir uma *User Experience* fluída e consistente, uma tarefa cuja dificuldade se agudizaria, considerando que alguns elementos visuais seriam incorporados em várias páginas, desenvolvidas por diferentes equipas, pelo que a separação de responsabilidades não era tão evidente como nos serviços do *backend*.

Com o intuito de alavancar esta arquitetura em camadas verticais, a Upwork decidiu investir esforço na melhoria das *pipelines* de integração existentes e rumar em direção à *Continuous Delivery*. Para levar este desígnio a bom porto, procuraram promover uma cultura de *DevOps*, na qual cada equipa tem autonomia e autoridade para implementar, testar e fazer o *deploy* das respetivas aplicações usando ferramentas de automação.

Atualmente, parte das equipas têm o processo completamente automatizado (CD), enquanto outras apesar de possuírem *pipelines* de integração ainda requerem uma ação manual para que uma *release* deseja desencadeada. A par disso a *Upwork* também se encontra a trabalhar na

monitorização e análise automática de cada vez mais métricas, que permitam potenciar a confiança e medir a *performance* de forma mais abrangente (Karamanlakis, 2017).

5.5.6.2 Dificuldades sentidas

A primeira dificuldade prendeu-se com o menu de navegação da aplicação, uma vez que as opções apresentadas variavam em função das permissões e do contexto de cada utilizador.

Parte desta lógica encontrava-se incorporada no *frontend* monolítico com um elevado nível de acoplamento e criar uma biblioteca limpa poderia implicar um esforço de *refactoring* demasiado alto. Paralelamente, pretendia-se adotar uma *framework* de PHP mais recente nos micro *frontends*.

Inicialmente a Upwork tentou resolver a questão através da criação de um *endpoint* no monolítico que recebendo a informação de autenticação no pedido devolvia apenas HTML contendo as opções de navegação aplicáveis (Nasiri, 2017).

Contudo, como o objetivo era substituir o monolítico e não o tornar uma dependência do novo *frontend*, os micro *frontends* deveriam delegar a lógica de navegação num serviço estritamente dedicado a essa finalidade. Assim, este serviço ao invés de devolver HTML com as várias entradas do menu de navegação, deveria retornar um documento JSON contendo os itens passíveis de ser exibidos. Estes dados eram posteriormente processados por uma biblioteca de *frontend* que produzia o HTML do menu, passível de ser servir o monolítico e os micro *frontends*, evitando duplicações.

Ora, colocar o monolítico a atuar como um serviço para desbloquear micro *frontends* enquanto se procede ao desenvolvimento dos serviços adequados para o substituir, provou ser uma decisão bem-sucedida.

O desafio seguinte prendeu-se com garantir a coerência e fluidez da interface apresentada ao utilizador, enquanto se procedia a uma atualização da *stack* tecnológica e a um *rebranding*, algo conseguido através do desenvolvimento de uma biblioteca de componentes partilhados, cujos recursos, JS e CSS, são versionados a cada *release* e podem ser referenciados pelos micro *frontends* no HTML.

A última dificuldade prendeu-se com o *routing*, pois parte dos URLs eram servidos por um sistema separado do que serve a aplicação principal, mais precisamente num *Load Balancer* do Nginx. Neste contexto, a criação e adição de micro *frontends* requeria efetuar alterações manuais à configuração do Nginx, o que se traduzia num aumento substancial de problemas decorrentes de erros, em atrasos no lançamento de novas *releases* e na necessidade de mobilizar técnicas mais avançadas de *deployment*.

Para resolver a questão a Upwork criou uma camada de *Load Balancing* no *Nginx*, cuja configuração é gerada automaticamente sempre que há atualizações, usando a ferramenta *Consul Template* (Nasiri, 2017).

Assim, para se tornar acedível cada micro *frontend* necessitava apenas de ser registado no Consul, uma ferramenta OSS que permite a descoberta e configuração de serviços (HashiCorp, sem data) .

5.5.7 Ikea

Ciente das limitações inerentes à adoção de uma arquitetura monolítica numa aplicação empresarial de comércio eletrónico, a multinacional sueca do ramo do mobiliário que nos finais de 2019 marcava presença em 56 países (IKEA, 2019) decidiu apostar na migração para micro serviços e micro *frontends*.

Ora, numa entrevista concedida a Stefan Tilkov no âmbito do podcast "*Conversations about Software Engineering*", Gustaf Nilsson Kotte, consultor na IKEA desde 2016, dá a conhecer as motivações e a abordagem de implementação da arquitetura de micro *frontends* no *site* da IKEA.

5.5.7.1 Motivações

O *frontend* nunca foi percecionado / pensado como um serviço, daí ter perdurado como monolítico. No entanto, começaram a verificar-se no *Frontend* Monolítico os problemas que já assolavam o *Backend* Monolítico e que acabaram em muitos casos por sentenciá-lo, nomeadamente o facto de não possível efetuar *releases* automaticamente, o crescimento desordenado da *codebase* e paralelamente do acoplamento e dos ciclos de desenvolvimento e *feedback* (Kotte, 2017).

Ora, a arquitetura de micro *frontends* permite trazer a *continuous delivery* para a esfera do *frontend*, e consequentemente viabiliza a capacidade de evolução independente.

Verifica-se, todavia, sobretudo nas empresas de grande dimensão, uma hegemonia da organização das equipas em camadas horizontais, numa espécie de rede de serviços especializados, ainda que o objetivo passe por quebrar o *Frontend* Monolítico.

Quanto ao número de serviços a implementar, não há nenhuma regra formal que o estipule. No entanto, em relação à dimensão das equipas, Gustaf Kotte sugere aplicar a Regra das Duas Pizzas, particularmente quando não se está familiarizado com o conceito e aplicação de micro *frontends*. Uma equipa pode deter mais do que um serviço, ainda que se recomende uma relação de um serviço por equipa, precisamente para evitar que as equipas fiquem com um novo monolítico em mãos.

Assim, num universo de 24 colaboradores a trabalhar num mesmo sistema, é preferível repartir estes colaboradores por duas equipas multidisciplinares, que tratam ambas de *backend* e *frontend*, para entregar valor mais rapidamente ao utilizador final.

É que num cenário em que se distribuam por duas equipas de *backend* e uma de *frontend* pode dar-se o caso de uma *user story* (US) já estar finalizada na camada de *backend* mas não poder seguir para cliente devido à interface gráfica ainda não estar fechada.

Analisando a situação numa perspetiva de alto nível, poderá considerar-se que ao aumentar a equipa de *Frontend* se vá resolver o problema, quando em boa verdade se está a caminhar a passos largos para um novo monolítico.

A aplicação da Lei de Conway é um bom ponto de partida para respeitar esta restrição na dimensão das equipas (Kotte, 2017).

Numa fase posterior e apesar de ser possível que cada detenha mais do que um serviço deve procurar manter-se uma relação de um para um entre serviço e equipa. Nesse ponto, para evitar que se resvale para uma arquitetura monolítica deve aplicar-se a Manobra Inversa de Conway (Kotte, 2017).

Assim, perante uma SPA desenvolvida usando uma *framework* JS na qual vão ser integrados micro *frontends* de duas equipas distintas, estas equipas devem colaborar na SPA integradora, por oposição a existir uma terceira equipa dedicada apenas à integração (Kotte, 2017).

Acrescenta que é a abordagem mais comum, até porque os micro serviços na sua génese são sobre diversidade e heterogeneidade arquitetural, o que impulsiona a utilização de diferentes tecnologias.

Apesar disso, deve averiguar-se se valerá realmente a pena suportar diferentes *frameworks* ou bibliotecas, em particular quando existe um número reduzido de equipas a trabalhar num projeto. Contudo, se o *frontend* estiver preparado para a integração agnóstica de tecnologias desde o início, será mais fácil e rápido encetar alterações na *stack* tecnológica sem que isso exija a reescrita integral da aplicação.

Assistindo-se a alterações frequentes no panorama do desenvolvimento *web*, inerentes ao aparecimento de novas soluções, Gustaf defende que a longo-prazo, é mais do que suficiente dar suporte a uma única biblioteca ou até optar por uma arquitetura monolítica em organizações de pequena escala como *startups*.

Por oposição, em organizações de maior dimensão, deve dar-se suporte a um conjunto alargado de tecnologias para que não seja necessário proceder à reescrita de parte significativa da aplicação em vários momentos do seu período de atividade. Além disso, partindo do pressuposto de que basear uma aplicação inteiramente numa única mesma tecnologia é arriscado ou imprudente, a única técnica de integração que permite obter no *frontend* os mesmos ganhos da adoção no *backend* de micro serviços passa pela transclusão (Kotte, 2017), que é precisamente a abordagem implementada no *site* do IKEA.

A transclusão pode ocorrer do lado do cliente, usando CSI, de que são exemplos as *iframes* ou as *tags* de imagem, ou ocorrer do lado do servidor, usando diretivas ESI ou SSI.

No que concerne o *Edge-Side Includes*, *Edge* diz respeito à última camada na qual se detém algum tipo de controlo sobre a aplicação ou infraestrutura e que corresponde tipicamente a uma CDN ou a uma camada de cache similar (Kotte, 2017).

A transclusão por via de diretivas ESI do lado do servidor processa-se de forma semelhante ao carregamento de uma imagem por elemento <img> do lado do cliente, uma vez que à semelhança do que acontece com o atributo "*src*", nas diretivas ESI é especificado o caminho para os fragmentos ou páginas.

Se o recurso pretendido não se encontrar em cache, o pedido será encaminhado para o servidor de origem.

5.5.7.2 Utilização de ESI no contexto da IKEA

O ESI no IKEA é a fundamental para adotar micro *frontends* (Kotte, 2018).

As páginas e os fragmentos (consistem em documentos HTML incompletos) são os conceitos aglutinadores da linguagem ESI, podendo cada equipa ter a seu cargo um conjunto de páginas e/ou fragmentos.

As páginas contêm referências ESI para os fragmentos, referências essas que podem cruzar as fronteiras entre equipas. Assim, uma página de Produto pode conter referências para os fragmentos de *Header* e *Footer*, o que torna estes fragmentos reutilizáveis.

Do mesmo modo, uma equipa, por exemplo, a responsável pelo Produto, pode além da página de detalhes do mesmo, disponibilizar fragmentos que correspondem a cartões do Produto, passíveis de ser utilizados por outras equipas nas respetivas páginas. Daqui facilmente se infere que a dificuldade não reside na divisão, mas na composição dos vários micro *frontends*, uma dificuldade minimizada com a utilização de ESI.

Outro ponto a favor do ESI prende-se a possibilidade de tirar partido da cache para melhorar significativamente a *performance*.

No caso do IKEA, um documento processado é armazenado por 15 minutos e até mesmo as respostas aos pedidos são guardadas em cache.

Deste modo, tudo o que é estático ou não sofre alterações com frequência pode ser diretamente obtido da cache, sem necessidade de *round-trips* adicionais. Isto introduz um ponto de viragem, uma vez que até então qualquer alteração implicava a invalidação de toda a página e solicitar tudo novamente ao servidor de origem (Kotte, 2018).

Tipicamente num *e-commerce* partes como *Header* e *Footer* são bem menos suscetíveis a sofrer alterações do que informação relativa a preço, pelo que podem ser armazenados em cache.

No que concerne a cache, apesar de permitir até o armazenamento de respostas e conteúdo dinâmico, convém ressalvar que nem tudo deve ser armazenado.

A possibilizar de personalizar a utilização da cache pode potenciar a *performance* de uma SPA, visto que devido à elevada interatividade faz sentido que a gestão de estado ocorra no cliente.

Inicialmente o *site* da IKEA era construído por um gerador de páginas estáticas. Esta solução simplificava a invalidação da cache, dado que perante alterações num documento era gerado um novo ficheiro que posteriormente era mantido em cache.

Atualmente, os resultados do processamento ESI são mantidos por 15 minutos enquanto os documentos subjacentes são armazenados 48 horas, ainda que possam ser invalidados antes caso se verifique alguma alteração (Kotte, 2018).

Kotte defende a utilização de geradores de *sites* estáticos para conteúdo e informação que sofrendo poucas alterações pode tirar partido da cache e de SPAs para conteúdo personalizado e sempre que a gestão de estado ocorra maioritariamente do lado do cliente.

Enaltecendo a capacidade de evolução independente da arquitetura de micro *frontends*, reconhece que para dar suporte a aplicações legadas, mantendo as suas particularidades, e integrá-las numa única aplicação deve preferir-se enveredar por esta abordagem (Kotte, 2018).

No que concerne uma eventual inconsistência entre micro *frontends* e/ou ocorrência e conflitos de *scripts* e estilos, Kotte sugere que pode ser colmatada através da criação de um ficheiro/biblioteca global de estilos colocada ao dispor de páginas e fragmentos, que se resuma ao essencial para que não se crie demasiado acoplamento.

Desconhecendo quem vai utilizar os fragmentos que desenvolvem, as equipas devem procurar assegurar a sua autocontenção e aquando da correção de erros se deve favorecer o *Mean-Time-To-Recovery* em detrimento do *Mean-Time-Between-Failures* (Stenberg, 2018).

Na perspetiva de Kotte, o conceito de autocontenção pressupõe que caso se pretenda usar um fragmento e seja necessário adicionar CSS, este seja incluído noutro fragmento que o referencie. Esta opção de incluir através de fragmentos independentes os recursos prende-se com o *cache busting*.

Quando estes ficheiros são armazenados em cache durante bastante tempo até expirarem, pode dar-se o caso de, entretanto, serem atualizados. Contudo, devido ao facto de armazenados em cache no *browser* dos visitantes, estes podem não ser capazes de verem estas atualizações repercutidas.

Usando *cache busting*, sempre que um ficheiro referenciado nos fragmentos possui uma nova versão, é gerada uma *hash* e as referências atualizadas, sem que as *tags* ESI, através das quais se integram os fragmentos, precisem sofrer quaisquer modificações.

Relativamente aos fragmentos produzidos pelas equipas de desenvolvimento, a IKEA não definiu normas quanto à *markup* permitida, mas promove-se a produção de fragmentos que não possuam um número excessivo de dependências para não comprometer a *performance* e nos quais seja possível recuperar rapidamente de falhas e erros.

Quanto à autonomia das equipas na tomada de decisão, as equipas devem evitar que as suas decisões entrem em rota de colisão com os contratos firmados ou lógica comum. Caso contrário, pode-se colocar em risco todo o sistema aplicacional.

Os fragmentos devem ser o mais circunscritos possível, pelo que é inconcebível que somente para desenvolver um *widget* de carrinho de compras (apresentado no *Header*) se recorra a uma *framework*, por exemplo Angular, pois poderia criar uma dependência generalizada, que atentaria contra o carácter agnóstico de tecnologia dos micro serviços e aumentaria substancialmente o tamanho do *bundle*.

Na perspetiva de Gustaf, o uso de ESI viabiliza a criação de soluções baseadas em micro *frontends* tirando proveito do SSR. Por omissão, o ESI carrega toda a página. No entanto, para otimizar a *performance* é possível através de *Client-Side Includes*, efetuar o carregamento de fragmentos à medida das necessidades (*lazy loading*).

Também pode dar-se o caso de existir um micro *frontend* destinado à pesquisa de um item, escrito usando uma *framework* de SPA, que contempla uma caixa de texto na qual é introduzida a expressão a procurar, sendo apresentados resultados à medida que são introduzidos novos caracteres. Neste caso, uma vez que o carregamento de fragmentos ocorre do lado do cliente, o ESI fica incumbido apenas por referenciar estilos e *scripts*, havendo uma separação clara entre inclusão de conteúdo e de recursos (Kotte, 2018).

Contudo, há abordagens mais declarativas de CSI, entre as quais, o h-include, uma biblioteca da autoria de Kotte inspirada no hinclude de Mark Nottingham, através da qual basta adicionar um elemento DOM via JavaScript para que depois seja incluído na página.

Num *site* de *e-commerce* o SEO assume particular importância.

O JavaScript não oferece garantias de que páginas renderizadas sejam indexadas pelos *crawlers* dos motores de pesquisa, algo que se torna mais problemático em *sites* nos quais a transclusão se resume a CSI.

Uma alternativa consiste na pré-renderização do lado do servidor e doravante efetuá-la do lado do cliente, mas há que ponderar se a complexidade introduzida valerá a pena, começando sempre por equacionar primeiro o SSR ou o CSR e só depois abordagens híbridas (Kotte, 2018).

Em síntese, o mais importante a reter da experiência da IKEA é que não existindo soluções *"one size fits all"*, deve ser efetuada uma análise dos *trade-offs* a nível organizacional e de serem pesados os prós e contras de diferentes abordagens, para se determinar a solução mais adequada.

5.5.8 DAZN

O DAZN é o primeiro serviço de *streaming* de desportos ao vivo e *on-demand* a nível mundial, daí ser, por vezes, daí ser vulgarmente designado "Netflix do desporto", contemplando no seu catálogo competições nacionais e internacionais («DAZN», 2020b).

Fundada em 2015 no Reino Unido, a DAZN está presente em 9 e prevê lançar em 2020 um serviço em inglês disponibilizado a mais de 200 países e territórios (DAZN, 2020a).

À semelhança da Netflix e do Spotify, também a DAZN dá suporte a uma vasta panóplia de dispositivos, entre os quais, PCs, *smartphones*, *smart* TVs, consolas de videojogos, etc.

Atualmente o desenvolvimento da plataforma está a cargo de um conjunto alargado de equipas presentes em diferentes localizações geográficas.

Perante num universo organizacional composto por várias equipas dispersas e a necessidade de dar resposta a requisitos como a *performance* e a heterogeneidade de dispositivos, sobretudo numa plataforma desta natureza, era necessário estudar minuciosamente a solução arquitetural e organizacional a implementar.

5.5.8.1 Abordagem arquitetural

Depois de efetuar a primeira *release*, constatou-se que seria necessário repensar a plataforma para escalar quer a organização, quer a plataforma de *streaming*.

Segundo Luca Mezzalira, VP de arquitetura na DAZN, para alcançar este objetivo foram analisadas e desenvolvidas POCs usando diversas abordagens e práticas comuns, com vista a determinar qual a forma mais sustentável de escalar a empresa e integrar novos colaboradores.

De todas as abordagens equacionadas, decidiram abraçar a arquitetura de micro *frontends* que, aliada a micro serviços, poderia revelar-se uma mais-valia para escalar as equipas e criar artefactos independentes e que são mapeados com os domínios de negócio.

Antes de proceder à implementação propriamente dita de micro *frontends*, especialistas de domínio, arquitetos e equipa de desenvolvimento uniram esforços em prol da modelação do negócio em conformidade com o DDD, tendo-se identificado os subdomínios apresentados no topo da figura abaixo, que posteriormente seriam orquestrados através de uma camada denominada Bootstrap (Mezzalira, 2019b).

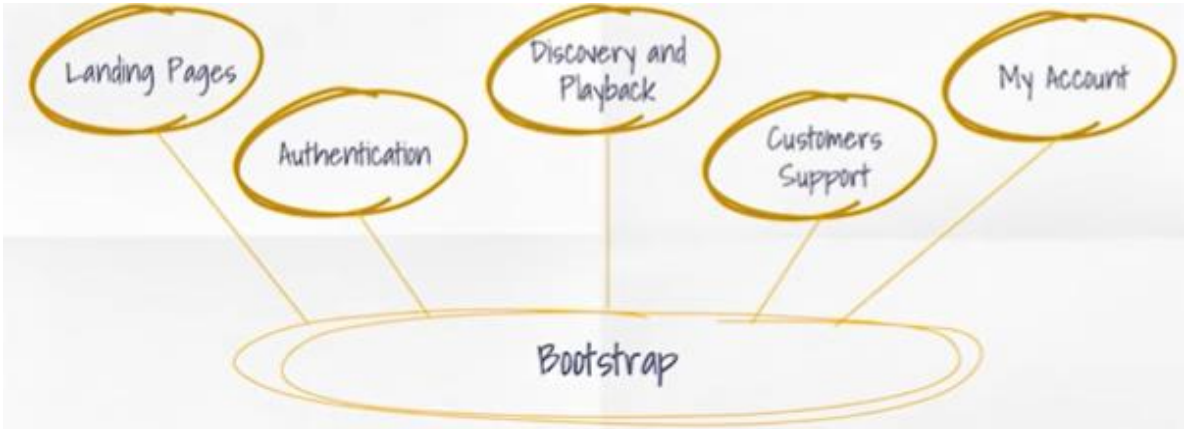


Figura 38 – Subdomínios de domínio da DAZN (Mezzalira, 2018)

Na perspetiva de Mezzalira, seguindo à risco os princípios do DDD, "identificar um micro *frontend* torna-se trivial". Contudo, o processo de levantamento de micro *frontends* varia em função do estado do projeto, isto é, de se tratar de um novo projeto ou de um projeto legado.

O primeiro cenário torna-se mais desafiante, uma vez que não se dispõe de dados que permitam inferir como os utilizadores consumirão o conteúdo disponibilizado. Já em projetos legados recorrendo ao *User-Centered Design* a tarefa torna-se mais simples (Mezzalira, 2019b).

Em matéria de granularidade, afirma que um micro *frontend* pode corresponder a uma mera página ou a uma aplicação baseada numa SPA (caso da DAZN) ou em SSR.

Apesar de ser crucial identificar os micro *frontends* à *priori*, estes evoluem com o negócio. Prova disso mesmo é que atualmente e contando já com um maior número de equipas de produto, estas continuem a ser enquadráveis nestes subdomínios.

Estes subdomínios são orquestrados através de uma camada designada Bootstrap, imediatamente descarregada ao aceder ao endereço da plataforma e responsável por:

- Inicializar a aplicação, a configuração específica para o país e dispositivo de acesso);
- abstrair as operações de I/O dos micro *frontends,* pois, sobretudo entre fabricantes e até entre mesmo modelos dos dispositivos suportados, verifica-se uma elevada variabilidade nas APIs disponibilizadas. Prescindindo desta abstração ao nível do Bootstrap, teria de ser desenvolvido um MFE para cada equipamento suportado;
- *routing* dos vários micro *frontends*, que são apresentados de acordo com o estado do utilizador. Assim, ao aceder ao *site* principal, caso o utilizador se encontre autenticado é encaminhado para "*Delivering & Playback*", caso contrário segue para a "*Landing Page*" ou para página de "Autenticação", se estiver a tentar aceder a uma rota específica protegida;
- Partilhar configurações transversais com múltiplos micro *frontends* e notificá-los aquando de alterações sobre dados globais (caso dos referentes à autenticação).

A solução criada pela DAZN permite às equipas efetuar o *deploy* para o ambiente pretendido de forma independente e quando pretendido, usando técnicas como o *Canary Releasing* (Mezzalira, 2018).

5.5.8.2 Balanço da adoção de micro *frontends*

Inicialmente formaram-se cinco equipas de produto, que em apenas uma semana conseguiram contribuir para a *codebase*, graças à sua autonomia e especialização num determinado subdomínio, que sendo perfeitamente delimitado e contido é mais facilmente apreendido.

Em matéria de escalabilidade, a DAZN constatou que na hora de escalar aplicações de *frontend* a maior dificuldade prende-se com escalar as equipas e não a infraestrutura.

No entanto, ao final de pouco tempo, eram já 100 os *developers* de diferentes localizações a trabalhar na plataforma, focados no desenvolvimento E2E de partes específicas da mesma, enquanto a escalabilidade da camada de orquestração estava a cargo de arquitetos de *software*.

A organização frisa ainda a importância de alcançar um *throughtput* positivo (taxa de transferência/produtividade), isto é, ter toda a equipa alinhada e a convergir para o objetivo, reduzindo o ruído / sobrecarga de comunicação e coordenação entre equipas.

Por último, a DAZN acredita que ao possibilitar a definição de *stacks* tecnológicas mais diversificadas talvez seja possível reduzir o número de saídas de *developers* das organizações motivadas pela vontade de aprender e aplicar novas tecnologias, ferramentas e abordagens (Mezzalira, 2018, 2019a).

5.5.9 Zalando

A Zalando é uma das maiores retalhistas de moda da Europa, tendo registado uma receita de 4.5 milhões de euros em 2017. Relativamente à plataforma segundo dados de 2018 contava com um catálogo de mais de 300 mil produtos e registava mais de 200 milhões de visitas por mês, das quais 75% através de dispositivos móveis (Kubyshkin, 2018).

Contando na atualidade com cerca de 2500 *developers*, pretende potenciar a produtividade e continuar o desenvolvimento de novas funcionalidades no *website*, que regista diariamente picos de tráfego.

5.5.9.1 Contexto

Inicialmente o *site* da Zalando possuía uma arquitetura monolítica, assente numa filosofia *"Thin Client, Fat Server"*, na qual a aplicação FE estava construída sobre um único serviço de *backend*, responsável por praticamente toda a lógica, desde a renderização de UIs à gestão de dados estatísticos.

À data, a aplicação demorava em média cerca de 1 hora a compilar, o que impactava negativamente o ciclo de desenvolvimento, a motivação e a produtividade dos *developers*, em especial os *frontend developers*, que se viam obrigados a esperar para testarem qualquer alteração (Kubyshkin, 2018).

5.5.9.2 Abordagem

A Zalando começou a equacionar adotar micro serviços na espera de alcançar autonomia e escalar a organização rapidamente.

Kubyshkin, colaborador da Zalando desde 2016 e atualmente arquiteto de *frontend*, afirma que quando entrou na organização esta contava com cerca de 150 *developers* e estava focada em tirar o máximo partido das pessoas para tornar a Zalando mais produtiva e capaz de integrar facilmente novas tecnologias nas suas soluções (Kubyshkin, 2018).

O processo de conversão para micro serviços teve como ponto de partida o levantamento de subdomínios e *bounded contexts*, em particular os *Core Domains*, para estruturar o código, atribuir as responsabilidades e delimitar os vários micro serviços para posteriormente entregar o seu desenvolvimento a equipas mais familiarizadas com a área de negócio correspondente.

Se é verdade que a conversão do *backend* em micro serviços permitiu suprir parte dos problemas, não é menos verdade que limitações que advém da interação do utilizador com a plataforma persistiram devido ao caráter monolítico do *frontend*.

Kubyshkin realça, no entanto, que uma arquitetura monolítica no *frontend* não é sinónimo de fracasso. Aliás, em projetos desenvolvidos por equipas de reduzida dimensão, como detêm um conhecimento holístico da solução, uma solução monolítica é suficiente (Kubyshkin, 2018).

Sendo inviável colocar todos os *developers* a trabalhar sobre o mesmo repositório e *codebase*, a Zalando lançou o Project Mosaic, que consiste num conjunto de serviços, bibliotecas e especificações que definem como os componentes devem interagir entre si, para suportar uma arquitetura baseada em serviços em *websites* de larga escala.

Nesse contexto, cada equipa é responsável por fornecer fragmentos HTML, isto é, partes de *markup*, passíveis de ser integradas numa página e obtidas através de chamadas a serviços *backend*, regra geral, desenvolvidos e mantidos pela própria equipa de ou serviços centralizados que tratam de *cross-cutting concerns* como Traduções, Autenticação, etc.

Relativamente aos serviços desenvolvidos pela Zalando para adotar esta arquitetura, comece-se por analisar o Layout Service, escrito em Go e inspirado no BigPipe do Facebook.

O Layout Service, denominado Tailor, é responsável pela composição dos fragmentos, estabelecendo o local onde cada um deve ser integrado e definindo com que serviços cada fragmento pode comunicar. Fornece através da sua API fragmentos de *markup* passíveis de ser agregados para obter uma página.

Quanto ao Router, designado Skipper, interceta os pedidos recebidos do *browser* do cliente e decide o que fazer com eles.

5.5.9.3 Benefícios da abordagem adotada

São inúmeros os benefícios da adoção de micro *frontends*, mais especificamente:

- A existência de um router global (reverse proxy) facilmente configurável a mediar o acesso aos vários serviços, capaz de redirecionar em função quer da rota quer do tráfego para a nova infraestrutura;

- O desenvolvimento do Skipper, uma ferramenta OSS que desempenha um papel idêntico ao do NGINX atualmente. Contudo, aquando do início de trabalhos no Skipper o fornecimento de conteúdo dinâmico ao NGINX só era possível utilizando uma linguagem muito específica, alvo de constantes alterações e com pouca adesão da comunidade. Entre as funcionalidades oferecidas pelo Skipper destacam-se a capacidade de fazer *targetting* e a atualização de rotas dinâmicas em poucos segundos, o que permite alternar rapidamente entre rotas do novo sistema e do antigo;

- A infraestrutura fornecida pela Zalando às equipas, permitiu-lhes efetuar *deployments* de forma independente e com a periodicidade desejada;

- A escalabilidade organizacional e aplicacional alcançada. Em 2018 a Zalando contava já com mais de 20 equipas de produto com cerca de 50 fragmentos a seu cargo (havendo equipas a trabalhar em mais do um fragmento). Por outro lado, a plataforma foi capaz de responder a picos de tráfego (que atingiram o marco de 1000 visitas à página por segundo);

- Melhoria significativa da *performance* traduzida num decréscimo da latência entre 20 e 30 milissegundos, motivado pelos serviços se encontrarem em *streaming*. Desta forma, basta que os *downstream services* respondam prontamente para que a página carregue rapidamente.

5.5.9.4 Questões e desafios levantados

Além dos benefícios inerentes à adoção de micro *frontends*, há determinados aspetos a considerar antes mesmo de adotar micro *frontends* (Kubyshkin, 2018).

Desde logo, a dificuldade em recrutar colaboradores com conhecimentos E2E e/ou com bases de DevOps. Assim, como consequência da inevitável curva de aprendizagem verifica-se uma desaceleração no *throughput* dos colaboradores enquanto se encontram em fase de integração.

Na conversão de monolítico para micro serviços pode verificar-se que alguns serviços não estão diretamente ligados a necessidades do cliente ou do negócio, por exemplo, abstrações de *storages* e processos, acabando muita da lógica num micro *frontend* ou mesmo disseminada por vários.

Para que a conversão ocorresse o mais rapidamente possível, o desenvolvimento da infraestrutura ficou a cargo da *Core Team*. Estando a escalabilidade intimamente ligada a *cross-cutting concerns* como monitorização e *logging*, é fundamental que a Core Team forneça às equipas de produto ferramentas e *templates* com qualidade, devidamente documentadas e eventualmente suportados por mecanismos de *scaffolding*, para que estas se possam dedicar inteiramente aquele que é seu real foco: desenvolver funcionalidades relacionadas com o respetivo domínio que criem valor para o cliente.

Outro aspeto a ter em mente é que centralização não é sinónimo de monolítico. O desenvolvimento de *routing*, *logging*, *tracking*, testes A/B, composição de páginas pode não ser entregue a equipas de produto, até porque são questões difíceis de endereçar. Deste modo, podem-se concentrar esforços no desenvolvimento de funcionalidades que vão entregar verdadeiro valor ao cliente.

Para viabilizar a conversão de um monolítico em micro serviços ou micro *frontends* é fundamental que exista uma infraestrutura que assegure a integração contínua e o que permita fazer chegar a implementação das *cross-cutting concerns* aos vários micro *frontends*.

Contudo, antes de partir para a implementação de micro serviços ou micro *frontends* há que determinar se faz sentido, com base nos requisitos e problema a endereçar, respondendo às questões apresentadas na tabela abaixo.

Tabela 11 – Questões a colocar antes de decidir adotar micro *frontends*

QUESTÃO	RESPOSTA DA ZALANDO
Qual o problema que se pretende endereçar usando micro *frontends*?	Potenciar o *throughput* num contexto de crescimento da organização, nomeadamente das equipas de desenvolvimento
É a infraestrutura proposta capaz de dar resposta a problemas transversais complexos ao invés de deixar essa tarefa para as equipas?	Num cenário de micro *frontends* deve investir-se em infraestrutura, criando condições para que o foco das equipas indica sobre a entrega de valor para o cliente
Qual é o modelo de contribuição para a infraestrutura?	A infraestrutura em si é um monolítico, por isso, não havendo diretrizes que estabeleçam como podem as equipas dar o seu contributo, eventuais alterações podem impactar as restantes equipas

Ora, pretendendo a Zalando manter esta rota de crescimento bem-sucedido, começou recentemente a trabalhar na Interface Framework, uma versão melhorada do Mosaic, em particular do Tailor, baseada em conceitos similares, mas mais focada em componentes e GraphQL (Colin, 2018).

5.5.10 Critical Techworks

A Critical Techworks (CTW) remonta a 2018, ano em que a BMW após analisar mais de 1000 empresas selecionou a Critical Software como a parceira ideal para auxiliar o colosso da indústria automóvel da Baviera a liderar a transformação digital neste setor através do desenvolvimento ágil de *software* para os veículos do futuro (Cordeiro, 2019).

Desde a sua fundação que a organização se rege por um modelo de trabalho fortemente impregnado pelo manifesto Agile, que define os princípios gerais de colaboração entre pessoas, no sentido de as equipas estarem alinhadas com a organização no essencial, nomeadamente em matéria de valores, mas concedendo-lhes autonomia e encorajando-as a complementar as diretrizes com o seu cunho pessoal.

Apresentando uma estrutura horizontal em que as hierarquias praticamente se desvanecem, o formal papel de chefe dá lugar ao de facilitador e o desenvolvimento de *software* é entregue a unidades, unidades essas que englobam um conjunto de equipas.

As unidades comportam-se como pequenas empresas pois detêm um ecossistema ou contexto independente no qual os elementos que o compõem cooperam e convergem em prol de uma visão e objetivos comuns.

Um dos princípios presentes no modelo de trabalho, afirma que a organização procura "*Feature Teams*", que "devem reunir todas as competências necessárias para entregar uma funcionalidade" sem que "cada um dos elementos da equipa necessite de deter conhecimento de todas as matérias" (P. V. da Silva et al., 2019).

Convicta de que "em conjunto as pessoas conseguem chegar mais longe", a organização propõe que as equipas de Produto sejam constituídas por *Product Owner*, equipa de *DevOps* (composta tipicamente por cinco *developers*, um dos quais assumindo simultaneamente o papel de *Scrum Master)*, cujos conhecimentos técnicos no global devem possibilitar o desenvolvimento *end-to-end* de aplicações e que preferencialmente devem estar na mesma localização geográfica), UX *Designer* (dependendo da complexidade da UI ou das interações com o utilizador) e partes interessadas na aplicação (P. V. da Silva et al., 2019).

Estas equipas devem ser capazes de confluir conhecimento de "negócio, inovação e qualidade", "desenvolver produtos com perícia e alinhados com os objetivos do produto", estar "focadas na entrega de incrementos de qualidade e que representem valor para os clientes (MVPs) tão rapidamente quanto possível" e é expectável que os *developers* sejam capazes de "desenvolver os seus conhecimentos de negócios para tomar as melhores decisões técnicas e antever necessidades futuras" bem como "criarem *pipelines* de entrega de elevada qualidade e que promova o *Continuous Deployment*".

Na CTW, as equipas passam de *DevOps* a *BizDevOps*, dado que além de terem a seu cargo a implementação e a idealização e construção da infraestrutura que viabilize a CD, os *developers* são incutidos a participar ativamente na tomada de decisões e no planeamento estratégico de um produto, daí a preocupação com a evolução do seu conhecimento do negócio.

Perante isto, pode-se afirmar sem reservas que quer a organização das equipas, quer os papéis e respetivas responsabilidades estão alinhadas com os princípios dos micro *frontends*.

Apesar de desde a sua fundação a CTW ter adotado esta organização multidisciplinar de equipas e, portanto, ser difícil proceder à comparação com uma realidade que não esta, alguns supervisores, entre os quais Dina Lopes, *Head of Interactions* da unidade na qual se insere a equipa que o autor deste documento integra, denotam um decréscimo significativo da produtividade em projetos nos quais o desenvolvimento das camadas de *Frontend* e *Backend* é entregue a diferentes equipas e/ou a colaboradores que se encontrem em localizações distintas.

Na sua ótica, quando tal se verifica, evidenciam-se dificuldades de comunicação e de coordenação.

Além disso, alerta para a criação de dependências ou constrangimentos que, por vezes, extravasam as competências das próprias equipas e que podem comprometer o planeamento inicialmente previsto devido a derrapagens na duração dos ciclos de desenvolvimento e *release*.

No mesmo sentido aponta João Gonçalves, *Tech Lead* da unidade do autor deste documento e CTO da Critical Techworks.

Com base em experiências profissionais anteriores, considera que a atribuição do desenvolvimento de *Frontend* e *Backend* a equipas distintas cria uma forte dependência entre as mesmas, sobretudo aquando do lançamento de uma nova *release* para produção.

Nessas condições, efetuar um *deploy* implica um período de indisponibilidade, no qual são introduzidas alterações que serão submetidas a validação. Ora, podendo abranger ambas as camadas, requerem a coordenação entre as equipas para garantir a integridade da nova versão da aplicação lançada para o cliente.

Já a entrega do desenvolvimento de funcionalidades a equipas multidisciplinares permite, na sua perspetiva, obter ciclos de desenvolvimento mais curtos e garantir a integridade do produto, promove a escalabilidade e a autonomia sustentadas numa infraestrutura tão automatizada quanto possível e inteiramente idealizada e concebida por cada equipa, que permita a monitorização, *logging* e *tracing* em tempo-real e a *Continuous Delivery*.

Além disso, a Critical Techworks tem movido esforços no sentido de colmatar desafios passíveis de ser despoletados pela autonomia que confere às equipas.

Nesse âmbito, com vista a uniformizar a *User Experience*, assegurando tanto quanto possível a consistência e coerência visual das aplicações, a empresa tem vindo a investir no desenvolvimento de um *Design System*.

Tratando-se a formação de silos de conhecimento de outro potencial problema, a organização tem procurado realizar periodicamente quer sessões nas quais uma determinada equipa dá a conhecer aos restantes colegas o trabalho que tem vindo a ser realizado, quer ações promovidas pelas *Communities of Practice* (CoP).

O sucesso da aposta da BMW é notório e se dúvidas houvesse, os números falam por si. No último trimestre de 2018, registou um volume de faturação que superou os três milhões de euros e tendo iniciado atividade com apenas 100 colaboradores, a organização estima até ao final de 2020 atingir a fasquia dos 1000 (Lusa, 2019).

6 Prova de Conceito

Este capítulo tem como objetivo:

- Contextualizar a prova de conceito a desenvolver;
- Proceder à análise e modelação de domínio, com vista a atingir os objetivos traçados e responder ao problema enunciado;
- Apresentar o *design* arquitetural preconizado para cada técnica de integração;
- Descrever a abordagem seguida na versão implementada e apresentar possíveis abordagens de implementação para as restantes versões da prova de conceito;
- Identificar vantagens e desvantagens das técnicas selecionadas.

6.1 Contexto

Recolhida e tratada a informação relativa à adoção de uma arquitetura de micro *frontends* no processo de desenvolvimento *web*, explanada nos capítulos anteriores, de modo a mobilizar o conhecimento adquirido e aplicá-lo na prática, propõe-se o desenvolvimento de uma prova de conceito (POC).

Esta prova de conceito consiste no desenvolvimento de uma aplicação *web* de *e-commerce* simplificada, denominada Tellco, que sendo construída segundo esta arquitetura, pressupõe a integração dos vários micro *frontends* a desenvolver e a comunicação entre si.

Serão produzidas várias versões desta prova de conceito, cujo ponto de variabilidade corresponderá em grande medida à técnica de integração seguida em cada versão, uma vez que relativamente à implementação dos MFEs propriamente dita não se preveem alterações substanciais entre versões.

Perante um conjunto tão vasto de técnicas e ferramentas de integração enunciadas nos capítulos anteriores, serão selecionadas aquelas que melhor se ajustam ao que se espera das aplicações *web* atualmente e especificamente aos requisitos pretendidos desta aplicação *e-commerce*.

Quanto às tecnologias a utilizar em cada MFE, dado o carácter agnóstico de tecnologia das técnicas de integração analisadas anteriormente, a seleção recairá sobre aquelas que pela sua natureza sejam mais propensas à adoção de uma arquitetura de MFE, dando primazia a opções com as quais o autor deste documento esteja mais familiarizado.

Relativamente aos objetivos desta prova de conceito, consiste em:

- Aferir a viabilidade de uma arquitetura de micro *frontends* face a outras abordagens;
- Determinar quais as técnicas de integração mais ajustadas a uma aplicação *e-commerce*;
- Determinar com base nas conclusões retiradas da implementação das várias versões em que contextos a adoção de MFEs é uma mais-valia e quais as vantagens e desafios.

6.2 Análise

Como referido anteriormente, a aplicação a desenvolver no âmbito desta prova de conceito corresponde a uma aplicação *e-commerce, que, portanto,* corresponde ao seu domínio.

Antes de proceder à implementação de uma aplicação com alguma complexidade, é necessário ter um bom conhecimento sobre o negócio no qual a aplicação se insere. Nesse sentido, o contributo dos especialistas de domínio (pessoas que detém um amplo conhecimento da área de negócio) no levantamento concetual é crucial.

No entanto, como referido na seção 2.1.6, para que se que estes conceitos possam ser aplicados no código é necessário passarem por um processo de modelação baseado na estreita colaboração entre especialistas de domínio e de *software*.

Segundo Evans, um perito reconhecido em DDD (Avram & Marinescu, 2007; Fowler, 2005), todos os modelos correspondem a uma simplificação que não é mais do que "uma interpretação da realidade que abstrai os aspetos relevantes para resolver o problema ignorando detalhes acessórios" (Evans, 2003).

Dado que a aplicação *e-commerce* não será desenvolvida em contexto real para um determinado cliente final, o processo de modelação não derivará de uma auscultação formal e colaboração com especialistas de domínio, mas sim da análise das funcionalidades e comportamentos verificados em plataformas de venda eletrónica como a Amazon e a Worten.

Estas aplicações, regra geral, abarcam conceitos como: produto, catálogo de produtos, cliente, carrinho de compras, recomendação ou sugestão, avaliação de produto, encomenda, inventário, pagamento, entrega e gestão de conta.

Sustentando-se nas palavras de Evans e tendo em conta os objetivos delineados para a prova de conceito, o autor deste documento deliberou no processo de modelação do domínio deixar de fora os conceitos de inventário, pagamento, entrega e gestão de conta, por considerar não acrescentarem valor ao que se pretende da prova de conceito, tendo se obtido o modelo de domínio apresentado abaixo.

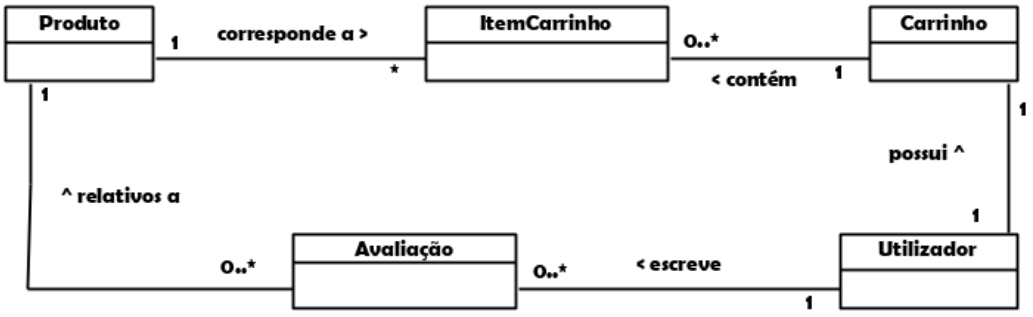


Figura 39 - Modelo de Domínio da POC

Assim sendo, na sequência do processo de modelação do domínio "aplicação *e-commerce*" e das considerações enunciadas na secção 2.1.6.1, apuraram-se os subdomínios e *bounded contexts* a desenvolver na prova de conceito, que podem ser consultados na Figura 40.

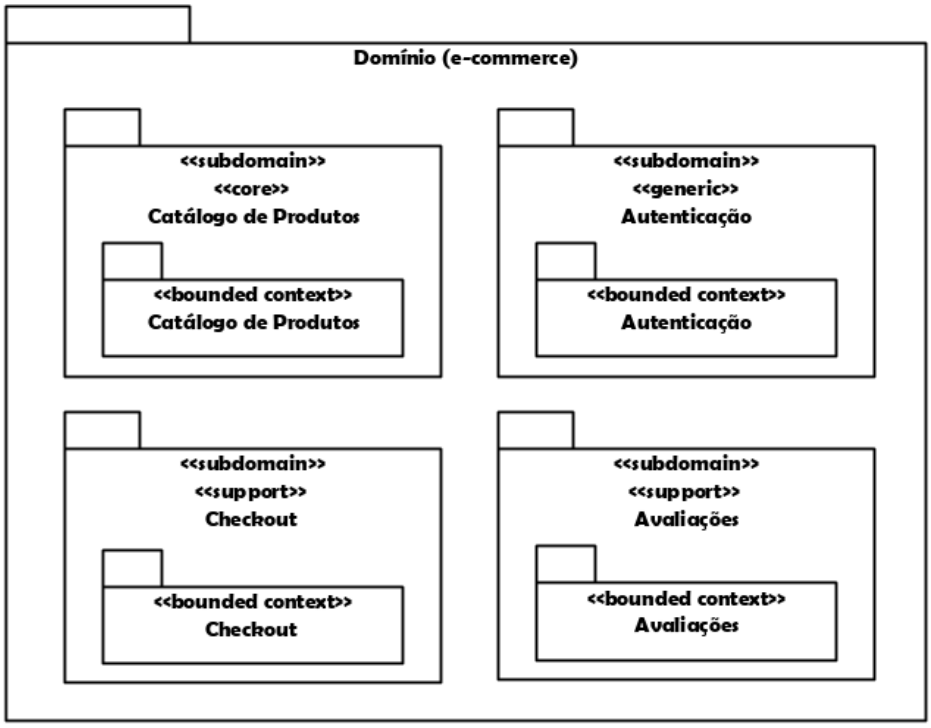


Figura 40 - Domínio, Subdomínios e *Bounded Contexts* da Prova de Conceito

No que se refere aos subdomínios apresentados na figura, retiram-se as seguintes ilações:

- O "Catálogo de Produtos" corresponde ao domínio principal do domínio, dado que o principal objetivo de uma aplicação *e-commerce* é precisamente apresentar produtos aos clientes, para que este os possa analisar e se assim desejar adquirir.

- Para tal, necessita primeiro de adicionar esses produtos ao carrinho que deve poder consultar e editar a qualquer momento. No entanto, fatores como as avaliações efetuadas a um produto podem influenciar o cliente na hora de clicar no botão de

adicionar ao carrinho de compras. Daí que ambos, isto é, "Avaliações" e "*Checkout*", possam ser considerados subdomínios de suporte.

- Por fim, apesar de se tratar de uma preocupação transversal, a existência de um sistema de autenticação não estando relacionada com detalhes específicos do negócio, pode basear-se num modelo/biblioteca existente, pelo que corresponde a um subdomínio genérico.

Quanto à missão, isto é, aquilo que se espera que os *bounded contexts* sejam capazes de proporcionar, estão descritas na Tabela 12.

Tabela 12 - *Bounded Contexts* da Prova de Conceito

BOUNDED CONTEXT	**MISSÃO**
CATÁLOGO	Listar os produtos existentes e apresentar os detalhes de um produto
AVALIAÇÕES	Apresentar avaliações relacionadas com o produto que o utilizador está a consultar (através da página de detalhes)
CHECKOUT	Fornecer uma boa experiência de *checkout*, neste caso permitir adicionar artigos ao carrinho e consultar/editar o mesmo
AUTENTICAÇÃO [12]	Permitir a um cliente registar-se e autenticar-se para poder adicionar artigos ao carrinho e fazer comentários a produtos

Tendo presente não só as missões definidas para cada *bounded context* e que os modelos que compõem o domínio "devem ser pequenos o suficiente para serem atribuídos a uma única equipa" (Avram & Marinescu, 2007), em contexto real, cada *bounded context* seria desenvolvido por uma equipa dedicada e especializada no subdomínio que o contém.

Em termos de interação entre MFEs, prevê-se a necessidade de interação entre diversos micro *frontends*, por exemplo, na adição de um produto ao carrinho de compras.

6.3 *Design* Arquitetural

Se relativamente ao aspeto visual dos diversos módulos aplicacionais não se preveem alterações significativas em função da técnica de integração, o mesmo não se pode afirmar relativamente à arquitetura, visto que está intimamente ligada não só à técnica em específico, mas também ao ambiente onde ocorre a integração.

[12] A autenticação é uma preocupação transversal a toda a aplicação, por isso, segundo os pressupostos da arquitetura de MFEs, num contexto real deveria ser desenvolvida ou por uma equipa dedicada a plataforma ou por uma das equipas de produto (consultar secção 4.4.3 para informação detalhada).

Perante um universo tão alargado de técnicas referidas neste estudo (consultar seção 5.1), que superam uma dezena, a implementação e experimentação na sua totalidade não seria exequível no âmbito desta prova de conceito.

Assim sendo, procedeu-se à seleção das técnicas a contemplar, com base nos seguintes critérios:

1. Devem ser abrangidas técnicas de integração quer *Client-Side*, quer *Server-Side*;

2. Deve ser dada prioridade a técnicas que melhor se ajustem ao desenvolvimento de aplicações *web* atuais;

3. Devem ser privilegiadas técnicas com algum processo de maturação ou que utilizem tecnologias inovadoras, para as quais existe documentação, suporte da comunidade e/ou se conhecem casos de adoção em contexto real.

Tendo em conta estes critérios, o autor propõe-se na POC a implementar as técnicas de "SPA Unificada com *App Shell* em 2 Níveis usando *Web Components*", "AJAX usando um servidor *web* partilhado" e "*Server Side Includes*".

Dado que a POC adotará a arquitetura orientada a MFEs, em termos arquiteturais corresponderá a uma aplicação organizada em camadas verticais, correspondentes aos MFEs previstos.

Visto que cada micro *frontend* será desenvolvido de forma *end-to-end*, numa perspetiva arquitetural de menor granularidade, isto é, de mais baixo nível, a respetiva camada englobará uma camada de interface com o utilizador, uma camada de micro serviços e uma camada de persistência de dados.

Posto isto, nas subseções seguintes serão introduzidas as arquiteturas idealizadas para cada uma técnicas contempladas na POC.

6.3.1 SPA Unificada com App Shell em 2 Níveis

No panorama atual, os utilizadores são cada vez mais exigentes com as aplicações usadas no seu quotidiano, esperando que seja apelativa, intuitiva, fluída e que reaja praticamente de imediato aos seus inputs.

Neste contexto, *frameworks* JS como React, Angular e Vue têm vindo a proliferar.

Tratando-se de aplicações isoladas, por norma SPAs, para serem integradas garantindo que as solicitações do utilizador são atendidas devidamente é necessária a existência de uma *app shell*.

No que concerne a arquitetura preconizada para a implementação desta técnica no contexto da POC, é apresentada na Figura 41 o diagrama de componentes, sendo que:

- O componente "*App Shell*" corresponde à aplicação-contentor da solução;
- A *App Shell* é responsável pela orquestração das diversas SPAs, que correspondem aos componentes Catálogo SPA, Avaliações SPA e Checkout SPA;
- Relativamente à "Autenticação" sendo transversal, optou-se por implementar os respetivos componentes visuais na própria *App Shell*;
- As SPAs de cada equipa incluem a UI, MS e DB relativa a cada MFE.

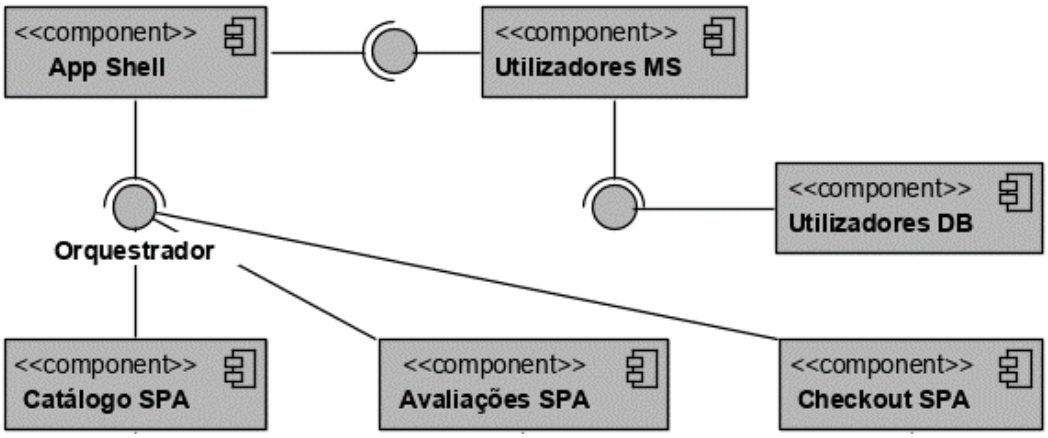


Figura 41 - Diagrama de Componentes da Técnica SPA Unificada

6.3.2 AJAX com Servidor *Web* Partilhado

Esta técnica de *Client-Side Integration* permite a integração de múltiplas páginas num único documento através de AJAX.

O conteúdo destas páginas é obtido através de um servidor *web* partilhado (componente *Web Server*) que interceta os pedidos HTTP e os remete para as aplicações das várias equipas com base na sua tabela de encaminhamento.

As aplicações respondem com fragmentos HTML, que apesar de passarem pelo servidor, seguem para o cliente (componente *Browser*) onde serão integrados (ver Figura 42).

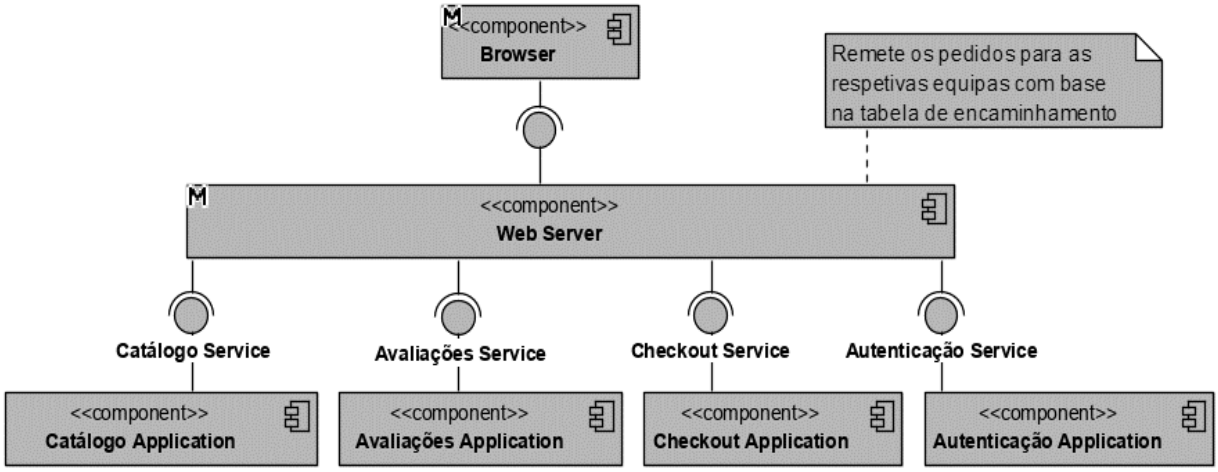


Figura 42 - Diagrama de Componentes da Técnica AJAX c/ Servidor *Web* Partilhado

6.3.3 *Server Side Includes*

Esta técnica de *Server-Side Integration* permite a integração de múltiplos fragmentos num único documento através de adição de diretivas SSI ao seu HTML. É uma opção certamente a equacionar em cenários em que o tempo de carregamento, SEO e acessibilidade pesam mais do que a interatividade.

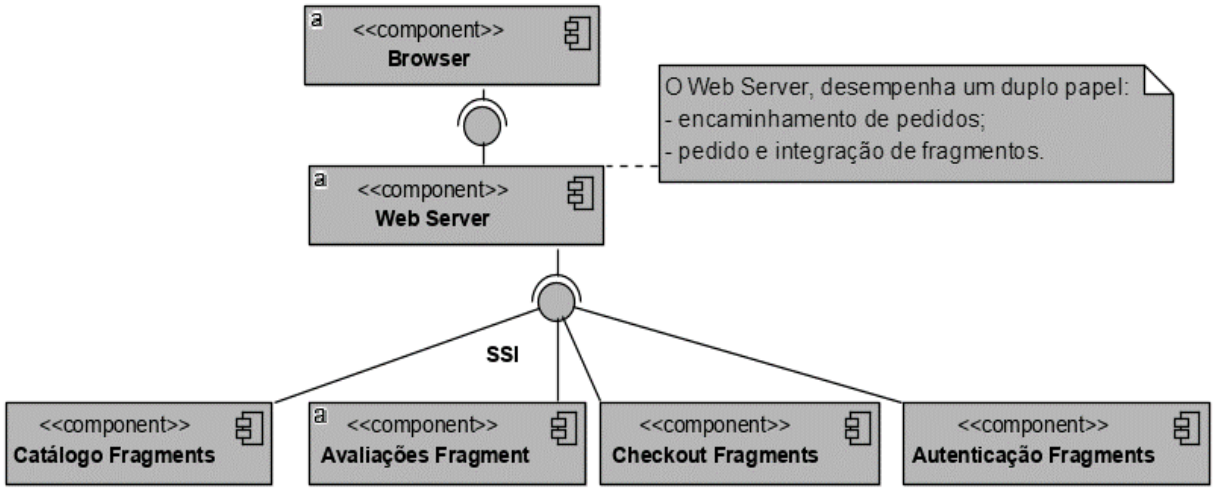


Figura 43 - Diagrama de Componentes da Técnica *Server Side Includes*

Supondo que se acede à página de detalhe de um produto (através do *Browser*), o servidor *web* partilhado analisa a rota e encaminha o pedido para a equipa responsável pelo catálogo. A aplicação gera o HTML para a página solicitada, incluindo diretivas SSI relativas aos fragmentos de comentários e do carrinho de compras e envia para o servidor. O servidor faz o *parsing* do HTML e ao detetar diretivas SSI extrai os URLS e solicita o conteúdo correspondente às equipas. Assim que todas as diretivas estiverem resolvidas, o servidor envia o documento HTML finalizado para o cliente para ser apresentado ao utilizador.

6.4 Implementação

Modelado o domínio e efetuado o *design* arquitetural da aplicação, procedeu-se à implementação da POC.

Apesar de inicialmente se ter previsto a implementação de várias versões, ao constatar que já dispunha de elementos suficientes para retirar conclusões, o autor deste documento decidiu implementar a POC aplicando apenas a técnica SPA Unificada com *App Shell* em 2 Níveis.

6.4.1 SPA Unificada com App Shell em 2 Níveis

Esta técnica de integração pressupõe a existência de uma aplicação centralizada na qual são integrados os vários micro *frontends* e são tratadas as *cross-cutting concerns*.

A *App Shell* é igualmente responsável pelo *routing*, ora para o router interno de cada equipa (*routing* com 2 níveis), ora para os fragmentos e páginas finais (*routing* com 1 nível).

Com vista a implementar uma aplicação orientada a MFEs adotando como técnica de integração a SPA Unificada com App Shell em 2 Níveis, descrita na seção 5.1.4, após alguma pesquisa o autor deste documento decidiu tirar partido da biblioteca Single-SPA.

Considerando os *bounded contexts* identificados anteriormente, a sua missão e as funcionalidades previstas e tendo como referencial o projeto-exemplo disponibilizado na documentação oficial do Single-SPA e desenvolvido em React, definiram-se as tecnologias a utilizar e procedeu-se ao levantamento dos MFEs necessários para desenvolver o *site* da Tellco.

6.4.1.1 Definição da *Stack* Tecnológica

Pretendendo o autor deste documento simular o desenvolvimento E2E, para retirar conclusões mais fidedignas da POC, apesar do foco incidir sobre a camada de *frontend*, também se procedeu ao desenvolvimento do *backend*, seguindo uma abordagem de micro serviços.

Para efeitos de controlo de versões, o código de cada micro serviço e de cada micro *frontend* foi colocado em repositórios alojados no Github.

Começando pela camada de *backend*, foram criados quatro micro serviços, mais precisamente, gestão de utilizadores (Kumar, 2020), catálogo de produtos, avaliações e carrinho de compras, cada um servido por uma base de dados independente, com exceção do serviço de carrinho de compras que além da base de dados de carrinho de compras também acede à base de dados de produtos, em conformidade com o padrão *Shared Database* (Richardson, 2018a).

Em termos tecnológicos, foi utilizado MongoDB nas bases de dados, que se encontram armazenadas no MongoAtlas, e Express, Typescript e Mongoose na implementação dos serviços, que foram *deployed* para o *Heroku*.

Do lado do *frontend*, foram usadas as seguintes tecnologias:

- single-spa conjuntamente com single-spa-layout para a integração dos micro *frontends*;
- react, como *framework* de implementação de cada micro *frontend*;
- SCSS na estilização (Dando, 2019);
- Jest, jasmine e enzyme nos testes automáticos;
- Webpack, import maps e SystemJS para gerir as dependências de cada micro *frontend*;
- Import-map-deployer para efetuar o *deployment* do *import map*;
- Circle-CI para a criação e execução da *pipeline* de CI e CD de cada micro *frontend*;
- Google-Cloud e JSDelivr como servidor de CDN para cada micro *frontend* e para cada dependência externa comum, respetivamente;

Relativamente às opções tomadas para o FE, a decisão de usar React prendeu-se com a familiaridade do autor deste documento com a tecnologia e por se tratar da *framework* mais amplamente utilizada, não só no projeto exemplo mais completo, mas também nos tutoriais de single-spa disponibilizados na documentação oficial, o que reduz a curva de aprendizagem inerente à aplicação de MFEs, já exacerbada pelo facto de se usar uma biblioteca recente para a concretizar.

A recomendação na documentação oficial foi também o critério de decisão que motivou a escolha de Webpack, *import maps*, System JS, import-map-deploy, Circle-CI, Google Cloud Platform e JSDelivr (Denning, 2019c, 2020a).

Relativamente aos estilos, a escolha recaiu por SCSS com o intuito de utilizando uma espécie de *namespace* garantir o isolamento.

Quanto às ferramentas de testes selecionadas correspondem às aplicadas em contexto profissional pelo autor.

6.4.1.2 Identificação de Micro *Frontends* e *Cross-Cutting Concerns*

Considerando os *bounded contexts* identificados anteriormente, a sua missão e as funcionalidades previstas e tendo como referencial o projeto-exemplo disponibilizado na documentação oficial do Single-SPA e desenvolvido em React, definiram-se as tecnologias a utilizar e procedeu-se ao levantamento dos MFEs necessários para desenvolver o *site* da Tellco.

Os micro *frontends* identificados estão apresentados na tabela abaixo.

Tabela 13 – Micro *Frontends* da aplicação Tellco usando Single-SPA

MICRO *FRONTEND*	***KEY* NO *IMPORT MAP***	**TIPO DE MFE**	**EQUIPA RESPONSÁVEL (EM TEORIA)**
NAVBAR	@msc-tellco/navbar	*Application*	Equipa de Gestão de Utilizadores
CATALOGUE	@msc-tellco/catalogue	*Application*	Equipa de Catálogo
REVIEWS	@msc-tellco/reviews	*Application*	Equipa de Avaliações
SHOPPING-CART	@msc-tellco/shopping-cart	*Application*	Equipa de Checkout
STYLEGUIDE	@msc-tellco/styleguide	*Utility-Module*	Equipa de Avaliações

Dado o número limitado de requisitos funcionais e não funcionais desta POC, não haveria necessidade de em contexto real criar uma equipa dedicada a manter a infraestrutura.

Alternativamente, como sugerido na secção 4.4.3, poder-se-ia proceder à distribuição de *cross-concerns* como a Autenticação, *Styleguide* e *Root Config* pelas várias equipas.

Assim sendo, o micro serviço de gestão de utilizadores e o micro *frontend* de Navbar, seriam competência da equipa de gestão de utilizadores.

Quanto ao *styleguide* ficaria a cargo da equipa de Avaliações, já que detém os componentes usados em mais pontos da aplicação, enquanto a *root config* poderia ser gerida pela equipa de Catálogo, uma vez que é responsável pela página inicial da aplicação *e-commerce*.

No que concerne o *routing*, o primeiro nível, responsável por adicionar ao DOM a *application* correspondente à rota, é gerido pelo single-spa-layout ao nível da *root config*, conforme apresentado no Anexo B, enquanto o segundo nível é gerido internamente a cada *application* pelo *router* do React.

Para uma melhor compreensão dos componentes e páginas desenvolvidos, podem ser consultadas no Anexo C imagens dos mesmos.

6.4.1.3 *Root Config*

Relativamente ao *root config* da aplicação Tellco, no ficheiro HTML foi incluído um *template* no qual são definidas as aplicações a ser *mounted* no DOM em função da rota, com recurso ao single-spa-layout, que são registadas no ficheiro JS.

6.4.1.4 Micro *Frontend* de *Navbar*

O micro *frontend* de *Navbar* é o único transversal a toda a aplicação. Contempla os componentes diretamente ligados à autenticação, nomeadamente as modais de *signin*/*login*, *signup* e *signout* a par da *Navbar*, a partir do qual podem ser acedidos.

Além destes componentes, a *navbar* integra o componente *"CartWidget"*, proveniente do MFE de carrinho de compras, onde são apresentados o total de itens atualmente no carrinho.

Fazendo os componentes relacionados com a autenticação parte integrante do MFE da *navbar*, é neste MFE que são processados os eventos de *signin*, *signup* e *signout*.

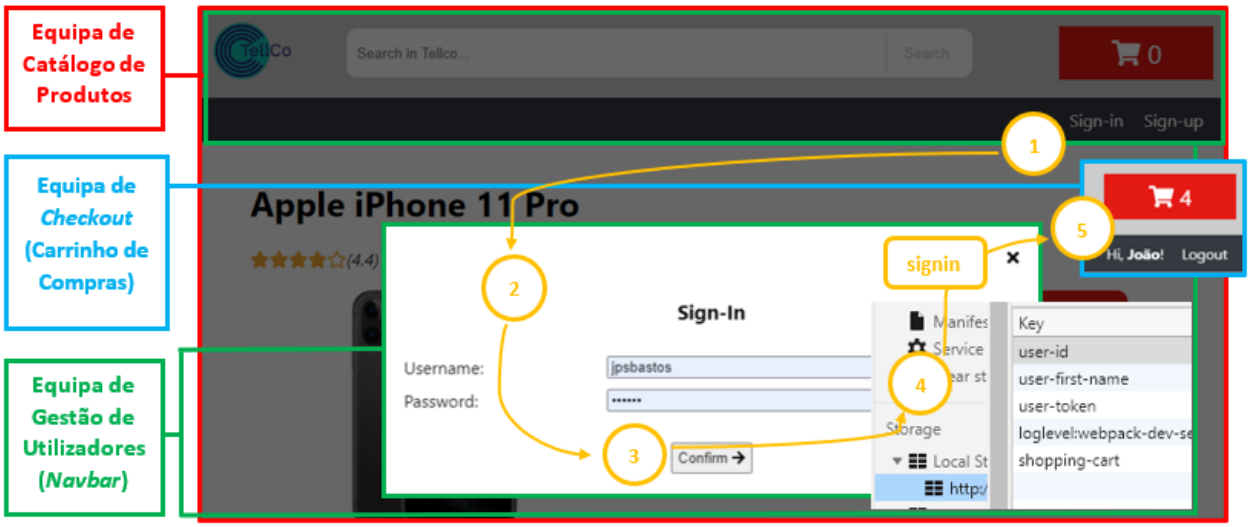


Figura 44 – Processo de *Login* na Aplicação Tellco

Relativamente ao processo de *signin*, apresentado na Figura 44, envolve as seguintes etapas:

1. O utilizador clica no botão "Sign-in" e é aberta a modal apresentada na figura;
2. Introduz o nome de utilizador e palavra-passe;
3. Ao clicar no botão "Confirm" é efetuada uma chamada ao serviço de gestão de utilizadores, que devolve os dados do utilizador, entre os quais o id e um *token* JWT;
4. Completado o pedido, esta informação é armazenada na *local storage* nas *keys* "user-id" e "user-token" respetivamente, para que os restantes MFEs tenham acesso a esta informação;
5. É instanciado um *Custom Event* do tipo "signin", que visa informar o "*CartWidget*", de que o utilizador se encontra autenticado para que este possa apresentar o total de itens que possui no seu carrinho.

6.4.1.5 Micro *Frontend* de Avaliações

O micro *frontend* de Avaliações contém apenas componentes não acessíveis através de rota, nomeadamente a "ReviewList" e o "StarRating".

Dado que todos os componentes foram escritos em React, a mesma *framework* do micro *frontend* que o incorpora, não foi necessário implementar este micro *frontend* como uma *parcel*.

6.4.1.6 Micro *Frontend* de Catálogo

Relativamente ao micro *frontend* de catálogo, é acessível a partir da *base route* e inclui os componentes "Card", "ImagePreview", a página inicial e a página de detalhe de um produto.

Na página inicial, carregada ao aceder à rota principal, é apresentado o catálogo de produtos, que inclui uma instância de "Card" por cada produto apresentado, que por sua vez, incorpora um componente "StarRating", proveniente do micro *frontend* de "Reviews".

Para poder ser incorporado, este componente React criado no micro *frontend* de Avaliações é exportado no ficheiro em que é definido e importado desse micro *frontend* usando ES Modules e tirando partido do SystemJS (ver excerto de código abaixo).

```
import { StarRating } from "@msc-tellco/reviews"
```

Código 11 – Importação de componentes de outros micro *frontends*

Dado que em todas as situações em que é usado o componente que o integra detém informação relativa ao produto, nomeadamente do *rating*, este é passado diretamente como atributo.

Quanto à página de produto, acessível através da sub-rota "products/:id", apresenta além de fotografias e informação técnica do próprio produto, o botão de adicionar o produto ao carrinho (componente "AddCartButton" importado do micro *frontend* de Carrinho de Compras), o *rating* (através da incorporação do componente "StarRating") e a lista de todas as avaliações (por via da incorporação do componente "ReviewsList", proveniente do micro *frontend* de "Reviews").

A adição de um novo produto ao carrinho requer que o utilizador se encontre autenticado, uma vez que para efeitos de simplificação existe uma relação de um para um entre utilizador e carrinho de compras.

Além disso, sendo um processo que envolve a interação entre página de produto e botão de adicionar ao carrinho e entre botão e *widget* de carrinho de compras para atualizar o total de itens no carrinho, requer a utilização de mecanismos de comunicação.

Sendo a relação entre página e botão do tipo "página para fragmento", a página de detalhe ao integrar este fragmento, passa-lhe informação, neste caso o id do produto, por atributo.

A adição de artigos ao carrinho é um processo que envolve a comunicação entre vários micro *frontends*. Ora, a Figura 45 apresenta as etapas e os MFEs envolvidos, que serão esmiuçados de seguida.

Figura 45 – Processo de Adicionar Novo Produto ao Carrinho de Compras na Aplicação Tellco

1. O utilizador procede ao *login* (caso ainda não esteja autenticado);

2. Ao clicar no botão "Add to Cart", como a relação entre o botão e o *widget* é do tipo "fragmento para fragmento", a comunicação é estabelecida por via da instanciação no botão de um *Custom Event* do tipo "update-cart", enviando como detalhe o id do produto;

3. O *widget* está à escuta de eventos deste tipo que, quando são despoletados, desencadeiam um pedido POST ao serviço de carrinho de compras para adicionar o produto;

4. *Completado* com sucesso o pedido, o número de itens contidos no carrinho é atualizado.

6.4.1.7 Micro *Frontend* de *Styleguide*

O micro *frontend* de *styleguide* visava centralizar estilos transversais à aplicação como cores e fontes.

Contudo, o autor deste documento não conseguiu que os restantes micro *frontends* conseguissem resolver as variáveis declaradas nos ficheiros SCSS do *styleguide*, pelo que este acabou por não ser utilizado e os valores destas variáveis replicados nos vários micro *frontends*.

6.4.1.8 Micro *Frontend* de Carrinho de Compras

O micro *frontend* de carrinho de compras possui apenas uma página, acessível a partir da rota "/cart", e detém os componentes "AddCartButton" (um botão que permite adicionar um produto cujo id é recebido como *prop*) e "CartWidget" (*widget* onde é apresentado o número de itens atualmente no carrinho de compras).

Quer a página, quer a utilização e/ou atualização destes componentes requerem que o utilizador esteja autenticado, confirmando o estado do utilizador com base na existência da key "user-id" na *local storage*. Caso o utilizador não esteja autenticado, estes componentes emitem um *Custom Event* do tipo "unauthorized" que ao ser escutado pela *navbar* despoletam a abertura da modal de login.

Assim, quando se acede à rota "/cart" a página de carrinho começa por verificar se o utilizador está autenticado. Em caso afirmativo, realiza um pedido HTTP ao serviço de carrinho para obter a informação relativa aos produtos existentes no carrinho, que são depois apresentados.

6.4.1.9 *Deployment* dos Micro *Frontends*

O *deployment* da SPA seguiu as recomendações da documentação oficial.

Nesse sentido, com recurso ao import-map-deployer procedeu-se ao *deployment* da primeira versão do *import map* para o Google Cloud Platform (GCP) (Google, 2020b).

Tratando-se o import-map-deployer de um Docker *container*, apesar de poder ser executado usando uma instalação local de Docker, para simular um ambiente de produção a execução deve ocorrer na *cloud* (Denning, 2020a). Para tal, foi criado na GCP um *cluster* de Kubernetes.

Alojado o import-map-deployer, procedeu-se também na GCP à criação de um *bucket*, denominado msc-tellco-single-spa, no qual foi inicialmente colocado o ficheiro import-map.json, que iria alojar o *import map*, ainda sem quaisquer *keys*.

Relativamente às *keys* relativas a dependências externas comuns aos vários micro *frontends*, como react, react-dom, axios ou dayjs, foram adicionadas através de pedidos HTTP à API do import-map-deployer (Denning, 2020h).

Quanto às *keys* relativas a cada um dos micro *frontends* que compõem a aplicação, foram adicionadas/atualizadas posteriormente aquando do seu *deploy* para o *bucket*.

O *deploy* de um MFE, como referido anteriormente, envolve duas fases: a publicação da nova versão do ficheiro JS correspondente e a atualização do valor da *key* no *import map* (Denning, 2020a), tendo-se optado pelo CircleCI para efetuar o *deploy* numa *pipeline* de CI e CD.

Ora, a integração do CircleCI diretamente com o repositório faz com que a *pipeline*, que contém os jobs apresentados na figura seguinte, seja despoletada sempre que sejam detetadas alterações no *branch* especificado.

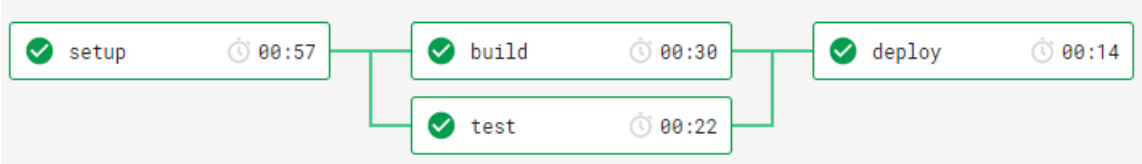


Figura 46 – Jobs da *Pipeline* de CI e CD de Cada Micro *Frontend* da Aplicação Tellco

Conforme demonstrado na figura, a *pipeline* abrange quatro jobs: *setup, build, test* e *deploy*.

Estes jobs são executados de forma sequencial, exceto os jobs de *test* e *build*, que correm em paralelo.

No job de *setup* são descarregadas as dependências através do comando npm install.

No job de *build* a aplicação é compilada (usando Webpack) enquanto no job de *test* são executados os testes unitários e o lint.

Por fim no job de *deploy*, tem lugar o *deployment*, que usando CircleCI é bastante rápido, permitindo agilizar o lançamento de novas versões dos micro *frontends*.

6.4.1.10 Principais Vantagens

Com base na POC desenvolvida, destacam-se como benefícios da utilização desta abordagem a capacidade de evolução independente, a experiência de desenvolvimento local, fomentar uma cultura de automação, o facto de usando SystemJS ser possível descarregar módulos ou dependências em micro *frontends* distintos uma única vez (com recurso a um eficiente sistema de cache) e a obtenção de uma *codebase* bastante modular, testável e com baixo acoplamento.

Quanto à capacidade de evolução independente, testemunhou-se a rapidez com que é executada a *pipeline* de integração e se efetua o *deployment*, o que permite decrementar os ciclos de lançamento.

Em termos de experiência para o *developer*, o autor deste documento ficou agradavelmente surpreendido com as potencialidades oferecidas pela adoção do single-spa seguindo a configuração recomendada, que permitem isolar falhas e diminuir os ciclos de desenvolvimento.

Verificou-se ainda que usando *import maps* era possível alterar em função do ambiente de execução os URLs a partir dos quais se obtêm os micro *frontends* da aplicação, o que permite que em contexto de desenvolvimento local, os *developers* apenas precisem de executar localmente o seu micro *frontend*, uma vez que os URLs dos restantes MFEs podem apontar para produção (Denning, 2019b).

Além disso, a equipa do single-spa criou o single-spa-inspector, uma extensão passível de ser usada nas *developers tools* do Google Chrome, que permite apresentar as *applications* registadas, apresentar o seu estado, fazer-lhes *mount*/*unmount*, fazer *overlay* das *applications* e até mesmo fazer *override* ao import-map (Denning, 2020g).

6.4.1.11 Principais Desafios e Limitações

No decorrer do desenvolvimento da POC, o autor foi confrontado com alguns desafios. Desde logo, o facto de o single-spa ser uma biblioteca recente e se encontrar em fase de desenvolvimento, fez com que a documentação ao dispor se resumisse à oficial, não havendo suporte da comunidade, exceto por via do canal de Slack criado pela equipa do single-spa.

Contudo, é de enaltecer a qualidade da documentação já produzida, redigida de forma objetiva e com recurso a excertos de código, que é complementada por exemplos de projetos implementados segundo esta abordagem e usando diferentes *frameworks* JS e por tutoriais e da autoria dos principais responsáveis pela biblioteca, nos quais são abordadas questões como a configuração de uma aplicação assente em single-spa e o *deployment* dos vários MFEs.

Tecnicamente falando, apesar dos ganhos obtidos com o uso de SystemJS, verificou-se, por vezes, que quando o projeto era compilado eram apresentados na consola alguns avisos relacionados com o tamanho da *bundle* e *alguma* lentidão no carregamento da página, que certamente poderia ser colmatado se o single-spa fornecesse opções que permitissem o *Progressive Enhancement*.

Além disso, apesar de na documentação oficial se anunciar que aquando do registo das várias *applications* é possível especificar conteúdo alternativo caso ocorra algum erro no seu carregamento, usando a versão mais recente do single-spa-react o autor foi confrontado com erros de Typescript, que após uma análise do package nos node_modules acredita deverem-se ao ficheiro de declarações não contemplar o atributo que supostamente deveria ser usado para definir este conteúdo. Assim sendo, constatou-se que perante a indisponibilidade ou falha de carregamento de um micro *frontend* a disponibilidade da aplicação ficou comprometida.

Outra das dificuldades sentidas prendeu-se com a criação do micro *frontend* de *Styleguide*, cujo objetivo seria o de centralizar cores, fontes e regras de estilo comuns a toda a aplicação. Apesar de se ter conseguido adotar o pré-processador SASS e isolar os estilos dos vários componentes e páginas, não se conseguiu que as variáveis e regras declaradas nos ficheiros de estilização do *Styleguide* fossem reconhecidas pelos MFEs, o que fez com que fossem replicadas.

Por fim, foi identificado um potencial problema a longo prazo, não relacionado especificamente com a utilização do single-spa, mas da adoção da arquitetura de micro *frontends*, sempre que fragmentos de diferentes micro *frontends* necessitarem de comunicar entre si.

Basicamente, requerendo este tipo de comunicação o uso de *Custom Events* ou similar, o autor alerta para a necessidade de não só atribuir aos tipos de eventos nomes com significado para o negócio, mas também de definir os tipos de eventos necessários para estabelecer a comunicação no âmbito do contrato firmado entre as equipas, ou de pelo menos se definir alguma forma de documentar explicitamente os eventos existentes.

Deste modo, pode-se evitar a criação de novos eventos quando já existam opções que permitam dar resposta aquilo que se pretende.

6.4.2 AJAX com Servidor *Web* Partilhado

O AJAX com Servidor *Web* Partilhado corresponde à técnica que se previa adotar na segunda versão da POC, que apresentar de estar inicialmente prevista, não foi implementada.

No entanto, nesta secção é descrita uma possível abordagem e sendo esta uma das técnicas de integração de MFEs mais utilizadas (C. Jackson, 2019), os relatos da sua adoção possibilitam a identificação das principais vantagens e desvantagens.

6.4.2.1 Possível Abordagem

O AJAX confere a cada equipa o poder de decidir a tecnologia na qual vai criar os seus fragmentos e páginas. Assim sendo, poderia utilizar-se JavaScript puro com o *template* a ser definido num ficheiro HTML externo ou uma biblioteca JS mais leve, uma vez que a responsabilidade pelo *routing* está a cargo de um servidor *web* centralizado.

Para desempenhar a função de servidor *web* centralizado poderia adotar-se o NGINX, dado que é OSS, existe um amplo suporte da comunidade e é bastante configurável.

6.4.2.2 Principais Vantagens

Por oposição à técnica aplicada na versão anteriormente, na qual a renderização, composição e *routing* eram tratadas do lado do cliente, o uso de AJAX com um servidor *web* partilhado, permite delegar o *routing* nesse servidor.

Relativamente à renderização e composição, recorrendo à transclusão do lado do cliente o AJAX consegue carregar fragmentos e injetá-los num determinado ponto do DOM, concedendo às equipas o poder de decidir como e quando serão renderizados e integrados os MFEs.

Cam Jackson afirma mesmo que “a flexibilidade que esta abordagem oferece, combinada com a capacidade de realizar *deploys* independentes”, a torna a sua escolha preferencial e “uma das mais amplamente utilizadas” (C. Jackson, 2019) .

Quanto à complementaridade do AJAX com um servidor *web* centralizado, permite colocar debaixo do mesmo domínio os micro *frontends* e serviços desenvolvidos, uma vez que interceta todos os pedidos efetuados à aplicação e com base nas rotas definidas nas tabelas de encaminhamento, encaminha o pedido para o respetivo MFE.

Além disso, a existência desta infraestrutura permite dar uma resposta centralizada às *cross-cutting concerns*, possibilita um carregamento mais rápido (que pode ser potenciado com recurso a cache) e oferece bons resultados em matéria de SEO (Geers, 2020a).

6.4.2.3 Principais Desafios e Limitações

A utilização de AJAX como técnica de integração acarreta a perda do isolamento e a necessidade de definir *namespaces* para evitar colisões de estilos e caso a aplicação requeira algum nível de

interatividade vai implicar a realização de sucessivos *round-trips* ao servidor para gerar uma versão atualizada do HTML apresentado ao utilizador (Geers, 2020a).

6.4.3 *Server Side Includes*

O SSI era a técnica de integração selecionada para a terceira versão da POC, que como referido anteriormente não foi implementada. Contudo, sendo uma técnica consolidada no mercado, foi possível chegar a uma possível abordagem, vantagens e desvantagens da sua aplicação.

6.4.3.1 Possível Abordagem

Como referido na secção 5.1.3.1, o SSI é uma das técnicas de renderização e integração no servidor, que neste caso se socorre de diretivas SSI para carregar conteúdo dinâmico em documentos HTML (Geers, 2020a). Sendo suportado pelos principais servidores *web*, isto é, NGINX, Apache e IIS, a escolha poderia recair sobre NGINX.

O motivo prende-se com poder ser usado como *reverse-proxy*, isto é, permitir efetuar o encaminhamento de pedidos com base no caminho especificado no URL, e carregar e integrar fragmentos num documento HTML que é depois enviado para o *browser*.

Além disso, de entre outras possibilidades oferecidas, o NGINX pode ser usado como *Load Balancer*, algo particularmente útil em períodos de sobrecarga (Linuxwize, 2019).

6.4.3.2 Principais Vantagens

Conforme abordado na secção 5.1.3.1, a utilização de diretivas SSI para efetuar a integração de micro *frontends* representa uma solução de primeira linha para alcançar melhores tempos de carregamento e apresentar em menos tempo uma página minimamente funcional no *browser*, seguindo os pressupostos do *Progressive Enhancement* (Geers, 2020b).

A par disso é facilmente indexado pelos *crawlers (*SEO) e permite assegurar a integridade e disponibilidade da aplicação perante a indisponibilidade ou falha no carregamento de fragmentos, que pode ser efetuado em paralelo ou *lazy loaded* (Geers, 2020a).

6.4.3.3 Principais Desafios e Limitações

Apesar das vantagens anteriormente elencadas, o SSI é difícil de configurar e não é uma opção viável para aplicações em que a rápida resposta às interações com o utilizador seja um requisito.

Por outro lado, o SSI não oferece isolamento no *browser* e a experiência de desenvolvimento local torna-se mais complexa ao depender da execução local de um servidor *web* com suporte a SSI (Geers, 2020a).

7 Experiências e Avaliação

Este capítulo visa:

- Formular as hipóteses de investigação que permitirão validar o tema do estudo;
- Identificar os indicadores de validação das hipóteses e as fontes de informação;
- Descrever os métodos de avaliação das hipóteses;
- Avaliar as experiências realizadas.

7.1 Definição de Hipóteses

Como referido anteriormente, o principal objetivo desta dissertação consiste no estudo da adoção de micro *frontends* no desenvolvimento *web*, com vista a determinar se a adoção desta abordagem de organização de *software* e equipas é viável, em que situações fará sentido adotá-la e quais as suas principais vantagens e desvantagens.

Os proponentes desta abordagem afirmam que a sua adoção permite agilizar o processo de desenvolvimento, efetuar *releases* independentes mais cedo e com maior frequência, maior disponibilidade das aplicações e dado o cariz autocontido dos micro *frontends* uma maior manutenibilidade, testabilidade, alta coesão e baixo acoplamento.

Assim, foram formuladas as seguinte hipótese nula e alternativa:

Hipótese nula (H0): A adoção de uma arquitetura de micro *frontends* não permite assegurar um processo de desenvolvimento *web* mais ágil do que as abordagens mais convencionais.

Hipótese alternativa (H1): A adoção de uma arquitetura de micro *frontends* permite um processo de desenvolvimento *web* mais ágil do que as abordagens mais convencionais.

7.2 Indicadores e Fontes de Informação

Nesta secção serão identificados os indicadores de avaliação das hipóteses, bem como as fontes de informação que permitirão proceder a esta apreciação.

7.2.1 Indicadores de Avaliação

Os indicadores de avaliação permitem corroborar ou refutar hipóteses, aferir o cumprimento dos objetivos traçados e determinar os contributos de um projeto. Neste caso serão aplicados os seguintes:

- Integrabilidade: avaliar a capacidade de integração dos micro *frontends* entre si e com a aplicação integradora;
- *User Experience*: aferir a consistência e coerência visual entre as *UIs* dos vários MFE*s*;
- Evolução independente: avaliar a capacidade dos micro *frontends* evoluírem de forma autónoma, isto é, poderem ser desenvolvidos de forma independente e lançados de forma isolada através da respetiva *pipeline* de integração;
- Autocontenção e robustez: avaliar o nível de acoplamento, separação de responsabilidades e resiliência/tolerância a falhas;
- Ciclos de desenvolvimento/*release*: avaliar o impacto da adoção de micro *frontends* nos tempos de desenvolvimento e lançamento de novas funcionalidades e/ou versões;
- Operacionalidade: avaliar o impacto da adoção de micro *frontends* na organização, nomeadamente no *throughput*, comunicação e coordenação das equipas;
- *Performance*: avaliar a *performance* da POC em termos de tempos de carregamento inicial, resposta aos *inputs* do utilizador, resposta a atrasos ou falhas e dimensão do *bundle* gerado.

7.2.2 Fontes de Avaliação

As fontes de avaliação permitem medir os indicadores identificados anteriormente. Este projeto contempla as seguintes:

- Conclusões retiradas da prova de conceito;
- Questionários direcionados a profissionais da área do desenvolvimento de *software*, estudantes de engenharia informática ou similar e entusiastas da temática;
- Casos de adoção de micro *frontends* em contexto real.

7.3 Métodos de Avaliação

Os métodos de avaliação são parte integrante do processo de avaliação de um projeto.

Consistem num conjunto de atividades que tendo presente os critérios/indicadores definidos anteriormente permitem proceder à avaliação das hipóteses formuladas e verificar qual é a que se verifica. Neste contexto, serão aplicados métodos baseados nas conclusões da POC, questionários e casos de estudo.

7.3.1 Conclusões retiradas da prova de conceito

Sendo o objetivo de uma prova de conceito o desenvolvimento e implementação de uma determinada ideia ou conceito com vista a aferir com base nos objetivos definidos para a POC a viabilidade, a própria POC constitui um método de avaliação.

A POC deste projeto visa determinar os impactos da arquitetura de micro *frontends* no processo de desenvolvimento *web*. Assumindo a integração dos MFEs um papel preponderante na adoção desta arquitetura, os MFEs da POC serão integrados aplicando um conjunto de técnicas selecionadas, que serão colocadas em práticas a fim de se retirarem conclusões relativamente a vantagens, desvantagens e cenários em que façam sentido.

Além disso, a POC em si destina-se a avaliar se de facto enveredar por este caminho permite um processo de desenvolvimento mais ágil, validando, dessa forma, as hipóteses enunciadas.

7.3.2 Questionários

Findo o desenvolvimento da POC serão desenhados questionários direcionados a empresas de desenvolvimento de *software*, profissionais, estudantes ou entusiastas desta área que já tenham aplicado ou pelo menos ouvido falar do conceito e tenham uma opinião informada sobre o mesmo.

Estes inquéritos deverão contemplar um conjunto de questões que permitam, com base no *feedback* dos inquiridos, identificar quais os princípios subjacentes a esta arquitetura que foram verificados, principais vantagens e desvantagens, técnicas mais populares e qual o seu grau de adequação aos requisitos, bem como as repercussões que a adoção desta abordagem teve do ponto de vista organizacional.

7.3.3 Casos de Adoção de Micro *Frontends*

Prevê-se ainda a identificação e análise de casos de adoção de micro *frontends* em contexto real, com vista a determinar o grau de satisfação dos principais intervenientes com esta abordagem e a identificar as principais mais-valias e lacunas.

7.4 Avaliação de Experiências

A aplicação dos métodos de avaliação descritos na secção anterior, mais concretamente o desenvolvimento da POC, a realização de um questionário e a análise de casos de adoção de micro *frontends* em contexto real, permitirão aferir a viabilidade da adoção desta abordagem no desenvolvimento *web*, em que contextos se assume como uma mais-valia e quais as suas principais vantagens e desvantagens.

Desse modo, com base na avaliação destas experiências será possível corroborar ou refutar as hipóteses formuladas.

7.4.1 Prova de Conceito

Relativamente à prova de conceito, como se comprova na secção 6.4.1 foi possível proceder à implementação de uma solução de acordo com o *design* arquitetural apresentado na secção 6.3.1.

Adotando "SPA Unificada com *App Shell* em 2 níveis" como técnica de integração de micro *frontends* foi criada uma aplicação *e-commerce* simplificada dividida em 6 subaplicações, das quais quatro correspondem aos *bounded contexts* identificados na secção 6.2, uma à tentativa de criar um *utility module* destinado a fornecer um guia de estilos transversal à aplicação e outra à *root config*.

No single-spa a *root config* atua como *app shell*, uma vez que é responsável pelo registo das várias *applications* e pela sua injeção/remoção do DOM em função da rota, delegando o *routing* de 2º nível nas *applications*.

Tratando-se a POC de uma aplicação *e-commerce* em que o desenvolvimento de diferentes funcionalidades, entre as quais o catálogo de produtos, que detém o serviço e BD dos produtos, e o carrinho de compras, que possui o serviço e BD do carrinho de compras, está a cargo de equipas diferentes, estas aplicações necessitam forçosamente de comunicar.

Recorrendo aos mecanismos de comunicação descritos na seção 5.4, foi possível transmitir informação entre fragmentos, páginas e fragmentos e vice-versa e notificar os vários micro *frontends* aquando da ocorrência de eventos aos quais tivessem subscritos. Além disso, foi possível partilhar os dados da autenticação por via de *local storage*.

Assim, tendo em conta que POC implementada vai de encontro à missão traçada para cada *bounded context* e dá suporte a todas as funcionalidades previstas, é seguro afirmar que a adoção de micro *frontends* é viável e adequada a uma aplicação *e-commerce*.

Relativamente às vantagens, detalhadas na seção 6.4.1.10, destacam-se como a capacidade de evolução independente, a cultura de automação, a rápida resposta às ações do utilizador e a obtenção de módulos de código bem delimitados, facilmente testáveis e escaláveis e que respeitam o SRP e a redução da dimensão do *bundle*.

Quanto aos principais desafios e limitações encontrados, explanados na secção 6.4.1.11, apesar do tempo investido em pesquisa e visualização de tutoriais relacionados com a configuração recomendada na documentação oficial do single-spa, o autor foi confrontado com a escassez de informação/documentação para *use cases* não triviais como a implementação de autenticação num contexto aplicacional distribuído e a criação de estilos globais para toda a aplicação, que não lhe permitiram criar variáveis de estilo globais e reutilizáveis usando SCSS.

Além disso, verificou-se alguma demora no carregamento inicial da página e a incapacidade da biblioteca em lidar com erros ou indisponibilidade dos micro *frontends*, que acabavam por colocar em causa a disponibilidade e integridade da aplicação perante falhas ou erros.

7.4.2 Questionários

Para aferir o grau de conhecimento da abordagem, a sua viabilidade e determinar as principais vantagens e desvantagens da sua adoção, foi disponibilizado um questionário, que pode ser consultado integralmente nos anexos, direcionado a profissionais do ramo de desenvolvimento de *software*, estudantes de engenharia informática, ciências da computação ou equiparado e autodidatas com interesse na área.

Este questionário foi desenvolvido com recurso à plataforma Google Forms, dada a simplicidade que oferece em termos de estruturação e de tratamento de resultados.

Após ter sido alvo de revisão pelo autor em reuniões com o seu orientador, dois dos seus superiores hierárquicos, um *Product Owner* e uma *Designer*, o questionário foi divulgado em contexto empresarial, em reuniões e por via de canais como o Yammer e o Teams, e nas redes sociais, mais especificamente no LinkedIn e grupo de Facebook dos estudantes de Engenharia Informática do ISEP.

Assim, entre 19 e 26 de junho do presente ano, foram recolhidas um total de 70 respostas validadas, na esmagadora maioria submetidas por profissionais da área de desenvolvimento de *software* (cerca de 86%), dos quais 44% exerce atividade há pelo menos 5 anos.

Em termos estruturais, o questionário está dividido em treze secções, perfazendo um total de 58 questões.

Destas secções, duas são meramente informativas, pois o questionário é iniciado com uma secção de introdução e termina com uma secção de agradecimento.

Das 11 secções que contém perguntas, 10 têm acesso restrito, sendo que destas 5 apenas podem ser respondidas por profissionais da área de desenvolvimento de *software* e 4 destinam-se aos inquiridos que possuam um perfil mais técnico (independentemente de serem estudantes, *developers*, arquitetos de *software*, *tech leaders* ou meros interessados na temática) e a secção 12, designada "Vantagens e Desvantagens da Adoção de Micro *Frontends*", pode ser respondida por trabalhadores do ramo (equipa de produto, *tech leaders,* arquitetos*, project managers, team managers*), estudantes e autodidatas.

Esta gestão de acesso às secções em função do perfil é efetuada com recurso a questões de controlo que conduzem o inquirido pelo questionário com base na sua resposta.

Dada a extensão do questionário, motivada pelo facto de se tratar de um tema recente, nesta avaliação serão analisadas apenas as questões consideradas mais pertinentes pelo autor para retirar conclusões.

No que concerne as secções destinadas apenas aos inquiridos com atividade profissional, foram recolhidas 60 respostas e selecionadas as questões abaixo.

Assim sendo, relativamente à frase "Considera mais vantajoso que as equipas de desenvolvimento sejam…", 78% dos inquiridos completou-a com "Multidisciplinares" enquanto apenas 22% com "Especializadas numa determinada camada aplicacional"

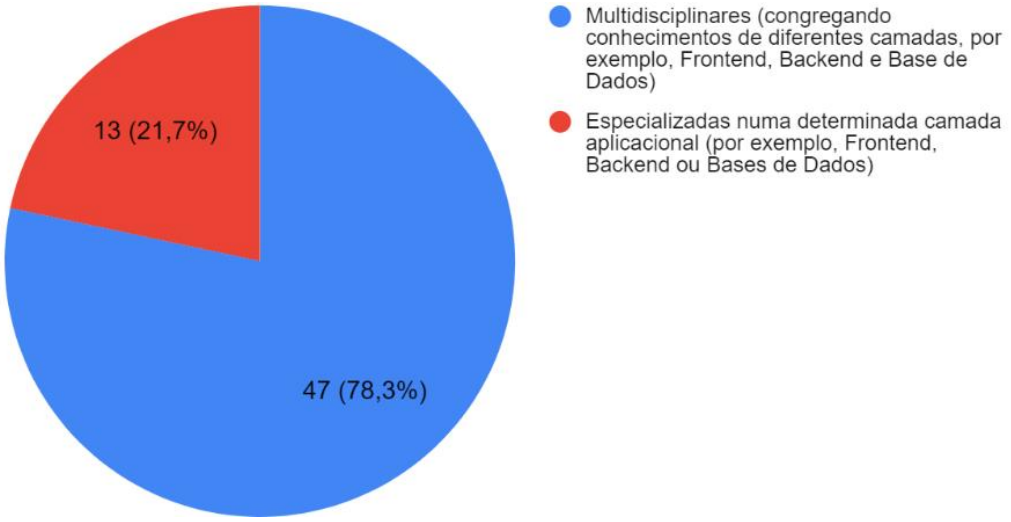


Figura 47 – Gráfico relativo à frase "Relativamente às equipas de desenvolvimento, considera mais vantajoso que sejam…"

Daqui se conclui que uma larga maioria considera que a organização em equipas multidisciplinares é uma mais-valia.

Sendo esta uma questão de controlo, os inquiridos que responderam multidisciplinares foram encaminhados para a secção seguinte (a 5ª) e os restantes para a secção 12.

Tendo sido solicitado aos inquiridos para classificarem numa escala de 1 (Discordo Totalmente) a 5 (Concordo Plenamente) a afirmação "A especialização destas equipas em determinadas áreas de negócio permite que identifiquem mais facilmente o que pode ser adicionado ao produto para gerar mais valor para o utilizador", 40 dos inquiridos atribuiu uma pontuação igual ou superior a 4.

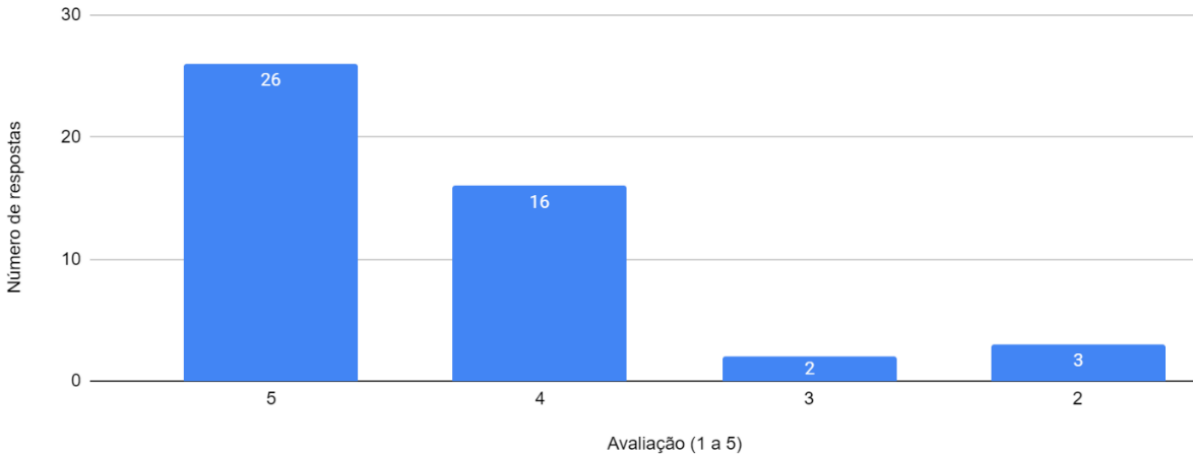


Figura 48 – Gráfico relativo à afirmação "A especialização destas equipas em determinadas áreas de negócio permite que identifique mais facilmente o que pode ser adicionado ao produto para gerar mais valor para o utilizador"

Relativamente à frase "Os ciclos de desenvolvimento por parte em equipas multidisciplinares face a equipas especializadas são, por norma, … ", só 13% a completaram com "Mais longos".

No que concerne a comunicação, questionados sobre se "Consideram que equipas multidisciplinares denotam uma melhoria significativa na comunicação e requerem um menor esforço de coordenação?", 80% indica que "Sim".

Relativamente à questão "Considera que o facto de as equipas serem multidisciplinares e autónomas explicita as responsabilidades de cada equipa", 73% julgam que "Sim".

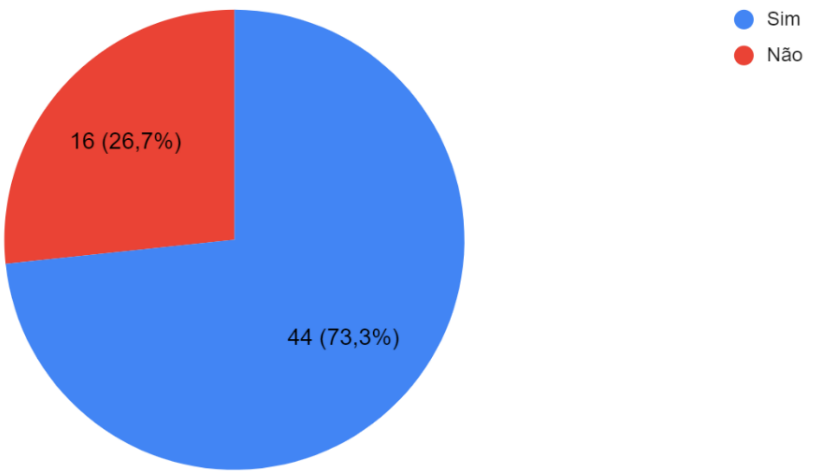


Figura 49 – Gráfico relativo à questão "Considera que o facto de as equipas serem multidisciplinares e autónomas explicita as responsabilidades de cada equipa?"

No entanto, questionados acerca do nível de autonomia que as equipas devem possuir, a percentagem dos que consideram que "A autonomia quase total para tomar decisões" é a aposta mais segura cai dos 73% da questão anterior para 62%.

No que concerne as implicações nas UIs da autonomia das equipas, 68% dos inquiridos considera que aumenta o risco de inconsistência entre as aplicações, ainda que de acordo com 95% possa ser minimizado com a existência de um *Design System* transversal à organização.

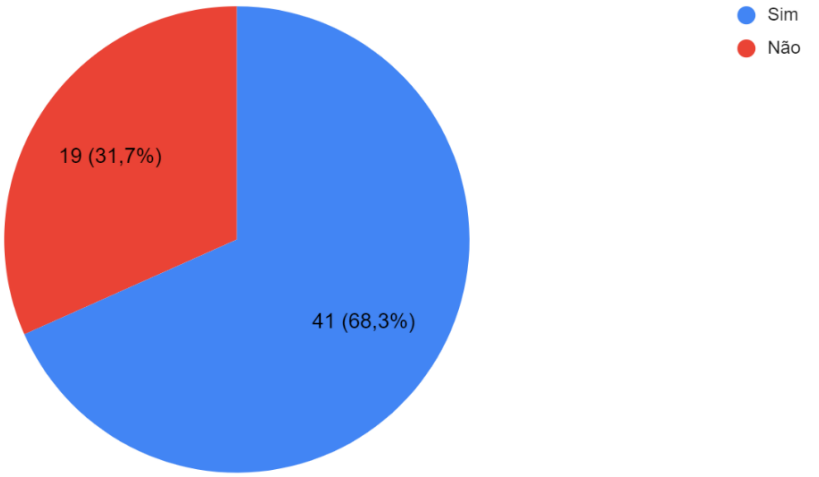


Figura 50 – Gráfico relativo à questão "Considera que facto de as equipas terem mais autonomia pode representar um risco acrescido de inconsistência de UIs entre as aplicações"

Relativamente à formação de silos de conhecimentos, 77% dos inquiridos considera que usando micro *frontends* há um maior risco, destacando como principais estratégias para minimizar a sua formação a "Criação e promoção de *Communities of Practice*" (30%), "Documentar com rigor as soluções desenvolvidas" (29%) e "Apresentar o que tem vindo a ser produzido a outras equipas" (27%).

Quanto à produção de MVPs, 72% dos inquiridos considera que detendo as equipas autonomia para decidir quando proceder ao *deploy*/*release* de novas funcionalidades ocorre num menor espaço de tempo.

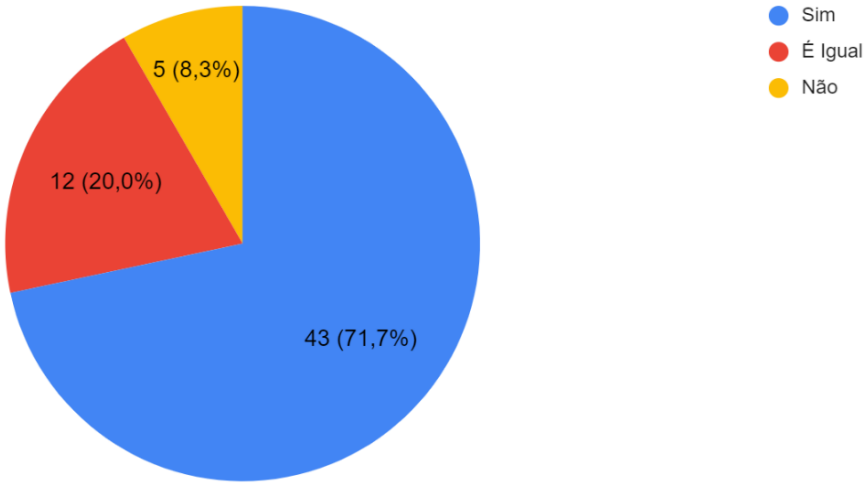


Figura 51 – Gráfico relativo à questão "Considera que tendo as equipas autonomia para decidir quando proceder ao *deploy/release* de novas funcionalidades que é possível produzir MVPs num menos espaço de tempo ?

Relativamente ao impacto da multidisciplinaridade na produtividade das equipas, 92% dos inquiridos consideram que é positivo.

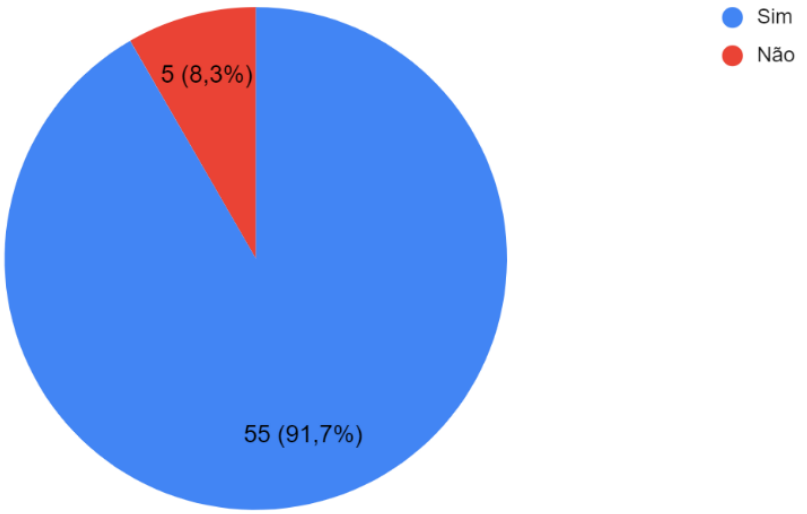


Figura 52 – Gráfico relativa à questão "De uma maneira geral considera que uma estrutura organizacional assente em equipas multidisciplinares aumenta a produtividade das equipas?"

No que concerne as secções que detêm questões de cariz mais técnico, evidencia-se, desde logo, o facto da arquitetura de micro *frontends* ter aparecido recentemente, que faz com que seja desconhecida por 38% dos inquiridos.

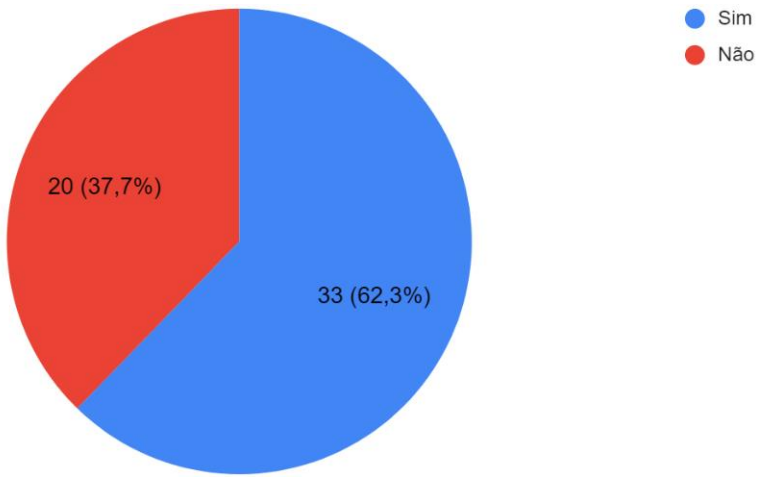


Figura 53 – Gráfico relativo à questão "Já tinha ouvido falar ou lido algo acerca de micro *frontends*?"

Dos 33 inquiridos que indicaram conhecer a abordagem, apenas 10 (30%) já a tinha aplicado, sobretudo com o propósito de promover alterações na *stack* tecnológica das aplicações (42%).

Relativamente ao tipo de projetos em que faz mais sentido utilizar micro *frontends*, 79% das respostas apontam para "Projetos de média/grande dimensão e/ou com um modelo de negócio vasto e complexo".

Quanto à "Integração e Comunicação de Micro *Frontends*", 68% dos inquiridos considera que é possível uma total integração dos micro *frontends* desenvolvidos por equipas distintas entre si e com a aplicação integradora e 81% que é possível estabelecer comunicação eficazmente sem que o custo seja incomportável.

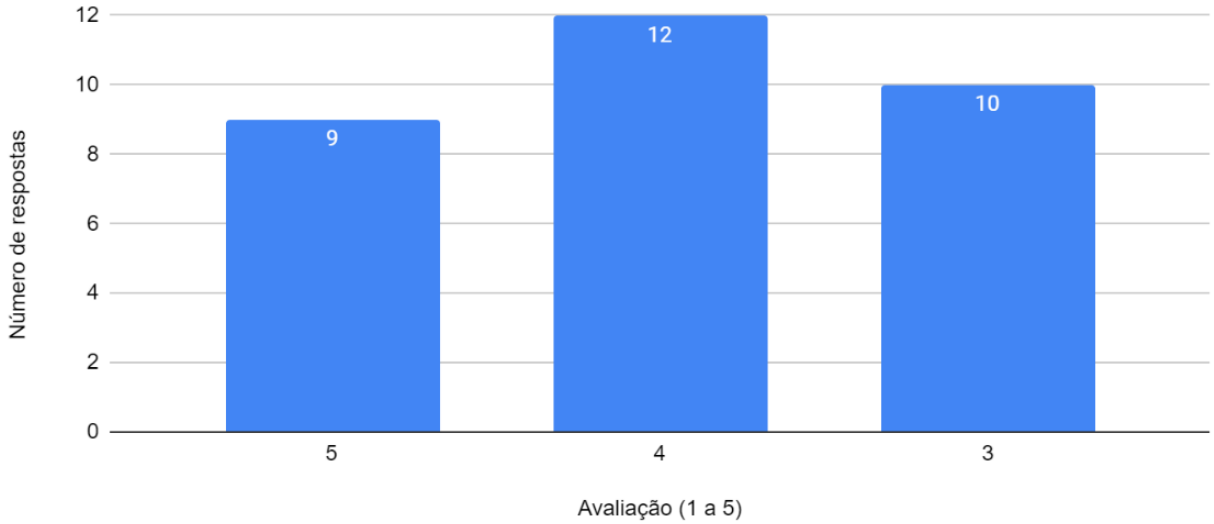


Figura 54 – Gráfico relativo à afirmação "É possível uma total integração de micro *frontends* desenvolvidos por equipas distintas entre si e com a aplicação integradora"

Constata-se ainda que a tendência para algum desconhecimento da abordagem persiste, visto que de um universo de 31 inquiridos, a técnica de integração mais votada obtém apenas 23 respostas, 77% dos inquiridos revela não conhecer nenhuma biblioteca de integração e a biblioteca de integração mais votada alcança apenas 4 votos.

Questionados sobre a técnica de integração mais adequada para satisfazer determinados requisitos, a escolha dos inquiridos recaiu sobre uma abordagem híbrida [13], para aplicações bastante interativas e reativas (48%) [14] e para aplicações em que quer o tempo de carregamento, quer a interatividade sejam importantes (68%).

Já para assegurar a autocontenção e isolamento dos micro *frontends*, 55% considera que a melhor solução é o uso de "*Web Components* (*Custom Elements* com *Shadow* DOM)" e existe uma resposta a sugerir a definição de convenções e a sua aplicação no código.

Relativamente aos mecanismos de comunicação, foi solicitado aos inquiridos que avaliassem numa escala de 1 a 5 o grau de dificuldade de estabelecer a comunicação entre UIs, entre UI e *backend* e entre serviços de *backend*, tendo-se registado uma dificuldade média de 3.1, 3.8 e 3.6 respetivamente.

Assim, no entender dos inquiridos o desafio maior reside na comunicação entre as UIs e os serviços da mesma equipa, algo que pode dever-se a uma má interpretação da questão, isto é, a uma eventual confusão com um cenário hipotético em que uma UI necessite de apresentar dados disponibilizados por serviços de outras equipas.

Em relação a este cenário, em que um micro *frontend* necessita de aceder aos serviços da própria equipa e simultaneamente aos de outras, apesar de alguma literatura aconselhar a implementação de um mecanismo de replicação de dados (Geers, 2020a), 80% dá uma pontuação de 1 a 3 ao seu custo-benefício, sendo a criação de um serviço *middleware* a preferência de 45% dos inquiridos, que é precisamente outra das recomendações da literatura, que sugere a implementação do padrão *Backend-for-Frontend* num contexto de micro *frontends* (C. Jackson, 2019).

No que concerne a capacidade de evolução independente e mais especificamente a liberdade de definição da *stack* tecnológica, 80% dos inquiridos considera que apesar de existir essa liberdade, isso não significa que se devam utilizar sempre tecnologias diferentes entre equipas.

[13] Integração *client-side* com pré-renderização *server-side*.
[14] Ainda que não muito descolada da opção "*Single Page Application* Unificada usando *App Shell*" (que alcança 32% dos votos).

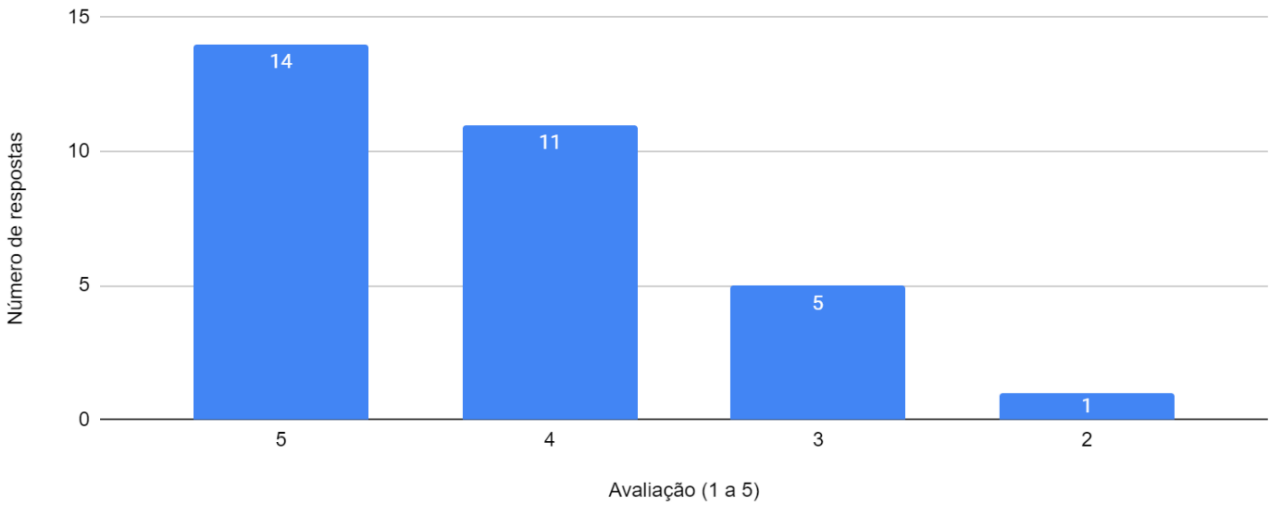


Figura 55 – Gráfico relativa à afirmação "A liberdade de definição de *stack* tecnológica por parte das equipas pode refletir-se num aumento significativo do *bundle* da aplicação já que apesar de ser possível usar diferentes tecnologias isso não significa que se deva fazê-lo sempre".

Quanto à migração, a abordagem de migração funcionalidade a funcionalidade é a escolha de 80% dos inquiridos, que numa escala de 1 a 5, atribuíram uma pontuação igual ou superior a 4.

Quanto ao *deployment*, 55% dos inquiridos considera que adotando uma estratégia de CI e CD é possível efetuá-lo de forma independente, isto é, sem depender da coordenação com outras equipas.

Relativamente às caraterísticas inerentes à adoção de micro *frontends*, os inquiridos consideram que permitem a autocontenção e escalabilidade mais eficiente (87%), favorecem a modularidade, testabilidade e isolamento de falhas (90%), maior tolerância a falhas (81%) e a produção de código mais limpo e com menor acoplamento (71%).

Por fim, da análise à secção de "Vantagens e Desvantagens da Adoção de Micro *Frontends*" constata-se que os inquiridos destacam como principais vantagens:

- Modularidade e baixo acoplamento (15%);
- Responsabilidades bem-definidas (15%);
- *Deploy* de forma independente (14%);
- Escalabilidade mais eficiente (12%);
- Ciclos de desenvolvimento, lançamento e *feedback* mais curtos (10%).

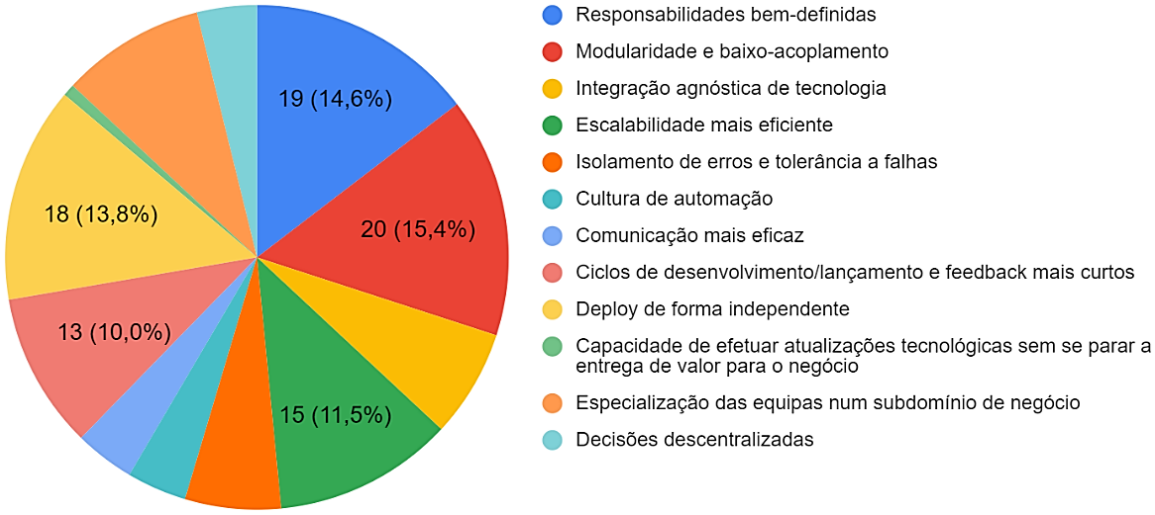


Figura 56 – Gráfico referente às "Vantagens do Uso de Micro *Frontends*"

Relativamente às desvantagens, as mais votadas foram:

- Redundância de código (14%);
- Maior complexidade técnica (14%);
- *Codebase* mais extensa (13%);
- Risco do *bundle* ficar demasiado extenso (13%);
- Silos de conhecimento (12%);
- Inexistência de *guidelines* pode causar inconsistência (8%).

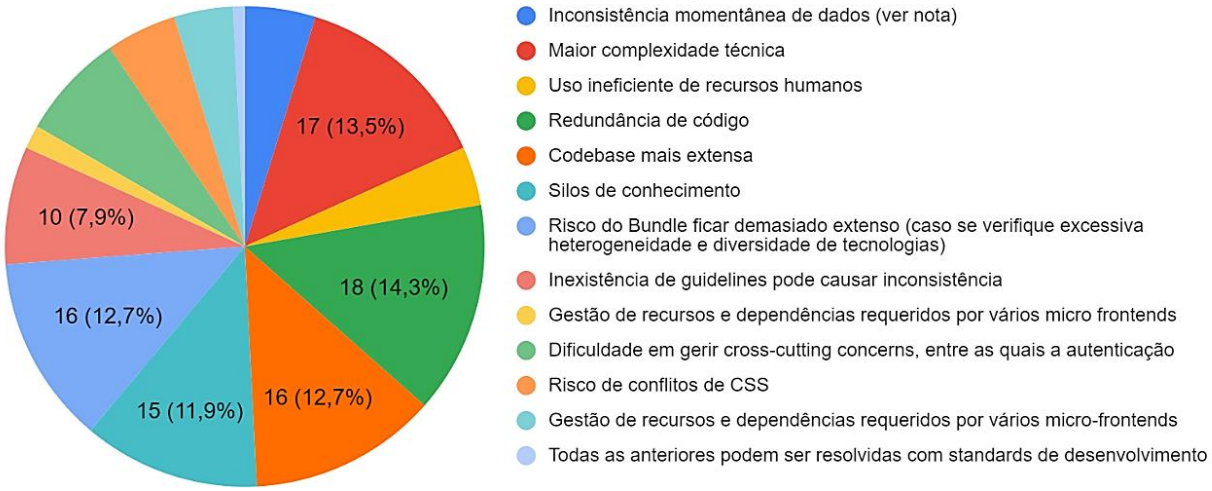


Figura 57 - Gráfico referente às "Desvantagens do Uso de Micro *Frontends*"

7.4.3 Casos de Adoção de Micro *Frontends*

Analisando os casos de adoção de micro *frontends* em contexto empresarial descritos na secção 5.5, constata-se que, em praticamente todas as organizações, partiu da necessidade de repensar a arquitetura das suas aplicações.

Esta necessidade prendia-se com mitigar problemas inerentes à adoção da abordagem puramente monolítica em projetos de alguma dimensão, nomeadamente:

- o crescimento desordenado da *codebase* que, detendo um elevado nível de acoplamento, era difícil de compreender na totalidade, de manter, alterar e estender e de complexa testabilidade, o que se traduzia no aumento dos ciclos de desenvolvimento;

- o elevado tempo de compilação e do qual resulta um único artefacto passível de ser *deployed*, o que naturalmente impedia a escalabilidade eficiente das aplicações;

- a dificuldade de escalar as equipas e fomentar a produtividade;

- problemas de *performance* decorrentes do tamanho elevado do *bundle* e da necessidade de carregamento inicial de todos os *assets* e dependências

- eventuais alterações na *stack* tecnológica implicarem, por norma, a reescrita integral das aplicações.

Outro dos motivos apontados prende-se com as organizações se reinventarem no sentido de serem capazes de produzir soluções inovadoras que lhes permitam diferenciar-se dos principais concorrentes.

Além disso, em organizações que começaram por efetuar a migração do *backend* para uma arquitetura de micro serviços, verificou-se que persistia a incapacidade de se efetuar o *deployment* independente das funcionalidades e consequentemente de lançar MVPs mais rapidamente.

Contudo, a Zalando relembra que antes de se decidir usar micro *frontends*, se deve determinar se esta abordagem permite dar resposta ao problema suscitado, se se reúnem condições para proceder ao desenvolvimento de uma infraestrutura automatizada, que permita tirar partido dos MFEs, e se é possível encetar eventuais ajustes na estrutura organizacional que se revelem imprescindíveis para fomentar a produtividade das equipas, e em caso afirmativo começar por definir os responsabilidades de cada equipa e firmar os contratos entre elas.

Nesta matéria, não sendo a arquitetura de micro *frontends* uma solução *"one size fits all"*, na perspetiva de várias organizações, deve preferir-se uma abordagem mais convencional quando há a garantia de que a UI terá apenas um conjunto fixo de componentes, que não sofrerá alterações significativas, cujo desenvolvimento pode ser entregue a uma única equipa, não

havendo perspetivas nem de crescimento da organização e/ou aplicação a médio-longo prazo nem de promover mexidas na *stack* tecnológica (Mezzalira, 2020).

Relativamente às vantagens que advém da adoção de micro *frontends*, as organizações realçam:

- responsabilidades bem-definidas;
- ciclos de desenvolvimento e lançamento mais curtos;
- a cultura de automação;
- a capacidade de evolução independente;
- alterações na *stack* tecnológica sem que isso implique a reescrita integral da aplicação;
- o menor risco de erros e indisponibilidade decorrentes do lançamento de novas versões, em particular usando técnicas como *Blue/Green* e *Canary Releasing*;
- escalabilidade mais eficiente;
- isolamento e tolerância de falhas;
- melhoria da *performance*, nomeadamente do tempo de carregamento e latência (neste caso, usando técnicas de integração *Server-Side*) e potenciada com recurso *ao code-splitting, lazy loading* e mecanismos de cache que evitam a realização de *round-trips* desnecessários ao servidor;
- existindo uma *app shell*, servidor *web* partilhado ou qualquer outro tipo de camada de integração dos vários micro *frontends* é possível abstrair questões transversais e tratar de forma centralizada *cross-cutting concerns* como a autenticação e autorização, o que permite que as equipas de produto estejam focadas na criação de valor para o seu subdomínio.

A par das vantagens, as organizações foram confrontadas com alguns desafios e identificaram algumas desvantagens da adoção de micro *frontends*, entre as quais:

- a complexidade técnica e tempo que exige em termos de infraestrutura, mais precisamente na conceção da *pipeline* de CI e CD e na implementação de mecanismos de monitorização, *logging* e *tracing* que verifiquem em tempo real a *performance* e ajudem a identificar eventuais problemas;
- a dificuldade em migrar de forma gradual aplicações com elevado nível de acoplamento;
- resistência em aderir aos princípios de MFEs e em aceitar a autonomia das equipas para tomar decisões;

- risco do *bundle* ficar demasiado grande, devido a existirem dependências duplicadas entre MFEs;
- risco de inconsistência visual e de colisões de *scripts* e de estilos;
- a existência de uma infraestrutura partilhada pode ser ótima para proceder alterações para todos ao mesmo tempo, mas em caso de erro constitui um ponto único de falha que provoca assim a indisponibilidade de toda a aplicação.

7.4.4 Síntese

Os resultados obtidos nas experiências aliados às conclusões retiradas do estudo permitem afirmar que a adoção de micro *frontends* em projetos de desenvolvimento *web* é viável.

Relativamente aos contextos em que se pode assumir-se como uma mais-valia, destacam-se o *software* desenvolvido de forma iterativo e com manutenção a longo prazo em projetos com alguma dimensão e/ou complexidade de negócio, desenvolvidos por várias equipas na mesma organização, ou se se pretender migrar uma aplicação legada de forma gradual.

Nesse sentido, desde que a organização no seu todo reúna conhecimentos de desenvolvimento E2E e seja possível proceder a alterações na organização das equipas para que se tornem multidisciplinares e se especializem numa determinada área de domínio, ficando responsáveis pelo desenvolvimento E2E da mesma, é possível agilizar o processo de desenvolvimento sem comprometer a qualidade da solução produzida.

Relativamente às vantagens da sua adoção, destacam-se a explicitação das responsabilidades de cada equipa, a produção de soluções altamente modulares e com baixo acoplamento, *codebase* limitada, de fácil testabilidade e manutenção e em conformidade com o SRP, a cultura de automação que possibilita a capacidade de efetuar *deployments* de forma independente, o que reduz os ciclos de desenvolvimento, lançamento e *feedback*, por não requerer a coordenação de equipas e permitir o lançamento de MVPs num menor espaço de tempo.

Quanto às desvantagens da sua adoção, destacam-se o risco do *bundle/payload* ficar sobrecarregado devido à duplicação de dependências entre micro *frontends*, da necessidade de proceder à reestruturação das equipas, que pode levantar uma onda de contestação, a complexidade técnica que exige em termos de *design* e conceção da *pipeline* de CI/CD, a curva de aprendizagem associada à aplicação de técnicas de integração de micro *frontends* e o risco de inconsistências visuais ou colisões de estilos, sobretudo quando os contratos firmados entre as várias equipas não contemplarem convenções de *namespacing* e normas de estilização.

8 Conclusões

Este capítulo destina-se a apresentar as conclusões a retirar do estudo sobre o tema e do desenvolvimento da prova de conceito, nomeadamente os objetivos superado, as limitações e desafios encontrados e o trabalho a realizar futuramente. Finalmente, é efetuada a apreciação final e global.

8.1 Objetivos Alcançados

O propósito deste trabalho consiste na recolha e análise de informação relativa à arquitetura de micro *frontends* para determinar se é viável, em que contextos a aplicação desta abordagem se assume como uma mais-valia.

Para tal, definiram-se quatro principais objetivos para retirar conclusões sólidas relativamente a esta temática.

Relativamente ao objetivo de "Recolher e estudar informação relativa a práticas de desenvolvimento *web*, dando particular ênfase a micro *frontends*", foi atingido com a produção dos capítulos "Contexto", "Micro *Frontends*" e "Integração e Comunicação de Micro *Frontends*", nas quais o tema foi minuciosamente analisado.

Quanto ao objetivo de "Determinar em que contextos a adoção de micro *frontends* poderá ser particularmente vantajosa", foi alcançado através da redação da secção 7.4, na qual com base no estudo, POC, questionário e casos de adoção foi aferida a viabilidade desta abordagem e identificados os contextos em que tal seria particularmente vantajosa, bem como foram identificadas as principais vantagens e desvantagens.

Em relação ao objetivo de "Comparar padrões de integração de micro *frontends*, identificando as principais vantagens e desvantagens, bem como os custos inerentes", foi igualmente

cumprido, uma vez que nas secções 5.1, 5.2 e 5.3 foram analisadas e comparadas as principais técnicas e bibliotecas de integração.

No que concerne o objetivo de "Implementar provas de conceito usando os padrões de integração selecionados", foi atingido através da implementação da prova de conceito usando a biblioteca single-spa, assente na técnica de integração "SPA Unificada com *App Shell* usando *Routing* em 2 Níveis".

Relativamente ao último objetivo, inicialmente estava previsto que fossem implementadas várias versões da POC usando técnicas de integração distintas.

No entanto, visto que a aplicação de *e-commerce* produzida é constituída por vários micro *frontends*, contendo páginas e fragmentos integrados em fragmentos e/ou páginas desenvolvidas de outros MFEs, em parte protegidos por autenticação, e que necessitavam de comunicar entre si, foi possível retirar conclusões quanto à viabilidade, vantagens e desafios decorrentes da adoção de micro *frontends*.

8.2 Limitações e Desafios

No que toca a limitações e desafios encontrados durante o desenvolvimento do estudo e produção da POC, é de salientar a inexistência de artigos científicos sobre o tema e a dispersão e superficialidade de informação em blogues e fóruns, que na maior parte dos casos se limita a apresentar o conceito e os pressupostos subjacentes, sem qualquer referência aos impactos da sua adoção nas organizações.

Apesar da bibliografia e os recursos à disposição serem limitados, o artigo da autoria de Cam Jackson no *site* de Martin Fowler, os livros "*Micro Frontends In Action"* de Michael Geers e *"Building Micro-Frontends"* de Luca Mezzalira, que ainda se encontram a ser redigidos, e várias apresentações sobre o tema em conferências tecnológicas, deram um forte contributo na produção de um estudo robusto e bastante abrangente.

Em relação à prova de conceito, não foi possível implementá-la em contexto empresarial entregando o desenvolvimento dos MFEs identificados a equipas distintas, o que não permitiu aferir as implicações da sua adoção na autonomia e produtividade das equipas.

Além disso, dado o carácter recente da biblioteca utilizada, a escassez de documentação e a falta de suporte da comunidade, o autor sentiu inicialmente alguma dificuldade em apreender os conceitos subjacentes ao single-spa e integrar nas páginas e/ou fragmentos de um micro *frontend* fragmentos provenientes de outros micro *frontends* e não foi capaz de no micro *frontend* de *styleguide* criar variáveis nos ficheiros de estilo e utilizá-las nos restantes MFEs.

Por outro lado, não encontrou informação relativa a boas práticas de implementação de autenticação usando single-spa.

8.3 Trabalho Futuro

Relativamente ao trabalho a desenvolver futuramente, à medida que o conceito de micro *frontends* for amadurecendo, prevê-se que as técnicas e bibliotecas de integração existentes evoluam e surjam novas opções que permitindo tornar esta arquitetura mais robusta, simples de colocar em prática e capaz de agilizar o processo de desenvolvimento, merecerão ser alvo de estudo e experimentação e farão com que mais organizações adotem esta abordagem.

Aliás, nesse contexto, o single-spa, biblioteca utilizada na implementação da POC revela-se uma solução bastante promissora para promover a adoção de MFEs, sendo já a aposta de empresas com uma presença consolidada no ramo do desenvolvimento *web*, como a Glintt.

Além disso, o trabalho desenvolvido pode ser complementado, implementando uma prova de conceito em contexto real, num cenário em que existam várias equipas multidisciplinares que fiquem responsáveis pelo desenvolvimento E2E dos MFEs, e promovendo melhorias na POC, nomeadamente nas *pipelines* criadas para os vários MFEs e na gestão de *cross-cutting concerns* como a autenticação e a partilha dos estilos e lógica contidos no micro *frontend* de *styleguide*.

8.4 Apreciação Global

O desenvolvimento deste projeto revelou-se uma mais-valia a nível académico, profissional e pessoal, pois se por um lado marca o encerramento de uma etapa importante da minha vida, por outro, me permitiu expandir horizontes relativamente ao desenvolvimento *web* e às abordagens existentes para conceber as aplicações.

Sendo o desenvolvimento *web* uma área em constante evolução, o estudo dos impactos da adoção de micro *frontends* constituiu indubitavelmente uma oportunidade para aprender e aplicar novos conceitos e técnicas, que certamente terão um impacto positivo no meu percurso profissional, permitindo-me participar mais ativamente na tomada de decisões, em particular as relacionadas com o *design* arquitetural e a definição de *stack* tecnológica e eventualmente a média-longo prazo em matéria de gestão e organização de equipas.

Tendo tido a oportunidade de abraçar um novo desafio profissional durante a realização deste documento, que implicou a aprendizagem de novas tecnologias e processos organizacionais, o estudo e a produção da POC exigiram bastante disciplina e uma gestão apertada de tempo, possível graças ao apoio incansável que recebi da minha família, amigos e colegas de trabalho.

Finalmente, considero que o balanço do trabalho desenvolvimento é francamente positivo, uma vez que, apesar das adversidades sentidas, todos os objetivos inicialmente delineados foram superados, produzindo-se um documento que resulta da recolha e tratamento de informação, da análise de casos de adoção da abordagem e do desenvolvimento de uma prova de conceito que mobiliza os conhecimentos adquiridos e que me permitiu lidar na primeira pessoa com algumas das limitações e desafios inerentes à adoção de micro *frontends*.

Referências

Airbnb. (sem data). *Airbnb Engineering & Data Science*. Obtido 16 de Maio de 2020, de https://airbnb.io/projects/hypernova/

Akamai. (sem data). *Edge Side Includes | Customer Support | Akamai*. Obtido 12 de Fevereiro de 2020, de https://www.akamai.com/us/en/support/esi.jsp

Alexa. (2020a, Fevereiro 19). *upwork.com Competitive Analysis, Marketing Mix and Traffic—Alexa*. https://www.alexa.com/siteinfo/upwork.com

Alexa. (2020b, Abril 28). *allegro.pl Competitive Analysis, Marketing Mix and Traffic—Alexa*. https://www.alexa.com/siteinfo/allegro.pl

Alija, N. (2017). Justification of Software Maintenance Costs. *International Journal of Advanced Research in Computer Science and Software Engineering*, *7*, 15–23. https://doi.org/10.23956/ijarcsse/V7I2/01207

Anderson, R. (2016, Abril 26). *Server Side Include <serverSideInclude>*. https://docs.microsoft.com/en-us/iis/configuration/system.webserver/serversideinclude

Apache. (sem data). *Apache httpd Tutorial: Introduction to Server Side Includes—Apache HTTP Server Version 2.4*. Obtido 14 de Fevereiro de 2020, de https://httpd.apache.org/docs/2.4/howto/ssi.html

Ast, M., & Gaedke, M. (2017). Self-contained web components through serverless computing. *Proceedings of the 2nd International Workshop on Serverless Computing*, 28–33. https://doi.org/10.1145/3154847.3154849

Atlassian. (sem data). *What is Continuous Integration*. Atlassian. Obtido 13 de Fevereiro de 2020, de https://www.atlassian.com/continuous-delivery/continuous-integration

Avram, A., & Marinescu, F. (2007). *Domain-Driven Design Quickly—A Summary of Eric Evans' Domain-Driven Design*. C4Media.

Babich, N. (2019, Outubro 18). User Centered Design Principles & Methods | Adobe XD Ideas. *Ideas*. https://xd.adobe.com/ideas/principles/human-computer-interaction/user-centered-design/

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture. *IEEE Software*, *33*, 1–1. https://doi.org/10.1109/MS.2016.64

Baldwin, C., & Clark, K. (2000). *Design Rules Volume I: The Power of Modularity* (Vol. 1). https://doi.org/10.2307/259400

Bali, R., Troshani, I., Goldberg, S., & Wickramasinghe, N. (2012). *Pervasive Health Knowledge Management*. Springer Science & Business Media.

Bankai. (2017, Abril 28). *A History of Disruption*. Medium. https://medium.com/@bankai_ux/history-of-ux-timeline-infographic-4a2035b5014a

Barutçu, S. (2006). *Quality Function Deployment In Effective Website Design: An Application In E-Store Design*. *7*, 41–63.

Basic, R. (2018, Março 20). *Bounded contexts and subdomains*. https://robertbasic.com/blog/bounded-contexts-and-subdomains/

Bidelman, E. (2019, Agosto 14). *Shadow DOM v1: Self-Contained Web Components | Web Fundamentals*. https://developers.google.com/web/fundamentals/web-components/shadowdom?hl=en

Björklund, C. (2019, Abril 15). *App Maintenance Cost Can Be Three Times Higher than Development Cost: EConnectivity Blog*. https://www.econnectivity.se/app-maintenance-cost-can-be-three-times-higher-than-development-cost/

Broadcast Channel API. (sem data). MDN Web Docs. Obtido 24 de Junho de 2020, de https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API

Brooks, F. P., Jr. (1995). *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc.

Bubbling and capturing. (2020, Junho 1). https://javascript.info/bubbling-and-capturing

Buck, A., Wilson, M., & Wasson, M. (2019, Janeiro 18). *Padrão de estrangulamento—Cloud Design Patterns | Microsoft Docs*. https://docs.microsoft.com/pt-pt/azure/architecture/patterns/strangler

Burkholder, B. (2018, Julho 6). *JavaScript SEO: Server Side Rendering vs. Client Side Rendering*. https://medium.com/@benjburkholder/javascript-seo-server-side-rendering-vs-client-side-rendering-bc06b8ca2383

Castro, F. (2019, Agosto 31). *Porto já é um hub tecnológico. E as empresas estão a contratar – ECO*. https://eco.sapo.pt/2019/08/31/porto-ja-e-um-hub-tecnologico-e-as-empresas-estao-a-contratar/

Clausing, D., & Hauser, J. (1988). *House of Quality*.

Cloudfare. (2016). *What Is a CDN? How Does a CDN work?* Cloudflare. https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

Cohen, D., & Costa, P. (2012). *Agile Software Development*.

Colin, J. (2018, Dezembro 6). *Front-End Micro Services*. Zalando Jobs. https://jobs.zalando.com/en/tech/blog/front-end-micro-services/

Cooney, D. (2016). *Webcomponents/HTML-Imports-and-ES-Modules.md at gh-pages · w3c/webcomponents*. https://github.com/w3c/webcomponents/blob/gh-pages/proposals/HTML-Imports-and-ES-Modules.md

Coplien, J. O., & Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development*. Prentice-Hall, Inc.

Cordeiro, R. (2019, Fevereiro 25). *CTW History*.

Dando, J. (2019, Janeiro 7). *Adding SASS/SCSS to your React + TypeScript project*. Medium. https://medium.com/@dandobusiness/adding-sass-scss-to-your-react-typescript-project-162de415b19a

DAZN. (2020a). *DAZN*. https://www.dazn.com/en-EU/l/sports/

DAZN. (2020b). Em *Wikipedia*. https://en.wikipedia.org/w/index.php?title=DAZN&oldid=963236411

Dehghani, Z. (2018, Abril 24). *How to break a Monolith into Microservices*. martinfowler.com. https://martinfowler.com/articles/break-monolith-into-microservices.html

Denning, J. (2019a, Dezembro 23). *Javascript tutorial: Import maps*. https://www.youtube.com/watch?v=Lfm2Ge_RUxs&list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU&index=3

Denning, J. (2019b, Dezembro 23). *Javascript tutorial: Local development with microfrontends, single-spa, and import maps*. https://www.youtube.com/watch?v=vjjcuIxqIzY&list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU&index=4

Denning, J. (2019c, Dezembro 24). *Deploying Microfrontends Part 1—Import Map Deployer*. https://www.youtube.com/watch?v=QHunH3MFPZs&list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU&index=5

Denning, J. (2020a, Janeiro 3). *Deploying Microfrontends Part 2—CI for in-browser modules*. https://www.youtube.com/watch?v=nC7rpDXa4B8&list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU&index=6

Denning, J. (2020b, Janeiro 6). *Javascript Tutorial—SystemJS intro*. https://www.youtube.com/watch?v=AmdKF2UhFzw&list=PLLUD8RtHvsAOhtHnyGx57EYXoaNsxGrTU&index=7

Denning, J. (2020c, Fevereiro 24). *Layout Engine*. https://single-spa.js.org/docs/layout-overview

Denning, J. (2020d, Fevereiro 24). *Single-spa · Building Applications*. https://single-spa.github.io/single-spa.js.org/docs/building-applications

Denning, J. (2020a, Fevereiro 24). *Single-spa · Getting Started Overview*. https://single-spa.github.io/single-spa.js.org/docs/getting-started-overview

Denning, J. (2020f, Fevereiro 24). *Single-spa · The single-spa ecosystem*. https://single-spa.github.io/single-spa.js.org/docs/ecosystem

Denning, J. (2020g, Fevereiro 24). *Single-spa-inspector*. https://single-spa.js.org/docs/devtools/#configuring-app-overlays

Denning, J. (2020h, Junho 26). *Single-spa/import-map-deployer*. https://github.com/single-spa/import-map-deployer

Denning, J., & McMurdie, J. (2020, Março 24). *Parcels*. https://single-spa.js.org/docs/parcels-overview

Denning, J., McMurdie, J., & Fedosov, V. (2020, Abril 20). *The Recommended Setup*. https://single-spa.js.org/docs/recommended-setup/

Denning, J., McMurdie, J., & Hawkins, T. (2020, Março 20). *Single-spa Microfrontend Types*. https://single-spa.js.org/docs/module-types/

Denning, J., McMurdie, J., & Wilson, A. (2020, Fevereiro 24). *Microfrontends Overview*. https://single-spa.js.org/docs/microfrontends-concept

Devs, W. (2015, Agosto 12). *Microservices and Isolation*. Western Devs. http://www.westerndevs.com/Microservices-and-isolation/index.html

Diaz, F. G. (2019a). *Ara Framework · Build Micro-frontends easily using Airbnb Hypernova*. "https://ara-framework.github.io/website/

Diaz, F. G. (2019b, Julho 5). *Strangling a Monolith application with Micro Frontends using Server Side Includes*. Medium. https://itnext.io/strangling-a-monolith-to-micro-frontends-decoupling-presentation-layer-18a33ddf591b

Dörnenburg, E. (2019, Junho 19). *Patterns for Micro Frontends—ThoughtWorks Talks Tech*. https://www.youtube.com/watch?v=tcQ1nWdb7iw

Drayton, C. (2018, Abril 26). *Agile Metrics Explained: Throughput*. https://www.mazzlo.co/blog/agile-metrics-throughput

Duffy, G. (2019). *What is network latency (and how do you use a latency calculator to calculate throughput)?* https://www.sas.co.uk/blog/what-is-network-latency-how-do-you-use-a-latency-calculator-to-calculate-throughput

ECMA International. (2019). *History of Ecma*. https://www.ecma-international.org/memento/history.htm

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software* (1.ª ed.). Pearson Education.

Facebook. (2010, Junho 4). *BigPipe: Pipelining web pages for high performance*. https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/

Fallon, C. (1980). *Value Analysis: Second Revised Edition*. Triangle Press.

Farrell, J., & Nezlek, G. S. (2007). Rich Internet Applications The Next Stage of Application Development. *2007 29th International Conference on Information Technology Interfaces*, 413–418. https://doi.org/10.1109/ITI.2007.4283806

Fernandes, T. (2017a, Março 6). *Spotify Squad framework—Part I*. Medium. https://medium.com/productmanagement101/spotify-squad-framework-part-i-8f74bcfcd761

Fernandes, T. (2017b, Março 18). *Spotify Squad framework—Part II - Product Management 101—Medium*. https://medium.com/productmanagement101/spotify-squad-framework-part-ii-c5d4b9398c30

Ficalora, J. P., & Cohen, L. *Quality function deployment and Six Sigma: A QFD handbook* (2nd ed). Prentice Hall.

Figus, M. (2015, Fevereiro 15). *OpenTable Tech UK Blog | Dismantling the monolith—Microsites at Opentable*. http://tech.opentable.co.uk/blog/2015/02/09/dismantling-the-monolith-microsites-at-opentable/

Finn. (2018, Fevereiro 6). *Podium*. https://tech.finn.no/images/2018-02-06-fagkveld-presentasjoner/2018-02-28%20Podium.pdf

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. (2004, Junho 29). *StranglerFigApplication*. martinfowler.com. https://martinfowler.com/bliki/StranglerFigApplication.html

Fowler, M. (2005, Dezembro 14). *EvansClassification*. https://www.martinfowler.com/bliki/EvansClassification.html

Fowler, M. (2010, Março 1). *Blue-Green Deployment*. martinfowler.com. https://martinfowler.com/bliki/BlueGreenDeployment.html

FrintJS. (sem data). *Documentation | Frint*. Obtido 16 de Fevereiro de 2020, de https://frint.js.org/docs

Frost, B. (2013, Junho 10). *Atomic Design*. https://bradfrost.com/blog/post/atomic-web-design/

Gałek, B., Walacik, B., & Wielądek, P. (2016, Março 12). *Managing Frontend in the Microservices Architecture*. Allegro.Tech. https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

Gao, K. (2015, Janeiro 15). *CSI vs SSI vs ESI: Concept and Our Usecases*. http://kgao.github.io/blogs/2015/01/15/CSI-ESI-SSI/

Gaunt, M. (2019, Agosto 9). *Service Workers: An Introduction | Web Fundamentals*. Google Developers. https://developers.google.com/web/fundamentals/primers/service-workers

Geers, M. (2018). *Micro Frontends—Extending the microservice idea to frontend development*. https://micro-frontends.org/

Geers, M. (2020a). *Micro Frontends In Action*. Manning.

Geers, M. (2020, Fevereiro 19). *What about App Shell? Who's in charge of? Conversation With Michael Geers via Twitter* [Comunicação pessoal].

Geers, M. (2020b). *Micro Frontends Integration Techniques Comparison Chart*.

Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP Theorem. *Computer*, *45*(2), 30–36. https://doi.org/10.1109/MC.2011.389

Google. (2020a). *Micro frontends—Explorar—Google Trends*. https://trends.google.pt/trends/explore?date=2016-10-01%202020-02-08&q=micro%20frontends

Google. (2020b, Junho 13). *Pushing and pulling images | Container Registry Documentation*. https://cloud.google.com/container-registry/docs/pushing-and-pulling

Gordon, E. K. (2018). *Isomorphic Web Applications*. Manning.

Grealou, L. (2015, Maio 11). *(15) Productivity vs Throughput | LinkedIn*. https://www.linkedin.com/pulse/productivity-vs-throughput-lionel-grealou/

Grigoryan, A. (2017, Abril 17). *The Benefits of Server Side Rendering Over Client Side Rendering*. https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8

HashiCorp. (sem data). *Consul by HashiCorp*. Consul by HashiCorp. Obtido 3 de Maio de 2020, de https://www.consul.io/index.html

Hauser, J., Griffin, A., Klein, R., Katz, G., & Gaskin, S. (2010). *Quality Function Deployment (QFD)*. https://doi.org/10.1002/9781444316568.wiem05023

Herrington, J. (2019, Agosto 19). *Micro Frontends and OpenComponents—Medium*. https://medium.com/@jack_42842/micro-frontends-and-opencomponents-ec10cfb200bd

Herrington, J. (2020, Março 13). *Single Spa + Federated Modules = Wow!* https://www.youtube.com/watch?v=wxnwPLLIJCY

Hombergs, T. (2018, Setembro 27). *What is Upstream and Downstream in Software Development?* Reflectoring.Io. https://reflectoring.io/upstream-downstream/

Humble, J., & Farley, D. (2010). *Continuous Delivery, Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

Hunt, A. (2000). *The Pragmatic Programmer*. Pearson Education.

Iframes and the postMessage Method. (sem data). Obtido 24 de Junho de 2020, de https://www.dyn-web.com/tutorials/iframes/postmessage/

IKEA. (2019, Dezembro). *One brand, lots of companies*. https://www.ikea.com/jp/en/this-is-ikea/about-us/one-brand-pub07af8e71

Isaac, L. (2014, Julho 9). *How to Setup HTML Server Side Includes SSI on Apache and Nginx*. https://www.thegeekstuff.com/2014/07/server-side-includes/

Jackson, C. (2019, Junho 19). *Micro Frontends*. https://martinfowler.com/articles/micro-frontends.html

Jackson, M. (2015, Junho 8). *Universal JavaScript*. Medium. https://cdb.reacttraining.com/universal-javascript-4761051b7ae9

Jorgé. (2016, Novembro 22). *Web Components and SEO*. https://react-etc.net/entry/web-components-seo

Kalske, M. (2017). *Transforming monolithic architecture towards microservice architecture* [M.Sc. Thesis]. University of Helsinki.

Karamanlakis, S. (2017, Janeiro 30). Upwork Modernization: An Overview. *Upwork Blog*. https://www.upwork.com/blog/2017/01/upwork-modernization/

KeyCDN. (2018, Outubro 4). *What is Cache Busting? - KeyCDN Support*. KeyCDN. https://www.keycdn.com/support/what-is-cache-busting

Kniberg, H. (2014, Março 27). *Crisp's Blog » Spotify Engineering Culture (part 1)*. https://blog.crisp.se/2014/03/27/henrikkniberg/spotify-engineering-culture-part-1

Koen, P. (2001). *Front End Innovation—What is the New Concept Development (NCD) model?* http://frontendinnovation.com/fei/what-is-the-new-concept-development-ncd-model?fbclid=IwAR3juHz20qI6jjEzgBKfduEEZ3VWs6LMk3K09n29sE0byNEe8iqdBJM2TsA

Koen, P. A., Ajamian, G., Boyce, S. D., Clamen, A., Fisher, E. S., Fountoulakis, S. G., Johnson, A., Puri, P. S., & Seibert, R. (2002). *1 Fuzzy Front End: Effective Methods , Tools , and Techniques*.

Kotte, G. N. (2017). *Microservice Websites*. https://gustafnk.github.io/microservice-websites/

Kotte, G. N. (2018, Julho 5). *Micro Frontends with Gustaf Nilsson Kotte—CaSE: Conversations about Software Engineering* (S. Tilkov) [Entrevista]. https://www.case-podcast.org/22-micro-frontends-with-gustaf-nilsson-kotte/transcript

Kubyshkin, D. (2018, Maio). *Zalando—Project Mosaic—Micro Frontends in Review*. https://www.youtube.com/watch?v=dKJenHVFTc4

Kumar, D. (2020, Maio 11). *Authentication in NodeJS With Express and Mongo—CodeLab #1—DEV*. https://dev.to/dipakkr/implementing-authentication-in-nodejs-with-express-and-jwt-codelab-1-j5i

LeanIX GmbH. (2017). *Modernizing IT with Microservices* [Technology]. https://www.slideshare.net/leanIX_net/modernizing-it-with-microservices

Lewis, J., & Fowler, M. (2014, Março 25). *Microservices*. https://martinfowler.com/articles/microservices.html

Linuxwize. (2019, Outubro 10). *Setting up an Nginx Reverse Proxy*. /post/nginx-reverse-proxy/

Lusa, A. (2019, Agosto 30). *Critical TechWorks estima atingir mil trabalhadores em Portugal em 2020*. Notícias ao Minuto. https://www.noticiasaominuto.com/economia/1312684/critical-techworks-estima-atingir-mil-trabalhadores-em-portugal-em-2020

Lynch, W. (2019, Fevereiro 8). *Scrum Philosophy: Release Early, Release Often—Warren Lynch—Medium*. https://medium.com/@warren2lynch/scrum-philosophy-release-early-release-often-a5b864fd62a8

Macquin, G. (2018, Março 21). *Single Page Applications vs Multiple Page Applications—Do You Really Need an SPA?* https://medium.com/@goldybenedict/single-page-applications-vs-multiple-page-applications-do-you-really-need-an-spa-cf60825232a3

Marcano, P., Veloso, M., & Wenzel, M. (2019, Novembro 5). *Choose between traditional web apps and single page apps | Microsoft Docs*. https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps

Martin, R. (2000). *Design Principles and Design Patterns*.

Martínez-Ortiz, A. L., Lizcano, D., Ortega, M., Ruiz, L., & López, G. (2016). A quality model for web components. *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*, 430–432. https://doi.org/10.1145/3011141.3011203

MDN contributors. (2019a, Março 23). *Window.postMessage()—Web APIs | MDN*. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

MDN contributors. (2019b, Abril 8). *Progressive Enhancement—MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement

MDN contributors. (2019c, Junho 14). *CustomEvent()—Web APIs | MDN*. https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent

MDN contributors. (2019d, Setembro 29). *JavaScript language resources*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources

MDN contributors. (2020, Fevereiro 11). *Progressive web apps (PWAs)*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

Mendes, W. (2018, Janeiro 25). *Wilson Mendes: Micro Frontend—A Microservice Architecture From Your Frontend Web Apps—JSConf.Asia 2018*. https://www.youtube.com/watch?v=Kphwg2IsJfA

Mezzalira, L. (2018, Outubro 4). *Micro Frontend Architecture—Luca Mezzalira, DAZN*. UXDX Conf, Dublin. https://www.youtube.com/watch?v=BuRB3djraeM

Mezzalira, L. (2019a, Abril 8). *Adopting a Micro-frontends architecture—DAZN Engineering—Medium*. https://medium.com/dazn-tech/adopting-a-micro-frontends-architecture-e283e6a3c4f3

Mezzalira, L. (2019b, Maio 21). *Identifying micro-frontends in our applications*. Medium. https://medium.com/dazn-tech/identifying-micro-frontends-in-our-applications-4b4995f39257

Mezzalira, L. (2020). *Building Micro-Frontends (Early Release)*. O'Reilly.

Miller, J., & Osmani, A. (2019, Fevereiro). *Rendering on the Web | Google Developers*. https://developers.google.com/web/updates/2019/02/rendering-on-the-web

Millett, S., & Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design*. John Wiley & Sons.

Mota, A. (2020, Julho 1). *Utilização de Micro Frontends na Glintt* [Comunicação pessoal].

Nagy, G. (2019, Outubro 16). *Introduction to micro frontends architecture*. Medium. https://levelup.gitconnected.com/brief-introduction-to-micro-frontends-architecture-ec928c587727

Nasiri, S. (2017, Maio 25). Modernizing Upwork with Micro Frontends. *Upwork Blog*. https://www.upwork.com/blog/2017/05/modernizing-upwork-micro-frontends/

Naso, A. (2018, Agosto 7). *The Complete Guide to A/B Testing Your Progressive Web App*. Mobify. https://www.mobify.com/insights/complete-guide-a-b-testing-progressive-web-app-2018/

Neary, A. (2017, Maio 16). *Rearchitecting Airbnb's Frontend*. Medium. https://medium.com/airbnb-engineering/rearchitecting-airbnbs-frontend-5e213efc24d2

Neoteric. (2016, Dezembro 2). *Single-page application vs. Multiple-page application*. https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

Neoteric. (2017, Abril 20). *How can you refactor a monolithic application into microservices?* https://medium.com/@NeotericEU/how-can-you-refactor-a-monolithic-application-into-microservices-2eef8e323840

Newman, S. (2015). *Building microservices*. O'Reilly.

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc.

NGINX. (2015). *The Future of App Development and Delivery | NGINX*. https://www.nginx.com/resources/library/app-dev-survey/

Norelus, E. (2019, Abril 28). *Implementing Domain-Driven Design for Microservice Architecture*. https://medium.com/design-and-tech-co/implementing-domain-driven-design-for-microservice-architecture-26eb0333d72e

Nut. (sem data). *nut-project/nut: A framework born for micro frontends*. Obtido 16 de Fevereiro de 2020, de https://github.com/nut-project/nut

Oliver, J. (2009, Abril 4). *DDD: Strategic Design: Core, Supporting, and Generic Subdomains*. Jonathan Oliver. https://blog.jonathanoliver.com/ddd-strategic-design-core-supporting-and-generic-subdomains/

Olsson, H., Alahyari, H., & Bosch, J. (2012, Setembro 7). *Climbing the" Stairway to Heaven" A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software*. Proceedings - 38th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2012. https://doi.org/10.1109/SEAA.2012.54

OpenComponents. (sem data). Obtido 15 de Fevereiro de 2020, de https://opencomponents.github.io/

Opencomponents/oc. (sem data). GitHub. Obtido 15 de Fevereiro de 2020, de https://github.com/opencomponents/oc

OpenLearn. (2019). *An introduction to web applications architecture: 1.1 Client–server architecture—OpenLearn—Open University—TT284_1*. https://www.open.edu/openlearn/science-maths-technology/introduction-web-applications-architecture/content-section-1.1

Osmani, A. (2019, Maio 19). *The App Shell Model | Web Fundamentals*. Google Developers. https://developers.google.com/web/fundamentals/architecture/app-shell

Pang, A. (2019, Dezembro 6). Top 10 ERP Software Vendors, Market Size and Market Forecast 2018-2023. *Apps Run The World*. https://www.appsruntheworld.com/top-10-erp-software-vendors-and-market-forecast/

Perez, J. (sem data). *Airbnb Engineering & Data Science*. Obtido 20 de Junho de 2020, de https://airbnb.io/projects/hypernova/

Pérez, J. M. (2019, Março 25). *Building Spotify's New Web Player | Labs*. https://labs.spotify.com/2019/03/25/building-spotifys-new-web-player/

Perrott, J. (2020, Maio 19). *Angular—ViewEncapsulation*. https://angular.io/api/core/ViewEncapsulation

Petrova, S. (2019). *The Latest Reports and Stats About Agile [2019] | Adeva*. https://adevait.com/blog/remote-work/adopting-agile-the-latest-reports-about-the-popular-mindset

Piral. (sem data). *Piral—Documentation*. Obtido 15 de Fevereiro de 2020, de https://docs.piral.io/tutorials/01-introduction

Pittet, S. (sem data). *Continuous integration vs. Continuous delivery vs. Continuous deployment*. Obtido 29 de Janeiro de 2020, de https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Podium. (2019a, Junho 13). *Getting Started | Podlet · Podium*. https://podium-lib.io/docs/podlet/getting_started

Podium. (2019b, Junho 14). *Getting Started | Layout · Podium*. https://podium-lib.io/docs/layout/getting_started

Podium. (2019c, Junho 14). *Handling podlet unavailability · Podium*. https://podium-lib.io/docs/layout/unavailable_podlets

Posnick, J. (2018, Maio). *Beyond SPAs: Alternative architectures for your PWA | Web*. Google Developers. https://developers.google.com/web/updates/2018/05/beyond-spa?hl=pt-br

Puzzle-js/puzzle-js. (sem data). GitHub. Obtido 15 de Fevereiro de 2020, de https://github.com/puzzle-js/puzzle-js

Ranchal, R., Mohindra, A., Manweiler, J. G., & Bhargava, B. (2015). Radical Strategies for Engineering Web-Scale Cloud Solutions. *IEEE Cloud Computing*, *2*(5), 20–29. https://doi.org/10.1109/MCC.2015.101

Raymond, E. (1999). *The Cathedral & the Bazaar: Musings On Linux And Open Source By An Accidental Revolutionary*. O'Reilly.

React. (sem data-a). *Higher-Order Components – React*. Obtido 16 de Maio de 2020, de https://reactjs.org/docs/higher-order-components.html

React. (sem data-b). *React.Component – React*. Obtido 16 de Maio de 2020, de https://reactjs.org/docs/react-component.html

React. (sem data-c). *Reconciliation – React*. Obtido 13 de Fevereiro de 2020, de https://reactjs.org/docs/reconciliation.html

Rich, N., & Holweg, M. (2000). Value Analysis—Value Engineering. *INNOREGIO Project*, 32.

Richards, M. (2015). *Software Architecture Patterns*. O'Reilly. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

Richards, M. (2016). *Microservices Antipatterns and Pitfalls*. O'Reilly.

Richardson, C. (2015a). *API gateway pattern*. https://microservices.io/patterns/apigateway.html

Richardson, C. (2015b). *Database per service*. https://microservices.io/patterns/data/database-per-service.html

Richardson, C. (2015c). *Decompose by subdomain*. https://microservices.io/patterns/decomposition/decompose-by-subdomain.html

Richardson, C. (2015d). *Monolithic Architecture pattern*. https://microservices.io/patterns/monolithic.html

Richardson, C. (2018a). *Microservices Pattern: Shared database*. microservices.io. http://microservices.io/patterns/data/shared-database.html

Richardson, C. (2018b). *Microservices Patterns*. Manning.

Robbins, C. (2014, Novembro 5). *Defining Isomorphic Javascript*. https://the-pastry-box-project.net/charlie-robbins/2014-november-5

Rocha, F. (2018, Agosto 14). *O que é Server Side Rendering e como usar na prática*. https://medium.com/techbloghotmart/o-que-%C3%A9-server-side-rendering-e-como-usar-na-pr%C3%A1tica-a840d76a6dca

Samsung. (2015, Janeiro 6). *[Infographic] History of Samsung Things*. https://news.samsung.com/global/infographic-history-of-samsung-things

SAP. (sem data-a). *LuigiProject—Getting Started*. Obtido 14 de Fevereiro de 2020, de https://docs.luigi-project.io/docs/getting-started

SAP. (sem data-b). *Luigi—The Enterprise-Ready Micro Frontend Framework*. Obtido 7 de Maio de 2020, de https://luigi-project.io/about

SAP. (2011, 2020). *2011-Present | SAP History | About SAP SE*. SAP. https://www.sap.com/corporate/en/company/history/2011-present.html

SAP/luigi. (2020). [JavaScript]. SAP. https://github.com/SAP/luigi (Original work published 2018)

Sato, D. (2014, Junho 25). *CanaryRelease*. https://martinfowler.com/bliki/CanaryRelease.html

Schneider, M. (2015, Junho 21). *No. "Isomorphic", in terms of topology, describes the relationship between a transformation applied…*. https://medium.com/@MattiSG/no-isomorphic-in-terms-of-topology-describes-the-relationship-between-a-transformation-applied-1796e69de0e2

Senders, P. (2017, Agosto 15). *Front-end Microservices at HelloFresh—HelloTech*. https://engineering.hellofresh.com/front-end-microservices-at-hellofresh-23978a611b87

Shanmugam, A. K., Vignesh. (2016, Maio 24). *Better streaming layouts for front-end microservices with Tailor*. O'Reilly Media. https://www.oreilly.com/ideas/better-streaming-layouts-for-frontend-microservices-with-tailor

Silva, P. V. da, Esteves, J., Gonçalves, J., & Cordeiro, R. (2019, Julho 8). *Critical Techworks—Working Model*.

Silva, M. (2014, Outubro 6). *JavaScript bubbling e capturing*. iMasters - We are Developers. https://imasters.com.br/front-end/javascript-bubbling-e-capturing

Smith, P. G., & Reinertsen, D. G. (1991). *Developing products in half the time*.

Sowiński, J. (2019, Julho 5). *Microfrontends—Part 2: Integration and communication*. https://medium.com/stepstone-tech/microfrontends-part-2-integration-and-communication-3385bc242673

Stenberg, J. (2018, Agosto 7). *Experiences Using Micro Frontends at IKEA*. InfoQ. https://www.infoq.com/news/2018/08/experiences-micro-frontends/

Strukchinsky, V., & Starkov, V. (2016, Fevereiro 19). *BEM — Block Element Modifier*. http://getbem.com/

Subramanian, H., Raman, A., & Raj, P. (2017). *Architectural Patterns*. Packt Publishing.

Teza, P., Dandolini, G., Souza, J., Miguez, V., Fernandes, R., & Miguel, P. (2015). Modelos de front end da inovação: Similaridades, diferenças e perspectivas de pesquisa. *Production*, *25*. https://doi.org/10.1590/0103-6513.148113

The Nova architecture (Universal Rendering) · Ara Framework. (sem data). Obtido 15 de Fevereiro de 2020, de "https://ara-framework.github.io/website/

TheDomainDrivenDesign.IO. (2019, Março 6). *What is Strategic Design? - DDD - The Domain Driven Design*. https://thedomaindrivendesign.io/what-is-strategic-design/

ThoughtWorks. (2015, Janeiro). *Inverse Conway Maneuver | Technology Radar | ThoughtWorks*. https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., & Edmonds, A. (2015). An architecture for self-managing microservices. *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, 19–24. https://doi.org/10.1145/2747470.2747474

Torre, F. (2017, Agosto 24). *Applying Conway's Law to improve your software development*. ThoughtWorks. https://www.thoughtworks.com/insights/blog/applying-conways-law-improve-your-software-development

UXPin. (2015, Outubro 1). *A Hands-On Guide to Mobile-First Design*. Studio by UXPin. https://www.uxpin.com/studio/blog/a-hands-on-guide-to-mobile-first-design/

Verma, E. (2020, Janeiro 21). *Understanding SAP Modules: SAP FI, SAP CO, SAP SD, SAP HCM and more*. Simplilearn.Com. https://www.simplilearn.com/sap-modules-sap-fi-sap-co-sap-sd-sap-hcm-and-more-rar111-article

Vernon, V. (2015). *Implementing domain-driven design*. Addison-Wesley.

Vernon, V. (2016). *Domain-Driven Design Distilled*. Pearson Education.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano Merino, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., & Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, *11*. https://doi.org/10.1007/s11761-017-0208-y

Webber, E. (2019). *Building Successful Communities of Practice*. Blurb.

Wolff, E. (2016, Novembro 29). *Self-contained Systems: A Different Approach to Microservices*. https://www.innoq.com/en/articles/2016/11/self-contained-systems-different-microservices/

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

Zalando. (sem data). *Project Mosaic—Frontend Microservices*. Obtido 16 de Janeiro de 2020, de https://www.mosaic9.org/

Zalando/tailor. (2020). [JavaScript]. Zalando SE. https://github.com/zalando/tailor (Original work published 2016)

Zerner, A. (2017, Abril 6). *Client-side rendering vs. Server-side rendering—Adam Zerner—Medium*. https://medium.com/@adamzerner/client-side-rendering-vs-server-side-rendering-a32d2cf3bfcc

Anexo A – Diagramas da Arquitetura Nova da Ara Framework

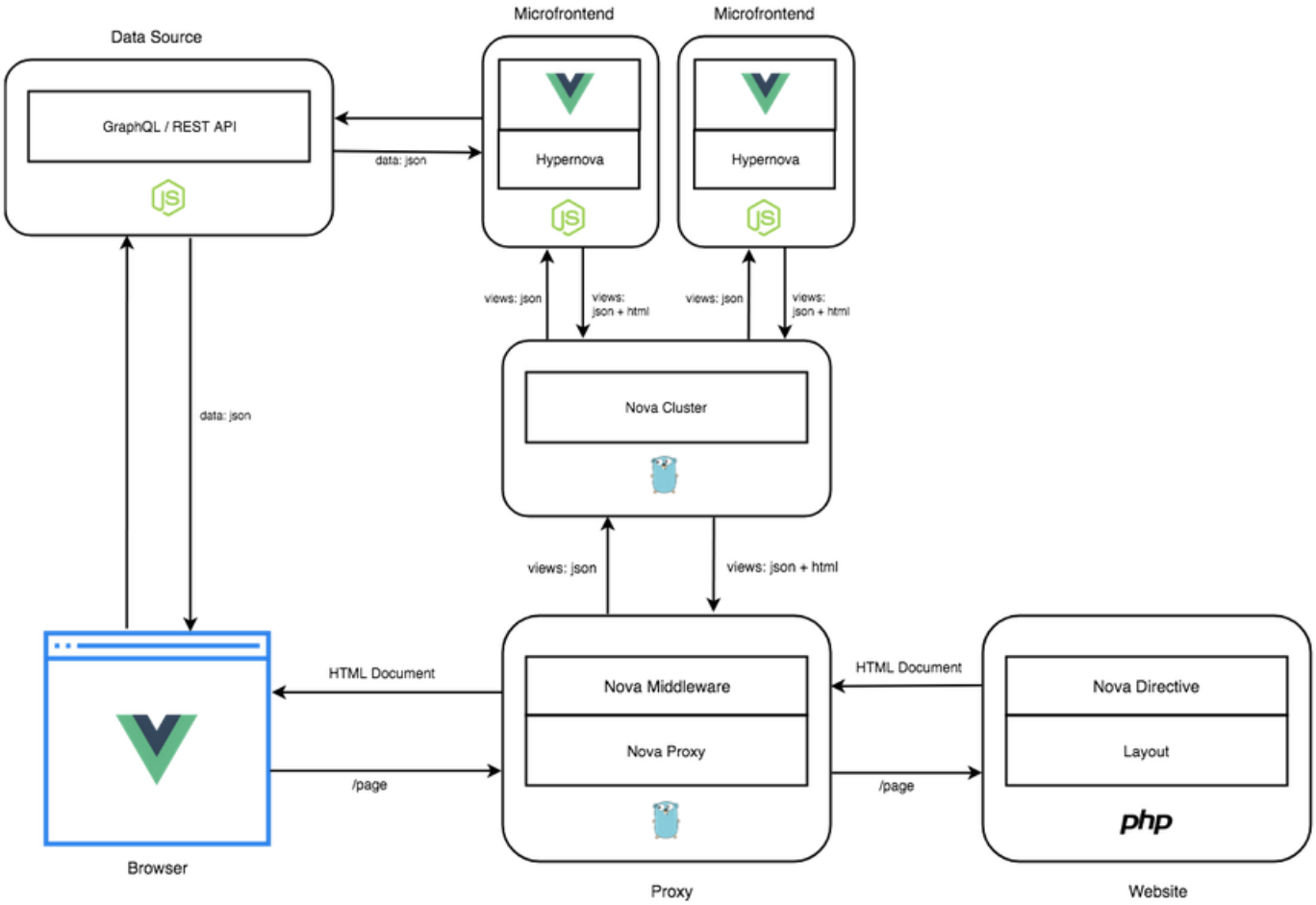


Figura 58 – Arquitetura Nova baseada em *Universal Rendering*

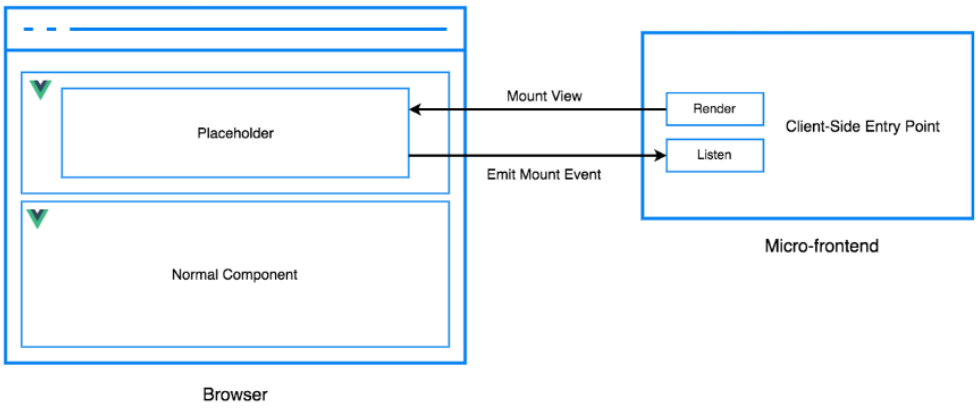


Figura 59 - Arquitetura Nova baseada em *Client-Side Rendering*

Anexo B – Utilização de Single-Spa-Layout na Aplicação Tellco

```
<template id="single-spa-layout">
  <single-spa-router>
    <nav>
      <application name="@msc-tellco/navbar"></application>
    </nav>
    <div class="main-content">
      <route path="cart">
        <application name="@msc-tellco/shopping-cart"></application>
      </route>
      <route path="/">
        <application name="@msc-tellco/catalogue"></application>
      </route>
      <route default>
        <h1 class="flex flex-row justify-center p-16">
          <p class="max-w-md">Nice try!</p>
        </h1>
      </route>
    </div>
  </single-spa-router>
</template>
```

Código 12 – Aplicação de single-spa-layout no ficheiro HTML da Aplicação Tellco para *Routing* de 1ºNível

Anexo C – Aplicação Tellco

Figura 60– *Navbar* da Aplicação Tellco

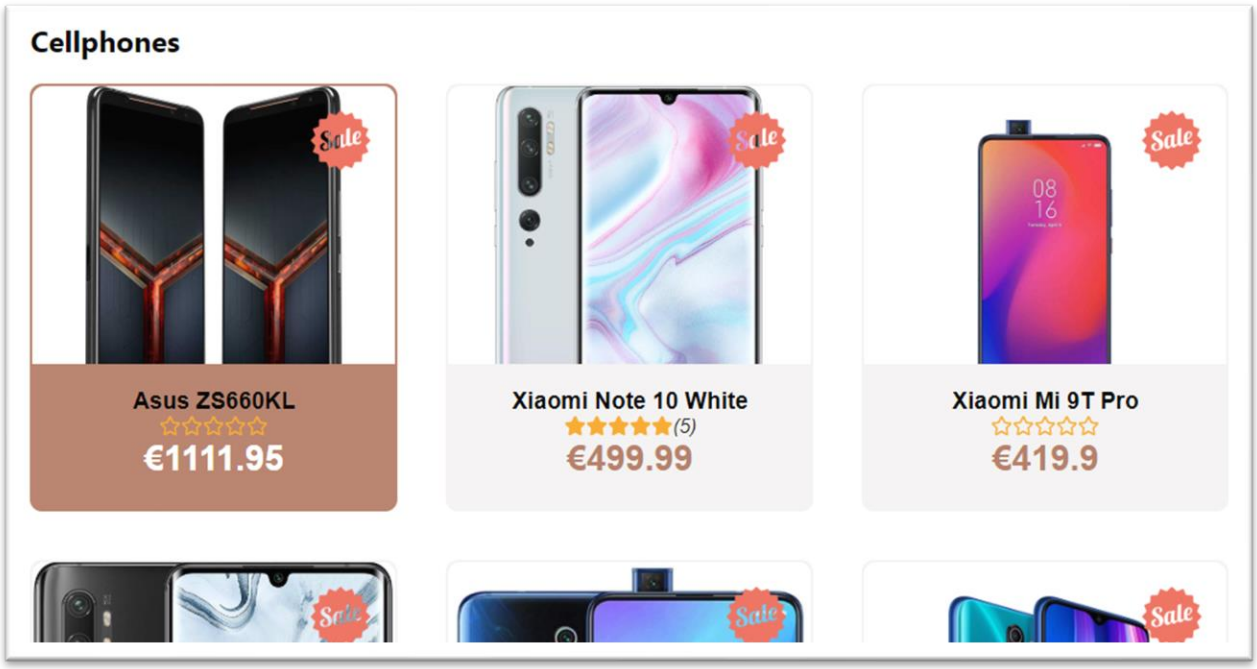


Figura 61– Catálogo de Produtos da Aplicação Tellco

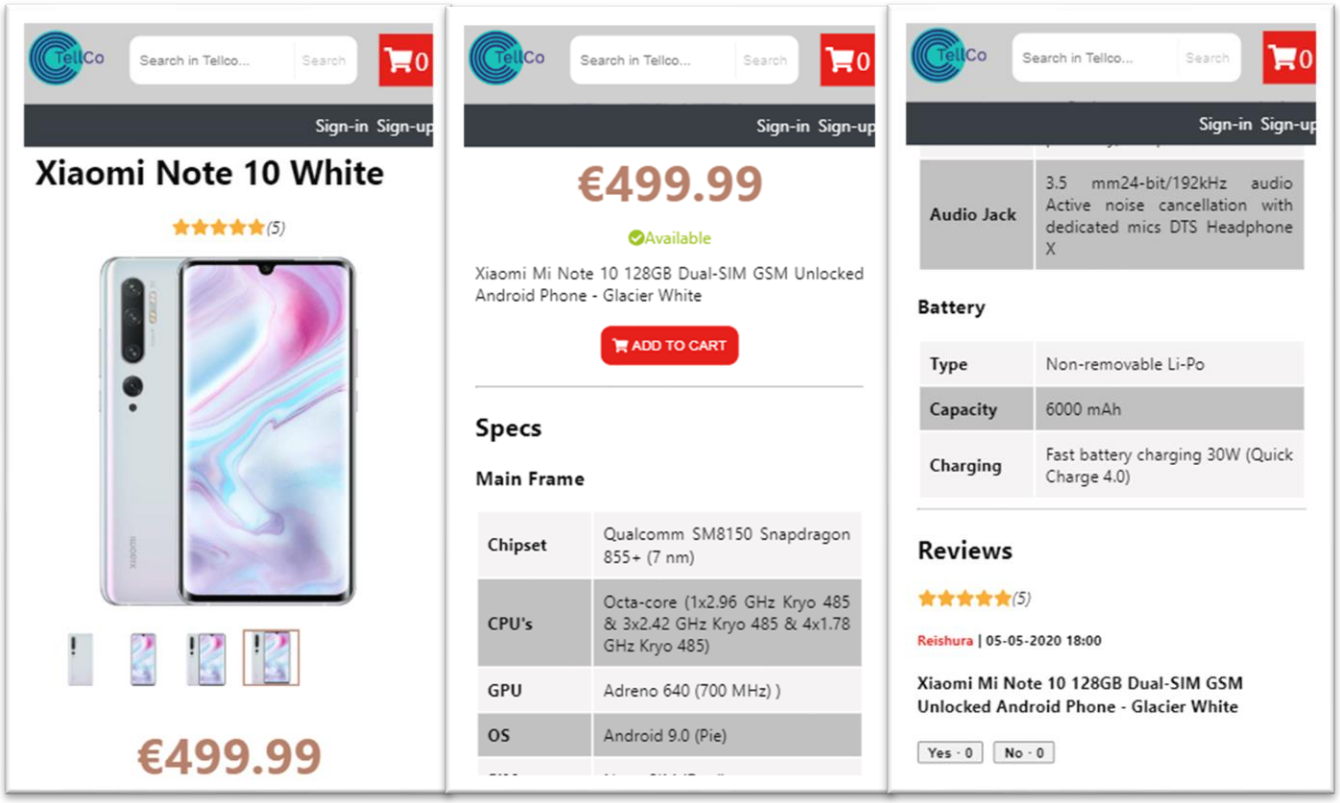


Figura 62 – Página de Detalhe de Produto da Aplicação Tellco

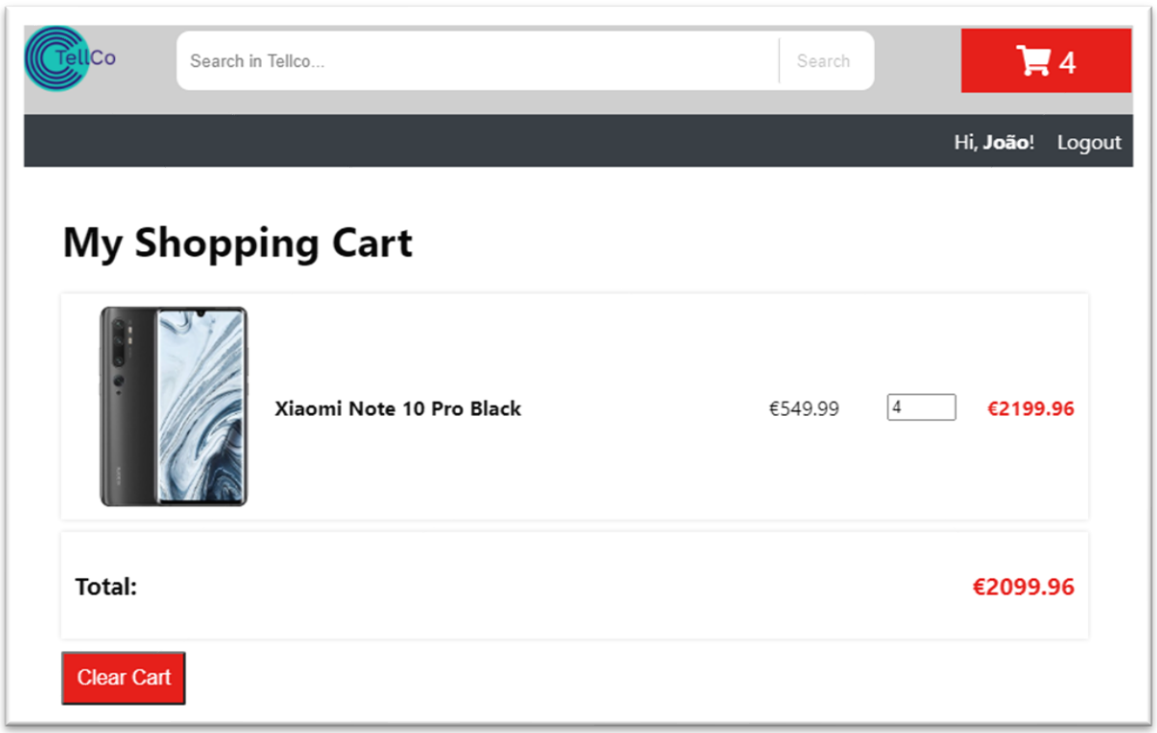


Figura 63 – Página e *Widget* de Carrinho de Compras da Aplicação Tellco

Anexo D – Questionário Aplicado

Secção 1 – Introdução do Questionário (Acesso Geral)

Desenvolvimento Web Orientado a Micro Frontends

O presente inquérito é realizado no âmbito de uma tese de mestrado cujo tema consiste no Desenvolvimento Web Orientado à Micro-Frontends e visa aferir a viabilidade e os impactos da adoção desta abordagem quer no processo de desenvolvimento de software, quer na organização, dinâmica e produtividade das equipas.

Um micro-frontend consiste numa camada vertical de uma aplicação que é desenvolvida de forma end-to-end (desde a base de dados até ao frontend) por uma equipa multidisciplinar especializada num determinado subdomínio de negócio, constituida por especialistas de domínio, especialistas de software e equipa de desenvolvimento (que inclui designers).
Sendo uma abordagem baseada na arquitetura de micro-serviços, os seus proponentes afirmam que conferindo uma maior autonomia às equipas ao longo do processo de desenvolvimento decrementa os ciclos de desenvolvimento e release, o que permite agilizar o lançamento iterativo e incremental de MVPs para o cliente e consequentemente criar valor e obter feedback contínuo.

Nesse sentido, este estudo é direcionado a colaboradores de empresas de desenvolvimento de software cuja função esteja direta ou indiretamente à produção de software, estudantes ou entusiastas do ramo da computação, engenharia e/ou desenvolvimento de software.

O tempo médio previsto para o preenchimento do questionário é de 4 a 10 minutos.

Secção 2 - Experiência Profissional (Acesso Geral) - 2 questões

1. Há quantos anos trabalha no ramo do desenvolvimento de software? *

Marcar apenas uma oval.

- 0 - 2 Anos
- 2 - 5 Anos
- > 5 Anos
- > 10 Anos

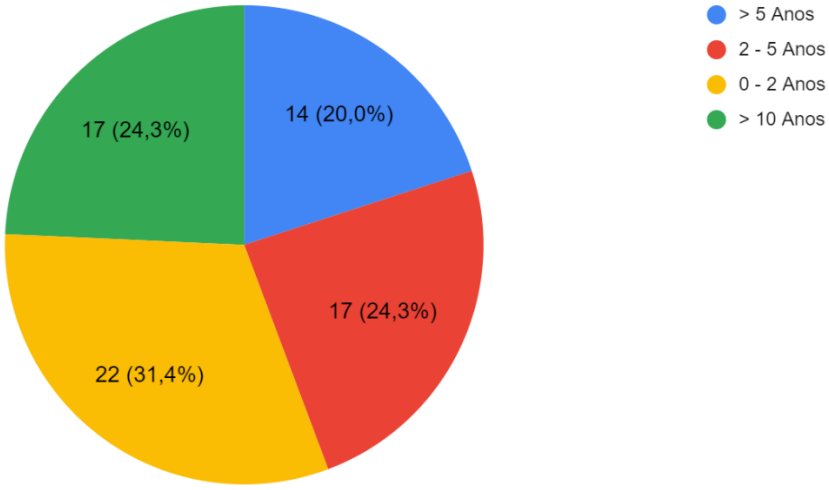


Figura 64 – Gráfico de resultados da questão 1 do questionário

2. Neste momento encontra-se a ... *

Marcar apenas uma oval.

- ◯ Trabalhar numa empresa dedicada ao desenvolvimento de software
- ◯ Estudar engenharia informática, ciências da computação ou similar *Avançar para a pergunta 26*
- ◯ Pesquisar e aplicar conceitos de desenvolvimento web por iniciativa ou gosto pessoal *Avançar para a pergunta 26*
- ◯ Nenhuma das anteriores *Avançar para a secção 13 (Final do Questionário)*

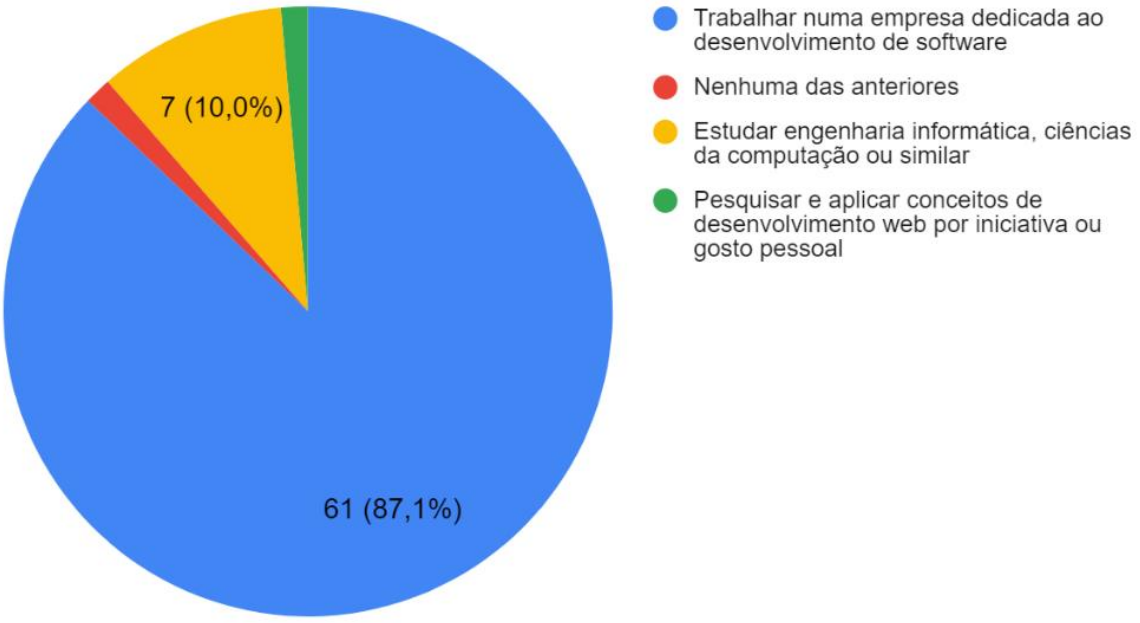


Figura 65 – Gráfico de resultados da questão 2 do questionário

Secção 3 – Categoria da Função Atual (Acesso restrito aos inquiridos que responderam "Trabalhar numa empresa dedicada ao desenvolvimento de *software*" na última questão da secção 2) - 2 questões

3. Já trabalhou segundo metodologias ágeis (SCRUM, Kanban, etc.) ? *

Marcar apenas uma oval.

- ◯ Sim
- ◯ Não

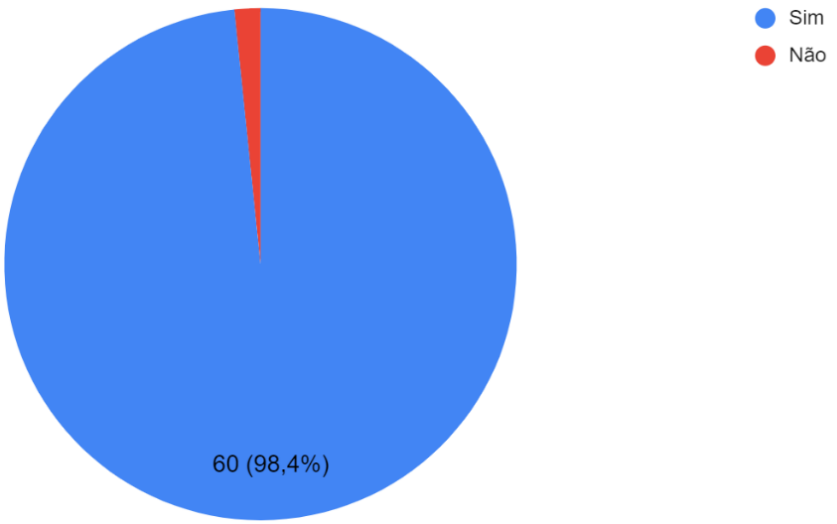


Figura 66 – Gráfico de resultados da questão 3 do questionário

4. A sua função atual pode ser enquadrada em: *

Marcar apenas uma oval.

- Equipa de Produto (Developers & QAs, Designers, Product Owners, Scrum Masters, etc)
- Arquiteto, Tech Lead, Team Manager ou Project Manager ou equiparado
- Outro *Avançar para a secção 13 (Final do Questionário)*

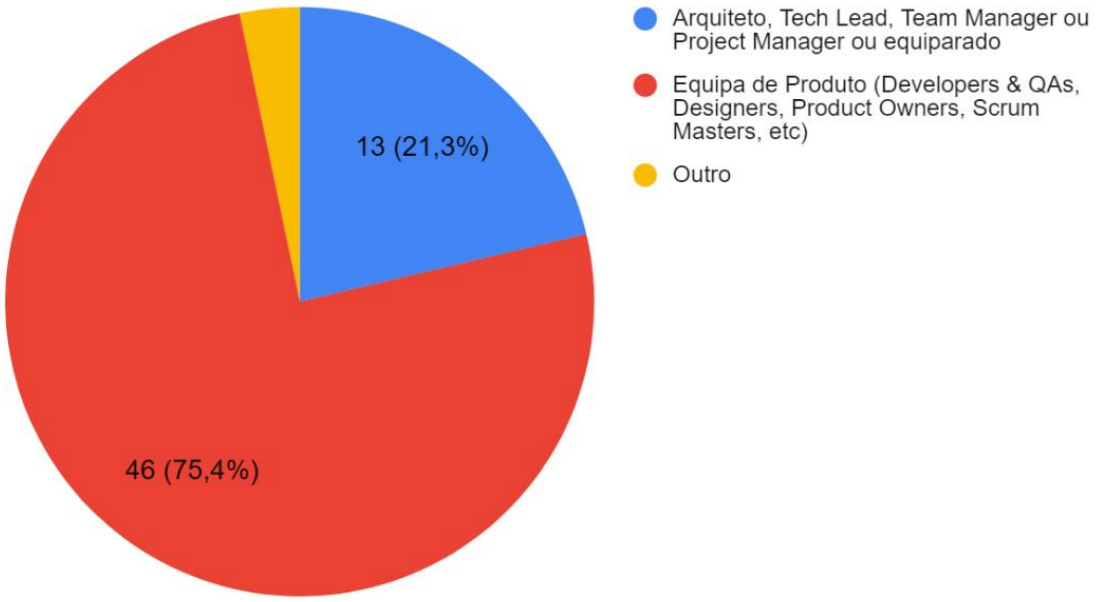


Figura 67 – Gráfico de resultados da questão 4 do questionário

Secção 4 – Modelo Organizacional (Acesso restrito a Arquitetos, *Tech Leaders, Team Managers, Product Manager* e Equipa de Produto) - 1 questão

5. Relativamente às equipas de desenvolvimento, considera mais vantajoso que sejam... *

Marcar apenas uma oval.

Multidisciplinares (congregando conhecimentos de diferentes camadas, por exemplo, Frontend, Backend e Base de Dados)

Especializadas numa determinada camada aplicacional (por exemplo, Frontend, Backend ou Bases de Dados) *Avançar para a pergunta 8*

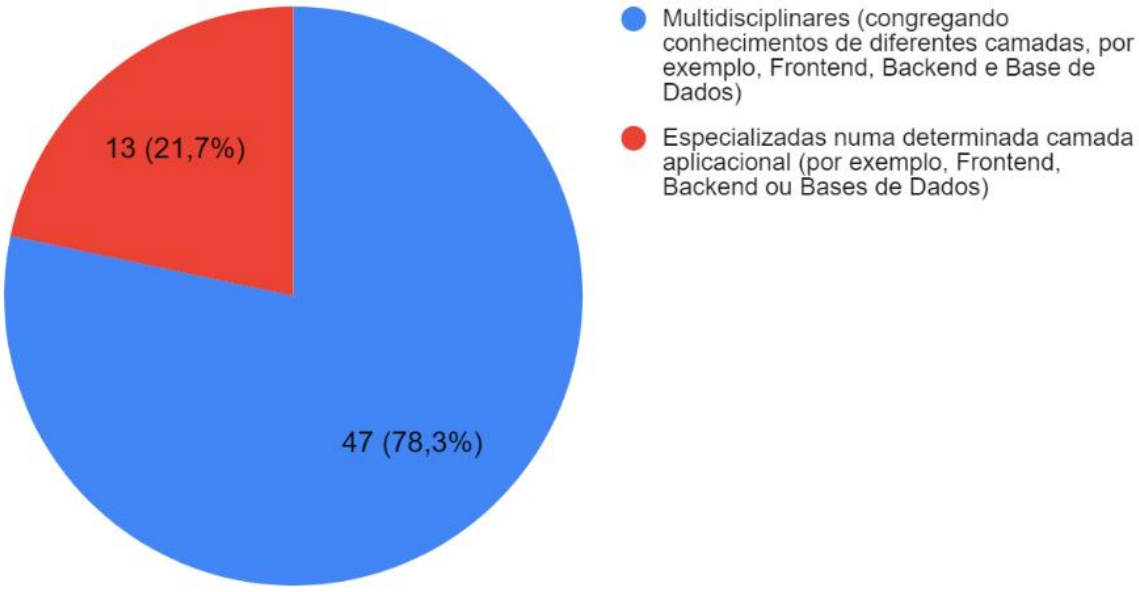


Figura 68 – Gráfico de resultados da questão 5 do questionário

Secção 5 – Equipas Multidisciplinares (Acesso restrito a Arquitetos, *Tech Leaders, Team Managers, Product Manager* e Equipa de Produto) – 2 questões

6. A especialização em determinadas áreas de negócio enaltece o sentimento de pertença, fazendo com que as equipas se identifiquem com o produto e se crie um compromisso mais forte entre a organização e os seus colaboradores.

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Totalmente						Concordo Plenamente

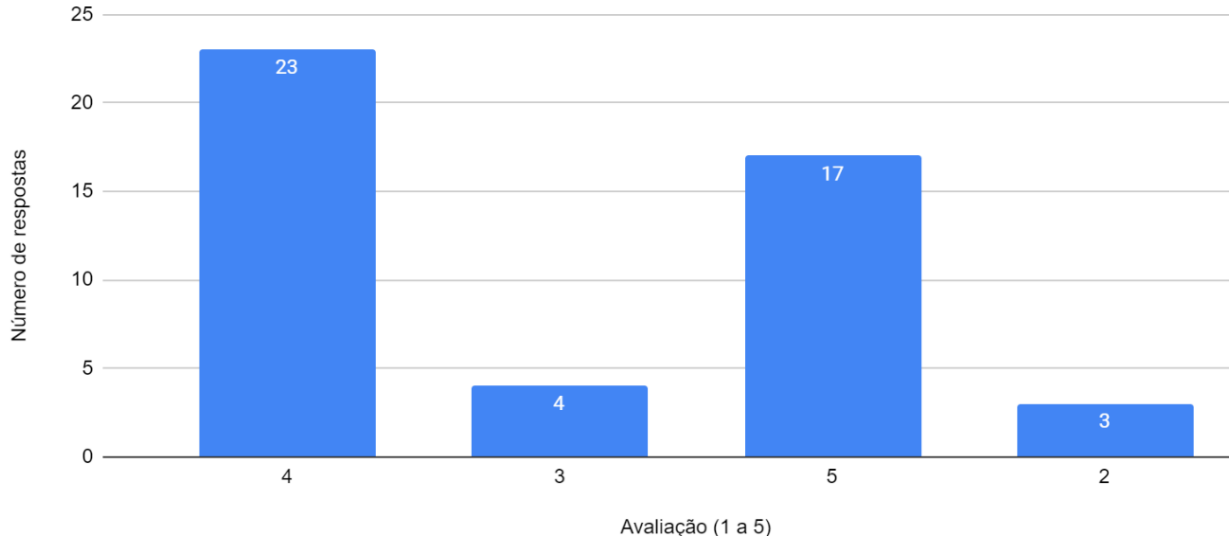


Figura 69 – Gráfico de resultados da questão 6 do questionário

7. A especialização destas equipas em determinadas áreas de negócio permite que identifiquem mais facilmente o que pode ser adicionado ao produto para gerar mais valor para o utilizador.

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Totalmente	◯	◯	◯	◯	◯	Concordo Plenamente

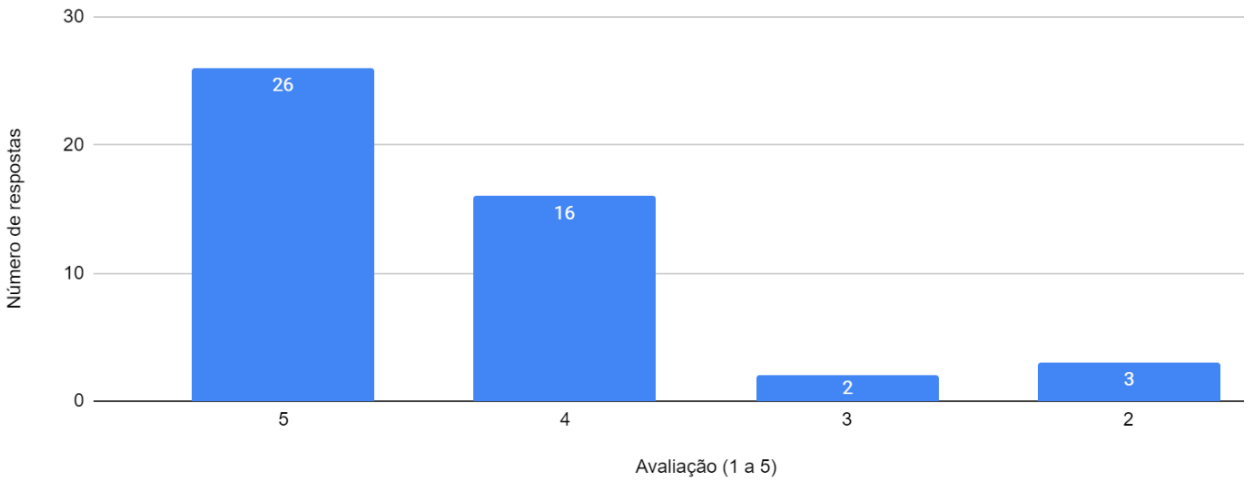


Figura 70 – Gráfico de resultados da questão 7 do questionário

Secção 6 – Adoção de Micro *Frontends* – Parte 1 (Acesso restrito a Arquitetos, *Tech Leaders, Team Managers, Product Managers* e Equipa de Produto) - 10 questões

8. Considera que a definição de equipas de produto que incluam Product Owner, QAs, Designers e Developers, etc. permite agilizar o processo de desenvolvimento no seu conjunto ou cria entropias ou bloqueios adicionais? *

Marcar apenas uma oval.

- Por norma, agiliza o processo de desenvolvimento
- Cria entropia ou bloqueios que podem traduzir-se em atrasos no processo de desenvolvimento

Figura 71 – Gráfico de resultados da questão 8 do questionário

9. Qual a constituição atual da sua equipa ? *

Marcar apenas uma oval.

- Equipa de Desenvolvimento e Product Owner
- Equipa de Desenvolvimento e Designers
- Equipa de Desenvolvimento, Product Owner e Designers
- Apenas Equipa de Desenvolvimento
- Outra

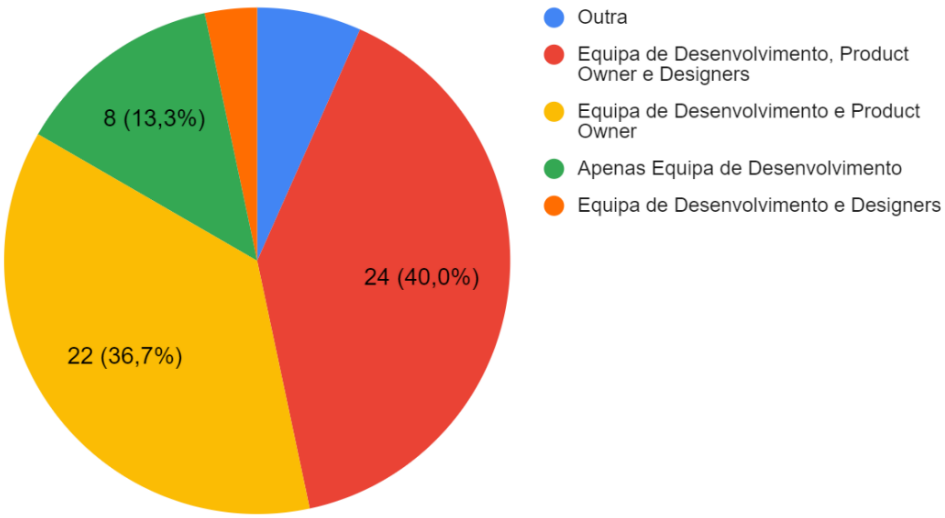


Figura 72 – Gráfico de resultados da questão 9 do questionário

10. Relativamente à duração dos ciclos de desenvolvimento por parte de equipas multidisciplinares face a equipas especializadas numa determinada camada, considera que são: *

Marcar apenas uma oval.

- Mais curtos
- Mais longos
- Similares

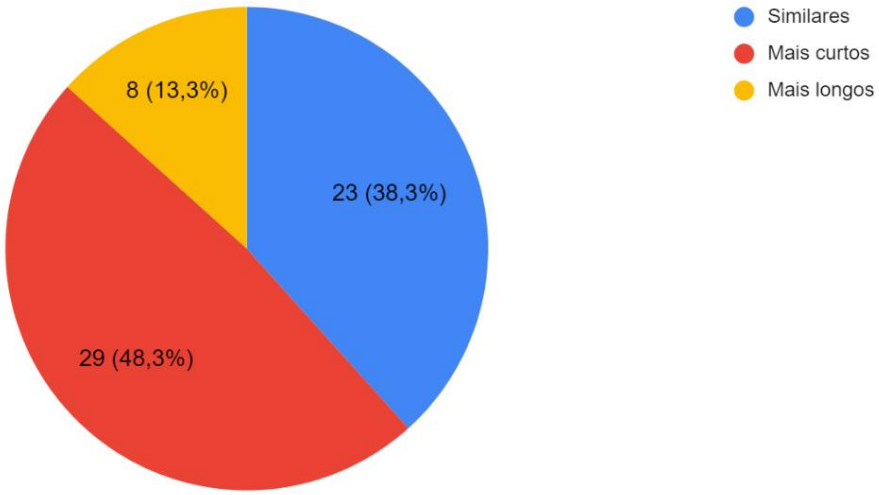


Figura 73 – Gráfico de resultados da questão 10 do questionário

11. Considera que equipas multidisciplinares denotam uma melhoria significativa na comunicação e requerem um menor esforço de coordenação? *

Marcar apenas uma oval.

Sim
Não

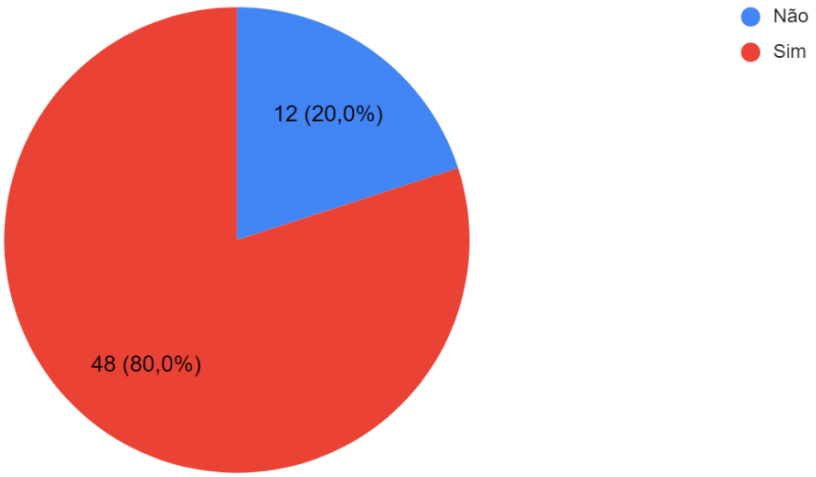


Figura 74 – Gráfico de resultados da questão 11 do questionário

12. O facto de as equipas serem multidisciplinares e autónomas faz com que sejam mais evidentes as responsabilidades de cada equipa? *

Marcar apenas uma oval.

Sim
Não

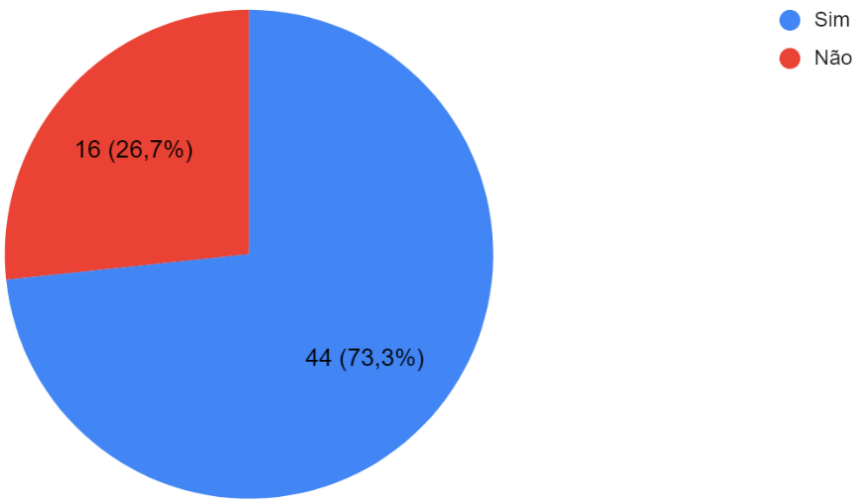


Figura 75 – Gráfico de resultados da questão 12 do questionário

13. Considera que esta organização de equipas permite dar uma resposta mais célere ao feedback obtido? *

Marcar apenas uma oval.

Sim

Não

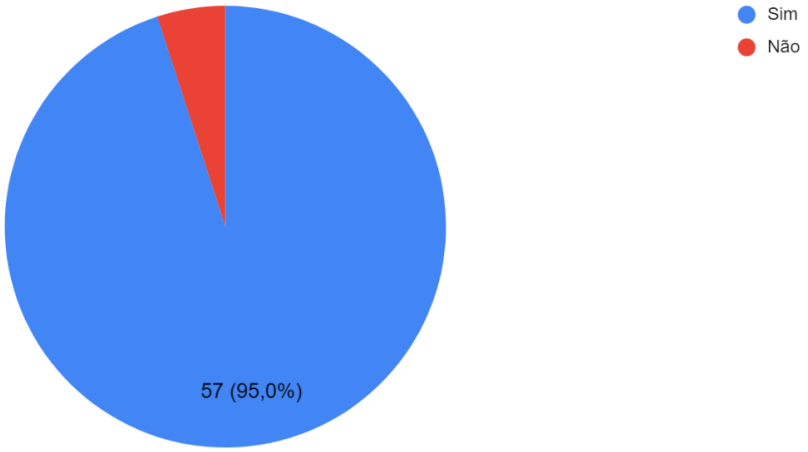


Figura 76 – Gráfico de resultados da questão 13 do questionário

14. No que toca à autonomia da equipa o que considera uma aposta mais segura? *

Marcar apenas uma oval.

Autonomia quase total para tomar decisões ainda que em conformidade com os compromissos firmados e guidelines organizacionais

Autonomia apenas para tomar decisões de implementação (estando a equipa submetida às diretrizes da organização e às decisões tomadas por especialistas/superiores hierárquicos)

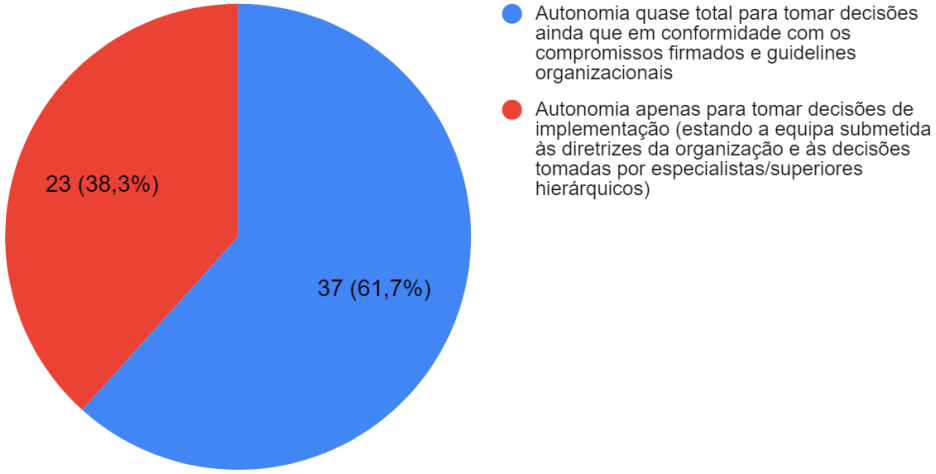


Figura 77 – Gráfico de resultados da questão 14 do questionário

15. Considera que a possibilidade de as equipas terem mais autonomia pode representar um risco acrescido de inconsistência de UIs entre as aplicações? *

Marcar apenas uma oval.

- Sim
- Não

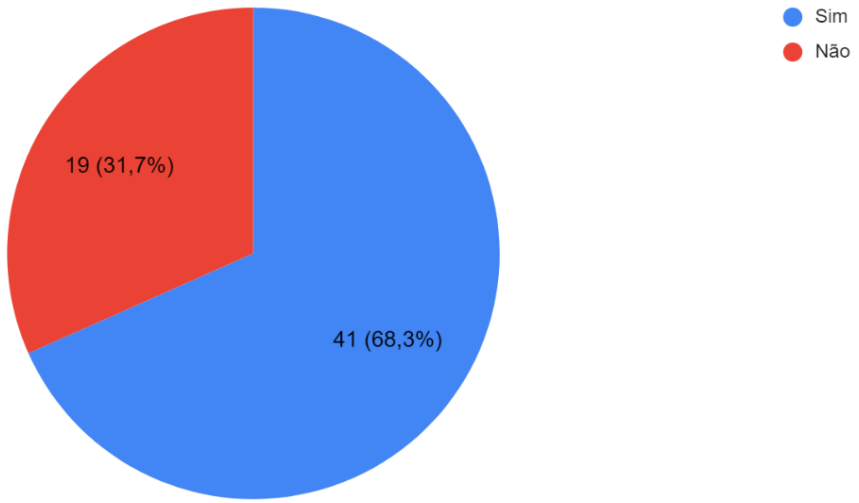


Figura 78 – Gráfico de resultados da questão 15 do questionário

16. Se respondeu "Sim" à questão anterior, considera que a existência de um Design System transversal à organização pode minimizar o risco de inconsistências visuais?

Marcar apenas uma oval.

- Sim
- Não

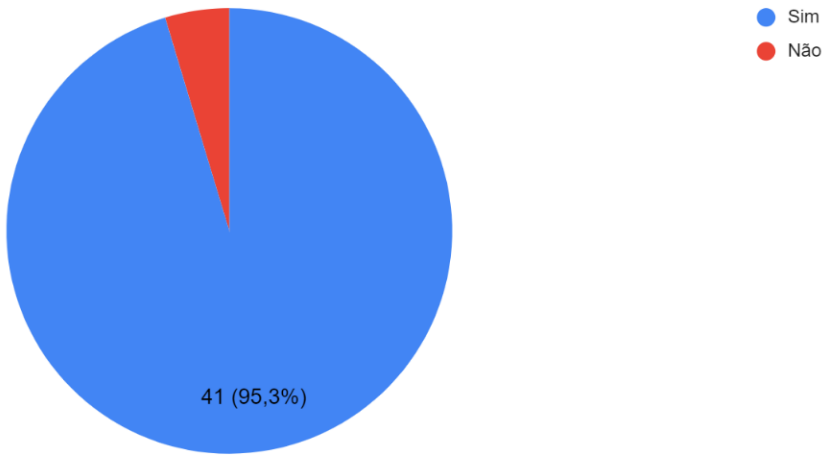


Figura 79 – Gráfico de resultados da questão 16 do questionário

17. A inclusão de Designers na equipa de Produto permite detetar e corrigir mais cedo inconformidades nas especificações visuais. *

Marcar apenas uma oval.

- Sim
- Não
- Não Sei

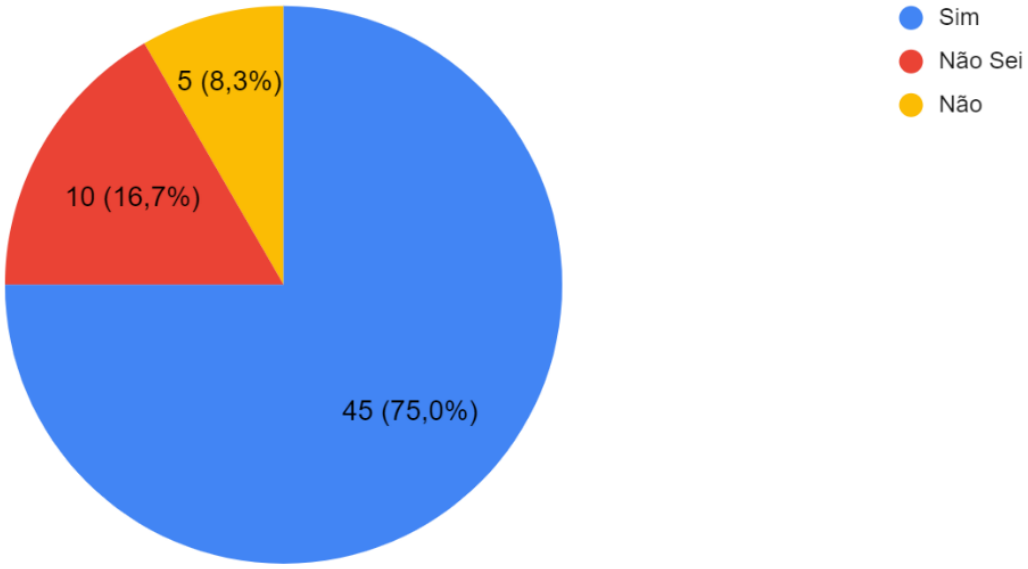


Figura 80 – Gráfico de resultados da questão 17 do questionário

Secção 7 – Adoção de Micro *Frontends* - Parte 2 (Acesso restrito a Arquitetos, Tech Lead, Team Managers, Product Manager e Equipa de Produto) - 8 questões

18. Considera que trabalhando as equipas de forma independente há um maior risco de se criarem "silos de conhecimento" ? *

Marcar apenas uma oval.

- Sim
- Não

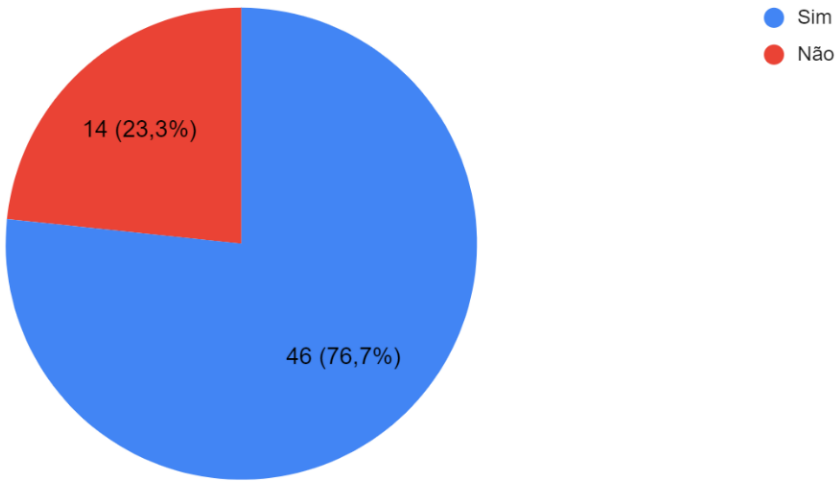


Figura 81 – Gráfico de resultados da questão 18 do questionário

19. Selecione das seguintes as duas estratégias que considera mais eficazes para minimizar a formação de silos de conhecimento *

Marcar tudo o que for aplicável.

- [ ] Criação e promoção de Communities of Practice (CoP)
- [ ] Identificadas as forças e fraquezas de uma equipa apostar na formação
- [ ] Apresentar o que tem vindo a ser produzido a outras equipas
- [ ] Documentar com rigor as soluções desenvolvidas

Outra: [ ] _____

Figura 82 – Gráfico de resultados da questão 19 do questionário

20. Considera que tendo as equipas autonomia para decidir quando proceder ao deploy/release de novas funcionalidades que é possível produzir MVPs num menor espaço de tempo? *

Marcar apenas uma oval.

- Sim
- Não
- É Igual

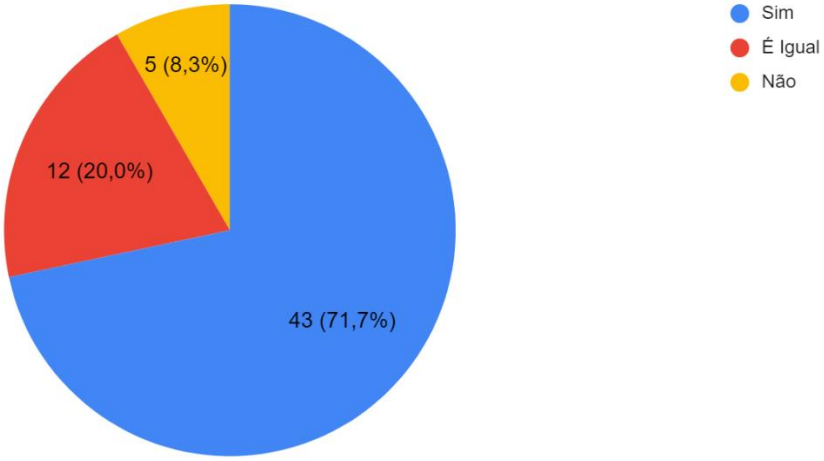


Figura 83 – Gráfico de resultados da questão 20 do questionário

21. Considera viável implementar uma organização baseada em equipas multidisciplinares em startups ? *

Marcar apenas uma oval.

- Sim
- Não
- Não Sei

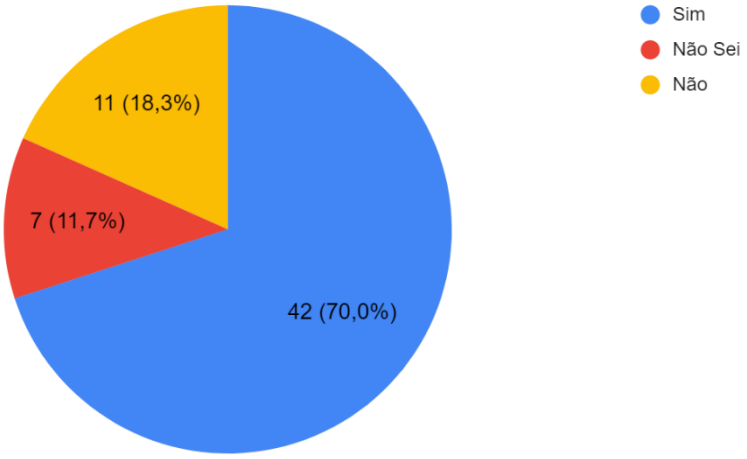


Figura 84 – Gráfico de resultados da questão 21 do questionário

22. Se respondeu "não" assinale qual o principal motivo

Marcar apenas uma oval.

- Reduzido número de colaboradores (<12)
- Volatilidade do modelo de negócio
- Outra: _____

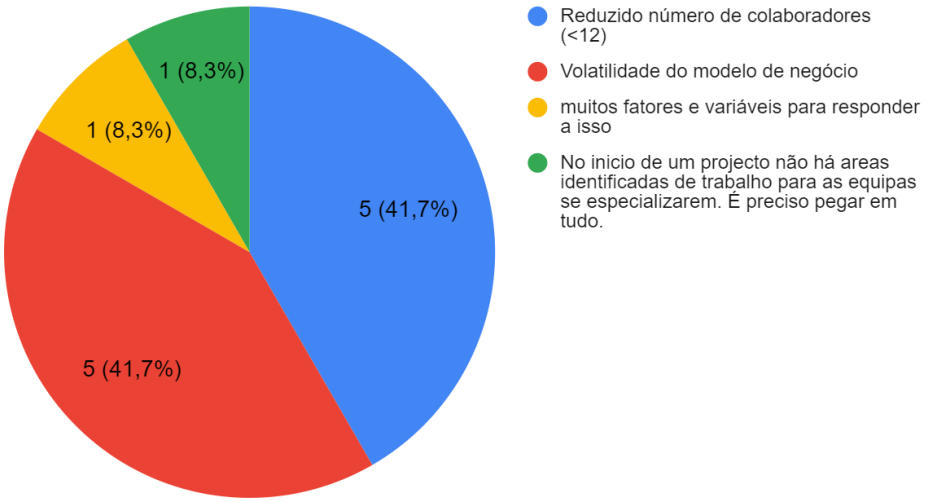


Figura 85 – Gráfico de resultados da questão 22 do questionário

23. Considera viável implementar uma organização baseada em equipas multidisciplinares em empresas mais consolidadas no mercado e/ou com projetos de maior dimensão? *

Marcar apenas uma oval.

- Sim
- Não
- Não Sei

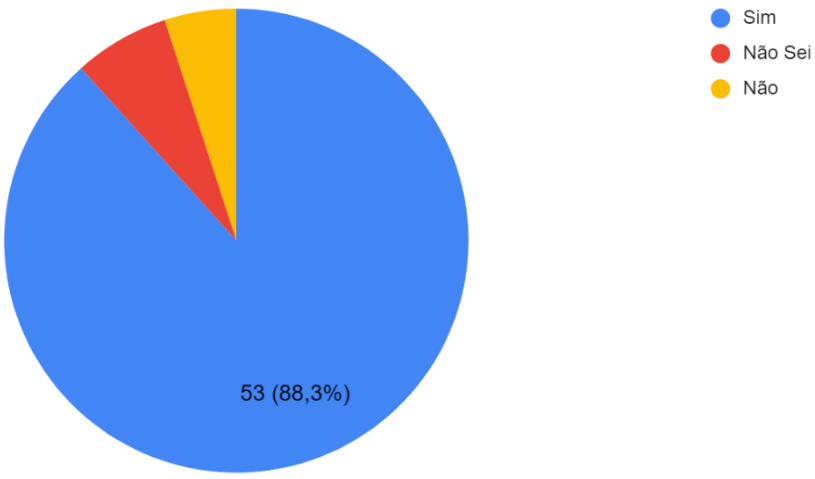


Figura 86 – Gráfico de resultados da questão 23 do questionário

24. De uma maneira geral considera que uma estrutura organizacional assente em equipas multidisciplinares aumenta a produtividade das equipas ? *

Marcar apenas uma oval.

Sim
Não

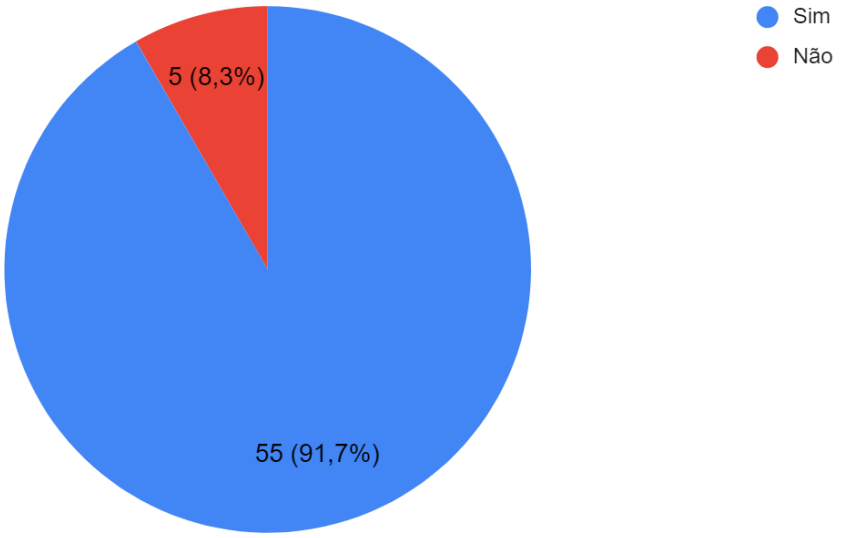


Figura 87 – Gráfico de resultados da questão 24 do questionário

25. Qual é a função que desempenha atualmente na sua equipa ? *

Marcar apenas uma oval.

Developer, Arquiteto de Software, Tech Lead ou equiparado
Product Owner, Scrum Master, Designer ou Outra *Avançar para a pergunta 57*

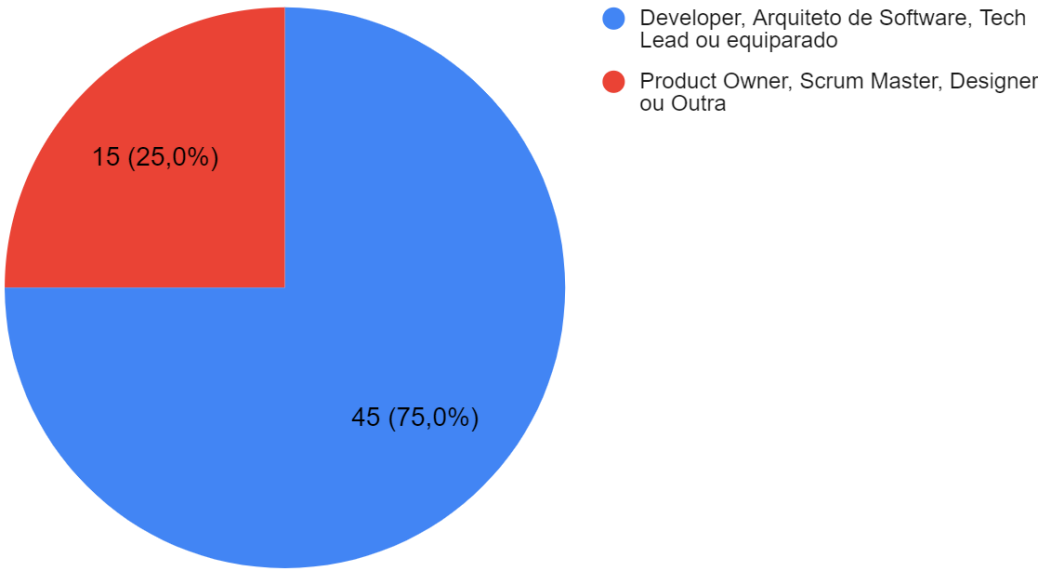


Figura 88 – Gráfico de resultados da questão 25 do questionário

Secção 8 – Familiaridade com o Conceito de Micro *Frontends* (Acesso restrito a *Developers,* Arquitetos, *Tech Leaders,* Estudantes de Eng. Informática e Autodidatas) - 2 questões

26. Com base nos seus conhecimentos de desenvolvimento de software, considera que tem um perfil de: *

Marcar apenas uma oval.

- Frontend Developer
- Backend Developer
- Full Stack
- Mobile Developer
- Desktop Developer
- Arquiteto de Software / Tech Lead
- Outro

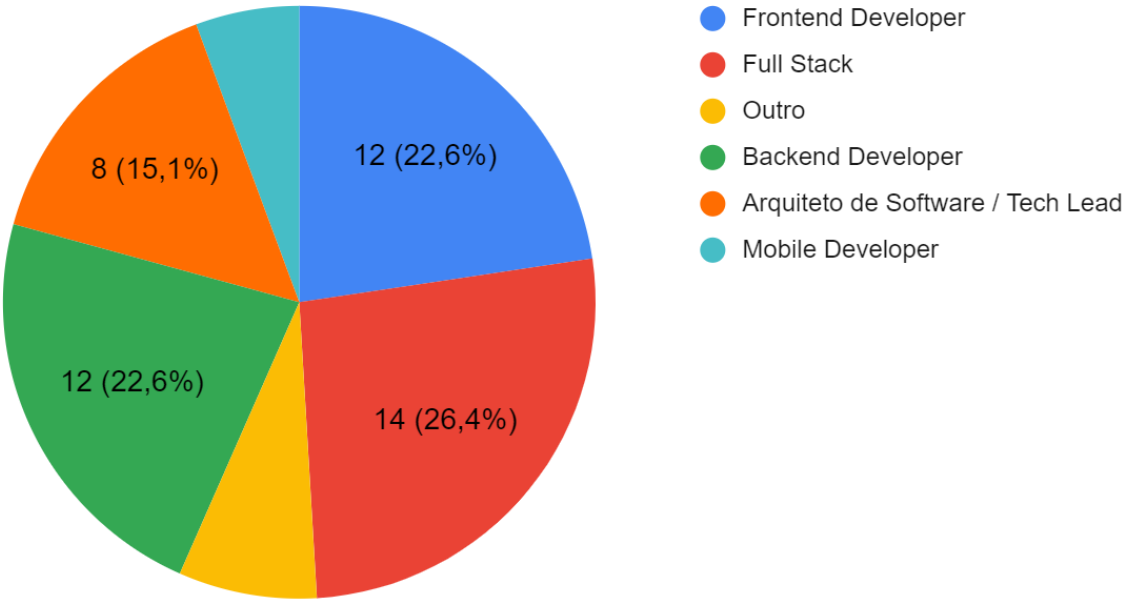


Figura 89 – Gráfico de resultados da questão 26 do questionário

27. Já tinha ouvido falar ou lido algo acerca do conceito de "Micro-Frontends" ? *

Marcar apenas uma oval.

- Sim
- Não *Avançar para a secção 13 (Final do Questionário)*

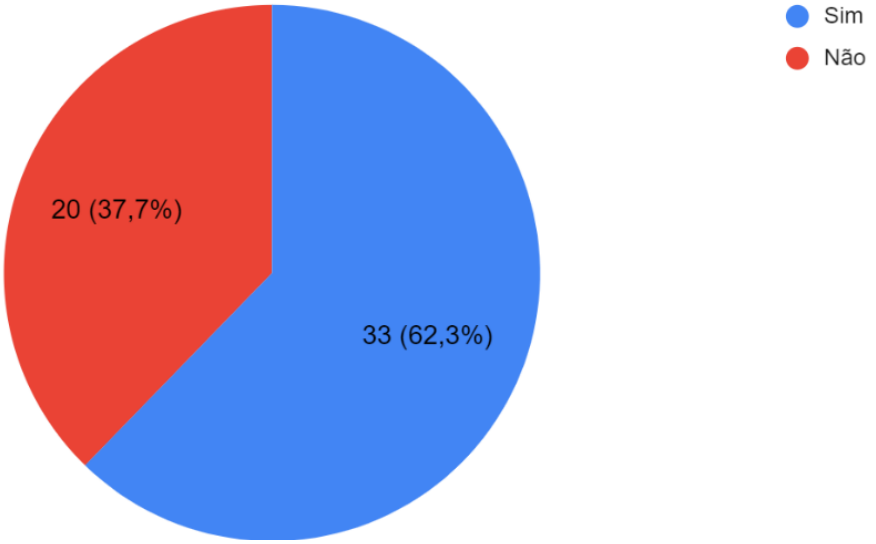


Figura 90 – Gráfico de resultados da questão 27 do questionário

Secção 9 – Aplicação de Micro *Frontends* em Projetos de Desenvolvimento (Acesso restrito aos inquiridos que indicaram conhecer Micro *Frontends*) - 3 questões

28. Já adotou Micro-Frontends em algum projeto de desenvolvimento de software ? *

Marcar apenas uma oval.

Sim
Não

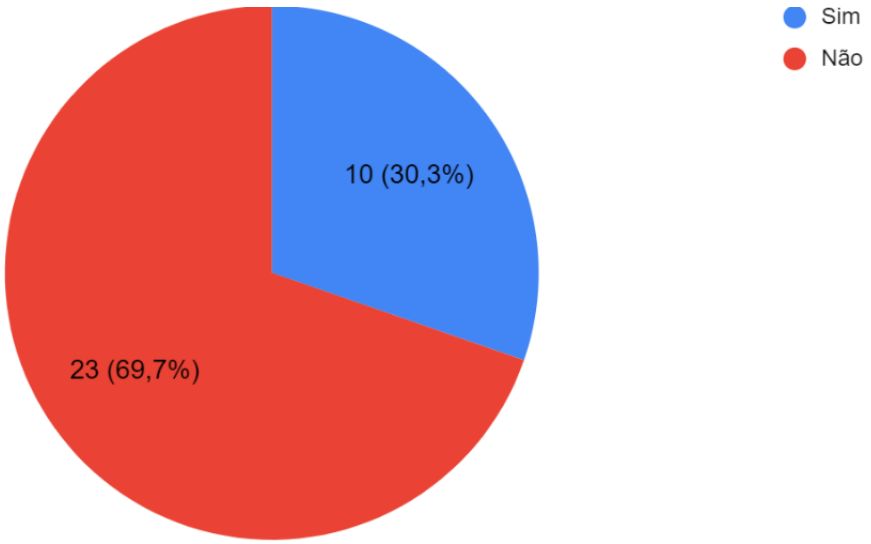


Figura 91 – Gráfico de resultados da questão 28 do questionário

29. Se respondeu "Sim" à questão anterior, descreva em poucas palavras o propósito do projeto, o que levou a optar por Micro-Frontends e quais as principais vantagens e desafios que identificou

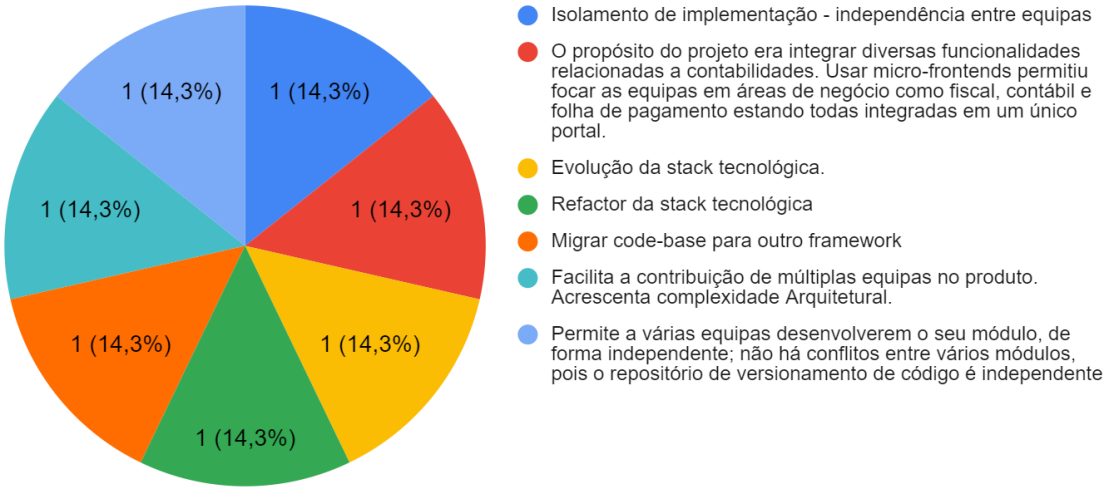


Figura 92 – Gráfico de resultados da questão 29 do questionário

30. Em que tipo de projetos de desenvolvimento web considera que faz mais sentido? *

Marcar apenas uma oval.

- ◯ Projetos de pequena dimensão e/ou com um modelo de negócio mais simples e limitado
- ◯ Projetos de média/grande dimensão e/ou com um modelo de negócio vasto e complexo
- ◯ Não usaria Micro-Frontends *Avançar para a pergunta 57*

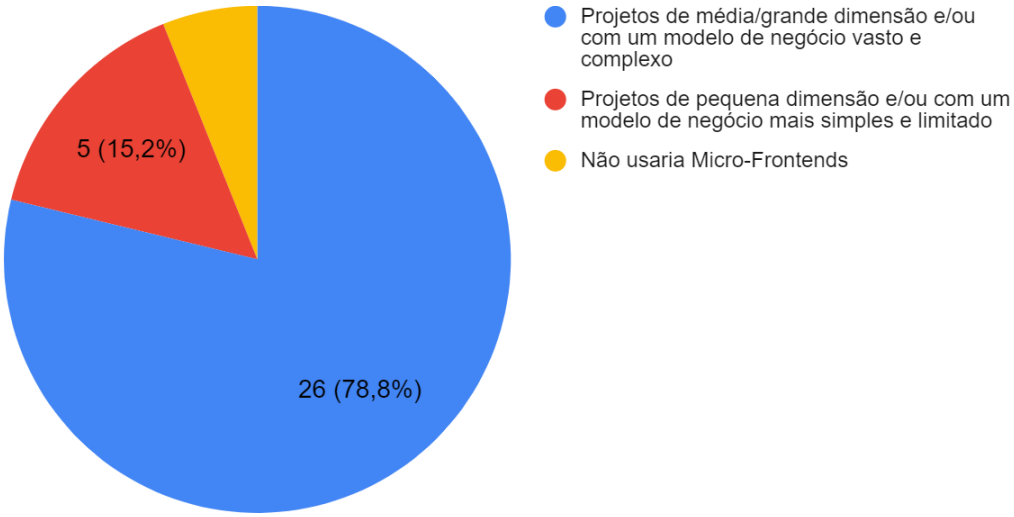


Figura 93 – Gráfico de resultados da questão 30 do questionário

Secção 10 – Integração e Comunicação de Micro *Frontends* (Acesso restrito aos inquiridos que indicaram conhecer Micro *Frontends*) - 17 questões

31. É possível uma total integração de micro-frontends desenvolvidos por equipas distintas entre si e com a aplicação integradora *

Marcar apenas uma oval.

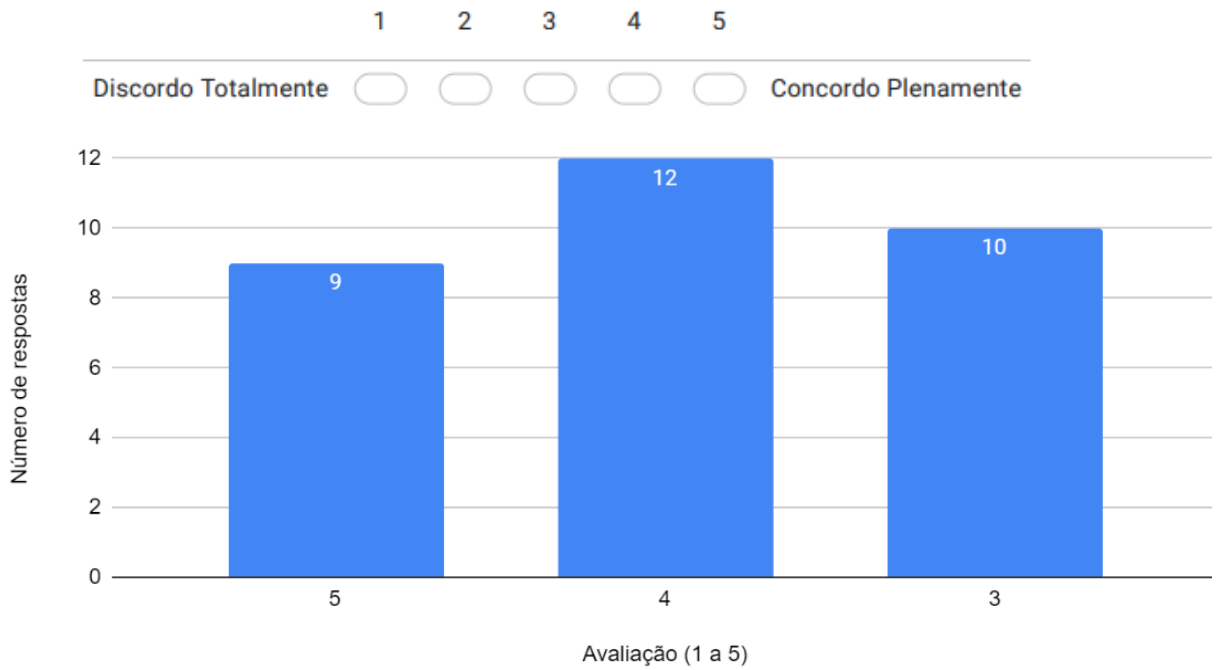


Figura 94 – Gráfico de resultados da questão 31 do questionário

32. Que técnicas de integração de micro-frontends conhece ? *

Marcar tudo o que for aplicável.

- [ ] Integração via Links (Client-Side)
- [ ] IFrames (Client-Side)
- [ ] AJAX (Client-Side)
- [ ] Web Components (Client-Side)
- [ ] Server Side Includes (Server-Side)
- [ ] Single Page Application Unificada usando App Shell (Client-Side)
- [ ] Edge Side Includes (Server-Side)

Outra: ☐ _____

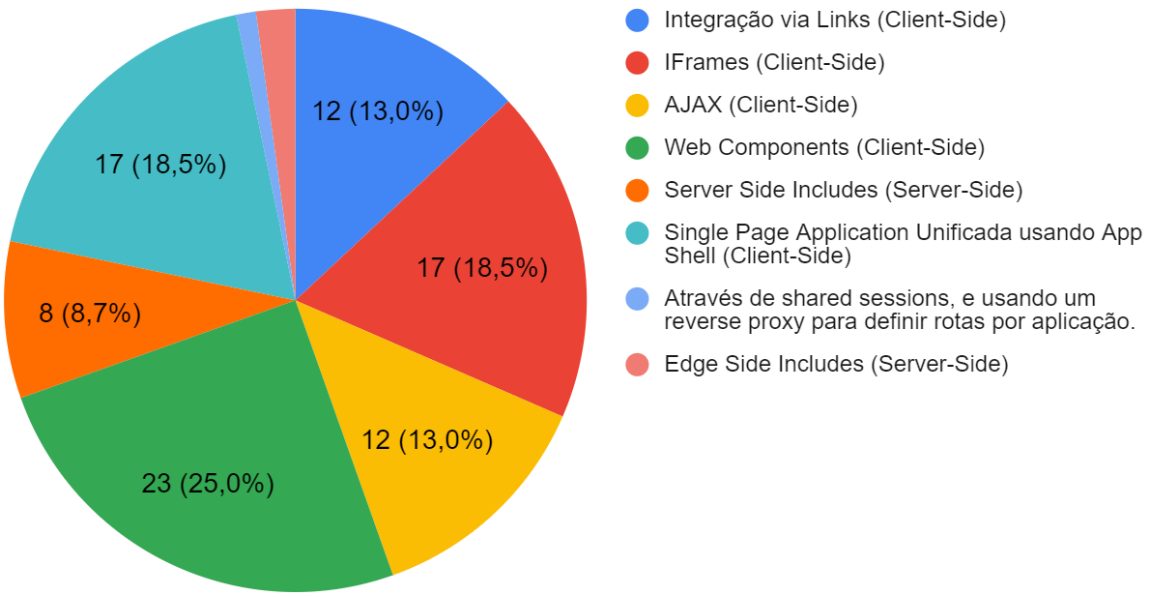


Figura 95 – Gráfico de resultados da questão 32 do questionário

33. Conhece alguma biblioteca de integração de micro-frontends ? *

Marcar apenas uma oval.

Sim

Não

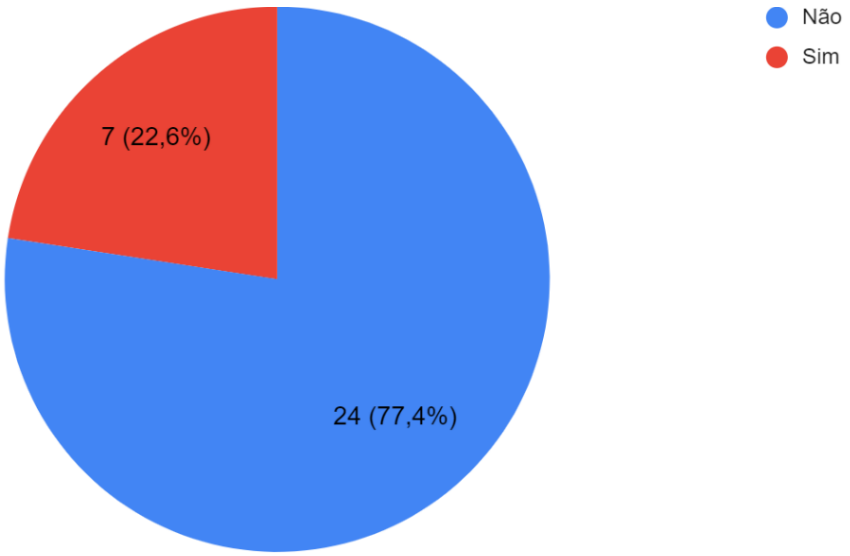


Figura 96 – Gráfico de resultados da questão 33 do questionário

34. Se respondeu "Sim" à questão anterior, que bibliotecas de integração de micro-frontends conhece ?

Marcar tudo o que for aplicável.

- [ ] Single-SPA (Client-Side)
- [ ] Luigi (Client-Side)
- [ ] Zalando Tailor (Server-Side)
- [ ] Podium (Server-Side)
- [ ] OpenComponents (Server-Side)

Outra: [ ] _____

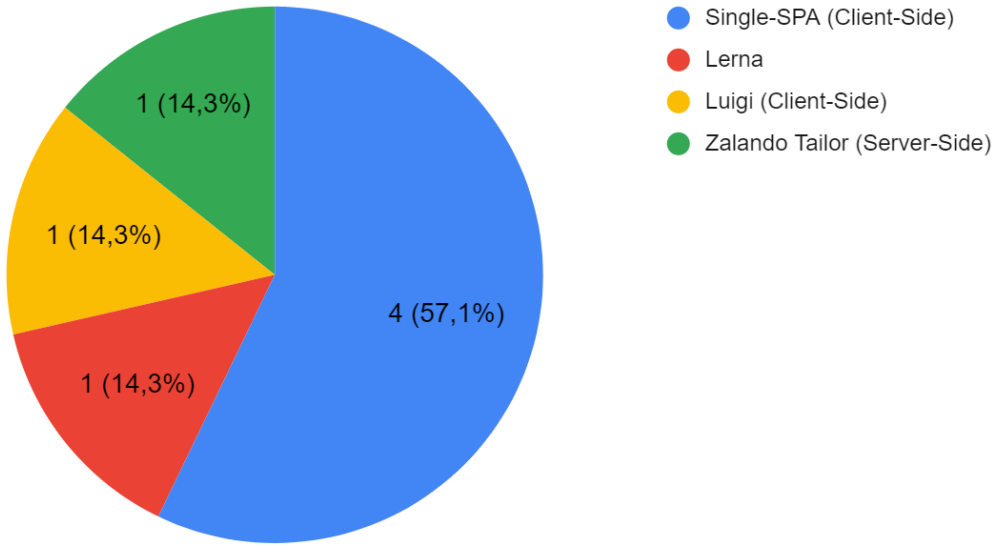


Figura 97 – Gráfico de resultados da questão 34 do questionário

35. Qual o tipo de técnica de integração que considera mais adequada para aplicações bastante interativas e reativas (como uma aplicação e-commerce)? *

Marcar apenas uma oval.

- ◯ Single Page Application Unificada usando App Shell
- ◯ AJAX
- ◯ Custom Elements
- ◯ Uma técnica Server-Side
- ◯ Uma abordagem híbrida em que a integração ocorre do lado do cliente complementada com pré-renderização no servidor
- ◯ Outra: _____

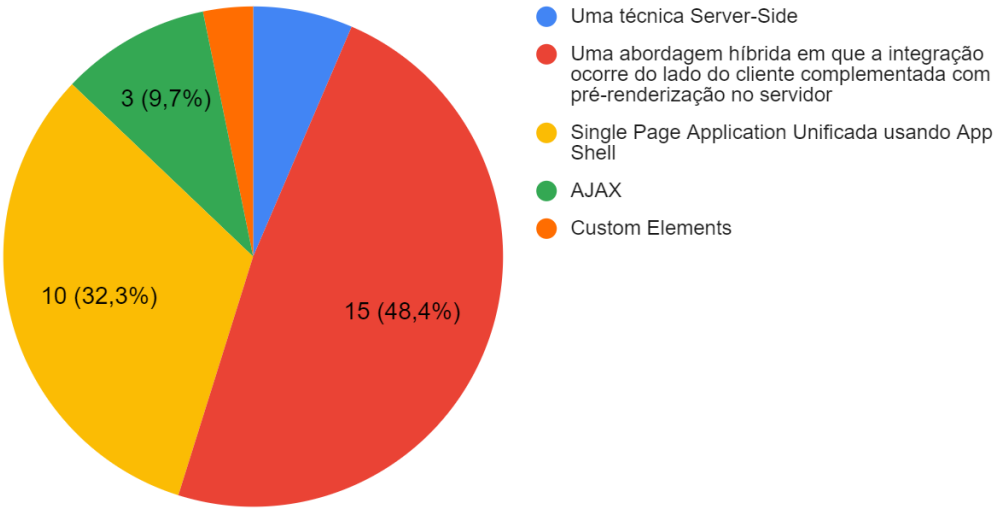


Figura 98 – Gráfico de resultados da questão 35 do questionário

36. Qual o tipo de técnica de integração mais adequada para aplicações em que quer o tempo de carregamento, quer a interatividade são importantes ? *

Marcar apenas uma oval.

- ◯ Uma técnica Client-Side
- ◯ Uma técnica Server-Side
- ◯ Uma abordagem híbrida em que a integração ocorre do lado do cliente complementada com pré-renderização no servidor
- ◯ Outra: _____

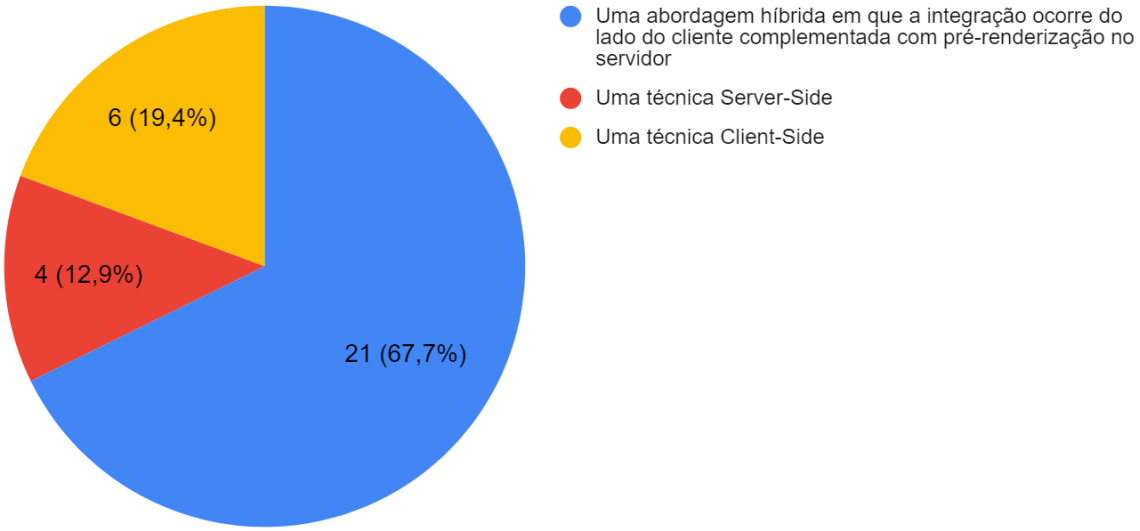


Figura 99 – Gráfico de resultados da questão 36 do questionário

37. Qual o tipo de técnica de integração que considera mais adequada para assegurar a autocontenção dos micro-frontends? *

Marcar apenas uma oval.

- IFrames
- Páginas independentes (com integração via Links)
- Web Components (Custom Elements com Shadow DOM)
- Outra: _____

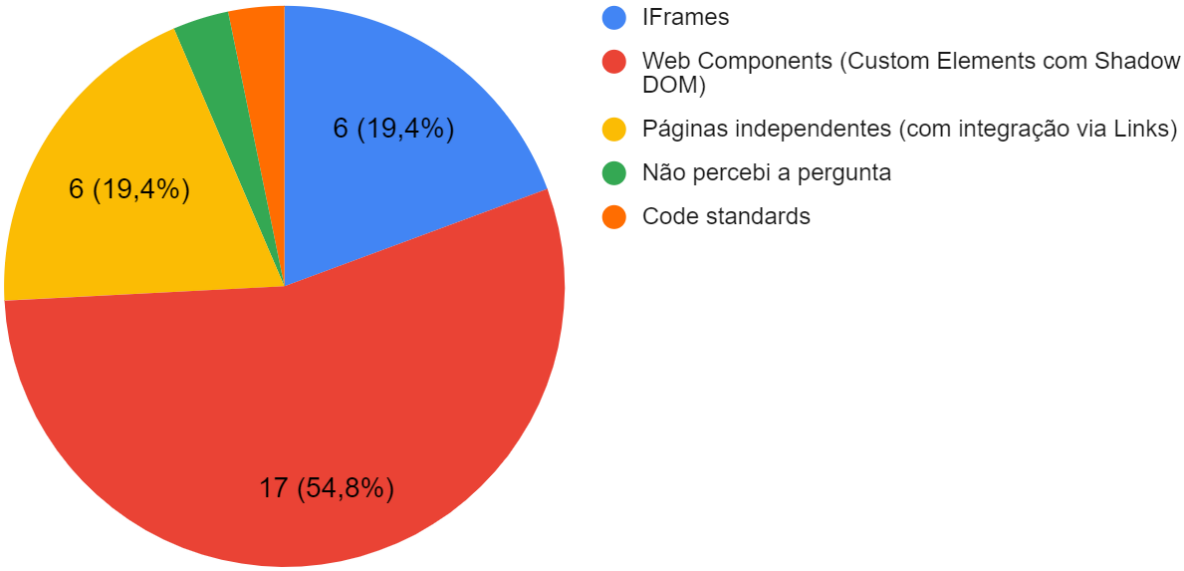


Figura 100 – Gráfico de resultados da questão 37 do questionário

38. É possível utilizando as técnicas de integração existentes efetuar a composição entre os vários micro-frontends e a navegação/transição suave entre si como se de uma Single Page Application vulgar se tratasse. *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Totalmente	◯	◯	◯	◯	◯	Concordo Plenamente

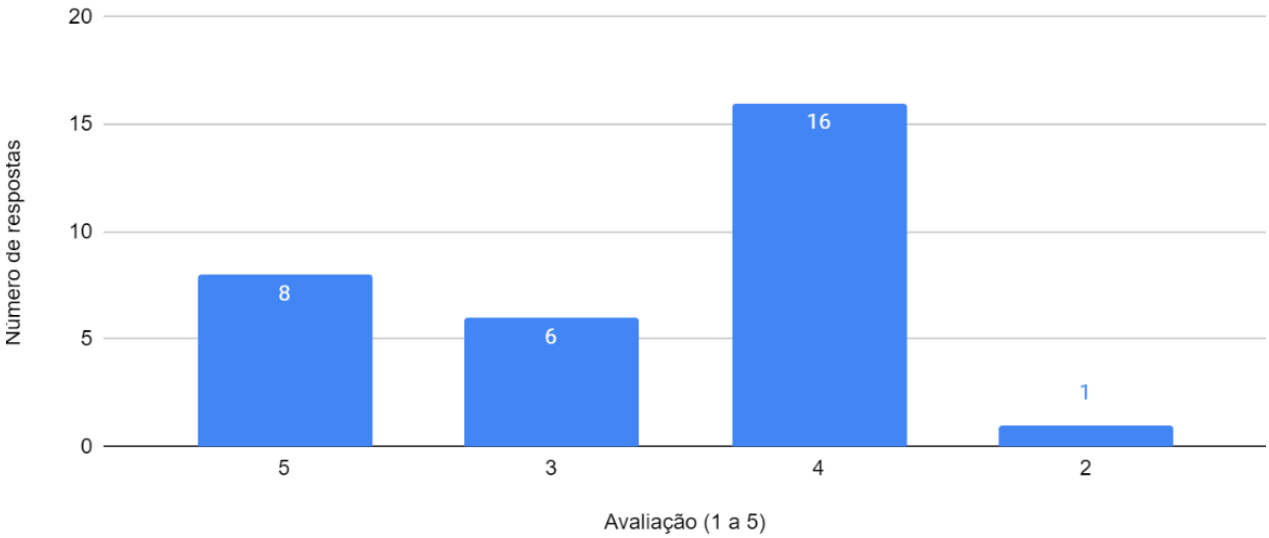


Figura 101 – Gráfico de resultados da questão 38 do questionário

39. Numa escala de 1 a 5, avalie a dificuldade de estabelecer comunicação entre UIs de diferentes micro-frontends *

Marcar apenas uma oval.

	1	2	3	4	5	
Bastante complexo	◯	◯	◯	◯	◯	Bastante simples

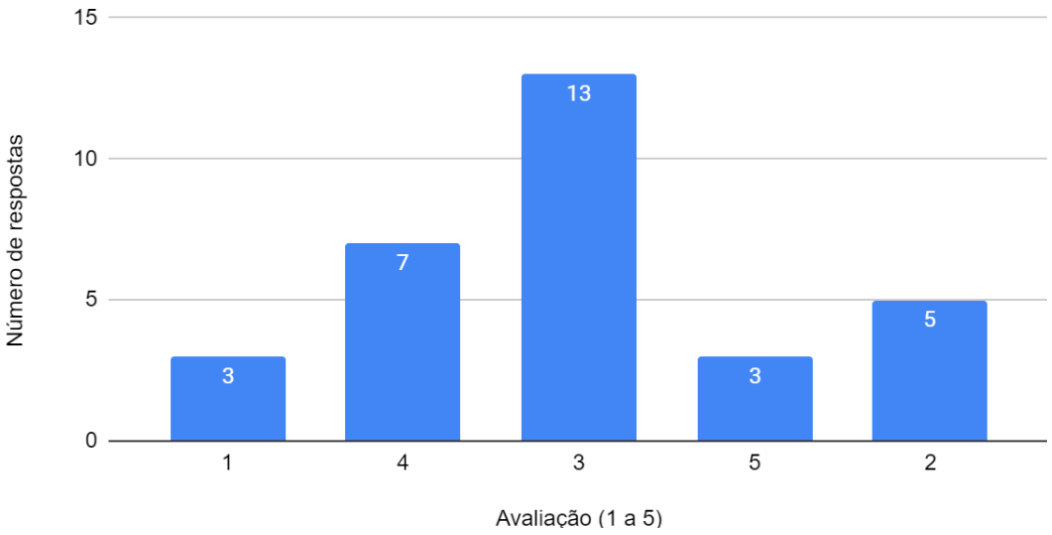


Figura 102 – Gráfico de resultados da questão 39 do questionário

40. Numa escala de 1 a 5, avalie o grau de dificuldade no estabelecimento de comunicação entre a UI e a infraestrutura backend desenvolvida pela mesma equipa *

Marcar apenas uma oval.

	1	2	3	4	5	
Bastante complexo	◯	◯	◯	◯	◯	Bastante simples

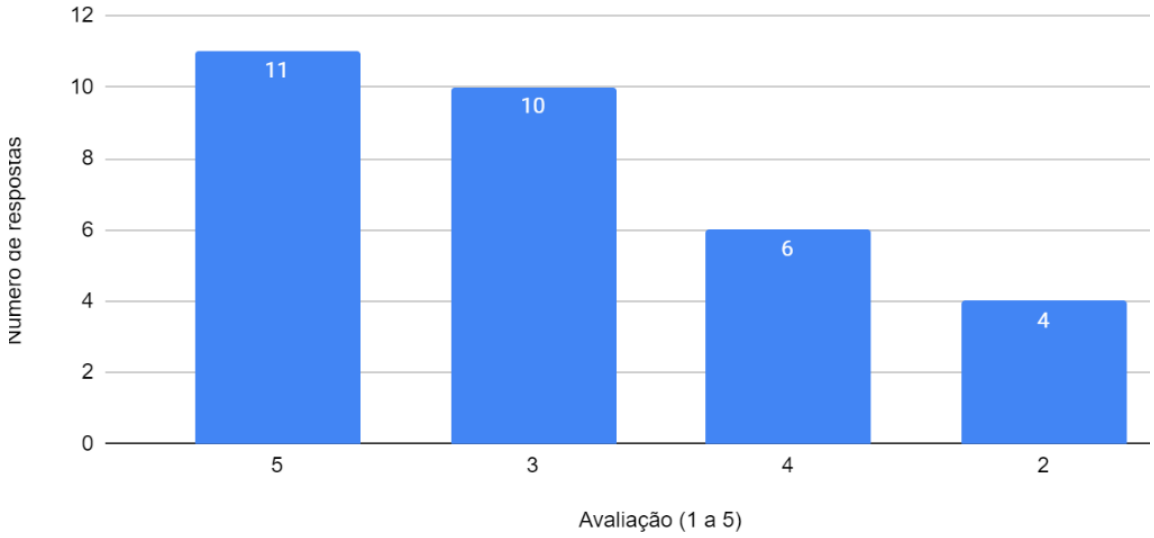


Figura 103 – Gráfico de resultados da questão 40 do questionário

41. Que mecanismos de comunicação entre UIs mantidas por diferentes fragmentos ou páginas conhece ? *

Marcar tudo o que for aplicável.

- ☐ Comunicação via atributos dos elementos
- ☐ Custom Events
- ☐ Window PostMessage API
- Outra: ☐ _____

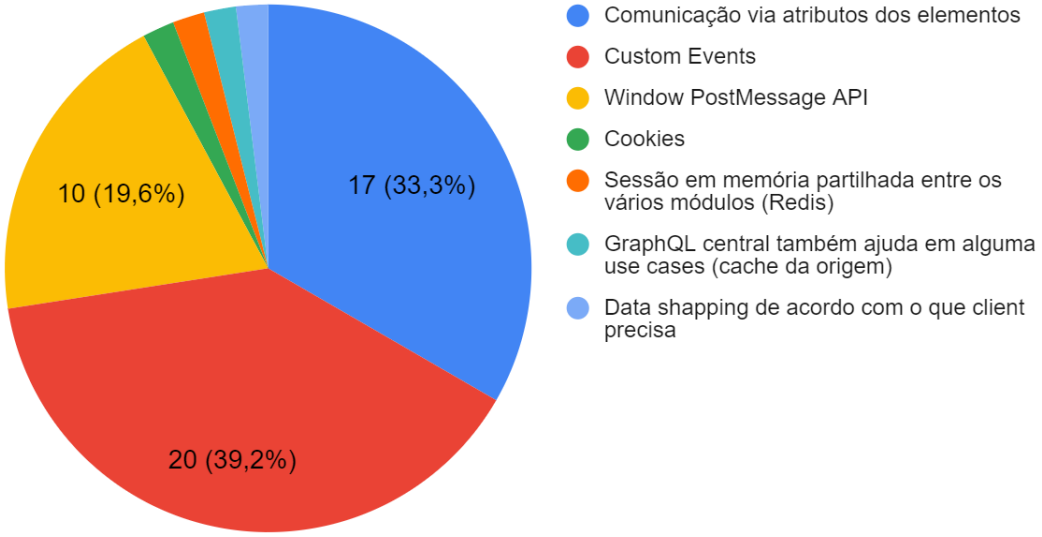


Figura 104 – Gráfico de resultados da questão 41 do questionário

42. Considera que os mecanismos de comunicação elencados na questão anterior podem criar acoplamento e sobrecarregar o código com listeners, emitters e dispatchers ? *

Marcar apenas uma oval.

Sim
Não

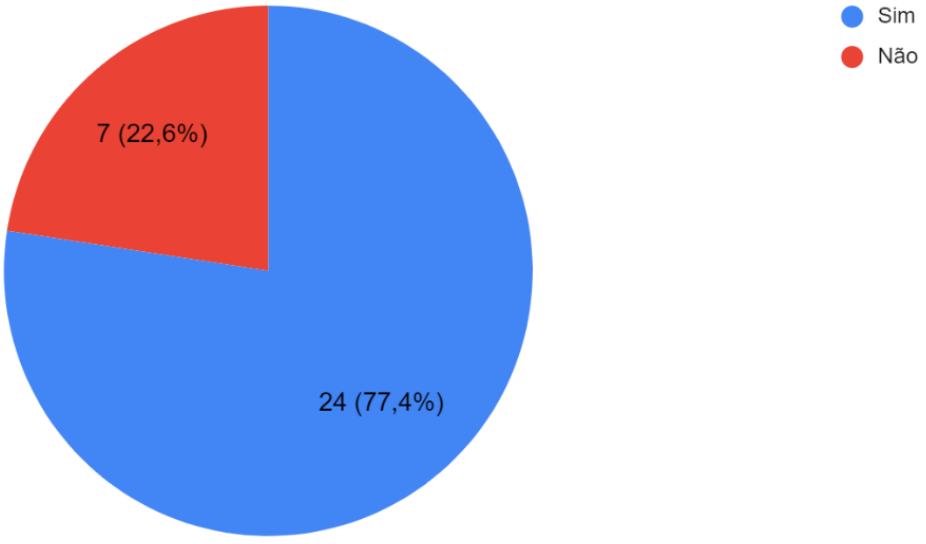


Figura 105 – Gráfico de resultados da questão 42 do questionário

43. Numa aplicação e-commerce, desenvolvida usando micro-frontends, na página de detalhes de um Produto apresentamos uma lista de comentários mantida por outra equipa. Que abordagem usaria ? *

Marcar apenas uma oval.

- ◯ Incluir no HTML da página a lista de comentários, enviando-lhe a informação do produto e delegando-lhe todo o resto
- ◯ Obter a informação e processá-la do lado da página e delegar apenas a renderização no componente

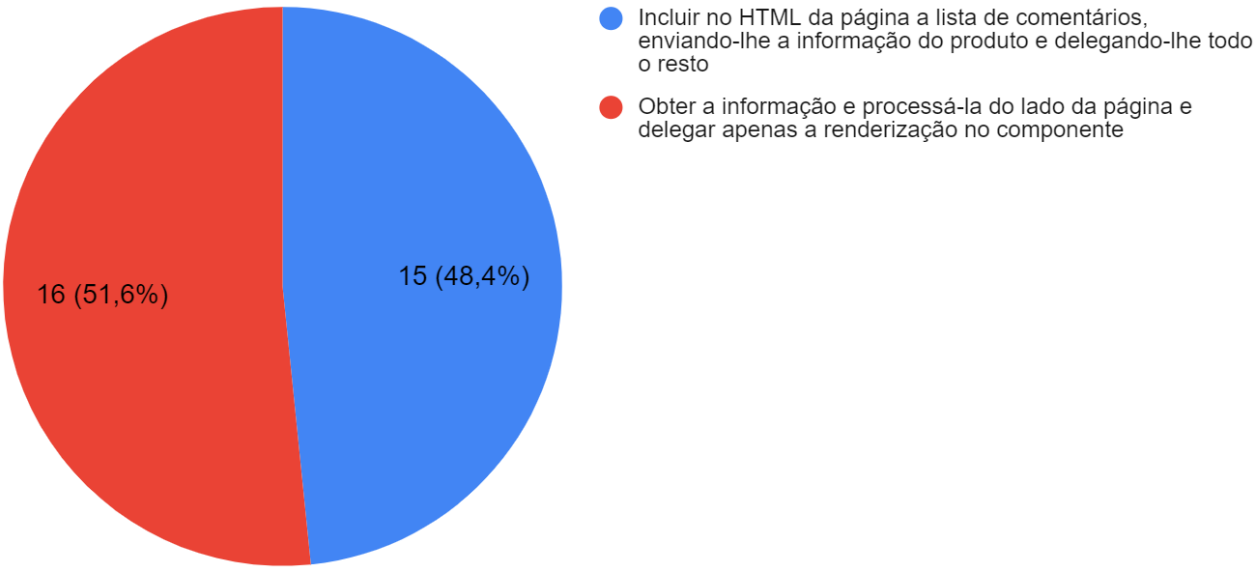


Figura 106 – Gráfico de resultados da questão 43 do questionário

44. Numa escala de 1 a 5, avalie a dificuldade no acesso a informação detida por um serviço por parte de outro serviço *

Marcar apenas uma oval.

	1	2	3	4	5	
Bastante complexo	◯	◯	◯	◯	◯	Bastante simples

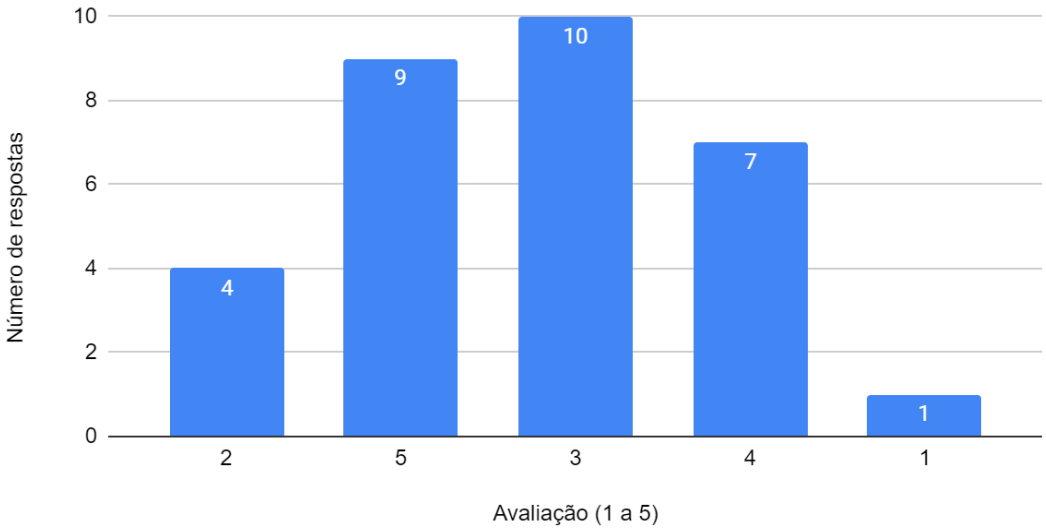


Figura 107 – Gráfico de resultados da questão 44 do questionário

45. Numa aplicação e-commerce, a equipa responsável pelo Carrinho de Compras necessita de aceder à informação dos Produtos no Carrinho , armazenada numa BD externa. Que abordagem usaria ? *

Marcar apenas uma oval.

- ◯ Criação de um serviço middleware que agregue a informação do carrinho com a informação dos produtos
- ◯ Chamada direta do serviço de produto no frontend
- ◯ Chamada direta do serviço de produto no backend
- ◯ Implementação de um mecanismo de replicação de dados (feeds, message bus, streams)

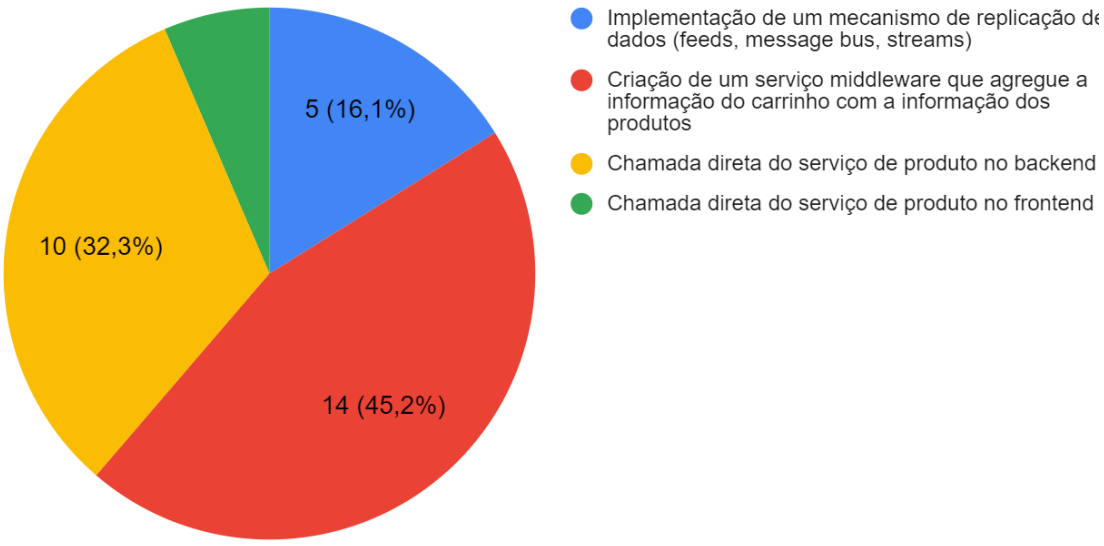


Figura 108 – Gráfico de resultados da questão 45 do questionário

46. Avalie numa escala de 1 a 5, o custo benefício de implementar um "Mecanismo de replicação de dados" para aceder a dados de uma BD externa. *

Relativamente à escala, 1 significa muito alta e 5 muito baixo

Marcar apenas uma oval.

	1	2	3	4	5	
Muito Alta	◯	◯	◯	◯	◯	Muito Baixa

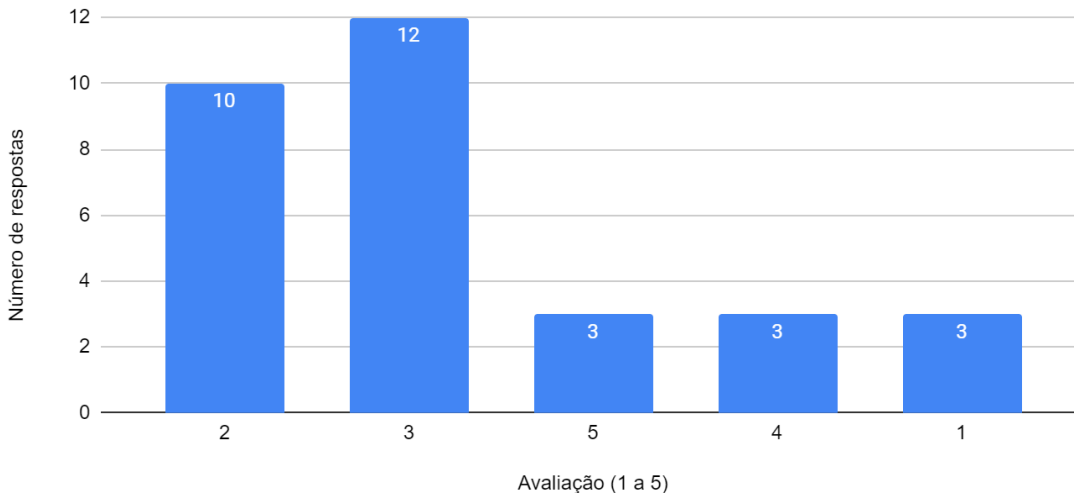


Figura 109 – Gráfico de resultados da questão 46 do questionário

47. Tendo em conta os mecanismos de comunicação entre micro-frontends existentes considera é possível estabelecê-la eficazmente sem que o custo seja incomportável? *

Marcar apenas uma oval.

◯ Sim
◯ Não

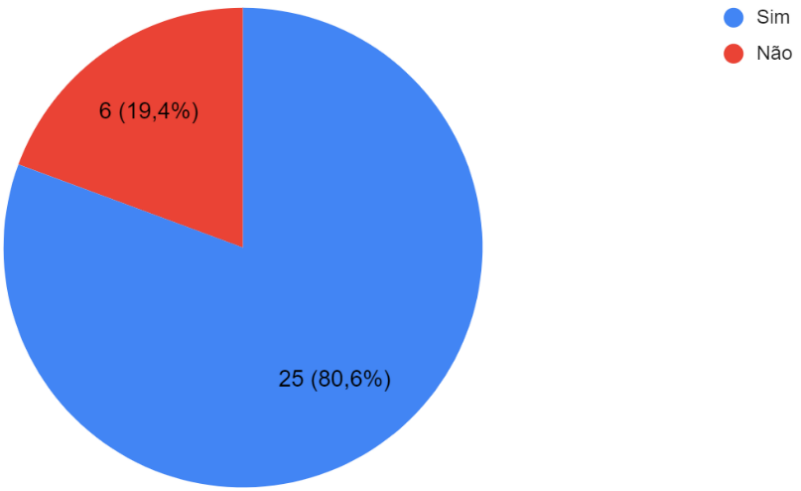


Figura 110 – Gráfico de resultados da questão 47 do questionário

Secção 11 – Micro *Frontends* e Capacidade de Evolução Independente (Acesso restrito aos inquiridos que indicaram conhecer Micro *Frontends*) – 9 questões

48. A liberdade de definição de stack tecnológica por parte das equipas pode refletir-se num aumento significativo do bundle da aplicação visto que apesar de ser possível usar diferentes tecnologias isso não significa que se deva fazê-lo sempre. *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Totalmente	◯	◯	◯	◯	◯	Concordo Plenamente

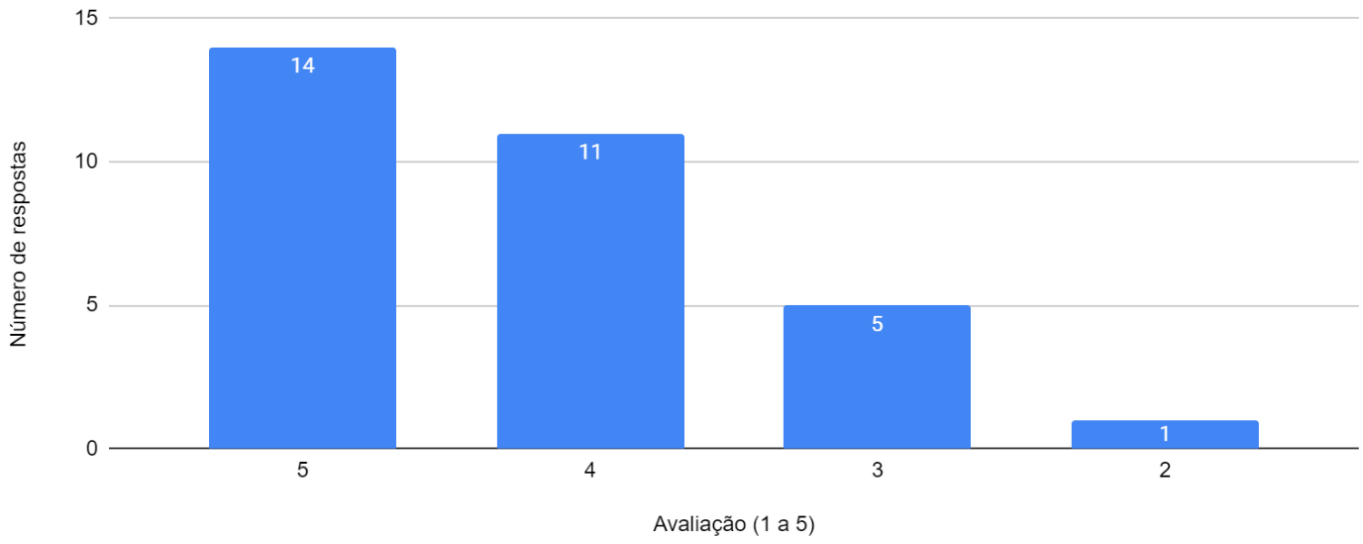


Figura 111 – Gráfico de resultados da questão 48 do questionário

49. É menos arriscado promover a migração tecnológica de uma aplicação funcionalidade a funcionalidade, mantendo-se o sistema antigo e o sistema novo em funcionamento, comparativamente a continuar a utilizar apenas o sistema antigo, enquanto a migração não estiver concluída *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Totalmente	◯	◯	◯	◯	◯	Concordo Plenamente

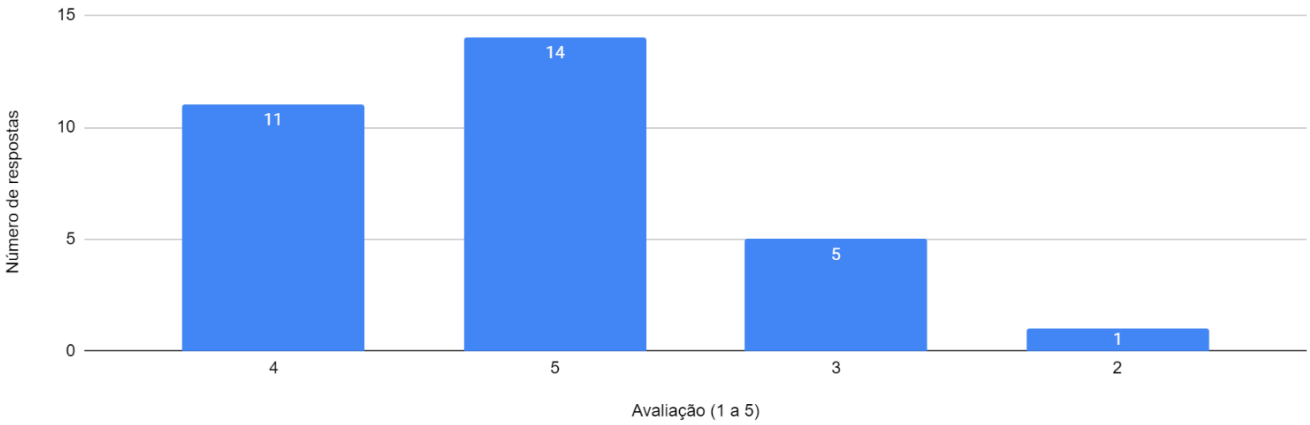


Figura 112 – Gráfico de resultados da questão 49 do questionário

50. Adotando uma estratégia de Continuous Delivery e Continuous Deployment cada equipa pode efetuar o deployment de forma independente, isto é, sem necessidade de coordenação com as outras equipas e com a periodicidade pretendida. *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Totalmente	◯	◯	◯	◯	◯	Concordo Plenamente

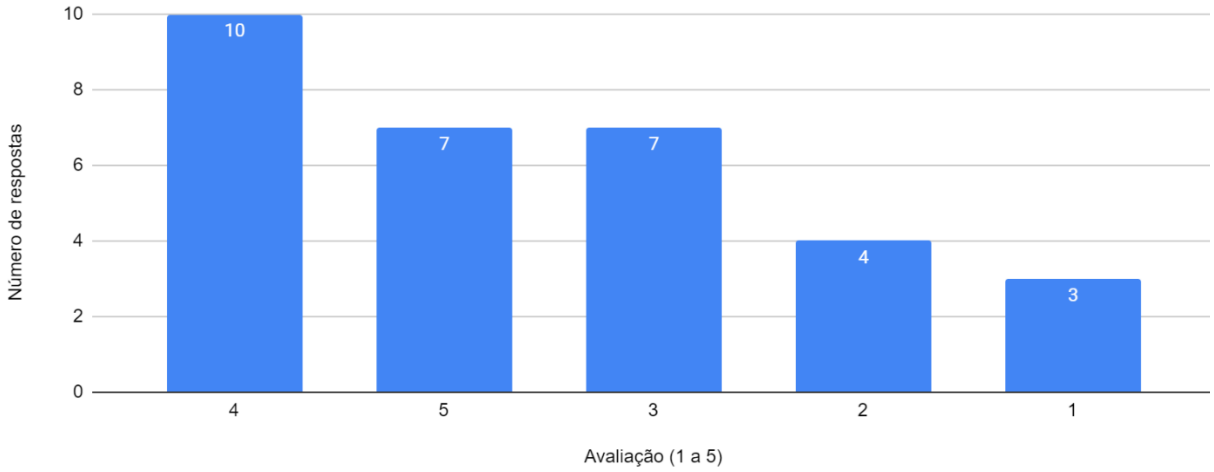


Figura 113 – Gráfico de resultados da questão 50 do questionário

51. Considera que o facto dos micro-frontends / micro-serviços possuirem uma codebase limitada, serem auto-contidos e deployable de forma independente permite uma escalabilidade mais eficiente ? *

Marcar apenas uma oval.

Sim
Não

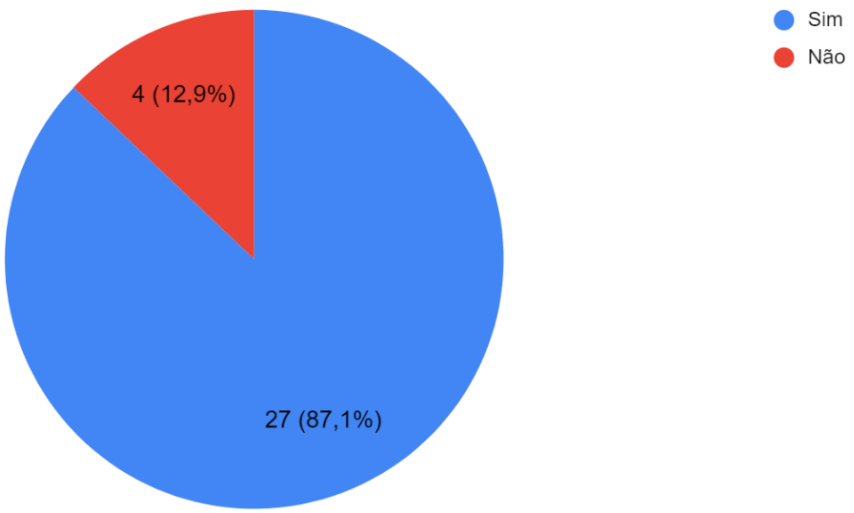


Figura 114 – Gráfico de resultados da questão 51 do questionário

52. Considera que o facto dos micro-frontends / micro-serviços serem altamente modulares favorece a testabilidade e o isolamento de falhas ? *

Marcar apenas uma oval.

Sim
Não

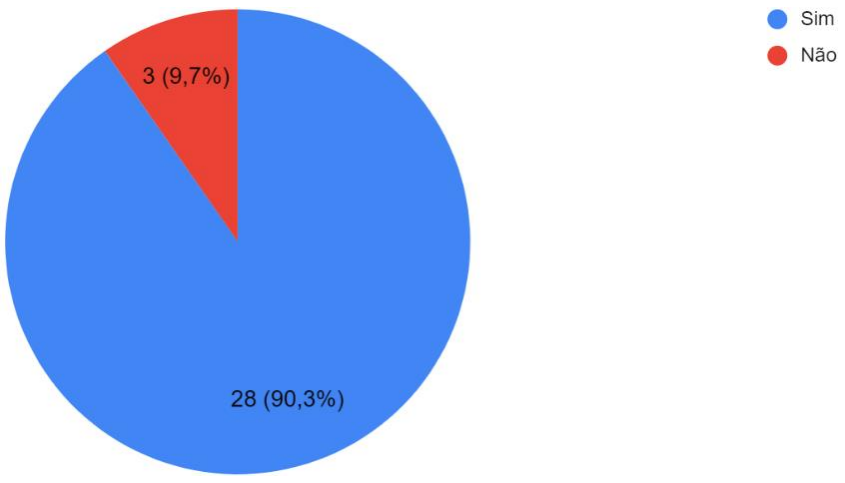


Figura 115 – Gráfico de resultados da questão 52 do questionário

53. Considera que usando micro-frontends/micro-serviços é possível assegurar um código mais limpo, facilmente extensível e manipulável e com menor acoplamento ? *

Marcar apenas uma oval.

- Sim
- Não

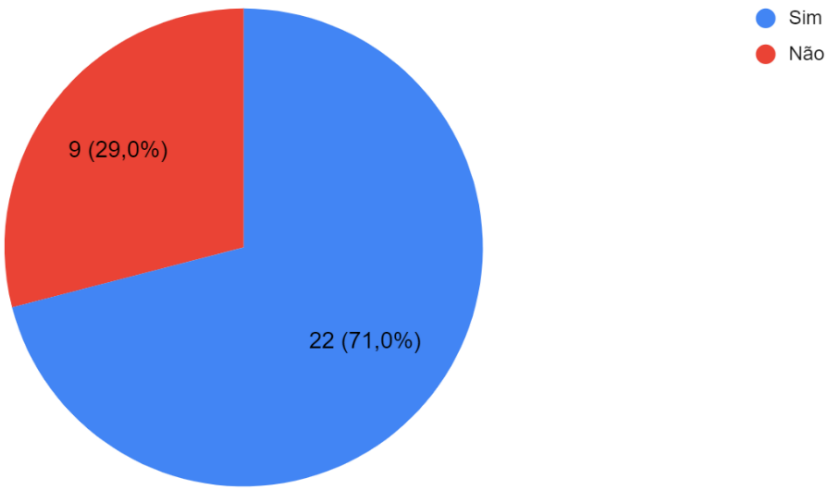


Figura 116 – Gráfico de resultados da questão 53 do questionário

54. Exigindo os micro-frontends uma infraestrutura que possibilite a Continuous Integration e a Continuous Delivery, considera que as vantagens que advém desta abordagem superam a complexidade que exige para ser concretizada ? *

Marcar apenas uma oval.

- Sim
- Não

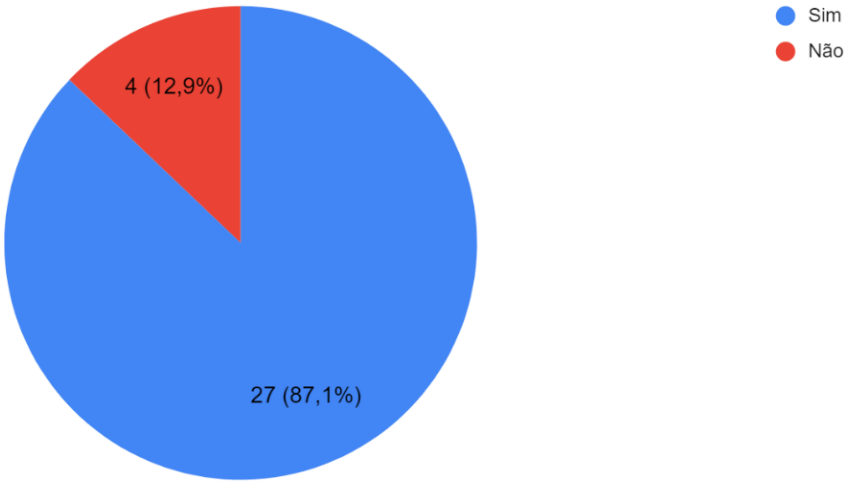


Figura 117 – Gráfico de resultados da questão 54 do questionário

55. Se respondeu "Não" qual é na sua opinião o principal problema ?

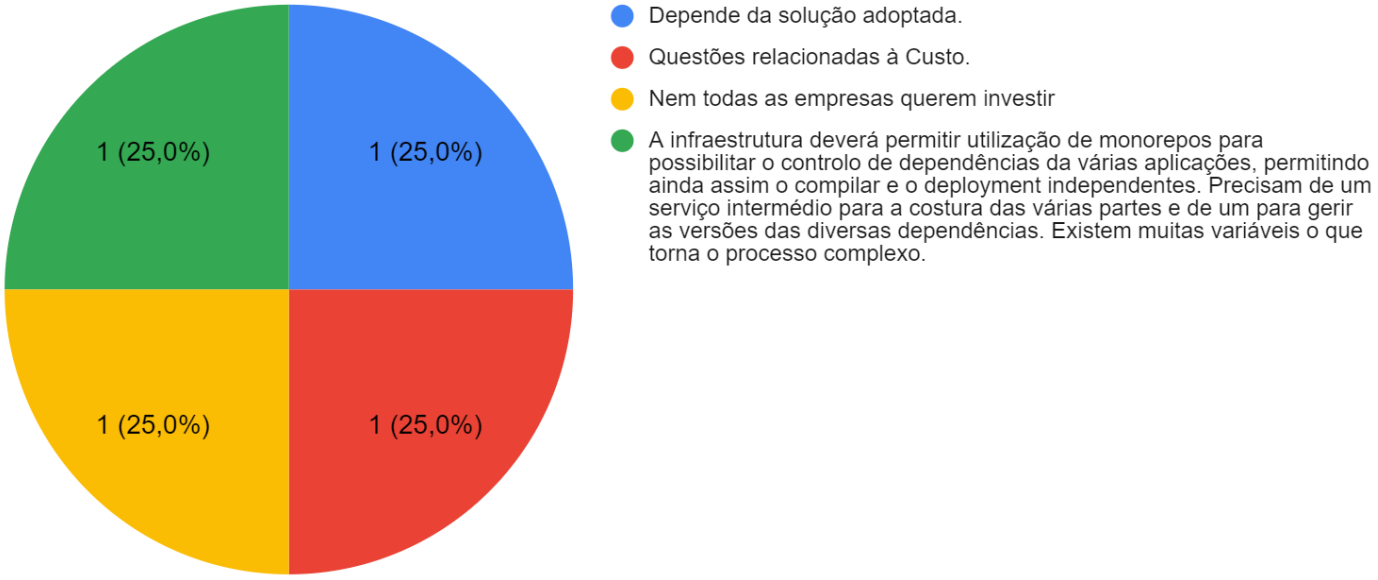


Figura 118 – Gráfico de resultados da questão 55 do questionário

56. Considera que usando Micro-Frontends/Micro-Serviços é possível obter uma maior resiliência e tolerância a falhas e/ou indisponibilidade de partes de uma aplicação? *

Marcar apenas uma oval.

Sim
Não

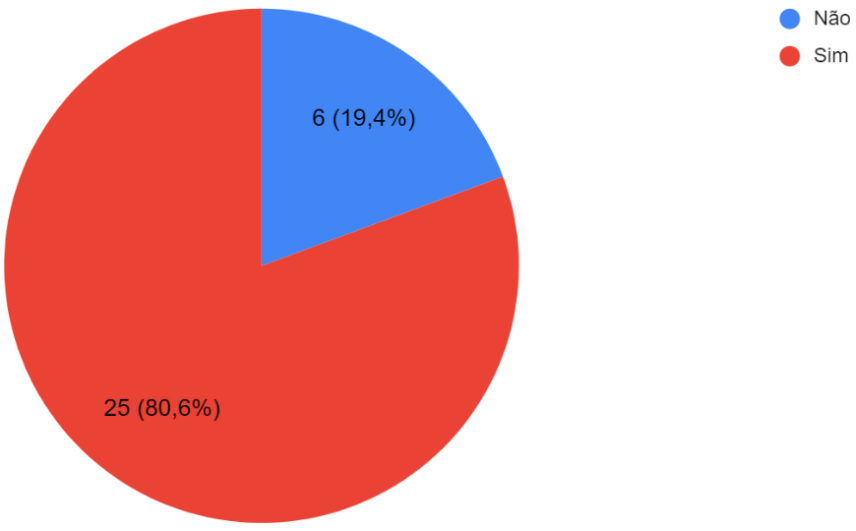


Figura 119 – Gráfico de resultados da questão 56 do questionário

Secção 12 – Vantagens e Desvantagens da Adoção de Micro *Frontends* (Acesso Restrito à Equipa de Produto, Arquitetos, *Tech Leaders, Project Managers, Team Managers*, Estudantes de Eng. Informática e Autodidatas) – 2 questões

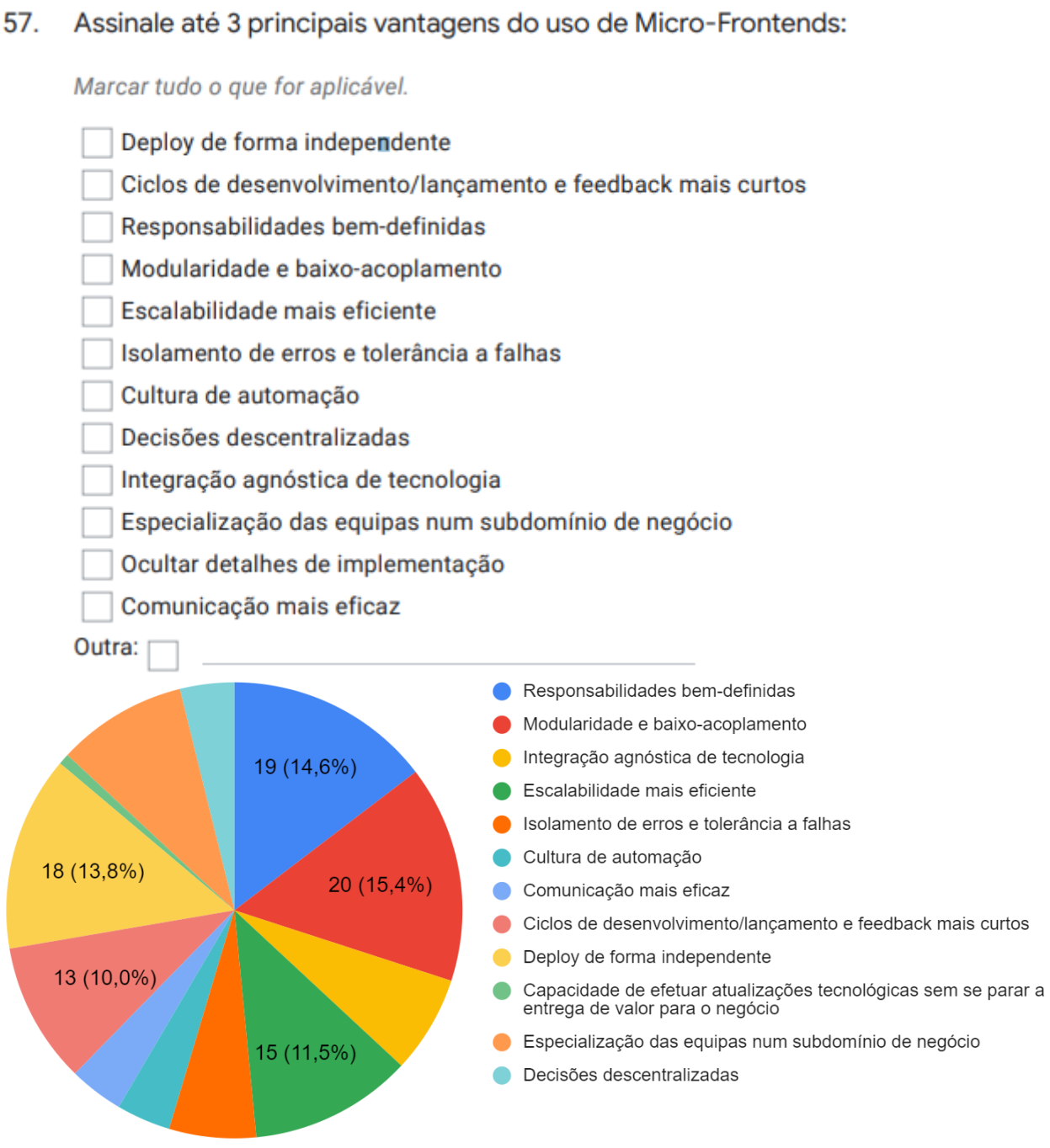


Figura 120 – Gráfico de resultados da questão 57 do questionário

58. Assinale até 3 principais desvantagens do uso de Micro-Frontends:

A arquitetura de micro-frontends pressupõe que cada equipa detenha a sua própria base de dados. Todavia, é frequente uma equipa necessitar de dados detidos por outras equipas, algo possível através da replicação e sincronização periódica dos mesmos entre serviços. Assim sendo, se algum serviço ficar indisponivel, tal não interfere com o funcionamento dos restantes. Além disso, a sincronização levando tempo e introduzindo latência pode causar uma consistência momentânea.

Marcar tudo o que for aplicável.

- [ ] Codebase mais extensa
- [ ] Redundância de código
- [ ] Inconsistência momentânea de dados (ver nota)
- [ ] Risco do Bundle ficar demasiado extenso (caso se verifique excessiva heterogeneidade e diversidade de tecnologias)
- [ ] Maior complexidade técnica
- [ ] Dificuldade em gerir cross-cutting concerns, entre as quais a autenticação
- [ ] Risco de conflitos de CSS
- [ ] Silos de conhecimento
- [ ] Inexistência de guidelines pode causar inconsistência
- [ ] Uso ineficiente de recursos humanos
- [ ] Gestão de recursos e dependências requeridos por vários micro-frontends

Outra: ☐ _____

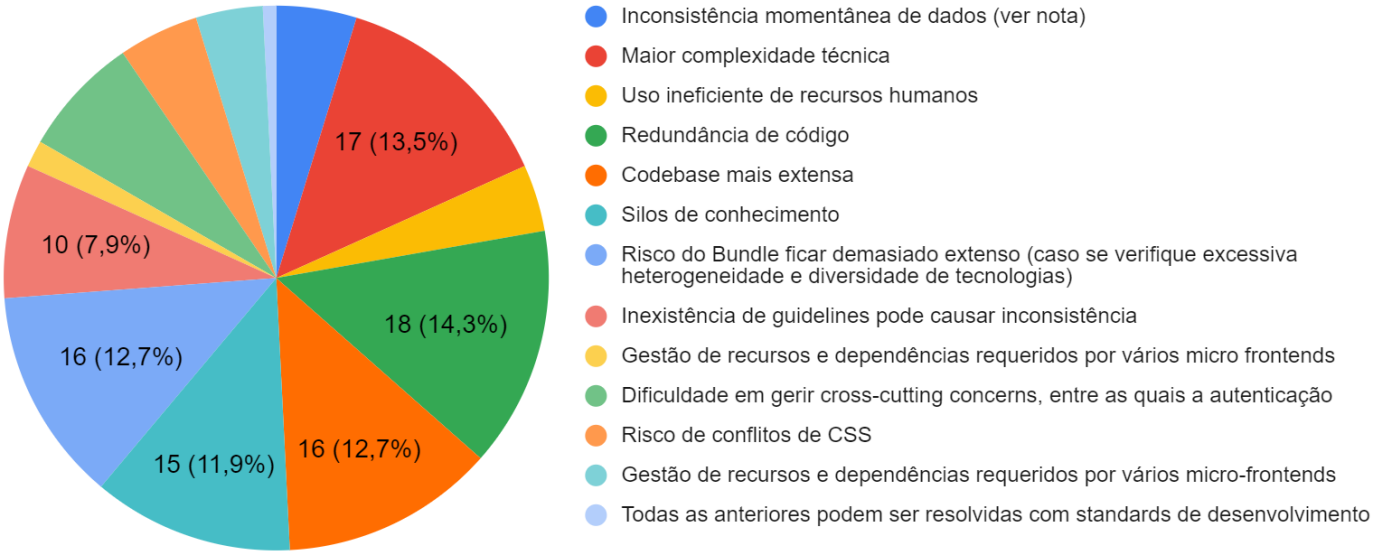


Figura 121 – Gráfico de resultados da questão 58 do questionário

Secção 13 – Agradecimentos/Final do Questionário (Acesso geral)

Final do Questionário

Queria agradecer o tempo dispendido na realização desde questionário e pelo forte contributo que deu na formulação de conclusões relativas à viabilidade da adoção de micro-frontends no processo de desenvolvimento web e à identificação das principais vantagens e desvantagens.