Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Um guia para *micro frontends*: estudo aprofundado e aplicação de caso de uso no *ecommerce*

Ricardo Hideki Hangai Kojo

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman
Cossupervisor: Me. Luiz Fernando O. Corte Real
Cossupervisor: Me. Renato Cordeiro Ferreira

São Paulo
24 de Dezembro de 2021

Agradecimentos

Darkness cannot drive out darkness; only light can do that. Hate cannot drive out hate; only love can do that.
— Martin Luther King Jr.

Em momentos tão difíceis, o que não faltam são agradecimentos. De forma resumida, agradeço a todos por tudo, e a tudo por todos.

Em mais detalhes, começo agradecendo aos meus pais por me oferecer tudo que precisei para chegar à universidade.

Agradeço aos meus amigos por me fazer rir, sorrir e me manter são neste ano.

Agradeço aos meus orientadores pela oportunidade, pelo acompanhamento próximo e por facilitar a realização de um trabalho tão extenso.

Agradeço ao Elo7 por permitir a realização deste trabalho, o que tornou o equilíbrio entre os âmbitos acadêmico e profissional possível.

Por fim, agradeço a todos que trabalharam e se sacrificaram pelo bem maior.

Resumo

Ricardo Hideki Hangai Kojo. **Um guia para *micro frontends*: estudo aprofundado e aplicação de caso de uso no *ecommerce***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

A arquitetura de *micro frontends* surgiu por volta de 2016, com o objetivo de aplicar as ideias de microsserviços para a camada de apresentação das aplicações. Nesse estilo arquitetural, o *frontend* é dividido em pequenas partes, que podem ir de um simples botão até uma página completa. Assim, espera-se melhorar aspectos como escalabilidade, resiliência e independência dos times, em troca de aumento da complexidade do sistema e sua infraestrutura. Este Trabalho de Conclusão de Curso possui como objetivo entender quando vale a pena adotar *micro frontends*, em particular no contexto da indústria. Para isso, foram levantados três objetivos secundários: entender o estado da arte da arquitetura, com o estudo das literaturas acadêmica e cinzenta; implementar esse estilo arquitetural no Elo7, empresa marketplace de produtos artesanais, que já adota microsserviços; documentar e avaliar a implementação, através de um questionário semiaberto com os desenvolvedores. A literatura acadêmica sobre a arquitetura ainda é escassa. Há falta de flexibilidade sobre a definição de um *micro frontend*, que é repetidamente atrelado à quebra de uma página em componentes. Também há carência de documentos sobre abordagens como *micro frontends* verticais e *micro frontends* sem interfaces. Um caminho para a melhoria seria aproximar a definição da arquitetura com a de seu predecessor, microsserviços. No Elo7, a necessidade de uma mudança arquitetural surgiu do grande acoplamento entre o sistema principal, um monolito Java, e outro sistema dedicado ao *frontend*. Também havia problemas com tecnologias descontinuadas e má experiência de desenvolvimento. Para solucionar esses problemas, foram adotadas a arquitetura de *micro frontends*, juntamente com os padrões *API Gateway* e *Backend For Frontend* (BFF), e tecnologias modernas, como Svelte e Fastify. O uso de *micro frontends*, apesar de bem sucedido, não era mandatório para atender às necessidades da empresa. De acordo com as análises das respostas ao questionário semiaberto, concluiu-se que outras alternativas, como um *frontend* monolítico, também atingiriam resultados equiparáveis. O que tornou a escolha por *micro frontends* a mais conveniente, no contexto da empresa, foi o fato de estar passando pelo processo de estrangulamento do monolito e adoção de microsserviços. Isso facilitou a implementação da arquitetura, por meio da reutilização da infraestrutura e do conhecimento compartilhado entre times.

Palavras-chave: Arquitetura de software. Estilos arquiteturais de software. Microsserviços. Micro frontends.

Abstract

Ricardo Hideki Hangai Kojo. **A micro frontends guide: in-depth study and application of an ecommerce use case**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

The micro frontends architecture was created around 2016, aiming to apply the core ideas of microservices to the presentation layer of the applications. In this architectural style, the frontend is divided into small pieces, which can range from a simple button to a whole page. Thus, it is expected to improve scalability, resilience, and team independence, in exchange for a complexity and infrastructure increase. The objective of this Capstone Project is to comprehend when it is worth it to adopt micro frontends, particularly in the context of the industry. To achieve this, other three secondary objectives were addressed: understanding the state of the art of micro frontends, through the study of academic and grey literature; implementing this architectural style at Elo7, a marketplace for handmade products, that already adopts microservices; documenting and assessing the implementation, supported by a mixed questionnaire to the developers. The academic literature is still sparse. It lacks flexibility about the definition of a micro frontend, which is repeatedly tied to the separation of a page into components. There is also a shortage of documentation about vertical micro frontends and interfaceless micro frontends. A path to improve is to approximate the architecture's definition to its predecessor's, microservices. At Elo7, the necessity of an architectural change came through the coupling between their main system, a Java monolith, and a dedicated frontend system. Furthermore, there were deprecated technologies and bad developer experience problems. To solve those, micro frontends architecture, along with the API Gateway and Backend For Frontend patterns, and modern technologies, such as Svelte and Fastify, were adopted. Although successful, the use of micro frontends was not mandatory to meet the company's needs. According to the mixed questionnaire answers analysis, other alternatives, namely a monolithic frontend, would also be able to reach comparable results. What made the choice to adopt micro frontends the most convenient, in the company's context, was the monolith strangulation and microservices adoption. It facilitated the implementation through infrastructure reuse and knowledge sharing between teams.

Keywords: Software architecture. Software architecture styles. Microservices. Micro frontends.

Lista de Abreviaturas

API	*Application Programming Interface*
BFF	*Backend For Frontend*
CAO	*Chief Architecture Officer*
CDN	*Content Delivery Network*
CSS	*Cascading StyleSheets*
CTO	*Chief Technology Officer*
gRPC	*gRPC Remote Procedure Call*
HTML	*HyperText Marking Language*
HTTP	*HyperText Transfer Protocol*
IME	Instituto de Matemática e Estatística
JSP	*JavaServer Pages*
TCC	Trabalho de Conclusão de Curso
URL	*Uniform Resource Locator*
USP	Universidade de São Paulo
WAR	*Web application ARchive*

Lista de Figuras

Lista de Tabelas

Sumário

Apêndices

Capítulo 1

Introdução

A arquitetura de microsserviços — surgida por volta de 2012 (Lewis e Fowler, 2014) — tem crescido em popularidade, tanto no campo acadêmico quanto na indústria, sendo adotada por empresas como Amazon, Netflix e Uber (Richardson, 2020). Essa arquitetura consiste em quebrar a aplicação em serviços pequenos, independentes e que trabalham em conjunto. O baixo acoplamento entre eles traz benefícios como manutenibilidade, escalabilidade e resiliência, características essenciais para grandes sistemas, que precisam ser confiáveis — sempre disponíveis e com o mínimo de falhas. No entanto, essa abordagem engloba apenas o *backend* da aplicação, deixando o *frontend* de lado (Peltonen *et al.*, 2021).

Assim, surge a arquitetura de ***micro frontends***, com o objetivo de aplicar os conceitos de microsserviços para a camada de apresentação das aplicações. Nessa abordagem, o sistema *frontend* é dividido em pequenas partes, que podem ser desde um simples botão até uma página completa. Essas partes são unidas em *client-side*, como o navegador do usuário, *edge-side*, como uma CDN ou *Content Delivery Network*, ou *server-side*, como um servidor na nuvem, produzindo um resultado único e coeso.

Há uma grande intersecção entre os benefícios e desafios de microsserviços e *micro frontends*. Logo, é racional cogitar adotar as duas arquiteturas, aproveitando o conhecimento e infraestrutura já disponíveis. No entanto, o conceito de *micro frontends* é recente — surgido em 2016 (Geers, 2017) — e existem poucos trabalhos acadêmicos relacionados. Ao contrário de microsserviços, que possui tecnologias e padrões de projeto específicos para essa arquitetura, há poucas ferramentas para *micro frontends*. Sendo assim, existem diversas oportunidades a serem exploradas.

Logo, o objetivo deste trabalho é identificar quando vale a pena adotar a arquitetura de *micro frontends*, em particular no contexto da indústria. Para isso, foram desenvolvidas atividades nos âmbitos teórico e prático.

A primeira parte foi entender o estado da arte dessa arquitetura através do estudo da literatura acadêmica, como artigos e livros, e cinzenta, como postagens em blogs e gravações de palestras. Por meio de uma análise crítica, foi possível identificar as fontes de conhecimento mais completas, os principais autores do tema e as deficiências nos materiais atuais.

Além do estudo, houve a implementação dessa arquitetura no **Elo7**, plataforma *marketplace* de produtos artesanais, que já utiliza microsserviços. Também fizeram parte desse objetivo documentar e avaliar a implementação: entender o motivo da mudança arquitetural, traçar o histórico de desenvolvimento, incluindo os novos sistemas e explicações para as decisões tomadas. Por fim, analisar os impactos de *micro frontends* à empresa, através de um questionário com os colaboradores envolvidos.

Esta monografia está dividida em seis capítulos, sendo o primeiro uma breve introdução ao tema e objetivos principais deste trabalho. No Capítulo 2 são apresentados conceitos essenciais para o entendimento deste Trabalho de Conclusão de Curso. O Capítulo 3 apresenta a empresa Elo7, os principais sistemas *frontend* e quais os motivos para a mudança arquitetural. No Capítulo 4, há o histórico da adoção de *micro frontends* na empresa, com detalhes dos sistemas criados, a situação atual da arquitetura e expectativas para o futuro. Para verificar as expectativas e observações dos desenvolvedores sobre a nova arquitetura foi feito um questionário semiaberto, descrito no Capítulo 5, que também inclui análises quantitativas e qualitativas sobre as respostas. Por fim, o Capítulo 6 compreende as conclusões da monografia acerca da literatura e do estudo conduzido em parceria com o Elo7.

Capítulo 2

Arquitetura de Software

O foco desta monografia é o estilo arquitetural de *micro frontends*, incluindo teoria e prática. Desta forma, o objetivo deste capítulo é apresentar os conceitos essenciais para a compreensão integral deste trabalho, assumindo conhecimentos básicos de computação e desenvolvimento de software.

2.1 Definição

A arquitetura de software pode ser definida como "o conjunto de estruturas necessárias para compreender um sistema, que inclui os elementos que o compõem, as relações entre eles, e suas propriedades." (Bass *et al.*, 2013). Assim, fazem parte de uma arquitetura os fundamentos, propriedades, regras e restrições que guiam o projeto.

O objetivo de defini-la é facilitar aspectos como desenvolvimento, implantação, operação e manutenção (Martin, 2019), além de ajudar na análise dos chamados atributos qualitativos: escalabilidade, resiliência, segurança, manutenibilidade, entre outros. Logo, decisões arquiteturais são essenciais para a evolução de um sistema. O crescimento desordenado e extrapolação dos limites da arquitetura levam o sistema a se tornar um *big ball of mud* (ou grande bola de lodo), termo criado por Foote e Yoder (1997).

É importante notar que não há um consenso na definição de arquitetura de software. A definição apresentada nesta seção é a que foi julgada mais clara e concreta, enquanto há diversas outras excessivamente abstratas. Por esse motivo, alguns desenvolvedores subestimam a importância da arquitetura para um sistema e sua longevidade.

Nas próximas seções, serão apresentados dois conceitos relacionados à arquitetura de software: estilos (2.2) e padrões arquiteturais (2.3). Esses conceitos são relevantes pois *micro frontends* é um estilo arquitetural, e um estilo possui um conjunto de padrões que podem ser implementados para resolver um problema num dado contexto.

2.2 Estilos Arquiteturais

Um estilo arquitetural determina um conjunto limitado de elementos e relações, dos quais é possível definir uma visão geral do projeto (Richardson, 2018a). Um sistema que respeita tais determinações pode ser considerado um membro daquele estilo arquitetural. Por fim, é importante notar que uma mesma aplicação pode adotar múltiplos estilos.

Ao escolher adotar um certo estilo arquitetural, é essencial se atentar às motivações para a adoção, aos benefícios trazidos e desafios relacionados. Qualquer decisão arquitetural traz grandes impactos, e como sabemos, não há balas de prata no desenvolvimento de software (Brooks, 1995): não existe um estilo arquitetural que funciona para todos os casos, ou que traga apenas vantagens.

Nas subseções serão apresentados três estilos arquiteturais: monolito (2.2.1), estilo muito utilizado na indústria e comumente mencionado no contexto de mudanças arquiteturais, microsserviços (2.2.2), arquitetura que deu origem aos *micro frontends*, e *micro frontends* (2.2.3), foco principal deste trabalho.

2.2.1 Monolito

Num monolito, a aplicação é estruturada numa única camada que contém toda a lógica necessária para o funcionamento integral do sistema, e que resulta num único arquivo executável. Num projeto Java[1], por exemplo, o resultado é um único arquivo WAR. As figuras 2.1 e 2.2 exemplificam a estrutura de uma aplicação monolítica.

Essa arquitetura se caracteriza por ser simples e direta, o que a torna ideal para novos projetos. No entanto, com o crescimento da equipe e da base de código, podem surgir problemas técnicos e organizacionais.

Dificuldades de manutenção, aumento da quantidade de *bugs*, desenvolvimento de novas funcionalidades lento, problemas de escalabilidade. Estes são alguns exemplos de problemas que podem surgir, e que baixam o moral dos desenvolvedores. Neste contexto, a arquitetura de microsserviços é uma alternativa (Richardson, 2018a).

Antes de introduzir a arquitetura de microsserviços, serão apresentadas as motivações, benefícios e desafios no uso de monolitos, de acordo com Richardson (2019).

Dentre as principais **motivações** para a adoção da arquitetura monolítica, estão:

- **Velocidade de desenvolvimento**: a velocidade de entrega de novas funcionalidade é essencial, principalmente para novos projetos e *startups*;
- **Simplicidade arquitetural**: o projeto deve ser simples de se entender e fácil de ser alterado, de forma que novos membros sejam capazes de se integrar e contribuir rapidamente.

Vale notar que os pontos acima são proeminentes no início do desenvolvimento, e vão diminuindo conforme o projeto aumenta.

[1] https://www.java.com/pt-BR/

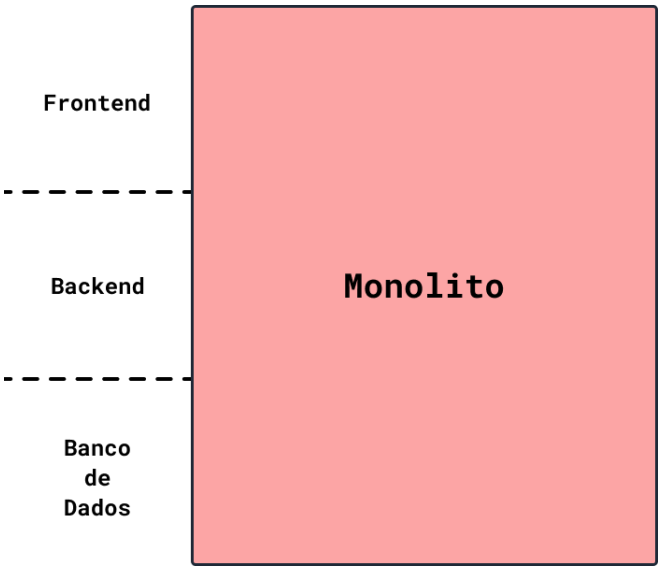

Figura 2.1: *Esquema representativo de um monolito, que engloba frontend, backend e comunicação com banco de dados.*

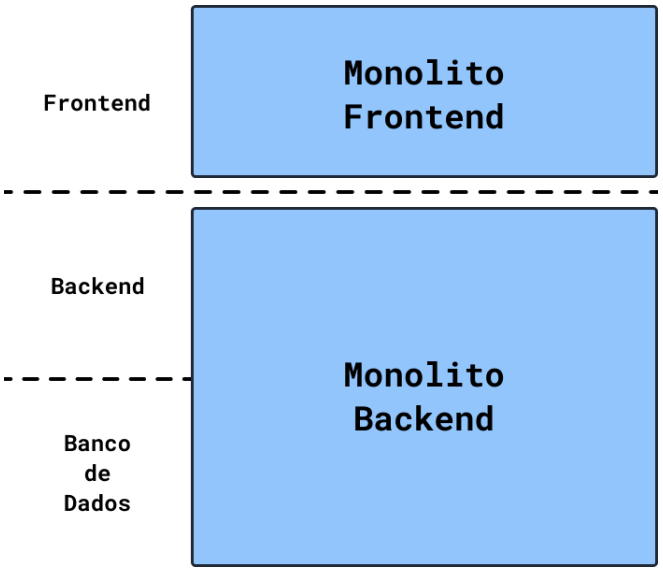

Figura 2.2: *Outra abordagem, na qual são criados dois monolitos: um dedicado ao frontend e outro ao backend (incluindo banco de dados).*

As principais **vantagens** deste estilo são:

- **Desenvolvimento simples**: é mais simples subir uma única aplicação do que múltiplas e não há necessidade de resolver problemas de comunicação entre sistemas. Além disso, há diversas ferramentas que facilitam o desenvolvimento de aplicações monolíticas;
- ***Deploy* simples**: há apenas um artefato a ser entregue em cada *deploy*;
- **Simples de escalar**: basta executar múltiplas instâncias da aplicação.

Por fim, os **desafios** relacionados são listados abaixo.

- **Troca de tecnologias**: a adoção de uma nova ferramenta ou linguagem de programação pode ser muito custosa em grandes projetos, o que é problemático caso a tecnologia utilizada se torne obsoleta;
- **Resiliência**: um único erro pode derrubar todo o sistema, o que afeta negativamente a disponibilidade.
- **Manutenção a longo prazo**: conforme o projeto cresce e mais desenvolvedores trabalham em cima dele, pode ocorrer a erosão arquitetural, em que a estrutura inicialmente planejada para o projeto vai se degradando, e aumenta a dificuldade de se manter decisões arquiteturais;
- **Escalabilidade não seletiva**: caso haja demanda para apenas uma funcionalidade, como busca de produtos por exemplo, não é possível escalar apenas uma parte do projeto.

2.2.2 Microsserviços

De acordo com Lewis e Fowler (2014):

> em resumo, o estilo arquitetural de microsserviços é uma abordagem na qual uma única aplicação é desenvolvida como um conjunto de pequenos serviços, cada um rodando seu próprio processo e se comunicando por meio de mecanismos leves, comumente APIs HTTP. Estes serviços são construídos sob as regras de negócio e com *deploys* independentes, feitos por um maquinário completamente automatizado. Há o mínimo de controle centralizado entre os serviços, que podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias para armazenamento de dados.

O desacoplamento entre microsserviços possibilita escalabilidade, uma das características principais dessa arquitetura. Assim, múltiplos times podem trabalhar de forma independente, cada um com seu serviço. Por outro lado, o sistema passa a ser distribuído, o que traz desafios complexos (Richards, 2015). Alguns exemplos são: comunicação, consistência de dados, testes entre serviços, além da necessidade de uma infraestrutura robusta que dê suporte aos múltiplos serviços.

Na literatura, a arquitetura de microsserviços é mencionada como uma alternativa a sistemas monolíticos que enfrentam problemas organizacionais e técnicos. A passagem da arquitetura monolítica para microsserviços pode ser feita através do *Strangler Fig Pattern*

(Fowler, 2004), no qual as funcionalidades presentes no monolito são gradualmente movidas para serviços independentes, até que o monolito se torne obsoleto e seja substituído pelo conjunto de microsserviços. Enfatizando: esse processo ocorre de forma gradativa, dado que uma reescrita completa do monolito é normalmente inviável, devido ao seu tamanho e complexidade. A figura Figura 2.3 mostra um exemplo de quebra de monolito em microsserviços, baseado nos exemplos da subseção anterior.

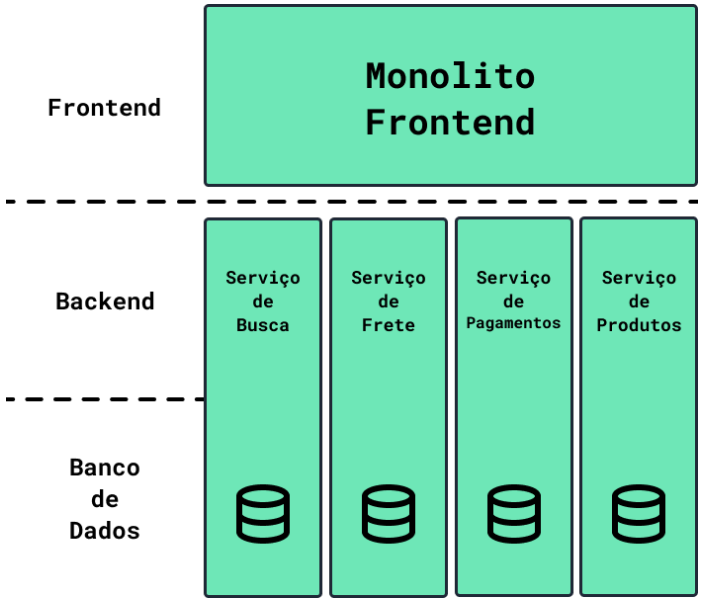

Figura 2.3: *Evolução da figura Figura 2.2, na qual o backend monolítico é quebrado em quatro microsserviços, cada um com seu próprio banco de dados.*

Abaixo, estão as motivações, benefícios e desafios relacionados a microsserviços, baseados no livro de Richardson (2018a).

As **motivações** para uso deste estilo são:

- **Crescimento do sistema:** aumento da complexidade, quantidade de *bugs* e código-fonte, problemas organizacionais;
- **Escalabilidade de times**: o aumento na quantidade de desenvolvedores não resulta num aumento de produtividade ou velocidade de entregas;
- **Escalabilidade de processamento**: é possível escalar apenas um serviço, ou seja, um domínio da aplicação, baseado na demanda;
- **Resiliência**: o sistema precisa estar disponível o máximo de tempo possível. Eventuais falhas não devem interferir na disponibilidade da aplicação.

Separar um projeto em partes menores traz **vantagens** como:

- **Aplicações poliglotas**: permite a experimentação e adoção de novas tecnologias. Cada serviço ou banco de dados pode utilizar a tecnologia mais adequada para o problema;
- **Autonomia**: os times passam a funcionar de forma independente, diminuindo a necessidade de ações coordenadas;

- **Independência**: com projetos independentes, vários aspectos são beneficiados: desenvolvimento, testes, manutenção, *deploy*;
- **Escalabilidade**: inclui o âmbito organizacional — é fácil aumentar os times de desenvolvimento — e estrutural — aumentar a capacidade da aplicação baseado na quantidade de usuários;
- **Resiliência**: caso ocorra um erro em um microsserviço, apenas esse serviço cai. Num sistema monolítico, construído completamente num único projeto, um erro derrubaria toda a aplicação.

Dos **desafios**, destaca-se a grande complexidade de se implementar e manter microsserviços. Abaixo, uma descrição dos desafios:

- **Complexidade**: num sistema distribuído, surgem novas questões: como garantir consistência entre múltiplos bancos de dados? Como abordar comunicação assíncrona entre serviços? Como testar a interação entre múltiplos serviços de forma consistente?
- **Infraestrutura**: é necessário configurar e manter uma infraestrutura que dê suporte à nova arquitetura, com múltiplos serviços funcionando simultaneamente. Note que cada serviço criado demanda configuração e monitoramento individual;
- **Divisão de limites de cada serviço**: para usufruir dos benefícios da arquitetura, é essencial evitar acoplamento entre serviços. Para isso, é necessário compreender e respeitar os limites de cada serviço. No entanto, encontrar esses limites também é um desafio;
- **Custo de comunicação em rede**: existe latência em cada chamada entre serviços, o que não acontecia quando todos os módulos ficavam no mesmo projeto. Também é importante se atentar à quantidade de chamadas, pois é possível sobrecarregar o próprio sistema.

2.2.3 Arquitetura de *Micro Frontends*

Micro frontends é "um estilo arquitetural no qual aplicações *frontend* independentes são unidas numa singularidade maior." (Jackson, 2019). O projeto é decomposto em pedaços menores e mais simples, que podem ser desenvolvidos, testados e entregues de forma independente, enquanto mantém a aparência de um único produto para o usuário final.

De acordo com Peltonen *et al.* (2021), "*micro frontends* estende a ideia da arquitetura de microsserviços e muitos dos princípios de microsserviços se aplicam para *micro frontends*." Logo, é natural propor a implementação das duas arquiteturas simultaneamente. Dado que os domínios são diferentes, *backend* e *frontend*, é possível aproveitar os conhecimentos sobre sistemas distribuídos e a infraestrutura existente, sem que haja conflitos entre os diferentes serviços. Por esses motivos, a abordagem de *micro frontends* é vista como o próximo passo após a implementação de microsserviços (Yang *et al.*, 2019).

As motivações, benefícios e desafios elencados por Peltonen *et al.* (2021) e Geers (2020) serão listados abaixo, iniciando pelas **motivações**:

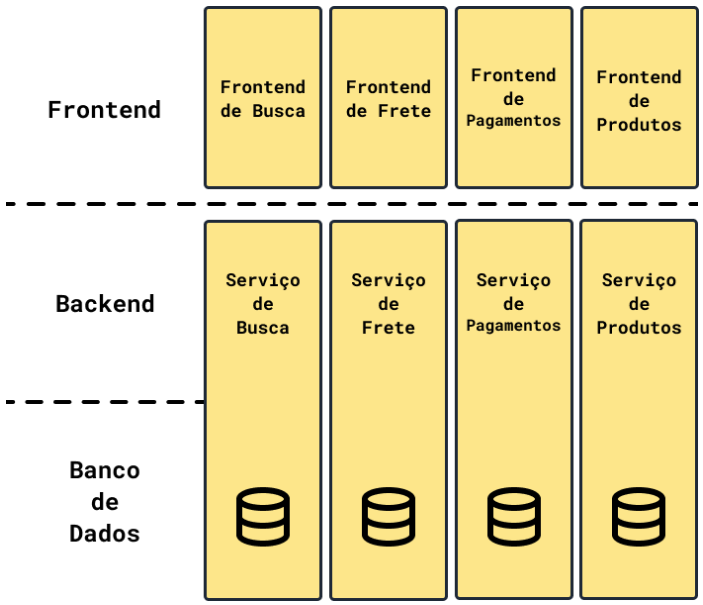

Figura 2.4: *Evolução da Figura 2.3, na qual o frontend monolítico é quebrado em quatro micro frontends, com contextos equivalentes aos microsserviços.*

- **Crescimento do *frontend***: o aumento da complexidade, quantidade de código e problemas organizacionais são os principais motivos para a adoção;
- **Escalabilidade de desenvolvimento**: inclui necessidade de *deploys* e times independentes, velocidade de entrega, falta de inovação e *onboarding* lento de novos desenvolvedores;
- **Seguir as tendências**: algumas empresas implementam microsserviços e *micro frontends* apenas para seguir as tendências do mercado.

Vale notar que os **benefícios** observados são similares aos de microsserviços. São esses:

- **Diferentes tecnologias**: cada *micro frontend* pode adotar as tecnologias mais adequadas para cada situação;
- **Times multidisciplinares**: com o sistema completamente dividido em serviços, tanto *backend* e *frontend*, é possível organizar os times de forma multidisciplinar. Times especializados executam apenas uma única função: infraestrutura, *backend*, *frontend*, design, marketing, etc. Já os times multidisciplinares contêm pessoas com diferentes especialidades, trabalhando em conjunto numa área do negócio: busca, recomendação, pagamento, etc;
- **Independência em diversos aspectos**: tanto os projetos quanto os times passam a ser independentes, o que facilita o desenvolvimento, testes e entrega de novas funcionalidades;
- **Escalabilidade**: inclui o âmbito organizacional — é fácil aumentar os times de desenvolvimento — e estrutural — aumentar a capacidade da aplicação baseado na quantidade de usuários;

- **Resiliência**: caso ocorra um erro em um *micro frontend*, apenas esse serviço cai. Num *frontend* monolítico, construído completamente num único projeto, um erro derrubaria toda a aplicação.

Da mesma forma, grande parte dos desafios são análogos, apesar de que alguns detalhes são excluivos de *frontend*:

- **Complexidade**: lidar com uma arquitetura distribuída, com múltiplos projetos e diferentes tecnologias;
- **Consistência de interfaces**: uma das premissas de *micro frontends* é que o resultado da união das partes deve ser coeso e indiscernível. Assim, todos os serviços devem compartilhar um padrão estético;
- **Governança**: definir qual time é responsável por um serviço pode se tornar um desafio quando múltiplos times o usam ou trabalham na mesma área de negócio;
- **Desempenho**: é necessário lidar com duplicação de código, dependências compartilhadas e quantidade de dados enviados ao usuário no momento em que os serviços são unidos.

De acordo com Mezzalira (2020), há quatro decisões a serem tomadas ao **implementar** *micro frontends*, as quais ele denomina *Micro Frontends Decision Framework*: Definir (*Define*), Compôr (*Compose*), Rotear (*Route*) e Comunicar (*Communicate*). Esses conceitos serão apresentados a seguir.

Primeiro, é necessário **definir** o que é um *micro frontend* dentro do projeto. Há duas abordagens principais: **horizontal**, mostrada na Figura 2.5, na qual cada *micro frontend* seria uma parte da página, ou **vertical**, como visto na Figura 2.6, na qual há um *micro frontend* por contexto ou domínio do negócio. Nessa segunda, um único *micro frontend* pode conter partes, uma página completa ou até múltiplas páginas. Essa decisão é essencial pois impacta diretamente as próximas.

Figura 2.5: *Exemplo da abordagem horizontal com cinco micro frontends: cabeçalho, menu, produtos recomendados, produtos em promoção e rodapé.*

O próximo passo é decidir como **compôr** os serviços em uma unidade coesa. Caso a escolha anterior seja por *micro frontends* horizontais, existem três abordagens principais: *client-side*, *edge-side* e *server-side*, sendo possível adotar apenas uma ou combiná-las. No

Figura 2.6: *Exemplo da abordagem vertical com três micro frontends: produto, carrinho e pagamento, cada um servindo sua própria página completa.*

caso da escolha por *micro frontends* verticais, essa decisão não é necessária, pois cada serviço já entrega a aplicação completa.

A composição *client-side* acontece no próprio navegador do usuário; a composição *edge-side* é feita em nível de CDN, enquanto a *server-side* é feita em nível de servidor, como mostra a Figura 2.7. Cada abordagem possui uma série de vantagens e dificuldades, além de usar tecnologias diferentes. Como esses tópicos são técnicos e mais próximos de desenvolvimento *frontend* do que de arquitetura, não entram no escopo deste trabalho.

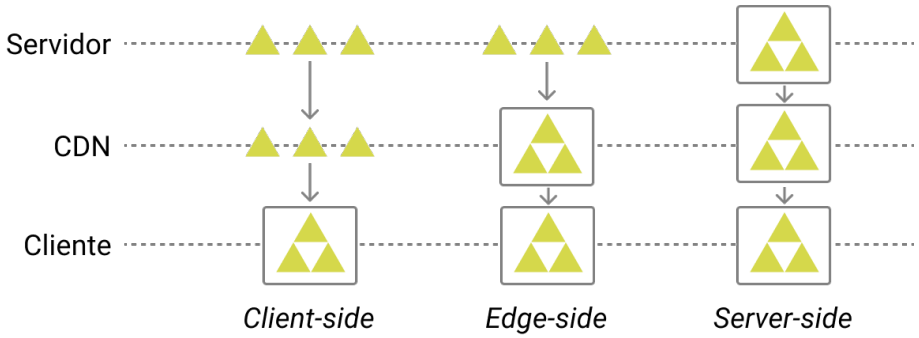

Figura 2.7: *Esquema das três abordagens de composição de micro frontends.*

Em seguida, o **roteamento** define como cada *micro frontend* é acessado. Comumente, cada serviço expõe um *endpoint* que retorna uma resposta HTML ao ser acessado via requisição HTTP. No caso de *micro frontends* horizontais, é necessário utilizar um roteador, camada responsável por fazer as chamadas para os diferentes serviços, que pode estar em nível de cliente, CDN ou servidor. Por outro lado, o caso vertical é novamente mais simples, bastando trocar de URLs para acessar outro serviço. Novamente, esse tópico não será aprofundado.

Por fim, há a **comunicação** entre *micro frontends*. Com a abordagem horizontal, isso é feito em nível de cliente através do envio de eventos no navegador, aproveitando de recursos nativos web. Para *micro frontends* verticais, isso depende das ferramentas utilizadas, mas é uma tarefa trivial dado que todos os componentes ficam no mesmo serviço.

2.3 Padrões Arquiteturais

Um padrão é uma solução para um problema que pode ser reutilizada num contexto específico. Padrões de software resolvem problemas arquiteturais ou de design definindo um conjunto de elementos de software que colaboram entre si. A principal utilidade de um padrão está na presença de um contexto definido, pois facilita discutir e avaliar sua aplicação (Richardson, 2018a).

Ao definir um padrão arquitetural, é comum especificar três aspectos:

- **Forças**: os problemas que devem ser tratados ao adotar este padrão;
- **Contexto resultante**: consequências de aplicar este padrão, que incluem vantagens, desvantagens e novos problemas introduzidos;
- **Padrões relacionados**: relações entre este e outros padrões arquiteturais. Um conjunto de padrões relacionados forma uma *pattern language* ou linguagem de padrões.

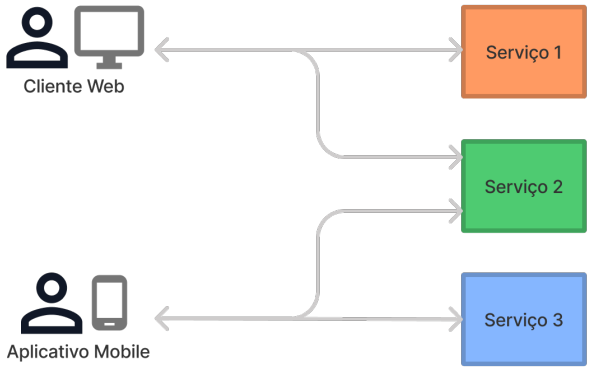

Figura 2.8: *Exemplo de sistema baseado em microsserviços com dois clientes e três serviços.*

O sistema apresentado na figura Figura 2.8 será usado como base para esta seção, pois pode apresentar uma série de problemas como:

- **Acoplamento**: cada cliente deve conhecer a API dos serviços que requisita. Assim, é necessário ter cuidado com o nível de acoplamento, de forma que uma mudança no serviço não cause problemas inesperados nos clientes;
- **Necessidades diferentes**: cada cliente pode necessitar de dados diferentes. Uma aplicação web, por exemplo, tende a mostrar mais informações do que aplicativos de celular;
- **Condições e tecnologias diferentes**: os clientes podem ser utilizados em *hardwares* diferentes. Celulares tendem a ter conexão de rede mais lenta do que desktops, que comumente possuem conexão cabeada. Outro exemplo: máquinas de cartão, para sistemas de pagamento, possuem processamento e memória limitados comparado a celulares. Além disso, cada serviço pode aceitar protocolos diferentes, o que traz a necessidade de tradução de protocolos.

Os padrões relevantes para esta monografia — *API Gateway* (2.3.1) e BFF (2.3.2) — e que foram essenciais na implementação de *micro frontends*, serão apresentados nas subseções

abaixo. Os dois são alternativas para resolver os problemas levantados acima.

2.3.1 *API Gateway*

API Gateway é um serviço que serve como ponto de entrada para outros serviços. Todas as chamadas feitas pelos clientes — por exemplo, um aplicativo nativo para celulares ou um site, para desktops — passam pela *gateway*, que além de roteá-las para cada serviço pode ser responsável por verificações de autorização e tradução de diferentes protocolos.

Este padrão pode ser utilizado no contexto de uma aplicação baseada em microsserviços, no qual existem diferentes clientes que acessam os múltiplos serviços. A comunicação com cada serviço pode ser feita utilizando protocolos diferentes, como HTTP e gRPC[2] (Richardson, 2018b).

Ao implementar o *API Gateway*, a responsabilidade de conhecer os serviços sai do cliente, simplificando-o e diminuindo a quantidade de requisições necessárias para que ele tenha as informações de que precisa. A Figura 2.9 exemplifica a implementação. Além disso, é possível expôr APIs específicas para cada cliente, o que será mostrado na subseção seguinte. Em contrapartida, a complexidade do sistema aumenta dado que a *gateway* é um serviço, que requer desenvolvimento, manutenção, testes e *deploy*.

Há diversas implementações para microsserviços relacionadas à *API Gateway*. Como esse não é o foco deste trabalho, esse ponto não será aprofundado.

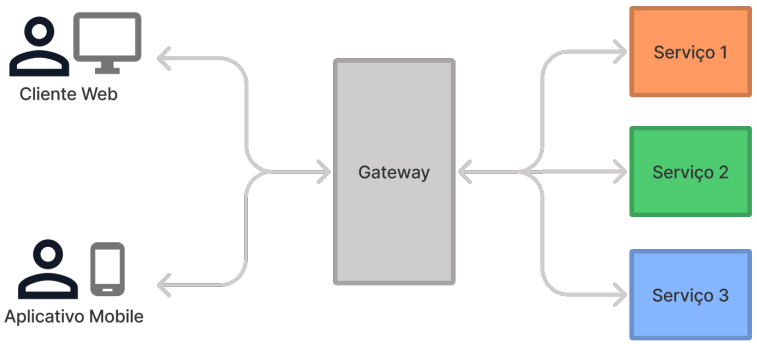

Figura 2.9: *Adaptação da Figura 2.8 com implementação de API Gateway.*

2.3.2 *Backend for Frontend (BFF)*

O padrão *Backend for Frontend* — documentado pela primeira vez por Calçado (2015) — é uma variação do padrão *API Gateway* (Richardson, 2018b), no qual cada cliente possui seu próprio *gateway*. Assim, é possível isolar as necessidades individuais de clientes e ter um time responsável pela aplicação cliente e BFF.

O contexto, benefícios e problemas relacionados são equivalentes aos de *API Gateway*.

[2] https://grpc.io/

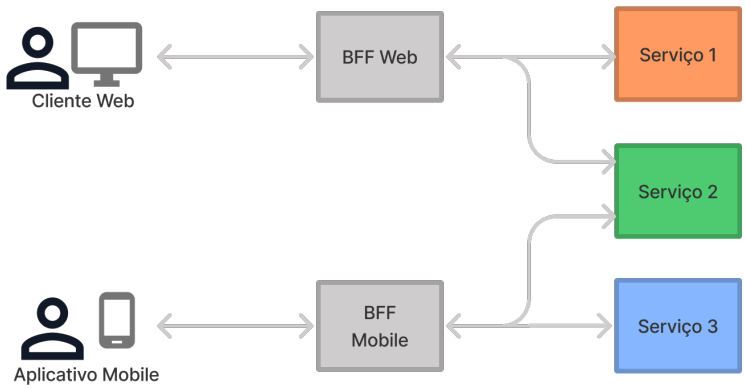

Figura 2.10: *Adaptação da figura Figura 2.8 com implementação de BFF.*

Capítulo 3

Frontend no Elo7

Por ser um *ecommerce*, o *frontend* é essencial para o sucesso do Elo7. O site precisa ser performático, simples de usar, disponível para *desktops* e dispositivos *mobile*. Neste capítulo serão apresentados a empresa Elo7 (3.1), seus principais sistemas para *frontend* (3.2) e a arquitetura antes da implementação de *micro frontends* (3.3).

3.1 Sobre o Elo7

O Elo7 foi fundado em 2008, com o objetivo de ser uma plataforma *marketplace* de produtos artesanais. Já em 2009, o site superou a marca de dez mil usuário e cem mil produtos vendidos, chegando a um milhão de usuários em 2013. De acordo com a própria empresa[1], sua missão é transformar a vida das pessoas através de um ambiente humanizado de compra e venda que conecta e inspira. Em 2021, a empresa foi adquirida pela Etsy[2], empresa estadunidense do mesmo ramo.

Em dezembro de 2021, a empresa contava com mais de 200 funcionários, sendo aproximadamente 50 da engenharia. Há dois times dedicados ao desenvolvimento *frontend*: Nymeros e Martell[3], com um total de nove membros.

Figura 3.1: *Logo da empresa.*

[1] https://www.elo7.com.br/sobre

[2] https://www.etsy.com/

[3] Os nomes dos times são referência à série de livros de fantasia épica "As Crônicas de Gelo e Fogo", conhecida também pela série de TV *Game of Thrones*.

3.2 Sistemas de *Frontend*

Antes da implementação de *micro frontends*, o Elo7 passava pelo processo de estrangulamento do monolito, conhecido como Marketplace. Ou seja, parte das funcionalidades de *backend* havia sido movida para microsserviços independentes. No entanto, as responsabilidades de *frontend* continuavam no monolito, sendo que alguns componentes eram servidos pelo Water Gardens, um projeto separado e dedicado apenas a esse propósito.

Nesta seção, serão apresentados os projetos Marketplace (3.2.1) e Water Gardens (3.2.2).

3.2.1 *Marketplace*

Marketplace é um sistema monolítico criado em Java, responsável por grande parte da lógica de negócio do Elo7. O projeto conta com mais de 4 milhões de linhas de código, 7,6 mil classes Java e 56 mil *commits* feitos por 96 contribuidores diferentes. Toda a parte de cadastro de usuários, produtos, chat, compra e venda é administrada pelo Marketplace.

O projeto é responsável pelo *backend* e *frontend*, e era o único ponto de entrada para todo o conjunto de sistemas do Elo7 — e consequentemente o único ponto de falha. As lógicas de negócio eram desenvolvidas em Java, enquanto as interfaces eram criadas utilizando JSP, Sass[4] e bibliotecas JavaScript desenvolvidas internamente.

Sua estrutura monolítica permitiu uma rápida evolução do sistema no início de seu desenvolvimento. Entretanto, após anos de manutenção — o sistema foi criado em 2012 —, desenvolver no Marketplace se tornou um problema para o Elo7. Seu tamanho, complexidade, alto tempo de *deploy* e erosão arquitetural estão entre os principais empecilhos. Assim, diversas funcionalidades como a busca de produtos, cálculo e gerenciamento de frete, e categorização de produtos foram movidas para microsserviços, como parte do processo de estrangulamento (*Strangler Fig Pattern*).

3.2.2 *Water Gardens*

Water Gardens, também conhecido como Water7 ou W7, é um sistema baseado em NodeJS[5] que serve componentes para o Marketplace. Esses componentes são criados utilizando HTML, Sass e DustJS[6], biblioteca criada pelo LinkedIn[7].

Este projeto surgiu a partir da demanda por um chat entre comprador e vendedor. Esse chat deveria ser enriquecido com informações advindas do *backend*, como status do pedido, situação de pagamento e frete, além de mostrar botões com ações possíveis para o usuário, como mostra a 3.2. Para isso, seria necessário criar uma interface reativa.

[4] https://sass-lang.com/

[5] https://nodejs.org/

[6] http://www.dustjs.com/

[7] https://www.linkedin.com/

A reatividade, no contexto do desenvolvimento *frontend*, é a capacidade da aplicação atualizar sua interface após uma mudança de estado (Mezzalira, 2018). Utilizando um chat como exemplo, ao receber uma nova mensagem, ela deve aparecer na tela. Caso a interface não seja reativa, seria necessário atualizar a página constantemente para verificar se novas mensagens chegaram, o que seria terrível para a experiência do usuário. Ter interfaces reativas é trivial com uso de *frameworks* modernos como React[8] ou VueJS[9]. No entanto, esses *frameworks* ainda não estavam disponíveis na época — início de 2016 — e a solução escolhida foi desenvolver bibliotecas internas.

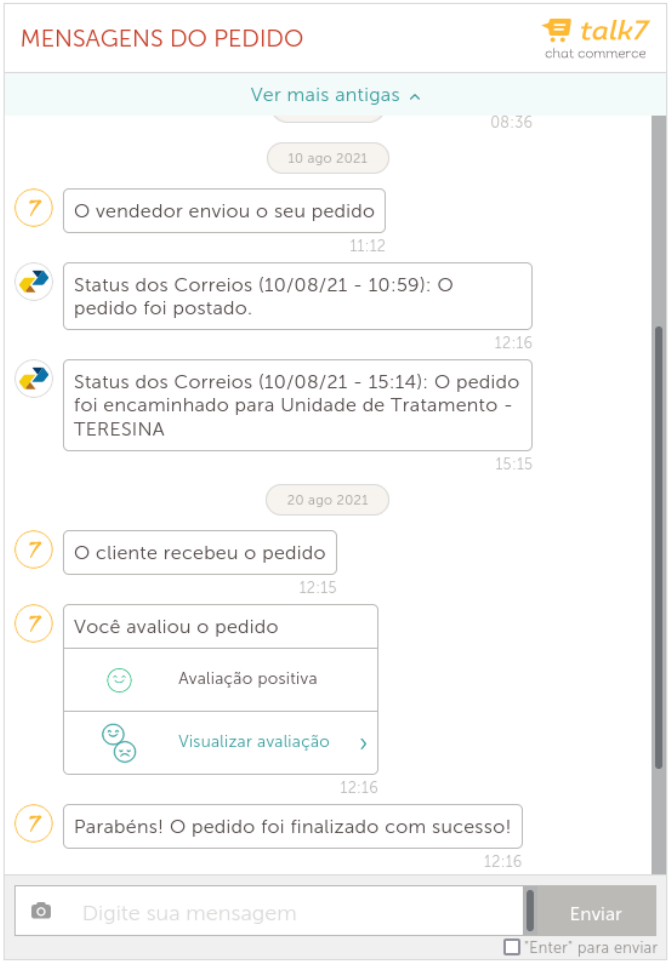

Figura 3.2: *Captura de tela do chat entre comprador e vendedor.*

Além da impossibilidade de reproduzir a reatividade dentro do monolito, uma das demandas técnicas era renderizar as mensagens do chat tanto em *client-side* quanto em *server-side*. Assim, foram criados o próprio Water Gardens, em NodeJS, e bibliotecas JavaScript para reatividade. O NodeJS foi escolhido pela possibilidade de usar isomorfismo, ou seja, utilizar a mesma base de código para cliente e servidor, atendendo a demanda técnica e evitando uma implementação dupla.

[8] https://reactjs.org/

[9] https://vuejs.org/

Por parte do Marketplace, foram feitas adaptações para utilizar o Water Gardens. Ainda assim, grande parte da página é produzida dentro do monolito, que delega algumas partes para o W7. Todos os dados necessários para montar os componentes são formatados pelo Marketplace e enviados para o Water Gardens. O W7 então monta o componente, devolvendo código HTML para o monolito. Esse HTML, que contém também a estilização e a lógica necessários para funcionamento do componente, é injetado na página, que é finalmente entregue ao usuário.

Na época, o projeto foi considerado um sucesso. Apesar das dificuldades e restrições encontradas ao desenvolver os dois sistemas em paralelo, as funcionalidades necessárias para a empresa foram atendidas, e os dois projetos continuaram a evoluir. No entanto, a biblioteca DustJS foi rapidamente descontinuada, logo ao final de 2016. Isso gerou uma série de problemas como dificuldade de manutenção, falta de documentação, falta de uma comunidade ativa, além de não ser atrativo para profissionais trabalhar com tecnologias depreciadas.

3.3 *Problemas Arquiteturais*

Devido às restrições impostas para o desenvolvimento do Water Gardens, houve um grande acoplamento com o Marketplace.

Primeiro, não havia mão de obra nem tempo hábil para retirar funcionalidades do monolito. Como o chat fica numa área restrita da aplicação, na qual é necessário login, o monolito cumpria o trabalho de autenticador e ponto de entrada para o W7. Logo, há uma inversão de responsabilidades, com o *backend* sendo acionado antes do *frontend*.

A segunda restrição era atribuir o mínimo de responsabilidades ao Water Gardens, devido à pouca experiência com NodeJS. Assim, toda a orquestração e formatação de dados era feita no Marketplace e repassada para o W7. Desta forma, em mudanças no *frontend* relacionadas a dados, o desenvolvedor era obrigado a trabalhar com o sistema *backend* em Java, o que podia não ser sua especialidade.

Assim, apesar de serem projetos diferentes, o processo de desenvolvimento e manutenção do Water Gardens ficou vinculado ao monolito. Esse problema arquitetural, somado ao uso de tecnologias descontinuadas, impulsionou as mudanças apresentadas no Capítulo 4.

Figura 3.3: *Esquema simplificado do fluxo de requisições para frontend antes da implementação de micro frontends.*

Capítulo 4

Implementando *Micro Frontends* no Elo7

Devido aos problemas apresentados no capítulo anterior, foi criado um plano para a implementação de uma nova arquitetura *frontend*, assim como a adoção de tecnologias modernas. Foi escolhida a arquitetura de *micro frontends*, que facilita o isolamento do *front*, além de possibilitar a quebra de interfaces em partes menores e o uso de diferentes tecnologias.

Neste capítulo, serão apresentados os principais sistemas implementados para essa mudança (4.1), a situação atual da arquitetura (4.2) e as possibilidades para o futuro (4.3).

4.1 Novos sistemas

Para implementar uma arquitetura de *micro frontends*, era necessária a criação de um serviço que funcionasse como *API Gateway*. Assim, o monolito deixaria de ser o principal ponto de entrada e passaria a ser requisitado apenas quando necessário. A *gateway* assume o papel de roteador, permitindo a chamada dos novos serviços de *frontend*.

Outro aspecto essencial foi a escolha das tecnologias utilizadas. Para selecionar quais satisfaziam as necessidades da empresa, foram feitos testes com múltiplos *frameworks* e bibliotecas, analisando desempenho e experiência de desenvolvimento. Esses testes foram feitos antes do início deste trabalho e não serão cobertos por falta de documentação.

Nas subseções abaixo, serão apresentados os projetos criados para a nova arquitetura, as tecnologias utilizadas e suas funções.

4.1.1 Brownie

Brownie é o projeto de renovação do blog técnico do Elo7[1]. O objetivo principal deste sistema é servir como prova de conceito das tecnologias *frontend* escolhidas: Svelte[2], um *framework* web moderno e eleito o mais amado pelos desenvolvedores em 2021[3], e Fastify[4], um servidor NodeJS de alta performance.

O desenvolvimento deste sistema se deu até que certas características como estrutura de diretórios, arquivos de configuração, uso de bibliotecas auxiliares, padrões de código e experiência de desenvolvimento fossem consideradas maduras o bastante para serem utilizadas em produção. Assim, foi criado um clone do Brownie, denominado Waffle, para que fossem desenvolvidas páginas do site Elo7. Os detalhes do projeto Waffle serão apresentados Subseção 4.1.4.

4.1.2 Nightfort

Nightfort é a implementação do padrão *API Gateway*, criado utilizando Zuul[5], biblioteca de código aberto disponibilizada pela Netflix[6]. Este projeto funciona como ponto de entrada principal para os sistemas Elo7. Todas as requisições feitas por usuários passam pela *gateway*, que as redireciona devidamente para outros serviços.

Além de rotear, a *gateway* também realiza verificações no conteúdo das requisições e atua na segurança de outros sistemas.

A implementação deste projeto foi mandatória para que os *micro frontends* pudessem assumir, parcialmente, a responsabilidade do monolito de servir interfaces.

4.1.3 The Reach

The Reach é o *design system* do Elo7. Um *design system* é um conjunto de padrões para criação de interfaces e componentes gráficos, que inclui cores, espaçamentos, fontes, ícones, entre outros. Ao seguir esses padrões, cria-se uma identidade visual para a marca, de forma que um cliente consegue identificar propagandas e sites baseando-se apenas em alguns elementos visuais (Hacq, 2018).

Do ponto de vista dos desenvolvedores, implementar um *design system*, ou seja, criar componentes *frontend* utilizando esses padrões, permite a reutilização de código, facilita o desenvolvimento e torna os *micro frontends* visualmente coerentes. Em outras palavras, apesar de serem projetos diferentes, desenvolvidos por times diferentes, o resultado da composição dos *micro frontends* não causa estranhamento aos usuários.

---

[1] https://blog.elo7.dev/

[2] https://svelte.dev/

[3] https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-web-frameworks

[4] https://www.fastify.io/

[5] https://github.com/Netflix/zuul

[6] https://netflix.github.io/

A implementação do The Reach é feita em uma biblioteca NodeJS, na qual os estilos e comportamentos podem ser reutilizados de forma simples. Os padrões e componentes já foram criados pelo time de design do Elo7, mas as atualizações na biblioteca são feitas apenas quando necessário, ou seja, quando a página a ser desenvolvida requer algo ainda não implementado.

4.1.4 Projetos Waffle

O primeiro projeto Waffle foi criado a partir do Brownie, com o objetivo de mover a página de busca de materiais para artesanato do monolito para *micro frontends*. Essa página foi escolhida pois é visualmente similar à página de busca de produtos, uma das principais do site, mas com menos acessos, em comparação. Após meses de desenvolvimento e refinamento, esse projeto foi colocado em produção e atualmente todo o fluxo de usuários da busca de materiais é atendido pelo Waffle.

Em seguida, foram criados outros dois projetos clones, para atender demandas internas do Elo7. Todos os sistemas criados a partir do Brownie são chamados de projetos ou sistemas Waffle, e cada um possui um nome complementar para ajudar a identificá-lo, como "Waffle Discovery" por exemplo. Além disso, todos os *micro frontends* atuais são projetos Waffle.

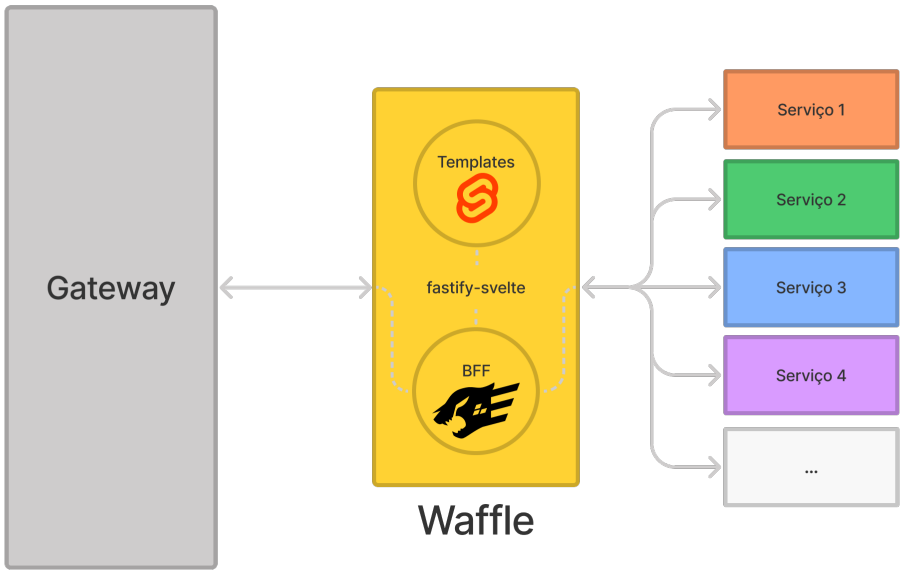

Figura 4.1: *Esquema arquitetural de um projeto Waffle.*

A figura 4.2 mostra a estrutura básica de um *micro frontend* no Elo7. As requisições de um usuário passam pela *API Gateway* e são redirecionadas para o Waffle. O BFF (*Backend For Frontend*), construído com Fastify, recebe a requisição e faz o roteamento interno. Ademais, cuida da orquestração e formatação dos dados, recebidos através de outros microsserviços. Esses dados são repassados para o *template* Svelte — que contém HTML, CSS e JavaScript — via uma biblioteca de código aberto desenvolvida pelo próprio Elo7: *fastify-svelte*[7]. A biblioteca também é responsável por permitir a renderização desses

[7] https://github.com/elo7/fastify-svelte

templates.

Essa é a arquitetura comum à todos os *micro frontends*, que totalizam três serviços até o momento. Um deles, o projeto mencionado no início desta subseção, é resultado do estrangulamento do monolito, enquanto os outros dois entregam páginas inéditas, suprindo novas demandas do time de produto.

4.1.5 *Waffle Core*

Após a criação de três projetos Waffle, os times de *frontend* puderam analisar quais módulos eram comuns a todos os serviços. Com este mapeamento, os trechos compartilhados foram movidos para uma nova biblioteca, um *framework* de código aberto para a criação de Waffles, conhecido como *Waffle Core*.

Uma das vantagens deste projeto é abstrair código de funcionamento interno dos Waffles — código que não é alterado com frequência — o que diminui a quantidade de arquivos e testes. Outra vantagem é a facilidade na criação de novos *micro frontends*, bastando apenas instalar o *framework* como uma dependência do NodeJS.

O projeto foi iniciado em outubro de 2021 e se encontra em estado inicial de desenvolvimento.

4.2 Arquitetura Atual

Com a concepção dos projetos citados anteriormente, a arquitetura de *frontend* da empresa mudou de forma significativa. A presença da *gateway* permite que outros serviços, além do monolito, façam o tratamento das requisições dos usuários, assumindo parte da responsabilidade de *frontend*. Assim, a geração de novos projetos Waffle se torna simples, necessitando de uma breve adição ao código da *API Gateway*.

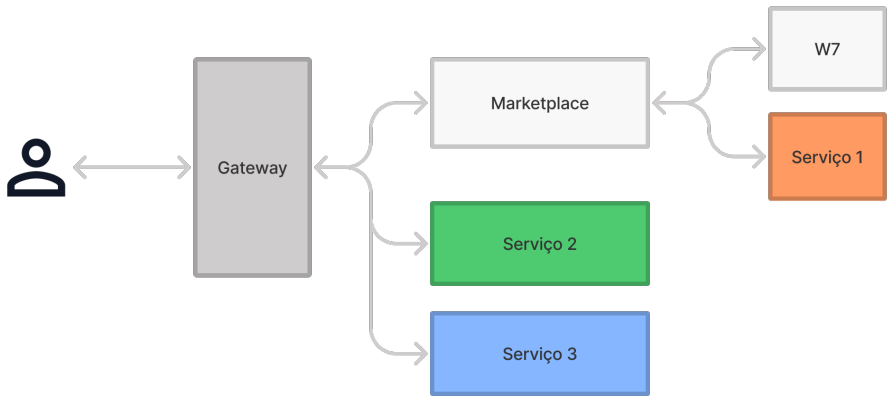

Figura 4.2: *Atualização do fluxo de requisições para frontend após a implementação de micro frontends.*

Sendo assim, a mudança arquitetural do Elo7 foi bem sucedida. A implementação de *micro frontends* verticais (veja a 2.2.3, permitiu a transferência de páginas do monolito para

os projetos Waffle, que utilizam tecnologias modernas e assim facilitam o dia a dia dos desenvolvedores. Também simplificou a geração de novos sistemas de *frontend*, dedicados aos novos produtos idealizados pela empresa.

É notável que o contexto no qual o Elo7 se encontrava, no meio de uma mudança arquitetural para microsserviços, favoreceu a implementação de *micro frontends*. Várias das tecnologias e infraestrutura utilizadas pelos microsserviços — e que estão fora do escopo deste trabalho — foram reaproveitadas para os projetos Waffle. Afinal, *micro frontends* também são microsserviços.

4.3 O Futuro da Arquitetura no Elo7

A evolução natural da arquitetura é mover mais páginas do monolito para os novos serviços, além da criação de mais *micro frontends*. Quanto à criação de novos serviços, não há planos para isso. Decidiu-se que a divisão de domínios seria baseada nos produtos da empresa. Assim, enquanto novos produtos não são concebidos, o foco dos desenvolvedores é incrementar os serviços já existentes.

Entretanto, há um gargalo: para que mais páginas sejam movidas para os projetos Waffle, é necessários mover funcionalidades do monolito para microsserviços. No momento, não há disponibilidade dos times de *backend* para criar esses novos serviços, e portanto não será possível requisitar os dados necessários para montar as interfaces. Para contornar essa limitação, os times de *back* e *frontend* devem definir contratos até o final de 2021. Com isso, será possível adiantar o desenvolvimento das interfaces, simulando chamadas para os futuros serviços.

É notável que, apesar de adotar microsserviços e *micro frontends*, o Elo7 não implementou times multidisciplinares de forma imediata. Essa possibilidade está sendo analisada com calma, seguindo a Lei de Conway Conway (1968), pois a mudança organizacional acopla a distribuição de times à arquitetura de software.

Isso significa que, caso ocorresse uma nova mudança arquitetural, como a reversão de microsserviços para monolito, documentada por Mendonça *et al.* (2021), seria necessário esforço organizacional e técnico. Por este motivo, é questionável que a adoção de times multidisciplinares é um benefício de *micro frontends*, e a literatura não cita provas de que tal mudança é sempre positiva.

Capítulo 5

Questionário com desenvolvedores

Para compreender melhor os impactos trazidos pela mudança arquitetural, um questionário foi aplicado entre os colaboradores do Elo7 direta ou indiretamente envolvidos com *frontend*. Assim, o objetivo deste capítulo é apresentar o questionário e os resultados obtidos.

A Seção 5.1 comenta a metodologia utilizada. Em seguida, a Seção 5.2 discorre sobre o perfil dos respondentes. Por fim, as Seção 5.3 e Seção 5.4 apresentam os resultados obtidos através de análises feitas sobre as respostas.

5.1 Metodologia

Um dos objetivos deste trabalho é avaliar a implementação de *micro frontends* feita pela empresa. Sendo assim, é razoável se voltar àqueles envolvidos: os desenvolvedores. Foi criado, então, um questionário semiaberto com o intuito de verificar quais eram os problemas enfrentados, as expectativas de mudança e os impactos percebidos pelos respondentes.

O questionário foi estruturado com o objetivo de tentar, ao máximo, separar os impactos da adoção de novas tecnologias dos impactos da nova arquitetura. Como essas duas modificações complexas ocorreram simultaneamente, era esperado que os desenvolvedores pudessem não ser capazes de dividir suas respostas sem esse tratamento cuidadoso.

Dezenove questões foram formuladas, sendo quinze abertas e quatro de múltipla escolha. Nos casos com múltipla escolha, foi utilizada uma escala de Likert.

Essas perguntas foram distribuídas entre cinco seções. A primeira seção foca em informações do respondente, como nome, experiência profissional e envolvimento com o *frontend* do Elo7. A segunda contém perguntas sobre a arquitetura pré-*micro frontends*. As seções três e quatro estão relacionadas ao uso de novas tecnologias e à mudança arquitetural, respectivamente. Por fim, a quinta seção enfatiza a compreensão da nova arquitetura.

As questões de múltipla escolha `3.1.` e `4.1.` pedem ao respondente considerar os impactos tecnológicos e arquiteturais, respectivamente, sobre sete aspectos:

- Contratação e *onboarding*;
- *Deploy*;
- Desenvolvimento entre múltiplos times;
- Entendimento do Projeto;
- Implementação de novas funcionalidades;
- Testabilidade;
- Velocidade de desenvolvimento.

Esses aspectos foram escolhidos devido à relevância para a empresa e por sua forte presença na bibliografia estudada.

Para ler o questionário na íntegra, veja o Apêndice A.

5.2 Perfil dos respondentes

Ao total, foram obtidas oito respostas. Detalha-se que houve coleta dos nomes dos respondentes, mas exclusivamente para controle e retorno dos resultados das análises. Na tabela abaixo, há a distribuição dos respondentes baseada em algumas características obtidas através da primeira seção do questionário.

Variáveis	Categoria	Frequência
Sexo	Masculino	87, 5%
	Feminino	12, 5%
Tempo na área de tecnologia	Até 5 anos	12, 5%
	De 5 a 10 anos	25, 0%
	Mais de 10 anos	62, 5%
Cargo no Elo7	CAO	12, 5%
	CTO	12, 5%
	Desenvolvedor de Software *Frontend*	50, 0%
	Líder de Time *Frontend*	25, 0%
Tempo no Elo7	Menos de 1 ano	25, 0%
	De 1 a 2 anos	25, 0%
	De 2 a 5 anos	12, 5%
	Mais de 5 anos	37, 5%

Tabela 5.1: *Tabela de qualificação dos respondentes.*

Da tabela, é possível retirar informações relevantes sobre a população. Destacam-se:

- Apenas uma mulher participou do questionário;
- A maioria dos respondentes possui longa experiência na área de tecnologia;
- Metade dos respondentes atuam como desenvolvedores *frontend*;
- Houve participação dos chefes da engenharia e líderes dos times *frontend*;
- Com relação ao tempo na empresa, a população é bem diversificada.

5.3 Análise quantitativas

Esta seção contempla a análise sobre as questões de múltipla escolha `1.2.`, `1.4.`, `1.6.`, `1.7.`, `3.1.` e `4.1.` do questionário.

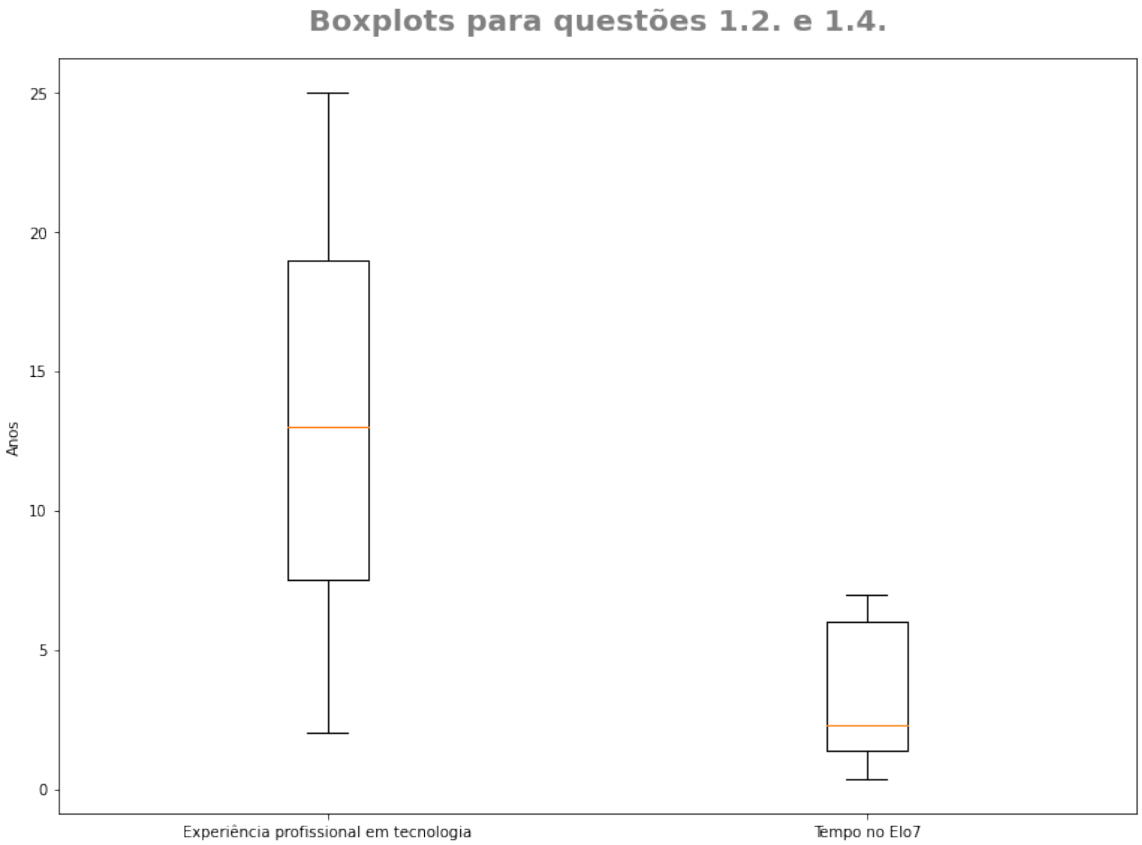

Figura 5.1: *Boxplots das questões 1.2. e 1.4., que envolvem tempo de experiência profissional e no Elo7.*

Como foi mencionado na seção anterior, é possível perceber em 5.1 uma grande disparidade quanto ao tempo de experiência na área de tecnologia, de dois até vinte e cinco anos, e na empresa, de um ano e nove meses até sete anos.

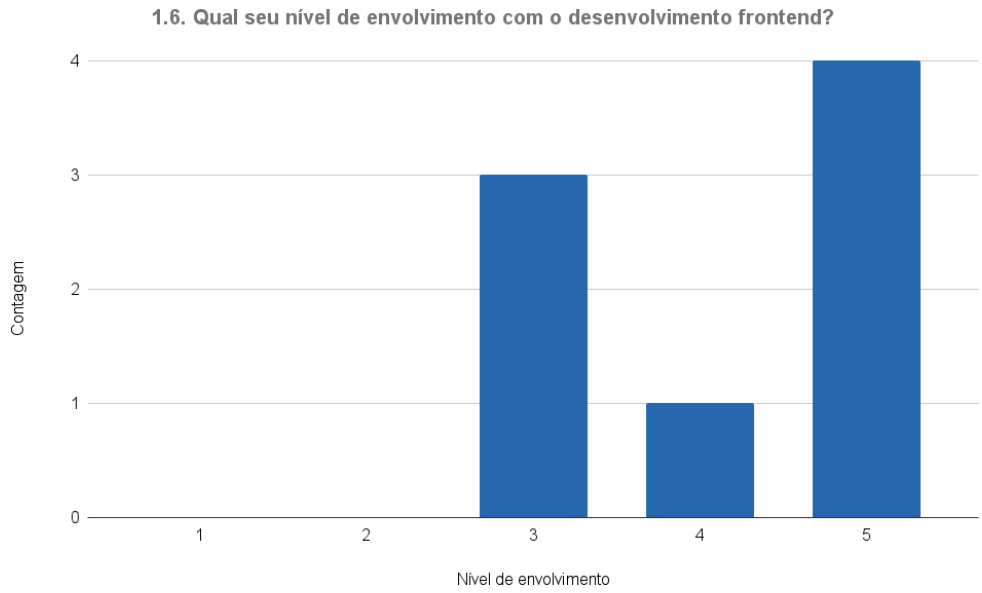

Figura 5.2: *Gráfico de barras sobre o nível de envolvimento com o desenvolvimento frontend, sendo 1 o mais baixo e 5 o mais alto.*

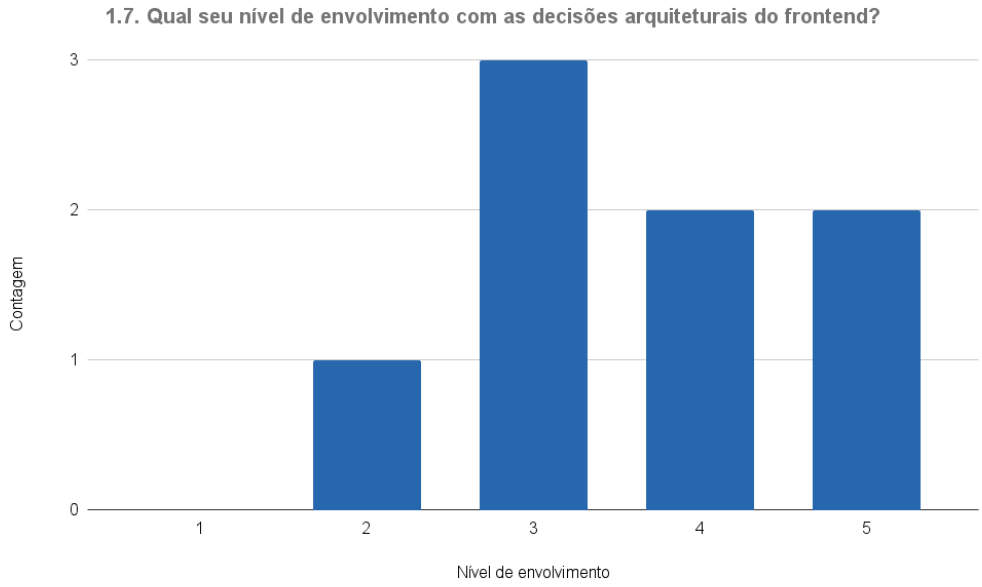

Figura 5.3: *Gráfico de barras sobre o nível de envolvimento com as decisões arquiteturais do frontend, sendo 1 o mais baixo e 5 o mais alto.*

Sobre as questões relacionadas ao envolvimento com o *frontend*, seja no desenvolvimento ou nas decisões arquiteturais, as respostas se concentram nos níveis mediano a alto. Esse resultado está dentro do esperado, dado que os desenvolvedores tem contato diário com o *frontend*, enquanto os chefes da engenharia possuem o conhecimento do todo e participam de forma indireta.

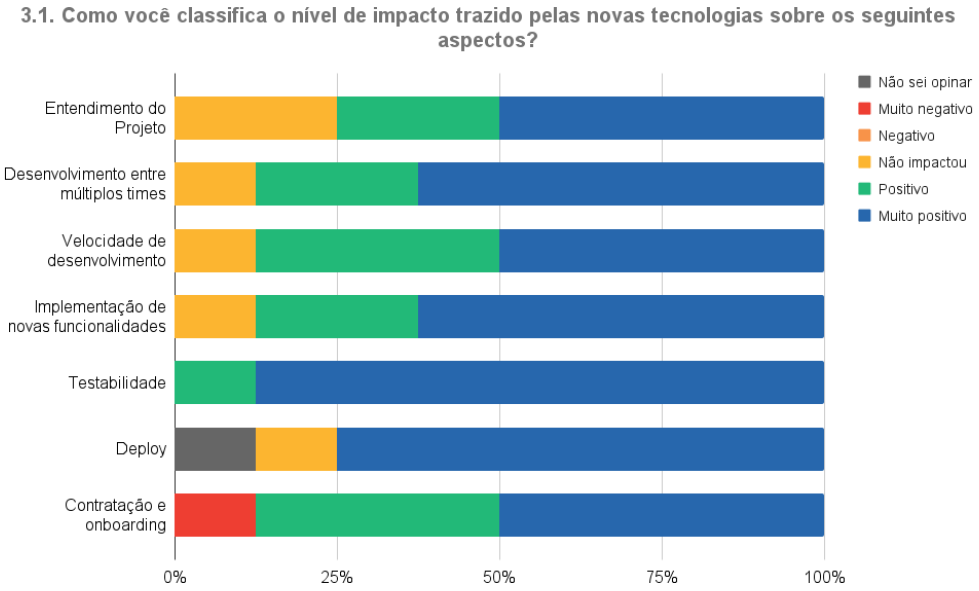

Figura 5.4: *Gráfico de barras horizontais sobre os impactos trazidos pelas novas tecnologias, baseado em porcentagem.*

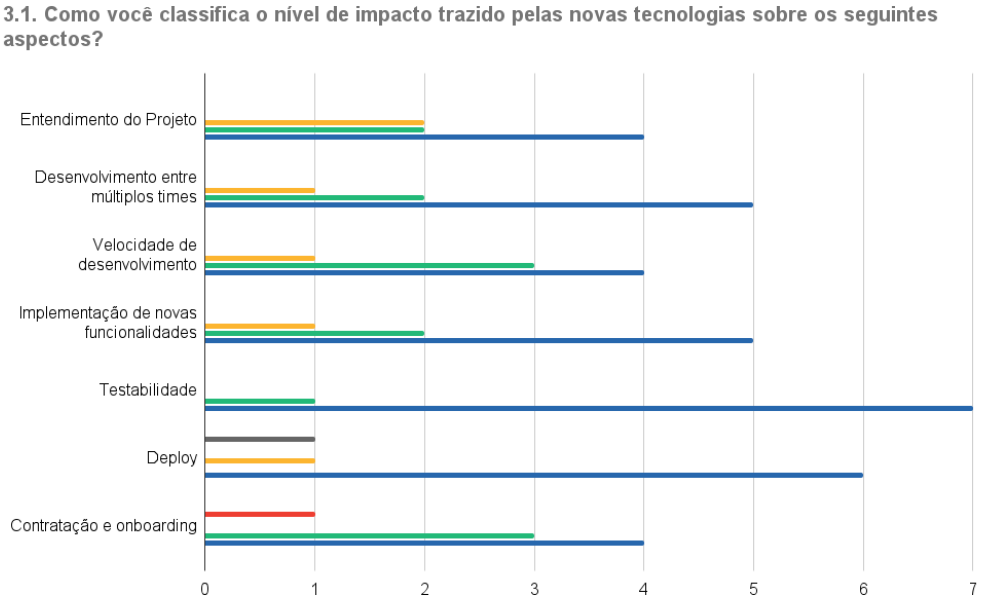

Figura 5.5: *Visualização alternativa à Figura 5.4, desta vez baseada em contagem.*

Analisando as novas tecnologias adotadas, as respostas foram dominantemente positivas ou muito positivas. Em apenas um aspecto, contratação e *onboarding*, aparece uma resposta contraditória. No entanto, ao examiná-la com mais detalhes, percebe-se que ela é, na verdade, complementar. Comparado ao DustJS, as tecnologias escolhidas são mais atraentes ao mercado de trabalho e de fácil aprendizado aos novos colaboradores. Por

outro lado, caso fossem selecionadas tecnologias amplamente utilizadas, como React, a quantidade de desenvolvedores experientes e que poderiam causar impacto em menos tempo na empresa seria maior. Dessa forma, quanto à contratação e *onboarding*, as novas tecnologias são superiores às anteriores, mas aquém do potencial de outras opções.

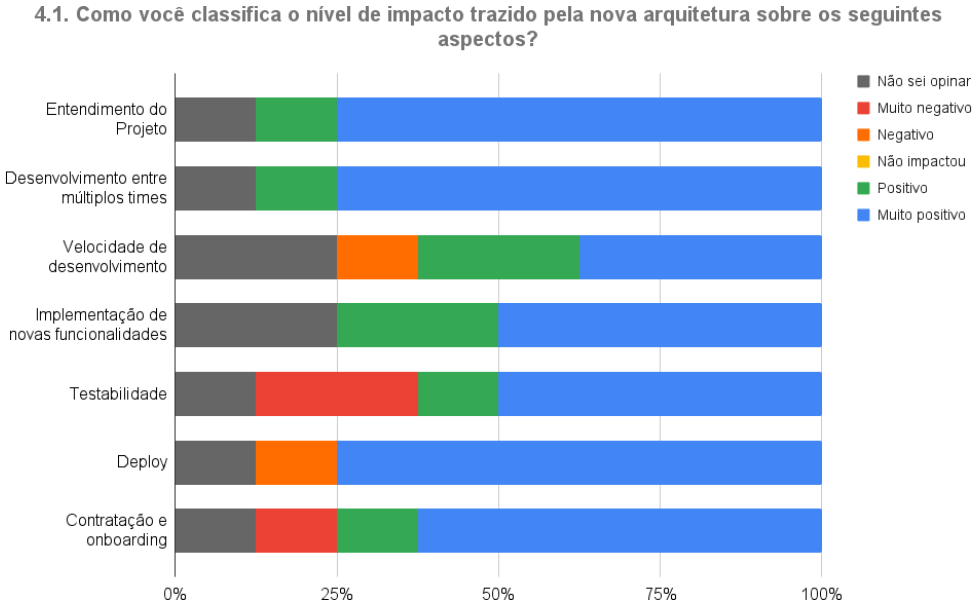

Figura 5.6: *Gráfico de barras horizontais sobre os impactos trazidos pela nova arquitetura, em porcentagem.*

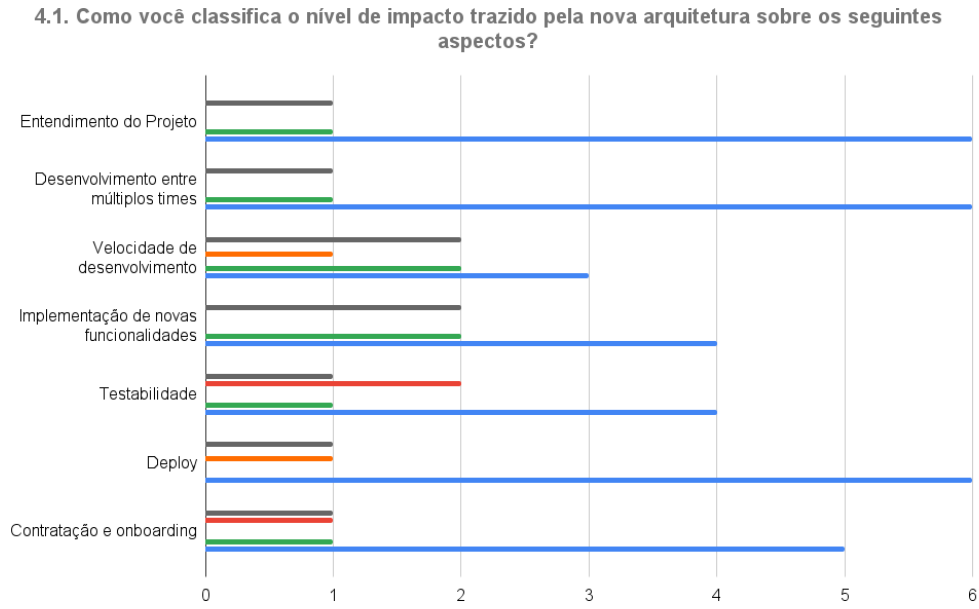

Figura 5.7: *Visualização alternativa à Figura 5.6, baseada em contagem.*

Enfim, sobre a nova arquitetura, nota-se o aparecimento de respostas "*Não sei opinar*", assim como a falta de "*Não impactou*". Isso demonstra escassez de clareza sobre os impactos arquiteturais comparados aos tecnológicos, que são mais palpáveis e presentes no dia a dia dos desenvolvedores.

Dos aspectos que receberam respostas negativas, destacam-se a testabilidade e contratação e *onboarding*. Sobre a testabilidade, da mesma forma que os testes passam a ser desacoplados do monolito, testes ponta a ponta num sistema distribuído — como é o caso de microsserviços e *micro frontends* — são naturalmente complexos. Quanto à contratação e *onboarding*, a apresentação dos serviços é menor e pode ser feita apenas quando necessário, mas o entendimento do sistema completo é notoriamente difícil, demandando mais tempo para familiarização.

5.4 Análise qualitativas

Esta seção contempla a análise sobre as questões abertas `2.1.`, `2.2.`, `2.3.`, `2.4.`, `3.2.`, `4.2.`, `5.1.`, `5.2.`, `5.3.` e `5.4.` do questionário.

Para as questões abertas, as respostas foram analisadas individualmente, e cada trecho considerado relevante foi codificado. Após passar por todas as respostas, os códigos foram agrupados em cinco temas: tecnologias antigas, tecnologias novas, arquitetura antiga, arquitetura nova e *micro frontends*. Cada tema possuía subdivisões, que serviram para agrupar os tópicos.

Sobre as tecnologias e arquitetura antigas, o propósito era entender quais os principais pontos positivos e negativos, além de verificar quais características os respondentes esperavam que seriam melhoradas. Ou seja, o foco é as expectativas dos participantes sobre as mudanças.

Sobre tecnologias e arquitetura novas, esperava-se obter as vantagens e desvantagens observadas pelos respondentes após a adoção de Svelte, Fastify e *micro frontends*. Neste caso, o objetivo é analisar os impactos sentidos pelos participantes em suas atividades diárias.

Por fim, o último objetivo era avaliar o entendimento sobre a definição de *micro frontends*, sobre a nova arquitetura e verificar se os participantes consideram que a arquitetura implementada no Elo7 é uma instância de *micro frontends*.

Note que, nas tabelas a seguir, há elementos destacados em negrito e com um asterisco. Eles representam os pontos mais citados dentre as respostas para aquele determinado tema. Nos casos em que não há destaque, não houve resposta dominante.

Tecnologias Antigas	
Vantagens	Componentização (no Water Gardens) Testabilidade com JSP Uso do NodeJS (no Water Gardens)
Desvantagens	**DustJS*** Experiência de desenvolvimento ruim Falta de funcionalidades modernas Uso de Java para *frontend* (por parte do Marketplace)
Aspectos que melhorariam com uso de novas tecnologias	Apoio da comunidade Comunicação entre sistemas **Contratação e *onboarding**** Divisão de responsabilidades Manutenibilidade Produtividade

Tabela 5.2: *Tabela de vantagens e desvantagens das tecnologias antigas, de acordo com os respondentes.*

Houve poucos comentários sobre as tecnologias antigas. É unânime a visão de que o uso da biblioteca DustJS é prejudicial ao desenvolvimento, e que a troca por um *framework* moderno e com suporte da comunidade ajudaria em diversos aspectos, como contratação e *onboarding*, o mais mencionado.

Tecnologias Novas	
Vantagens	Contratação e *onboarding* *Deploy* via infraestrutura atualizada **Familiariadade*** Experiência de desenvolvedor (DX) Melhor que Water Gardens Testabilidade com biblioteca Storybook Velocidade de desenvolvimento
Desvantagens	Não é padrão de mercado

Tabela 5.3: *Tabela de vantagens e desvantagens das tecnologias novas, de acordo com os respondentes.*

É notável que há apenas um comentário negativo sobre as novas tecnologias, que é um ponto relacionado à contratação e *onboarding*, já explicado na seção anterior. Sobre as vantagens, o destaque é para a familiaridade com as funcionalidades de *frameworks* atuais. O DustJS foi descontinuado ao final de 2016, e apesar de algumas atualizações feitas em outubro de 2021, o projeto não mostra sinais de que possa retornar. O Svelte, *framework* escolhido, é apenas um de vários que surgiram após o DustJS e que oferecem reatividade, desempenho, extensa documentação e suporte da comunidade.

Assim, era esperado que a quantidade de vantagens fosse maior que a de desvantagens. Entretanto, a falta de desvantagens pode ser sinal de um viés atrelado à familiaridade com a nova tecnologia.

Arquitetura Antiga	
Vantagens	Comunicação simples entre componentes W7 *Deploy* independente de Marketplace e W7 Está tudo num único lugar Isomorfismo Modelagem e uso dos dados Páginas performáticas e leves Reaproveitamento de CSS
Desvantagens	**Acoplamento do Water7 com o Marketplace*** Acúmulo de responsabilidades do monolito sobre W7 Alta curva de aprendizado para novos colaboradores Burocracia na comunicação entre Marketplace e W7 Comunicação entre componentes Configuração local do projeto Dependências internas *Deploy* demorado Dificuldade de evolução técnica Divisão dos componentes do *frontend* Falta de documentação Falta de flexibilidade Fluxo de dados entre Marketplace e W7 Infraestrutura antiga Observabilidade para o *frontend* Problemas de manutenibilidade Testes de interface Verbosidade de código
Aspectos que melhorariam com a adoção de uma nova arquitetura	**Acoplamento*** Aumento da produtividade Experiência de desenvolvimento Manutenibilidade Responsabilidade de estilização

Tabela 5.4: *Tabela de vantagens e desvantagens da arquitetura antiga, de acordo com os respondentes.*

Quanto à arquitetura antiga, é evidente a grande quantidade de desvantagens. No entanto, ressalta-se a possibilidade de alguns dos pontos levantados serem relacionados às tecnologias. Como citado na seção sobre metodologia, era esperado que houvesse inconsistências nas respostas e confusões entre as questões sobre tecnologia e arquitetura. Os trechos foram adicionados à tabela sobre arquitetura antiga pois foram mencionados em perguntas arquiteturais.

Das vantagens, a maioria está relacionada ao Water Gardens, que atinge seus objetivos de forma satisfatória, mas sofre com as limitações arquiteturais e tecnologia ultrapassada.

Sobre as desvantagens, a mais citada é o acoplamento entre os sistemas de *frontend*. Este é o principal motivo para a mudança arquitetural, e isso é reforçado por ser o aspecto a ser melhorado mais citado.

	Arquitetura Nova
Vantagens	Atrai mais desenvolvedores e o *onboarding* fica mais simples *Deploy* mais rápido Desenvolvimento entre times Entendimento do projeto fica mais transparente, apesar da complexidade Escalabilidade Possibilidade do uso de novas tecnologias Velocidade e facilidade de implementação
Desvantagens	Evolução depende da extração de serviços do monolito *Onboarding*, por causa do aumento da complexidade Conversão dos dados, que era feita pelo Marketplace, afeta velocidade

Tabela 5.5: *Tabela de vantagens e desvantagens da arquitetura nova, de acordo com os respondentes.*

Mais uma vez, há mais citações de vantagens do que desvantagens. Há alguns pontos que aparecem nos dois lados, mas são novamente complementares: o uso de uma arquitetura com popularidade emergente atrai desenvolvedores e o *onboarding* é simplificado pela apresentação de serviços pequenos. Por outro lado, a complexidade de um sistema distribuído torna o *onboarding* mais longo. Além disso, a velocidade de implementação aumenta, apesar de haver mais código a ser implementado.

5.2. Dentro do seu conhecimento sobre a arquitetura de *micro frontends*, você acha que a nova arquitetura implementada pode ser considerada *micro frontends*?	
Sim	37, 5%
Não	50, 0%
Não sei opinar	12, 5%

Tabela 5.6: *Resultados da questão 5.2.*

É curioso que metade das pessoas não compreende que a nova arquitetura possa ser considerada *micro frontends*. Conclui-se que há dois fatores primários para isso: a definição de *micro frontends* na literatura e a falta de menções à abordagem vertical.

As definições encontradas induzem o leitor a entender que é obrigatório dividir uma única página em múltiplos componentes para ser um *micro frontend*, o que não é verdade. Assim como em microsserviços, não há uma definição estrita do que deve ser um serviço.

Logo, é possível que existam *micro frontends* que sequer entregam interfaces. Por exemplo, um *micro frontend* para autenticação de usuários.

Se há dúvida que o exemplo anterior é, na realidade, um microsserviço, lembre-se: um *micro frontend* também é um microsserviço. De fato, nem todo microsserviço é um *micro frontend*, mas se o projeto é utilizado exclusivamente no contexto de *frontend* e é governado por um time *frontend*, é natural que seja visto como um *micro frontend*.

Esses fatos aliados à falta de material sobre a abordagem vertical, do *Micro Frontends Decision Framework*, contribuem para a falta de flexibilidade quanto ao entendimento da arquitetura.

Tecnologias novas, arquitetura mantida	
Impactos se manteriam totalmente	62, 5%
Se manteriam parcialmente	37, 5%
Não se manteriam	0, 0%

(a) *Respostas relacionadas à questão 5.3.*

Arquitetura nova, tecnologias mantidas	
Impactos se manteriam totalmente	0, 0%
Se manteriam parcialmente	50, 0%
Não se manteriam	50, 0%

(b) *Respostas relacionadas à questão 5.4.*

Tabela 5.7: *Categorização das respostas das questões 5.3. e 5.4.*

Finalmente, estudamos os resultados para a seguinte pergunta: caso as novas tecnologias fossem adotadas sem alterar a arquitetura, os impactos citados em outras respostas se manteriam? E no caso em que a arquitetura fosse alterada, mas as tecnologias mantidas? É possível verificar que a resposta para a primeira pergunta divide opiniões entre sim e parcialmente, enquanto a segunda se mantém igual entre parcialmente e não.

Assim, interpreta-se que o uso de novas tecnologias é visto como indispensável pelos respondentes. Como foi dito anteriormente, é esperado que haja um viés positivo quanto às tecnologias por parte dos desenvolvedores, dado que seus impactos são mais palpáveis.

Por outro lado, a nova arquitetura pode ser vista como um benefício secundário ou bônus. Por este motivo, conclui-se que apesar da implementação bem sucedida de *micro frontends*, essa não era a única alternativa arquitetural. Um sistema *frontend* monolítico, por exemplo, funcionaria, desde que fosse possível desacoplar as interfaces do monolito e utilizar tecnologias modernas. Outro ponto que fortalece esse argumento é o fato de que apenas uma pessoa mencionou escalabilidade, um dos principais benefícios de microsserviços, e consequentemente de *micro frontends*.

Capítulo 6

Conclusão

Este Trabalho de Conclusão de Curso tinha como objetivo entender quando vale a pena adotar *micro frontends* na indústria. Para isso, foram traçados três objetivos secundários. O primeiro era entender o estado da arte da arquitetura de *micro frontends* através do estudo da literatura acadêmica, como artigos e livros, e da literatura cinza, como publicações em blogs e gravações de palestras. O segundo era implementar *micro frontends* no Elo7, plataforma *marketplace* de produtos artesanais. Por fim, o último objetivo era avaliar a implementação através de pesquisas com os desenvolvedores da empresa.

A literatura sobre *micro frontends* ainda é insuficiente. Faltam referências ao conceito de *micro frontends* verticais — ou baseados em domínios — apresentado por Mezzalira (2020). Essa abordagem é simples e facilita a implementação da arquitetura dado que problemas como composição, roteamento e comunicação se tornam triviais.

Outro problema é a falta de diversidade e flexibilidade na definição de um *micro frontend*. A maioria das definições estudadas atrelam o uso da arquitetura à quebra de uma página em componentes, o que causou confusão aos times do Elo7. Apesar de ter adotado a abordagem vertical — antes mesmo de conhecer o conceito — os desenvolvedores não consideraram a nova arquitetura como *micro frontends* pela falta de composição, foco de literaturas como a de Geers (2020). Assim, é necessário aproximar a definição da arquitetura com a de seu predecessor, microsserviços, além de documentar mais possibilidades, como a criação de *micro frontends* sem interfaces.

Quanto à implementação feita pela empresa, todos os pré-requisitos foram atendidos e a adoção da nova arquitetura foi um sucesso. O padrão *API Gateway* permitiu que outros sistemas funcionassem como porta de entrada, dividindo as responsabilidades de *frontend* do monolito e resolvendo o problema de acoplamento. O padrão BFF possibilitou a comunicação com serviços de *backend*. Por fim, as tecnologias Svelte e Fastify melhoraram significativamente a experiência de desenvolvimento.

Apesar de bem sucedida, a escolha por *micro frontends* não era mandatória para o Elo7. Seria possível atingir resultados equiparáveis adotando outras arquiteturas, como um *frontend* monolítico, por exemplo. O fato da empresa estar passando pelo processo de estrangulamento do monolito e adoção de microsserviços foi o que facilitou a implementação da arquitetura, por meio da reutilização da infraestrutura e do conhecimento

compartilhado entre times. Logo, a escolha por *micro frontends* não era a única solução possível, mas era a mais conveniente no dado contexto.

Infelizmente, o livro *Building Micro Frontends*, escrito por Mezzalira (2021), um dos principais autores sobre o assunto e o mais relevante para este trabalho, foi lançado apenas na segunda quinzena de novembro de 2021. Não houve tempo hábil para adquirir e estudar esse livro, o qual deve suprir vários dos tópicos mencionados neste capítulo. Sendo assim, esse pode ser o ponto de partida para uma eventual sequência do trabalho.

Espera-se que este TCC possa ser útil à qualquer um que esteja estudando ou implementando *micro frontends*, por conter: teoria e prática, um histórico da implementação no contexto de *ecommerce* e uma visão crítica sobre a literatura. Para o Elo7, espera-se que o desenvolvimento deste trabalho tenha ajudado na discussões e decisões arquiteturais, e que sirva como documentação para desenvolvedores atuais e futuros.

Apêndice A

Questionário sobre impactos trazidos pelas mudanças arquiteturais no Elo7

O texto abaixo é uma transcrição do questionário, criado utilizando Google Forms[1], enviado aos colaboradores.

Este questionário faz parte do meu Trabalho de Conclusão de Curso, que tem como tema entender o estado da arte da arquitetura de *micro frontends*, assim como analisar, avaliar e contribuir para sua implementação feita no Elo7.

O TCC está sendo executado sob a orientação do professor Alfredo Goldman do IME-USP. Também fazem parte deste trabalho como coorientadores: Luiz Fernando O. Corte Real, líder de um time de *frontend* no Elo7 e mestre em Ciência da Computação pelo IME-USP, e Renato Cordeiro Ferreira, líder do time de *machine learning engineering* no Elo7 e doutorando em Ciência da Computação pelo IME-USP.

O objetivo do questionário é:

- embasar e direcionar conclusões sobre o uso de *micro frontends* no Elo7;
- entender os impactos trazidos pela arquitetura para o dia a dia dos desenvolvedores;
- ajudar nas decisões arquiteturais, trazendo pontos levantados na literatura;
- disseminar esse conhecimento para os times de *frontend* e resto da engenharia.

As perguntas estão divididas em cinco seções. Procure ler cada seção integralmente antes de começar a responder. Note que todas as perguntas são obrigatórias.

É importante ressaltar que todos os dados coletados pelo questionário serão usados exclusivamente para embasar o TCC em questão, sem a publicação de dados sensíveis. Além disso, todos os participantes serão anonimizados.

Ao prosseguir, você concorda com essas condições.

[1] https://docs.google.com/forms/

1. Sobre sua experiência e papeis

Essa seção procura entender melhor sua experiência profissional e as atividades realizadas no dia a dia de trabalho.

- **1.1** Qual seu nome?

  O nome será usado para identificar o entrevistado e contatá-lo em caso de dúvidas. Não será usado ou divulgado na monografia.

- **1.2** Há quanto tempo trabalha na área de tecnologia/engenharia (tempo total, em anos)?

- **1.3** Qual seu cargo atual no Elo7?

- **1.4** Há quanto tempo trabalha no Elo7 (em meses ou anos)?

- **1.5** Quais atividades você desempenha durante o dia a dia de trabalho?

- **1.6** Qual seu nível de envolvimento com o desenvolvimento *frontend*?

	1	2	3	4	5	
Nenhum						Alto

- **1.7** Qual seu nível de envolvimento com as decisões arquiteturais do *frontend*?

	1	2	3	4	5	
Nenhum						Alto

2. Sobre o desenvolvimento anterior às mudanças arquiteturais de *frontend*

O Projeto Farewell foi criado com o objetivo de tornar o Marketplace obsoleto através do *Strangler Fig Pattern*. Assim, surgiram diversos serviços, como os Waffles, a fim de assumir responsabilidades do monolito.

Antes do Projeto Farewell e da criação dos Waffles, todo o *frontend* era servido pelo Marketplace - monolito em Java, com JSP - com ajuda do Water Gardens (ou Water7) - projeto NodeJS que usa a biblioteca DustJS.

Nesta parte, gostaríamos de entender o contexto e analisar suas EXPECTATIVAS em relação às mudanças ocorridas no desenvolvimento *frontend*.

Entenda por "arquitetura antiga" o uso do Marketplace junto com Water Gardens para o desenvolvimento *frontend*.

- **2.1** Quais eram os pontos positivos da arquitetura antiga?

  Cite o que funcionava bem com o uso do Marketplace com Water Gardens.

- **2.2** Quais eram os problemas enfrentados na arquitetura antiga?

  Cite os problemas relacionados ao uso do Marketplace com Water Gardens.

- **2.3** Dentre os pontos abordados em 2.2, quais você acha que podem ser resolvidas pelo uso de novas tecnologias?
- **2.4** Dentre os pontos abordados em 2.2, quais você acha que podem ser resolvidas pela adoção da nova arquitetura?

3. Sobre os efeitos trazidos pelo uso de novas tecnologias

Como parte do Projeto Farewell, foram criados projetos baseados na nova arquitetura de frontend, denominada Waffle. Dentre esses projetos, fazem parte: *waffle-discovery*, *waffle-enquete7*, *waffle-buying* e o *atelier-front*.

Uma das características desses projetos é o uso de novas tecnologias, como Svelte, Fastify e Storybook.

Nesta seção, gostaríamos de analisar suas OBSERVAÇÕES em relação aos efeitos que o uso de novas tecnologias trouxe para o dia a dia de desenvolvimento *frontend*.

- **3.1** Como você classifica o nível de impacto trazido pelas novas tecnologias sobre os seguintes aspectos?

	Não sei opinar	Muito negativo	Negativo	Não impactou	Positivo	Muito positivo
Entendimento do projeto						
Desenvolvimento entre múltiplos times						
Velocidade de desenvolvimento						
Implementação de novas funcionalidades						
Testabilidade						
Deploy						
Contratação e onboarding						

- **3.2** Comente os pontos que julgar relevantes sobre suas respostas da questão anterior.

4. Sobre os efeitos trazidos pela nova arquitetura

Além do uso de novas tecnologias, também foi implementada uma nova arquitetura. Assim, houve diversas alterações na relação entre *front* e *backend*.

Dentre as alterações, fazem parte:

- criação de um *gateway* (Nightfort);
- extração de funcionalidades do Marketplace para microsserviços independentes;
- *frontend* passa a ser servido, parcialmente, pelos projetos Waffle;
- cada Waffle faz as *requests* necessárias para os serviços de *backend* e monta as páginas adequadamente;
- Marketplace continua servindo as outras páginas.

Nesta seção, gostaríamos de analisar suas OBSERVAÇÕES em relação aos efeitos que a nova arquitetura trouxe para o dia a dia de desenvolvimento *frontend*.

Entenda por "nova arquitetura" todas as mudanças citadas acima.

- **4.1** Como você classifica o nível de impacto trazido pela nova arquitetura sobre os seguintes aspectos?

	Não sei opinar	Muito negativo	Negativo	Não impactou	Positivo	Muito positivo
Entendimento do projeto						
Desenvolvimento entre múltiplos times						
Velocidade de desenvolvimento						
Implementação de novas funcionalidades						
Testabilidade						
Deploy						
Contratação e onboarding						

- **4.2** Comente os pontos que julgar relevantes sobre suas respostas da questão anterior.

  Procure explicar suas respostas: por que impactou muito negativamente? Ou quais foram os impactos positivos?

5. Sobre a compreensão da nova arquitetura

- **5.1** Para você, o que é um *micro frontend*?
- **5.2** Dentro do seu conhecimento sobre a arquitetura de *micro frontends*, você acha que a nova arquitetura implementada pode ser considerada *micro frontends*?

- **5.3** Se as novas tecnologias fossem implementadas sem alterar a arquitetura, você acha que os impactos citados se manteriam? Por quê?

  Ou seja, se fosse criado um monolito *frontend* usando Svelte e Fastify, por exemplo.

- **5.4** Se a nova arquitetura fosse implementada sem o uso de novas tecnologias, você acha que os impactos citados se manteriam? Por quê?

  Ou seja, se a nova arquitetura fosse implementada mantendo o uso de Java com JSP ou NodeJS com DustJS.

Encerramento do questionário

Muito obrigado por responder!

As análises das respostas serão enviadas para você assim que estiverem prontas.

Referências

[Bass *et al.* 2013] Len Bass, Paul Clements e Rick Kazman. *Software Architecture in Practice*. 3a. Addison Wesley, 2013 (citado na pg. 3).

[Brooks 1995] Frederick Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, 1995 (citado na pg. 4).

[Calçado 2015] Phil Calçado. *The Back-end for Front-end Pattern (BFF)*. https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html. Último acesso em 03/11/21. 2015 (citado na pg. 13).

[Conway 1968] M. E. Conway. "How do committees invent". Em: *Datamation* 14.4 (1968), pgs. 28–31 (citado na pg. 23).

[Foote e Yoder 1997] B. Foote e J. Yoder. "Big ball of mud". Em: 1997 (citado na pg. 3).

[Fowler 2004] Martin Fowler. *StranglerFigApplication*. https://martinfowler.com/bliki/StranglerFigApplication.html. Último acesso em 14/08/2021. 2004 (citado na pg. 7).

[Geers 2017] Michael Geers. *Micro Frontends*. https://micro-frontends.org/. Último acesso em 21/06/2021. 2017 (citado na pg. 1).

[Geers 2020] Michael Geers. *Micro Frontends in Action*. 1a. Manning, 2020 (citado nas pgs. 8, 37).

[Hacq 2018] Audrey Hacq. *Everything you need to know about Design Systems*. https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969. Último acesso em 23/12/2021. 2018 (citado na pg. 20).

[Jackson 2019] Cam Jackson. *Micro Frontends*. https://martinfowler.com/articles/micro-frontends.html. Último acesso em 21/06/2021. Jun. de 2019 (citado na pg. 8).

[Lewis e Fowler 2014] James Lewis e Martin Fowler. *Microservices*. https://www.martinfowler.com/articles/microservices.html. Último acesso em 21/06/2021. Mar. de 2014 (citado nas pgs. 1, 6).

[Martin 2019] Robert C. Martin. *Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software*. Alta Books Editora, 2019 (citado na pg. 3).

[Mendonça *et al.* 2021] Nabor C. Mendonça, Craig Box, Costin Manolache e Louis Ryan. "The monolith strikes back: why istio migrated from microservices to a monolithic architecture". Em: *IEEE Software* 38.5 (2021), pgs. 17–22. doi: 10.1109/MS.2021.3080335 (citado na pg. 23).

[Mezzalira 2018] Luca Mezzalira. *Front-End Reactive Architectures*. Apress, 2018 (citado na pg. 17).

[Mezzalira 2020] Luca Mezzalira. *Lessons from DAZN: Scaling Your Project with Micro-Frontends*. https://www.infoq.com/presentations/dazn-microfrontend/. Último acesso em 29/09/2021. 2020 (citado nas pgs. 10, 37).

[Mezzalira 2021] Luca Mezzalira. *Building Micro-Frontends*. O'Reilley Media, Inc., 2021 (citado na pg. 38).

[Peltonen *et al.* 2021] Severi Peltonen, Luca Mezzalira e Davide Taibi. "Motivations, benefits, and issues for adopting micro-frontends: a multivocal literature review". Em: *Information and Software Technology* 136 (2021), pg. 106571. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106571. url: https://www.sciencedirect.com/science/article/pii/S0950584921000549 (citado nas pgs. 1, 8).

[Richards 2015] Mark Richards. *Software Architecture Patterns*. 1a. O'Reilley Media, Inc., 2015 (citado na pg. 6).

[Richardson 2018a] Chris Richardson. *Microservices Patterns*. 1a. Manning, 2018 (citado nas pgs. 4, 7, 12).

[Richardson 2018b] Chris Richardson. *Pattern: API Gateway / Backends for Frontends*. https://microservices.io/patterns/apigateway.html. Último acesso em 27/10/2021. 2018 (citado na pg. 13).

[Richardson 2019] Chris Richardson. *Pattern: Monolithic Architecture*. https://microservices.io/patterns/monolithic.html. Último acesso em 23/12/2021. 2019 (citado na pg. 4).

[Richardson 2020] Chris Richardson. *Who is using microservices*. https://microservices.io/articles/whoisusingmicroservices.html. Último acesso em 30/06/2021. 2020 (citado na pg. 1).

[Yang *et al.* 2019] Caifang Yang, Chuanchang Liu e Zhiyuan Su. "Research and application of micro frontends". Em: *IOP Conference Series: Materials Science and Engineering* 490 (abr. de 2019), pg. 062082. doi: 10.1088/1757-899x/490/6/062082. url: https://doi.org/10.1088/1757-899x/490/6/062082 (citado na pg. 8).