UNIVERSIDADE FEEVALE

JONAS RAFAEL COLLING

ARQUITETURA MONOLÍTICA PARA MICROSSERVIÇOS

Novo Hamburgo
2022

JONAS RAFAEL COLLING

ARQUITETURA MONOLÍTICA PARA MICROSSERVIÇOS

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Juliano Varella de Carvalho

Novo Hamburgo
2022

RESUMO

Esse trabalho apresenta uma proposta de transformação de módulos de uma plataforma de *e-commerces* de arquitetura monolítica em microsserviços. A empresa detentora da plataforma lançou as vendas *online* no ano de 2014 para apenas uma de suas marcas. Com o sucesso nas vendas *online*, publicou outros *e-commerces* para o restante de suas marcas, todos eles na mesma plataforma. Atualmente a plataforma abriga oito *e-commerces* e três aplicativos. Para a expansão do produto foi necessário um aumento abrupto da equipe técnica, trazendo consigo alguns problemas que antes não existiam. Problemas como alto acoplamento de funcionalidades e grande dependência entre diferentes *squads*. A metodologia utilizada para desenvolvimento deste projeto foi a *Design Science Research* (DSR). A proposta de transformação está baseada na literatura e em trabalhos relacionados a este assunto, encontrados por meio da utilização de alguns métodos de revisão sistemática. Foi construído o planejamento de toda a arquitetura de microsserviços para substituir a atual plataforma, com a apresentação de um plano para execução da transformação. Junto a isso, um módulo da plataforma foi escolhido e implementado para expor as dificuldades a se enfrentar neste tipo de transformação. O objetivo da transformação é que a nova arquitetura resolva problemas críticos que se encontram nesse tipo de aplicação monolítica de alta complexidade. A nova arquitetura deve permitir que a plataforma siga em crescimento, com entregas em prazo menor de tempo e com maior qualidade.

Palavras-chave: Microsserviços. Arquitetura de software. Transformação de arquitetura. Evolução de sistemas legados. Arquitetura monolítica.

ABSTRACT

This work presents a proposal for transforming modules of an e-commerce platform of monolithic architecture into microservices. The company that owns the platform launched online sales in 2014 for only one of its brands. With the success in online sales, it published other e-commerces for the rest of its brands, all of them on the same platform. Currently, the platform houses eight e-commerces and three applications. For the expansion of the product, an abrupt increase in the technical team was necessary, increasing some problems that did not exist before. Issues such as high functionality of features and large dependencies between different squads. The methodology used for the development of this project was Design Science Research (DSR). The transformation proposal is based on the literature and on related works to this subject, found through the use of some systematic review methods. A plan for the entire microservices architecture was built to replace the current platform, with the presentation of a plan for executing the transformation. In addition, a module of the platform was chosen and implemented to export the difficulties to face in this type of transformation. The goal of the transformation is that the new architecture solves critical problems that are found in this type of monolithic application of high complexity. The new architecture should allow the platform to continue growing, with deliveries in a shorter period of time and with higher quality.

Keywords: Microservices. Software architecture. Architectural transformation. Evolutions of legacy systems. Monolithic architecture.

LISTA DE FIGURAS

Figura 1 – *Design* do monolito. 13
Figura 2 – *Design* microsserviço 15
Figura 3 – Processo de filtragem 20
Figura 4 – Organização da estrutura da plataforma 32
Figura 5 – Infraestrutura ambiente produtivo 33
Figura 6 – Detalhamento dos módulos da plataforma 39
Figura 7 – Modelagem estratégica DDD 45
Figura 8 – *Context maps* 46
Figura 9 – Estrutura de microsserviços 47
Figura 10 – Tecnologias da camada *frontend* 51
Figura 11 – Tecnologias *backend* 52
Figura 12 – Respostas para decidir sobre a estratégia de transformação dos módulos 54
Figura 13 – Camada atual *marketplace* 59
Figura 14 – Detalhamento de módulos sem *marketplace* 62
Figura 15 – Desenho arquitetura microsserviços *marketplace* 64
Figura 16 – Estrutura completa, *marketplace* e plataforma 66

LISTA DE TABELAS

Tabela 1 – Arquitetura microsserviços x monolítica 18

Tabela 2 – Termos de pesquisa 19

Tabela 3 – Diretriz de pontuação para serviços 24

Tabela 4 – Compilado dos artigos selecionados 30

Tabela 5 – Glossário dos domínios da plataforma 43

Tabela 6 – Detalhamento da estimativa do módulo *marketplace* 57

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AWS	Amazon *Web Services*
HTTP	*Hypertex Transfer Protocol*
JSON	*JavaScript Object Notation*
DDD	*Domain-Driven Design*
CI	Integração Contínua
CD	Entrega Contínua
CPU	*Central Process Unit*
DTO	*Data Transfer Object*
ERP	*Enterprise Resource Planning*
RAM	*Random Access Memory*
JSP	*Java Server Pages*
HTML	*HyperText Markup Language*
SASS	*Syntactically Awesome Style Sheets*
CSS	*Cascading Style Sheets*
REST	*Representational State Transfer*
SAC	Serviço de Atendimento ao Consumidor
SOAP	*Simple Object Access Protocol*
UI	*User Interface*
UX	*User Experience*
XML	*Extensible Markup Language*
JMS	*Java Message Service*
WSDL	*Web Service Description Language*

SUMÁRIO

1 INTRODUÇÃO 9
2 ARQUITETURAS MONOLÍTICA E MICROSSERVIÇOS 12
2.1 Características arquitetura monolítica 12
2.2 Características arquitetura microsserviços 14
2.3 Arquitetura microsserviços x monolítica 16
3 TRABALHOS RELACIONADOS 19
3.1 Os artigos selecionados 20
3.1.1 *From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture* (Gouigoux; Tamzalit, 2017) 21
3.1.2 *Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System* (Mi; Ma; Lu, 2020) 23
3.1.3 *Monoliths to microservices - Migration Problems and Challenges: A SMS* (Valepucha; Flores, 2021) 25
3.1.4 *Microservices migration patterns* (Balalaie; Heydarnoori; Jamshidi et al. 2018) 27
3.2 Compilado dos artigos selecionados 29
4 ESTRUTURA ATUAL DA PLATAFORMA DE *E-COMMERCES* 31
4.1 Arquitetura 31
4.2 Tecnologias em uso 33
4.2.1 Backend 33
4.2.2 Frontend 34
4.3 Módulos principais existentes 34
4.3.1 *Core* da aplicação 34
4.3.2 Lojas (sites *frontend*) 34
4.3.3 Web service 35
4.3.5 Integração *marketplace in* 36
4.3.6 Trocas e devoluções 36
4.3.7 Integração de canais 36
4.3.8 Sistema de recomendação de produtos 37
4.3.9 Marketing digital 37
4.3.10 *Gateway* de pagamento 37
4.3.11 Cotações de frete 38
4.3.12 ERP (*Enterprise Resource Planning*) 38
4.4 Modelo de trabalho 39
4.5 Principais problemas 41
5 CONSTRUÇÃO ARQUITETURAL DA PLATAFORMA DE *E-COMMERCES* 43
5.1 Definição de microsserviços com DDD 43
5.2 Proposta da estrutura de microsserviços 46
5.2.1 *Frontend web* sites 47
5.2.2 *Store Sales dashboard* 47
5.2.3 *Panel* CMS 48
5.2.4 *Panel backoffice* 48
5.2.5 API *e-commerce* 48
5.2.6 API CMS 48
5.2.7 API *Store Sales* 48

5.2.8 API *backoffice* 48
5.2.9 API *marketplace* 48
5.2.10 *Marketplace in* 49
5.2.11 *Marketplace out* 49
5.2.12 *Order* 49
5.2.13 *Customer* 49
5.2.14 *Product* 49
5.2.15 *Jobs* 49
5.2.16 *Payment gateway* 49
5.2.17 *Search engine* 50
5.2.18 *Notification* 50
5.2.19 *Quotation freight* 50
5.2.20 ERP 50
5.2.21 Product recommendation 50
5.3 Principais tecnologias selecionadas para a transformação 51
6 ESTRATÉGIA PARA TRANSFORMAÇÃO DA PLATAFORMA 54
6.1 Ordem de implementação 54
6.2 Módulo selecionado para implementação 56
6.3 Implementação do módulo de gerenciamento de *marketplace* 58
6.3.1 Protocolo de comunicação entre microsserviço e monolito 60
6.3.2 Adaptações na plataforma de *e-commerces* 60
6.3.3 Integração por mensageria 61
6.3.4 Implementação dos microsserviços para gestão de *marketplace* 62
7 CONSIDERAÇÕES FINAIS 67

1 INTRODUÇÃO

Com a evolução do mercado e das pessoas, as necessidades sistêmicas são modificadas a cada dia, os sistemas precisam ser construídos com mais rapidez e se tornam cada vez maiores e mais complexos. Para a construção dos sistemas, a engenharia de software precisa desenvolver novas técnicas para enfrentar os desafios e fornecer o software mais completo (SOMMERVILLE, 2019).

Definir a arquitetura do software é um dos itens de maior importância no planejamento de um sistema. Na sua evolução eles podem rapidamente se tornar muito complexos e caros na manutenção (SOMMERVILLE, 2019).

A arquitetura monolítica foi uma abordagem tradicional no desenvolvimento dos sistemas, usado em grande escala em empresas como Amazon e Ebay (LAURETIS, 2019, tradução nossa). Segundo SARITA; SEBASTIAN (2017, tradução nossa) a arquitetura monolítica é uma maneira fácil e comum de desenvolver e implantar um aplicativo de software, como uma unidade única de código, tendo todas funcionalidades reunidas. Essa vantagem normalmente é válida para pequenas ou médias aplicações, onde existe um pequeno time de tecnologia durante sua evolução.

Os sistemas legados monolitos normalmente estão sempre crescendo em tamanho e complexidade, tornando sistemas monstruosos após algum tempo de desenvolvimento e as desvantagens desta arquitetura acabam sendo maiores que as vantagens (KAZANAVICIUS; MAZEIKA, 2019, tradução nossa).

De acordo com SARITA; SEBASTIAN (2017, tradução nossa), aplicações monolíticas têm grandes limitações:

> - O forte acoplamento entre os módulos impedirá um aplicativo de estar pronto para SaaS (Software como Serviço), onde houver a necessidade de publicar as mudanças várias vezes ao dia. Atualizar um único módulo garante a necessidade de construir e implantar todo o aplicativo novamente;
> - Difícil de alcançar escalabilidade para recursos conflitantes para diferentes módulos dentro de um aplicativo;
> - Difícil de adotar novas linguagens e estruturas;
> - Entrega e *deploy* contínuo.

Por outro lado, a arquitetura em microsserviços procura resolver alguns problemas e limitações da arquitetura monolito (PRASANDY *et al.*, 2020, tradução nossa). Nessa abordagem, o software consiste em pequenos serviços independentes, que se comunicam usando APIs (*Application Programming Interface*) bem definidas. Como são executados de

forma independente, cada serviço pode ser atualizado, implantado e escalado para atender a demanda de funções específicas de uma aplicação.

Essa forma de desenvolvimento em serviços independentes traz alguns benefícios para a manutenção dos sistemas, como por exemplo: agilidade, pois os microsserviços podem ser atualizados de forma independente, sem a necessidade de reiniciar o aplicativo completamente; a falha de um serviço afeta apenas aquele módulo e seus consumidores; cada microsserviço pode ser escalável de forma separada, de acordo com a necessidade da aplicação (KAZANAVICIUS; MAZEIKA, 2019, tradução nossa).

De acordo com KAZANAVICIUS; MAZEIKA (2019, tradução nossa), a transformação do sistema monolito deveria ser modernizado para microsserviços quando:

- Monolito torna-se muito grande ou complexo para manter ou ampliar;
- Modularidade e descentralização são importantes aspectos;
- Preferência por benefícios de longo prazo em comparação aos de curto prazo.

A plataforma de *e-commerces* a ser utilizado na transformação utiliza arquitetura monolito. O sistema foi implementado a partir de uma plataforma de *e-commerce* de mercado. Plataforma de comércio eletrônico, que contempla venda de produtos para pessoas físicas e empresas. Sem customizações, a plataforma já entrega serviços para gerir o *e-commerce*, porém normalmente cada empresa tem suas peculiaridades no modelo de negócio. Pensando nisso, é possível customizar todas funcionalidades, ou até mesmo a adição de novas.

O sistema foi lançado no ano de 2014. Naquele momento apenas um *e-commerce* era hospedado nesta aplicação e uma equipe composta de oito pessoas atuava na evolução dele. Atualmente, existem oito *e-commerces* e três aplicativos *mobile* rodando nesta estrutura. Para sustentar este ambiente tecnológico, mais de sessenta pessoas estão distribuídas em oito *squads*.

O projeto cresce desde o lançamento. Funcionalidades foram adicionadas e customizadas, integrações com plataformas externas foram criadas, *APIs* desenvolvidas e o aumento de complexidade é cada vez maior.

Mesmo sem customizações, os módulos desta plataforma estão altamente acoplados, com uma forte dependência entre eles. Esta característica gera uma dependência não só no sistema, mas também entre as *squads*. As modificações se tornam complicadas, pois o desenvolvedor não impacta apenas a funcionalidade de sua responsabilidade, as demais *squads* têm grandes chances de serem afetadas. Esse nível de dependência somado a toda complexidade

do sistema dificulta a entrega de software contínuo, tornando a implantação e desenvolvimento lentos e dolorosos (SARITA; SEBASTIAN, 2017, tradução nossa).

Este trabalho propõe a transformação de módulos da plataforma de *e-commerces* de arquitetura monolítica para uma arquitetura moderna em microsserviços. O trabalho, através da utilização de boas práticas da arquitetura microsserviço, tem por objetivo indicar melhores maneiras na realização desse tipo de transformação, levando em conta que a evolução do sistema atual não para durante a execução. Além disso, também se direciona para a resposta da seguinte questão de pesquisa: Com a aplicação da nova arquitetura, é possível diminuir a complexidade e o acoplamento entre *squads* nas entregas de novas funcionalidades no *e-commerce* já existente?

Este trabalho está dividido em seis capítulos. O primeiro é esta introdução. No segundo capítulo é abordada a discussão da literatura sobre características de arquiteturas monolíticas e microsserviços. No capítulo três são apresentados os trabalhos relacionados ao assunto, apresentando abordagens sobre como executar esse tipo de trabalho. No quarto capítulo é descrita a atual organização do sistema em que este trabalho atua, detalhando a arquitetura, módulos, formato de trabalho e principais problemas. No capítulo seguinte, a nova camada arquitetural é apresentada, onde acontece a quebra de microsserviços e a definição das tecnologias macro. No sexto capítulo é exposta a estratégia para a transformação; é construído o plano de entregas por fases, seguindo uma ordem de implementação; e, além disso, é descrito qual módulo foi escolhido para implementação, juntamente com o detalhamento da fase de desenvolvimento. Por fim, o último capítulo expõe as considerações finais do trabalho.

2 ARQUITETURAS MONOLÍTICA E MICROSSERVIÇOS

Existem vários tipos de sistemas de software, para diferentes necessidades, usuários e funcionalidades. Não faz sentido buscar métodos ou técnicas universais para a engenharia de software, pois cada software necessita de diferentes abordagens. Todas as aplicações necessitam da atuação da engenharia de software para projetar a arquitetura, planejar a evolução e atender as necessidades propostas (SOMMERVILLE, 2011).

O software segue em evolução com o passar do tempo, não importa seu tamanho ou complexidade. As mudanças em sua volta motivam esse processo. Alterações ocorrem em todo momento, com correções de erros, adaptações ao cliente e negócio, ou até mesmo quando o sistema necessita de reengenharia (PRESSMAN, 2016). A reengenharia é a reconstrução do software de uma forma diferente, a fim de adequar o sistema com o momento em questão (CHIKOFSKY, CROSS, 1990, tradução nossa). Segundo Pressman, a reengenharia se faz necessária quando o software apresenta problemas com muita frequência, pois leva mais tempo para ser reparado e já não representa a tecnologia mais atualizada. Isso se chama manutenção e ela se torna cada vez mais difícil à medida que os anos passam.

Atualmente, ao falar sobre sistemas modernos, geralmente se imagina a utilização de microsserviços. Além disso, há uma tendência de modernizar e migrar aplicações para arquitetura de microsserviços, pensando na sua evolução. Porém arquiteturas monolíticas e de microsserviços possuem suas vantagens e desvantagens, portanto, antes de definir o caminho a seguir, certas considerações devem ser feitas, principalmente, conhecer as características de cada uma delas (VELEPUCHA, FLORES, 2021).

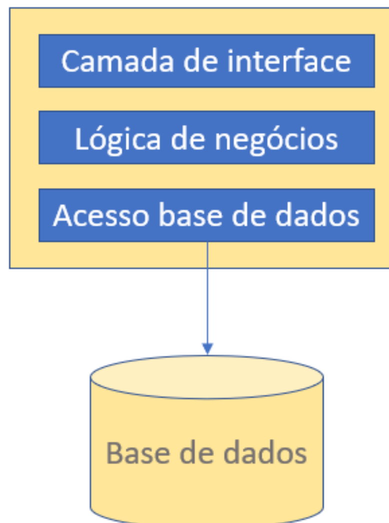
2.1 CARACTERÍSTICAS ARQUITETURA MONOLÍTICA

Segundo Chan (2017), a palavra “monolito” foi originalmente usada pelos gregos antigos para descrever um único bloco de pedra do tamanho de uma montanha. Embora a palavra seja usada de forma mais ampla hoje, a ideia permanece a mesma.

No ambiente de software, arquitetura monolítica é caracterizada por um sistema que possui apenas um arquivo como resultado (VILLAMIZAR, 2015). É uma aplicação com base única de código, que oferece diferentes serviços embutidos. Serviços que se comunicam entre sistemas externos e podem ser consumidos por diferentes interfaces (AL-DEBAGY; MARTINEK, 2018, tradução nossa).

O *design* desta abordagem pode ser representado pela Figura 1, onde as camadas de interface, lógica de negócios e acesso a base de dados ficam agrupadas em um único serviço.

Figura 1 – *Design* do monolito



Fonte: https://docs.microsoft.com/pt-br/

Este tipo de arquitetura é considerado um modo de implementação tradicional. Normalmente é utilizada como ponto de partida para o lançamento de um sistema, já que é o caminho mais rápido para a implementação (MICROSOFT, 2019a). Alguns benefícios para esse tipo de arquitetura são:

- Facilidade na criação. O código todo está contido em uma base única compartilhada, onde todas as camadas podem se encontrar;
- Simplicidade de depuração e localização de inconformidades. A execução do código fica em um único processo;
- Facilidade de avaliar alterações ou novas funcionalidades, em função da complexidade da arquitetura ser menor.

Na implantação desses tipos de sistemas, a aplicação inteira é empacotada em um único artefato (KURYAZOV; JABBOROV; KHUJAMURATOV, 2020, tradução nossa). Para a gestão de infraestrutura, a complexidade é consideravelmente baixa, já que o sistema tem apenas um artefato empacotado. Para escalar a aplicação, basta adicionar mais algumas instâncias deste artefato com um distribuidor de carga na frente (MICROSOFT, 2021b, tradução nossa). O lado negativo dessa abordagem é que ao escalar o sistema, a aplicação como um todo é escalada. Isso não gera um problema sistêmico, mas na maioria dos casos apenas

algumas partes dos sistemas têm a real necessidade de escalar, enquanto o resto segue com baixo consumo de recursos (MICROSOFT, 2021b, tradução nossa).

Uma característica relevante na utilização de uma arquitetura monolítica, é que os módulos do sistema ficam fortemente acoplados, com muita dependência entre si. O software normalmente se torna muito complexo conforme o crescimento, ocasionando uma alta dificuldade na adição de novos recursos (KURYAZOV; JABBOROV; KHUJAMURATOV, 2020, tradução nossa).

Apesar de ser um tipo de implementação mais simples de realizar e gerenciar, conforme o sistema cresce, as limitações se tornam visíveis. Kempf (2021) descreve uma lista das principais limitações:

- Baixa tolerância a falhas. Um erro em uma parte do sistema pode derrubar o aplicativo inteiro;
- Implantação demorada. Cada publicação nova requer uma recompilação completa;
- A utilização de novas linguagens e ferramentas é limitada ao que já está em uso;
- Caro para escalar. À medida que o número de acessos aumenta, o provisionamento de recursos será feito para todo aplicação e não apenas para as partes impactadas.

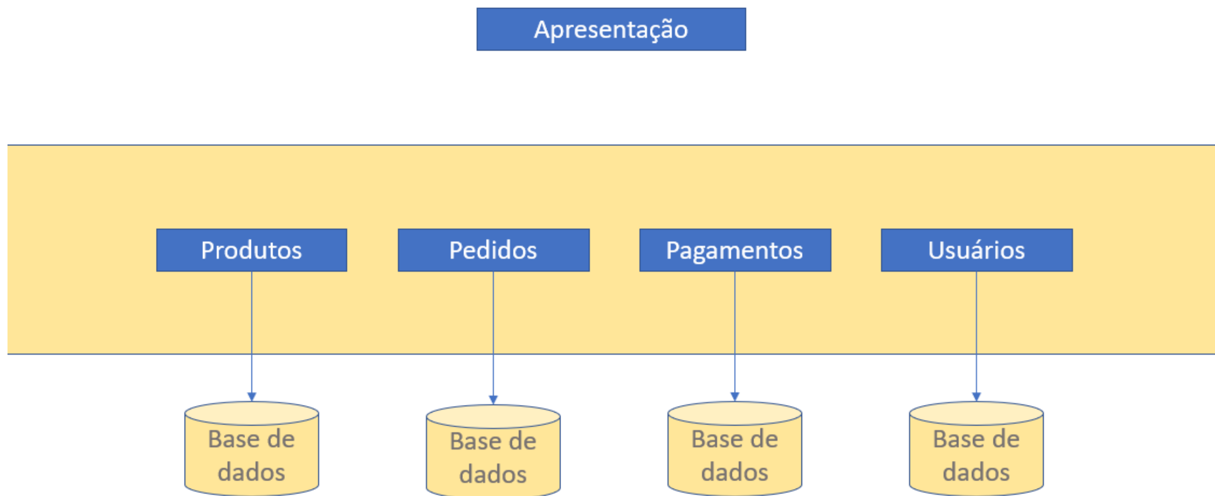
2.2 CARACTERÍSTICAS ARQUITETURA MICROSSERVIÇOS

A arquitetura em microsserviços é uma arquitetura moderna, que procura resolver alguns problemas e limitações da arquitetura monolito (PRASANDY *et al.*, 2020, tradução nossa). Este conceito de arquitetura se inseriu na comunidade de software por volta de 2014.

Microsserviços consiste em uma abordagem de arquitetura e organizacional do desenvolvimento de software, onde o sistema é composto por pequenos serviços independentes que se comunicam usando APIs bem definidas. Esses serviços são desenvolvidos por pequenas equipes autossuficientes (AWS, ANO).

O *design* da abordagem microsserviço pode ser representado pela Figura 2. Ela deixa clara a ideia de independência entre os módulos de um sistema, onde cada um funciona de forma isolada com sua própria base de dados.

Figura 2 – *Design* microsserviço.



Fonte: https://docs.microsoft.com/pt-br/

Com característica independente, o sistema que utiliza esse tipo de arquitetura pode ter seus serviços atualizados e implantados sem que haja algum impacto para a aplicação. Além disso, o sistema é facilmente escalado com maior aproveitamento de recursos, concentrando o aumento de força de máquina nos pontos de gargalo do sistema (AWS, ANO).

Carnell (2021) descreve que a arquitetura em microsserviços tem as seguintes características:

- A lógica do aplicativo é dividida em pequenos componentes com limites de responsabilidades bem definidos que se coordenam para entregar uma solução;

- Cada componente tem um pequeno domínio de responsabilidade e é implantado de forma totalmente independente do outro. Os microsserviços devem ter responsabilidade para uma única parte de um domínio de negócios. Além disso, o microsserviço deve ser reutilizável em vários aplicativos;

- Os microsserviços se comunicam com base em alguns princípios básicos e empregam protocolos de comunicação leves, como HTTP (*Hypertex Transfer Protocol*) e JSON (*JavaScript Object Notation*) para troca de dados entre o consumidor e o provedor de serviços;

- A implementação técnica subjacente do serviço é irrelevante porque os aplicativos sempre se comunicam com um protocolo de tecnologia neutra (JSON é o mais

comum). Isso significa que um aplicativo nesta arquitetura pode utilizar diferentes linguagens de programação em cada microsserviço.

- Por natureza são pequenos, independentes e distribuídos. Permitem organizações que tenham pequenas equipes de desenvolvimento com áreas bem definidas de responsabilidade. Essas equipes podem trabalhar em direção a um único objetivo, como manter um aplicativo, mas cada equipe é responsável apenas pelos serviços em que eles estão trabalhando.

Embora os benefícios na utilização desse tipo de arquitetura sejam bem relevantes, o nível de complexidade e conhecimento técnico para gestão desses sistemas cresce consideravelmente. KAINZ (2020, tradução nossa) descreve algumas características:

- Duplicação de código: como os microsserviços precisam ser independentes uns dos outros, não há como evitar a repetição de métodos internos;
- Versões de bibliotecas: diferentes serviços podem utilizar a mesma biblioteca, porém normalmente utilizam versões diferentes. Isso acontece, pois, os lançamentos de novas versões de cada serviço acontecem em momentos diferentes;
- Execução e monitoramento: O monitoramento pode se tornar complicado. Uma única chamada pode atingir vários serviços. Eles podem estar em servidores diferentes ou até mesmo em localizações geográficas diferentes.

2.3 ARQUITETURA MICROSSERVIÇOS X MONOLÍTICA

Como já foi descrito, para se ter sucesso na implementação de um sistema, o tipo de arquitetura é um fator muito importante a ser analisado. A evolução dos sistemas pode rapidamente torná-los muito complexos e caros de manter (SOMMERVILLE, 2019). A evolução de software é a capacidade de alterar, de forma rápida e confiável, um sistema de software para adaptá-lo às mudanças do ambiente, aos novos usuários e às necessidades do mercado, bem como manter seu desempenho operacional (BENNETT; RAJLICH, 2000).

Antes do conceito de microsserviços evoluir, a maioria dos aplicativos foram construídos usando um estilo arquitetônico monolítico. (Carnell, 2021, tradução nossa). Tal característica é encontrada em grande parte dos sistemas legados disponíveis na atualidade. Em muitos casos a migração da arquitetura tem sido a estratégia das empresas para modernizar e atualizar as aplicações, estratégia que se denomina Reengenharia de Software. Um dos

principais desafios da Reengenharia de Software é garantir a equivalência do software nas novas versões (GRUBB; TAKANG, 2003).

A decisão a qual caminho seguir na definição da arquitetura de um sistema deve levar em conta vários fatores. Fatores sobre tecnologias disponíveis no mercado, informações sobre o produto a ser construído e detalhes da equipe que irá construir e evoluir o produto. De acordo com KAINZ (2020, tradução nossa), alguns itens importantes são listados para questionar em uma análise desse tipo:

- Latência: uma aplicação em monolito não sofre com latência de rede, já que todas requisições são locais. Para microsserviços quanto maior for a cadeia de chamadas para outros serviços, maior será a penalização em latência, pois os dados necessitam percorrer pela rede entre os serviços da aplicação;

- Complexidade do desenvolvimento: a independência dos serviços em arquiteturas de microsserviços pode ser um gerador de complexidade, já que vários códigos-fonte podem ser envolvidos, usando diferentes estruturas, em diferentes linguagens de programação;

- Complexidade da execução: o monitoramento de aplicações monolíticas é consideravelmente simples, já que normalmente os dados estarão concentrados em um arquivo apenas. Já em microsserviços, pode envolver a verificação de vários serviços. Para realizar o rastreio completo de uma requisição, é necessário deixar todos os arquivos de log na ordem de execução entre os serviços envolvidos. Para gerenciar as transações, os microsserviços também necessitam de cuidados, por exemplo, uma nova tentativa de chamada mal implementada pode executar um pagamento duas vezes;

- Utilização de recursos: Microsserviços permitem o uso de recursos de maneira mais inteligente. Em momentos de sobrecarga de acessos, normalmente os serviços afetados são poucos. Com isso, nesse tipo de estrutura, é possível escalar apenas os serviços que estão sofrendo. Para escalar o monolito é necessário subir novas instâncias da aplicação, gerando desperdício de recursos, já que funcionalidades não afetadas na sobrecarga terão recursos reservados na infraestrutura;

- Escalabilidade: escalar um monolito é possível. Pode ser por novas instâncias ou execução de vários threads. Para microsserviços isso também é possível, porém conforme destacado no item anterior, é feito com menos recursos. Escalar microsserviços é normalmente

mais rápido. No caso do número de acessos de um sistema crescer bruscamente, a chance de enfrentar indisponibilidade diminui;

- *Time to market*: é o tempo entre a definição de negócios de um recurso até que ele seja disponibilizado publicamente. Por questões de tamanho e acoplamento, os monolitos normalmente são mais difíceis de implantar. Por outro lado, implantar microsserviços com frequência ou continuamente é mais fácil. Por questões de dependência e responsabilidades bem definidas, eles são mais simples de testar e fáceis de localizar os pontos de alterações.

Em resumo ao que foi descrito nos tópicos acima, foi criada a Tabela 1, onde é indicado qual tipo de arquitetura é mais favorável, baseado em algumas características encontradas em aplicações de *software*.

Tabela 1 – Arquitetura microsserviços x monolítica.

Característica	**Microsserviços**	**Monolítica**
Latência	-	Favorável
Complexidade de desenvolvimento	-	Favorável
Complexidade de execução	-	Favorável
Utilização de recursos	Favorável	-
Escalabilidade	Favorável	-
Time to market	Favorável	-

3 TRABALHOS RELACIONADOS

Para realizar o levantamento bibliográfico do trabalho foram utilizados alguns métodos de uma revisão sistemática. O objetivo era reunir trabalhos semelhantes a este. Inicialmente foi elaborada uma questão base para conduzir as buscas. O questionamento formado foi: Quais estratégias são utilizadas para transformação de um sistema de arquitetura monolítica em microsserviços? A busca foi aplicada a partir de três plataformas referenciais de apoio em trabalhos científicos, *Web of Science*, ACM e IEEE Xplore. A Tabela 2 detalha os termos de pesquisa utilizados em cada plataforma:

Tabela 2 – Termos de pesquisa utilizados na revisão de literatura.

Base de dados	**Termo de pesquisa**
Web of Science	((AB="microservices" OR AB="monolith") AND (TI="transf" OR TI="migration"))
ACM	(Abstract:"microservices" OR Abstract:"monolith") AND (Title:"transf" OR Title:"migration")
IEEE Xplore	("Abstract":"microservices" OR "Abstract":"monolith") AND ("Document Title":"transf" OR "Document Title":"migration")

A partir dos resultados encontrados, foram aplicados alguns critérios de inclusão e exclusão de registros.

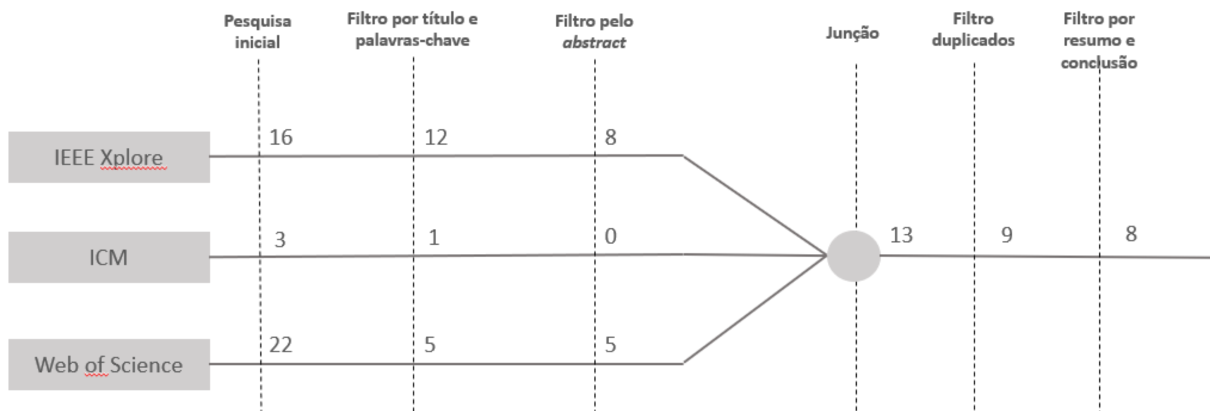
Critérios de inclusão e exclusão:

- Artigos publicados a partir do ano de 2016 (últimos 5 anos de pesquisa);
- Informação relacionada apenas com a área de engenharia de software;
- Apenas artigos;
- Apenas artigos escritos em inglês.

A pesquisa resultou em um total de quarenta e um (41) artigos a partir das três plataformas utilizadas. Foram feitos alguns procedimentos metodológicos (filtragem) com o intuito de buscar artigos mais próximos ao objetivo deste trabalho, onde o artigo expusesse as

estratégias, técnicas e métodos adotados na transformação, além dos desafios enfrentados. O primeiro critério foi a leitura do título e palavras-chave, que totalizou a remoção de vinte e três (23) artigos. Feito isso, a leitura do *abstract* foi realizada, removendo mais cinco (5) trabalhos. Nos treze (13) trabalhos restantes, havia quatro (4) duplicados, que foram removidos também. Após isso, a leitura da introdução e conclusão removeu mais um (1) artigo e, por fim, foi executada a leitura completa dos oito (8) artigos restantes.

Figura 3 – Processo de filtragem dos artigos selecionados.



Fonte: criação própria

A leitura completa dos artigos procurava a resposta para as perguntas abaixo. Trinta e três (33) trabalhos foram eliminados por não se adequarem com os questionamentos preestabelecidos.

- Qual foi o objetivo da migração da arquitetura do sistema ou da escrita do artigo?
- Para o entendimento da estratégia executada no trabalho, quais os métodos e técnicas indicados?
- A fim de mapear os riscos encontrados em um projeto deste porte, quais desafios técnicos são encontrados na transformação?
- Após a migração, quais os impactos obtidos no time técnico?
- Quais os impactos para a área de negócio?

3.1 OS ARTIGOS SELECIONADOS

Ao fim do processo de refinamento dos trabalhos relacionados, foram selecionados oito (8) artigos para estudo do processo de migração de software. O estudo resultou em um

conhecimento de estratégias, técnicas, dificuldades e benefícios que a comunidade tem enfrentado até o momento de escrita deste trabalho.

Os artigos *From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture* (Gouigoux; Tamzalit, 2017) e *Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System* (Mi; Ma; Lu, 2020) foram executados a partir da migração de sistemas existentes, a fim de relatar como foi o processo migratório, desde o planejamento até a fase de manutenção.

Os trabalhos *Monoliths to microservices - Migration Problems and Challenges: A SMS* (Valepucha; Flores, 2021) e *Microservices migration patterns* (Balalaie; Heydarnoori; Jamshidi et al. 2018) descreveram, baseados na literatura, problemas e soluções que devem ser considerados em projetos de migração de arquiteturas.

O trabalho *Microservices Migration in Industry: Intentions, Strategies, and Challenges* (Fritzsch; Bogner; Wagner et al. 2019) foi descrito com base em entrevistas com pessoas de papel chave em projetos migratórios. O trabalho reúne intenções, estratégias e desafios enfrentados por todos entrevistados.

O artigo *Microservices: Migration of a Mission Critical System* (Mazzara; Dragoni; Bucchiarone et al. 2018) não será aprofundado neste trabalho por não trazer informações construtivas para o propósito. Os autores realizaram um comparativo das duas arquiteturas de um sistema e não expuseram informações sobre o processo de migração.

O trabalho *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture* (Balalaie; Heydarnoori; Jamshidi) não será detalhado pelo fato de seu foco maior estar na camada de infraestrutura e não no software e no processo de transformação.

Outro artigo não detalhado será *"Functional-First" Recommendations for Beneficial Microservices Migration and Integration Lessons Learned from an Industrial Experience* (Goigoux; Tamzalit, 2019) que faz relatos do mesmo projeto de migração que o artigo *From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture* (Gouigoux; Tamzalit, 2017) e lista as mesmas dificuldades, estratégias e benefícios enfrentados.

3.1.1 *From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture* (Gouigoux; Tamzalit, 2017)

Os autores descreveram como foi a migração do principal sistema da empresa MGDIS SA. Ela é fornecedora de software na França, que trabalha em aplicativos voltados para coletividades públicas, contribuindo no gerenciamento do ciclo de vida, pagamento de subsídios financeiros e bolsas de estudos. As principais características desse mercado são a grande importância da interoperabilidade de sistemas com o ciclo de vida do produto, geralmente em torno de dez anos.

O principal objetivo da empresa na época, era reduzir o custo de interoperabilidade, além de permitir a melhora do sistema em longo prazo por meio de colaboração ou incorporação de recursos externos de software. A estratégia para alcançar este objetivo, era deixar os componentes do sistema autônomos e com o máximo de desacoplamento.

A tomada de decisão para definir os detalhes técnicos na transformação foram baseadas no equilíbrio entre os custos de garantia de qualidade e o custo de implantação. Com a granularidade dos serviços adequados, é simplificado o acompanhamento da qualidade, principalmente pela implementação de testes unitários e integrados, desde que sejam verdadeiramente independentes. O nível de granularidade dos serviços não pode ser levado ao nível mais fino possível, pois a complexidade para implantação aumenta conforme o sistema cresce na quantidade de serviços para execução. A premissa para obter o equilíbrio proposto foi o desenvolvimento de automações para validação de qualidade e em processos de implantação.

Na execução do projeto foram enfrentados alguns desafios apontados por Gouigoux e Tamzalit (2017). O primeiro deles, como dividir o antigo monolito em partes granuladas? De acordo com os autores, muita literatura explica a abordagem de microsserviços, mas isso é mais frequentemente citado no contexto de aplicativos para *web*, de alto volume de acessos, mas o caso do MGDIS não é exatamente esse. A equipe não sabia o nível de granularidade ideal para aquele contexto. A definição desta divisão foi baseada na abordagem funcional da aplicação, agrupando os microsserviços por funcionalidades. Outro desafio enfrentado foi a mudança na forma de implantação do aplicativo, já que os métodos tradicionalmente utilizados em arquitetura monolítica não serviam mais. Por último, a orquestração dos serviços, fazê-los conversar e integrar as informações entre o sistema do cliente com o MGDIS. A orquestração dos serviços progrediu através de tentativas e erros. Primeiramente foi testado uma camada ESB (*Enterprise Service Buses*), funcionando como um *gateway* entre o sistema do cliente e o MGDIS, para conectar as duas pontas de forma facilitada. A avaliação do uso desta camada resultou que a solução era mais adequada para grandes organizações, onde o investimento em

tecnologia é maior. A análise foi de que o custo na utilização se torna alto, pois necessita de um time técnico capacitado para esse tipo de integração. A preocupação para que a orquestração fosse bem definida foi para diminuir os riscos na reintrodução do acoplamento entre sistemas e camadas. A solução definida para o desafio foi manter o MGDIS com uma integração passiva, com exposição de *webhooks*, deixando a definição de como a integração do sistema do cliente seria feita com o MGDIS a cargo da empresa contratante.

> De acordo com IBM (2021a), um ESB, ou barramento de serviço corporativo, é um padrão de arquitetura por meio do qual um componente de software centralizado executa integrações entre aplicativos. Ele realiza transformações de modelos de dados, lida com conectividade, realiza roteamento de mensagens, converte protocolos de comunicação e, potencialmente, gerencia a composição de várias solicitações. O ESB pode disponibilizar essas integrações e transformações como uma interface de serviço para reutilização por novos aplicativos.

Um grande benefício que a transformação trouxe para o time técnico foi no reaproveitamento de serviços. Sem levar em consideração serviços comuns como autenticação e monitoramento de atividades, que quase sempre são compartilhados, cerca de um terço dos serviços já tem mais de um uso e dez entre setenta serviços são usados em mais de dois contextos diferentes.

A equipe de negócios também desfrutou do reaproveitamento de serviços. O impacto foi em relação ao tempo de lançamento de novos aplicativos semelhantes. O desenvolvimento da primeira aplicação (após *go live* da transformação), que implementou boa parte dos microsserviços, consumiu 160 dias de desenvolvimento. O segundo, precisou apenas adicionar alguns recursos em serviços já existentes, levou 50 dias para ser desenvolvido. O terceiro, teve impacto apenas na camada de integração e em configurações, levou 10 dias para desenvolver. Este benefício não havia sido pensado no planejamento da transformação de arquitetura. Além disso, houve diminuição de uma pessoa na equipe de suporte após a transformação, contabilizando uma economia de aproximadamente 70.000 € por ano.

3.1.2 *Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System* (Mi; Ma; Lu, 2020)

Os autores aplicaram a transformação de alguns módulos de um sistema legado de arquitetura monolítica para microsserviços. A execução foi feita a partir de um sistema do governo dos Estados Unidos, o sistema *Green Button*, que serve para a população acessar informações detalhadas do seu consumo de energia. O sistema foi lançado em 2012 e tem seu código aberto no repositório GitHub.

A transformação foi feita apenas pelos autores do trabalho, não foi realizada pela equipe responsável do sistema e não foi publicada em ambiente produtivo para os usuários acessarem. O objetivo do trabalho realizado foi explorar e identificar o melhor método de execução para esse tipo de transformação. Como estratégia de transformação, foram utilizadas duas técnicas. *Strangler Fig* e DDD (*Domain-Driven Design*).

O padrão *Strangler Fig* é um processo confiável e incremental para refatorar código. Método pelo qual um novo sistema cresce sobre um sistema antigo até que o sistema antigo seja estrangulado e possa simplesmente ser removido. A grande vantagem dessa abordagem é que as alterações podem ser incrementais, monitoradas o tempo todo e a chance de algo quebrar inesperadamente é bastante baixa. O sistema antigo permanece em vigor até que se tenha certeza de que o novo sistema está operando conforme o esperado, e então é uma simples questão de remover todo o código legado (FOWLER, 2004).

DDD é um tipo de metodologia de desenvolvimento de software baseada no conhecimento de domínio. É o conceito de que a estrutura e a linguagem do código de software (nomes de classe, métodos de classe e variáveis de classe) devem corresponder ao domínio do negócio (EVANS, 2016).

O método DDD foi utilizado na análise do sistema monolítico original, com objetivo de decompor o sistema em várias partes do ponto de vista da lógica de negócios, para identificar os microsserviços que seriam construídos. Com o DDD aplicado é possível identificar as entidades e seus relacionamentos no sistema legado. Essa etapa é essencial para converter módulos existentes em novos microsserviços.

Com o desenvolvimento incremental, onde cada módulo do sistema é abordado por vez, os autores enfatizam a importância em definir a ordem na qual os serviços devem ser extraídos. É feita uma avaliação dos benefícios que cada serviço extraído trará. A avaliação considera cinco critérios, onde cada funcionalidade do sistema monolítico é avaliada. Os critérios em questão estão representados pela Tabela 3.

Tabela 3 – Diretriz de pontuação para serviços.

Critério	**Pontuação**	**Benefícios**
Funcionalidade prestes a ser modificada na evolução do sistema legado.	0 ou 40.	Evitar realizar modificações na aplicação monolítica.
O serviço tem problemas de	0 até 20.	Evitar o consumo de recursos

performance.		indevidos.
O serviço tem funções essenciais do sistema (*core*).	0 até 20.	Se o serviço extraído inclui funções essenciais, à extração geralmente traz mais benefícios como escalabilidade ou manutenção.
O serviço está relacionado com um serviço migrado.	0 até 10.	É mais simples extrair um serviço que tem relação com um já migrado.
O serviço já é separado da estrutura.	0 até 10.	A complexidade dessa extração é menor, já que tem menor acoplamento.

Com os microsserviços definidos, a implementação já pode ser iniciada. A comunicação da nova estrutura com a antiga deve ser feita através de API. A ideia é que o usuário não seja impactado na migração, já que o serviço extraído só será removido após a migração.

Os principais impactos técnicos e de negócio apontados pelos autores seriam na melhora para escalar a aplicação e na facilitação de manutenção e evolução. Porém tais impactos não foram aplicados, pois a migração não foi executada no ambiente real do sistema.

3.1.3 *Monoliths to microservices - Migration Problems and Challenges: A SMS* (Valepucha; Flores, 2021)

O objetivo dos autores ao descrever este artigo foi de realizar uma revisão da literatura para identificar os problemas e desafios que surgem no processo de migração de arquitetura, e apresentar informações de tal forma que sirva de referência para os profissionais que desejam realizar este processo migratório. Os autores não realizaram a transformação na prática, mas trazem uma lista de informações importantes para se considerar no processo.

Existem recomendações que devem ser levadas em consideração antes de realizar o processo de migração. Por exemplo, se uma aplicação monolítica tem pouca funcionalidade de negócio e é pequena, talvez seja mais conveniente mantê-la como está e apenas realizar uma atualização para as versões mais recentes do *framework*, versão de linguagem de programação e otimizações de código. Porém, se uma aplicação é grande e complexa de manter, implementar mudanças é demorado, a empresa precisa utilizar tecnologias recentes, a recomendação é realizar a modernização da aplicação.

A lista de desafios a se considerar estão organizados nos tópicos abaixo:

- Decomposição do monolito: existem modelos e métodos que apoiam a decomposição. Atributos como granularidade, acoplamento e coesão entre os microsserviços devem ser considerados nesta fase.

- Reorganização das equipes de trabalho: no processo de migração a empresa normalmente enfrenta problemas organizacionais na equipe. Normalmente as responsabilidades de toda a equipe são divididas por tecnologias, por exemplo, uma equipe é responsável pelo *frontend*, outra pelo banco de dados. Ao migrar para microsserviços, o desafio é se organizar de forma que as diferentes equipes de trabalho passem a ser responsáveis pelos serviços. Trabalhar nessa nova modalidade terá uma curva de aprendizado, porém, os benefícios são previstos a médio e longo prazo. Ao criar microsserviços que são fracamente acoplados, a equipe responsável pode fazer melhores estimativas das mudanças.

- Incorporar o uso de tecnologias e práticas recomendadas: parte do sucesso da migração para microsserviços é a seleção e o uso adequado de tecnologias e práticas que já foram testadas com sucesso. Uma recomendação é que os serviços usem mecanismos de comunicação leves. Devido à natureza altamente distribuída, conforme o número aumenta, eles se tornam mais difíceis de gerenciar e monitorar. Para isso, recomenda-se a utilização de contêineres em Docker e a administração de microsserviços com Kubernetes. O desafio não está somente em incorporar essas tecnologias, mas em treinar a equipe para fazer uso correto delas. Os grandes benefícios dessas tecnologias estão em uma melhor resiliência, pois se um microsserviço cair ou for inibido, outro pode ser criado imediatamente, escalonando sob demanda e imediatamente, elevando o número de compilações e versões diárias através do uso de CI (Integração Contínua) e CD (Entrega Contínua).

- Realizar migração em iterações: uma estratégia muito arriscada é reescrever toda a funcionalidade em microsserviços em um único projeto de migração, pois esse processo pode levar muito tempo. Além disso, corre-se o risco de migrar funcionalidades que não são mais utilizadas pelo negócio, que não são mais necessárias. O desafio é planejar e priorizar adequadamente a funcionalidade do negócio a ser migrada por meio da realização de reuniões com usuários de negócios chave. A utilização de metodologias ágeis para as entregas é uma prática recomendada. Os benefícios com essas estratégias é que as funcionalidades de negócio mais utilizadas da aplicação estariam migradas antes. Uma técnica para este fim é a DDD, para identificar o fluxo de negócio funcional.

- Segmentação e consistência na divisão da base de dados: geralmente o monolito leva a ter uma única base de dados, e no envio de uma solicitação, várias etapas são executadas realizando um *commit* ou *rollback*, garantindo a consistência de dados. Ao migrar para microsserviços, a recomendação é de ter um banco de dados por microsserviço. Descentralizar os dados leva ao problema de garantir a consistência dos mesmos. Para combater isso, existem propostas como o uso de mensagens baseadas em eventos.

- Validações pré-migração: microsserviços nem sempre são a melhor alternativa. Antes de realizar a migração você deve validar as funcionalidades que a aplicação possui. Se for pequena, com poucas funcionalidades e baixa complexidade, a recomendação é deixá-la como está e apenas aplicar melhorias adequadas. Caso a aplicação for grande e fazer alterações nela é um processo complexo, sendo necessário altos custos de investimento, a migração é o caminho recomendado. Tomar a decisão correta permite otimizar o uso de recursos, tempo e custo.

3.1.4 Microservices migration patterns (Balalaie; Heydarnoori; Jamshidi et al. 2018)

Neste trabalho os autores relatam um conjunto de padrões de migração e rearquitetura, coletados de projetos de migração de software em escala industrial. Padrões que auxiliam a planejar uma reestruturação mais eficiente e eficaz. Os padrões estão descritos nos tópicos abaixo:

- Integração contínua: considerando a adoção de microsserviços, o número de serviços a serem compilados aumenta consideravelmente, e o processo deve ser automatizado. Cada vez que uma modificação é feita no repositório do serviço, a compilação deve ser feita. O trabalho consiste em buscar o novo código, executar testes, construir os artefatos e enviar os artefatos para o repositório para em algum momento ser utilizado em publicações nos ambientes.

- Conhecer o sistema atual: é importante ter conhecimento geral do sistema antes da migração. Ter domínio sobre as funcionalidades auxilia muito no momento de quebra do monolito em serviços. Dominar a arquitetura da aplicação é importante para ter maior reaproveitamento de bibliotecas e funções existentes.

- Decompor o monolito: a metodologia DDD deve ser utilizada para identificar subdomínios de negócio que o sistema está operando e cada subdomínio pode ser considerado um microsserviço. Este passo é importante, pois uma decomposição ruim pode trazer problemas de *performance*.

- Compartilhamento de métodos: existem duas opções recomendadas para reutilizar métodos comuns. Criar bibliotecas para códigos utilitários. Esta abordagem é melhor quando raramente necessita de mudança. A segunda opção é separar o método como um serviço.

- Utilização de registro de serviços: cada serviço rodando pode ter uma lista de várias instâncias. O balanceador de carga necessita conhecer cada instância e seus endereços no momento de direcionar as requisições. A solução para este problema pode ser a utilização de ferramentas como Eureka, Consul, Apache Zookeeper, entre outras.

- *Circuit Breaker*: utilizar esse mecanismo é importante para tornar os serviços mais resilientes. O mecanismo é posto no lado consumidor. Ele monitora as respostas recentes do provedor de serviços e irá agir quando o número de respostas de falha ultrapassar o limite definido. A ação pode ser retornar um código de resposta específico ou até mesmo retornar dados em cache mais recentes. O mecanismo fará esse processo até que o provedor do serviço volte a ficar disponível.

- Monitoramento: cada serviço deve ter sua ferramenta de monitoramento independente. Ferramentas que fornecem informações como uso de CPU (*Central Process Unit*) e memória RAM (*Random Access Memory*) são instaladas nos microsserviços. Em tempo de execução, as informações são coletadas e enviadas para um servidor de monitoramento, onde são armazenadas para serem consultadas com eficiência, utilizando um servidor de indexação para compilar os dados e serem utilizadas por uma ferramenta de visualização.

3.1.5 *Microservices Migration in Industry: Intentions, Strategies, and Challenges* (Fritzsch; Bogner; Wagner et al. 2019)

Neste trabalho os autores realizaram uma série de entrevistas com pessoas responsáveis por realizar a migração de diferentes sistemas para microsserviços. O objetivo do projeto era descobrir quais eram as intenções das empresas ao migrar a aplicação, quais as estratégias adotadas e quais os desafios técnicos e organizacionais. As entrevistas foram realizadas na Alemanha.

Entre as entrevistas realizadas, a maior parte das intenções pelas quais se procurava a migração era por causa de problemas de manutenção. Os problemas citados eram: aplicar modificações se tornou caro; alterações são propensas a efeitos colaterais; aplicar análise nas funcionalidades é muito difícil; não se tem mais a visão e conhecimento geral do sistema. Além disso, problemas de operabilidade foram mencionados. Problemas no rastreio de erros, tempos

muito altos para inicialização e dificuldades em aplicar atualizações em geral. Também foi citado o consumo elevado de recursos de infraestrutura e tempo para lançamento de novos recursos no mercado.

A estratégia de migração predominante para os sistemas investigados foi de reescrever a aplicação existente. A metade dos casos utilizou o padrão *Strangler Fig* para substituir gradualmente o sistema existente. Os quadros de tempo para a migração tiveram variação de um ano e meio até três anos ou mais. Em vários casos foram montadas equipes grandes para o desenvolvimento, em um projeto entrevistado foram formadas equipes de até quarenta pessoas. Como estratégia para decompor o monolito, boa parte das empresas utilizaram a abordagem de decomposição por funcionalidade. Apesar de ser bastante mencionado pela literatura, apenas três empresas utilizaram o método DDD para este fim.

Como desafios técnicos, encontrar o nível de decomposição correta e a falta de experiência na área de construção de uma arquitetura em microsserviços foram identificadas com frequência nos projetos. Foi relatado também um esforço elevado para garantir segurança na nova estrutura e construir uma arquitetura resiliente com serviços tolerantes a falhas. No entanto, alguns casos mencionaram que o maior desafio não era técnico, pois sempre existem pessoas, talvez externas, que podem dominar o assunto.

As migrações investigadas enfrentaram um desafio na transição de modelos de processo tradicionais para metodologias ágeis. Normalmente os meses iniciais dessa reestruturação causam atritos constantes na colaboração. Empresas utilizam formas diferentes para lidar com este desafio, o importante é conseguir aplicar uma mudança de mentalidade para metodologias ágeis.

3.2 COMPILADO DOS ARTIGOS SELECIONADOS

A Tabela 4 mostra um compilado dos artigos que foram detalhados neste trabalho.

Tabela 4 – Compilado dos artigos selecionados.

Artigo	Objetivo	Estratégia de transformação	Desafios	Impacto técnico/negócio
Gouigoux;	Custo de	-	Decomposição do	Reaproveitamento

Tamzalit, 2017.	interoperabilidade e melhorar o sistema para longo prazo.		monolito, forma de implantação e orquestração de serviços.	de serviços e retorno de investimento.
Mi; Ma; Lu, 2020.	Explorar e identificar o melhor método de execução de transformação.	*Strangler Fig* e DDD	-	Melhora na escalabilidade e manutenção.
Valepucha; Flores, 2021.	Identificar problemas e desafios.	-	Decomposição do monolito, organização da equipe, novas tecnologias, entre outros.	-
Balalaie; Heydarnoori; Jamshidi et al. 2018.	Auxiliar na reestruturação mais eficiente e eficaz.	-	Decomposição do monolito, Integração contínua, domínio sobre sistema, entre outros.	-
Fritzsch; Bogner; Wagner et al. 2019.	Descobrir as intenções, estratégias e desafios das empresas na migração de arquitetura.	*Strangler Fig*.	Decomposição do monolito, inexperiência dos profissionais, arquitetura resiliente, entre outros.	Impacto na transição do processo de desenvolvimento das equipes.

4 ESTRUTURA ATUAL DA PLATAFORMA DE *E-COMMERCES*

A empresa detentora da aplicação é líder de vendas no setor em que atua, com mais de 48 anos de história. A sua atuação não é apenas no ramo digital de forma *online*, ela tem forte atuação em vendas por lojas físicas também. Comercializa atualmente mais de 11 milhões de itens por ano. Lançou as vendas *online* no ano de 2014, publicando o *e-commerce* para uma de suas marcas. Nos anos seguintes, com o sucesso nas vendas *online*, começou a disponibilizar outros *e-commerces* para o restante de suas marcas. O faturamento nestes canais de vendas passou a ser cada vez mais relevante para a empresa com o passar dos anos.

Atualmente existem oito *e-commerces* e três aplicativos *mobile* que estão rodando na estrutura. As aplicações vendem produtos de moda, com grande parte dos produtos voltados para o gênero feminino. Todas essas aplicações estão implementadas em uma única plataforma, com exceção da camada frontend dos aplicativos *mobile*, que estão separados.

Um e-commerce e um aplicativo funcionam na modalidade de *marketplace*, tendo mais de setenta *sellers* vendendo produtos de várias marcas do setor de moda. O restante (sete *e-commerces* e dois aplicativos *mobile*) atuam no formato simples, onde vendem apenas produtos da marca que o sistema representa.

O número de acessos simultâneos, somando todas as aplicações, gira em torno de cinco mil e quinhentas pessoas entre dispositivos *mobile* e *desktop* em dias normais. Em dias de campanha promocional esse número aumenta significativamente, tendo um aumento de até 110% no número de acessos.

Para sustentar e evoluir as aplicações, o time tecnológico é composto de mais de sessenta pessoas distribuídas em oito *squads*. Existem ainda equipes operacionais e uma equipe de suporte operacional que trabalham para manter os ambientes das aplicações em funcionamento.

4.1 ARQUITETURA

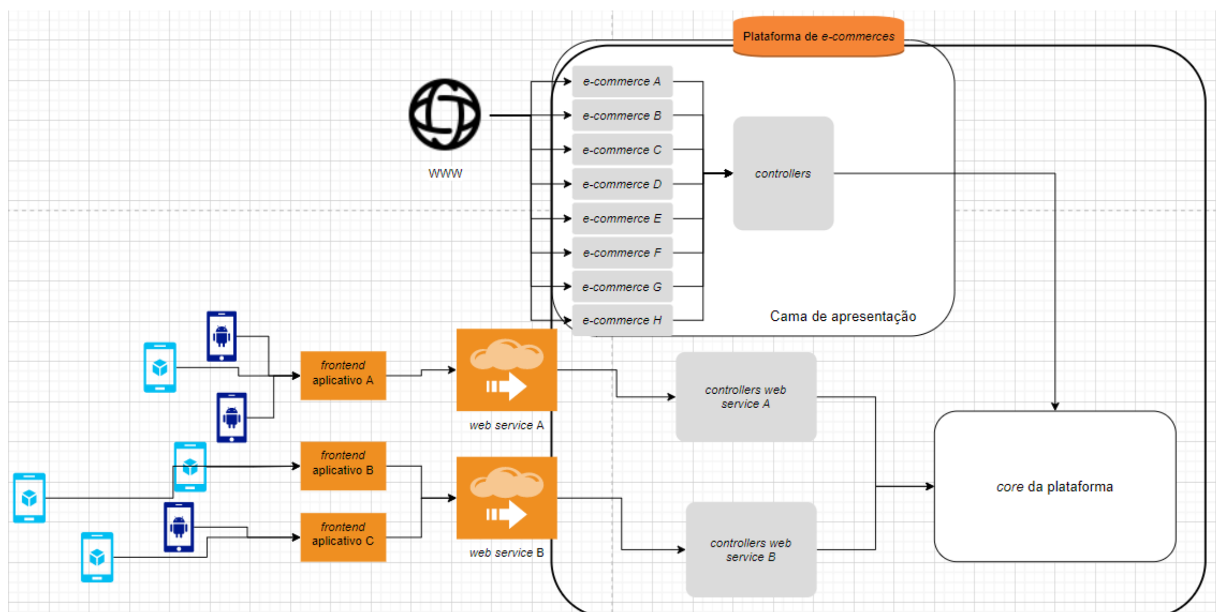
No momento da escrita deste trabalho, a estrutura desses sistemas usa a arquitetura monolítica, rodando a partir de uma plataforma de *e-commerces* de mercado. É uma plataforma de comércio eletrônico, que contempla venda de produtos para pessoas físicas e empresas. O código da aplicação é organizado utilizando o conceito Jigsaw, onde o desenvolvimento é organizado em módulos.

> De acordo com DevMedia (2016), a ideia no projeto Jigsaw é que os módulos sirvam como uma camada adicional de abstração, acima do nível de *package*, permitindo que se declare quais são os pacotes que serão exportados para módulos que dependam do módulo que está sendo desenvolvido. Com isso, os pacotes que não estiverem declarados como exportados ficarão visíveis somente por classes que estejam dentro do mesmo módulo.

A camada *frontend* de cada um dos oito sites de *e-commerce* está separada por módulos diferentes, onde cada site contém o seu específico. Telas como *checkout* e carrinho, têm sempre as mesmas funcionalidades, têm seu código unificado, onde apenas a camada de estilo é específica para cada site. Isso auxilia muito na manutenção. Com exceção dessas duas telas, cada site tem o seu próprio código fonte, mesmo que a estrutura dos sites seja parecida. A camada *backend* que responde os sites está unificada, contemplando todos eles por meio do mesmo código fonte.

Além dos sites de *e-commerce*, a aplicação ainda responde três aplicativos *mobile*, onde apenas a camada *backend* está neste projeto. Grande parte dos serviços que servem os sites também são reaproveitados para os aplicativos. Isso contribui para facilitar a manutenção e centralizar regras de negócios para ambos canais de vendas.

Figura 4 – Organização da estrutura da plataforma.



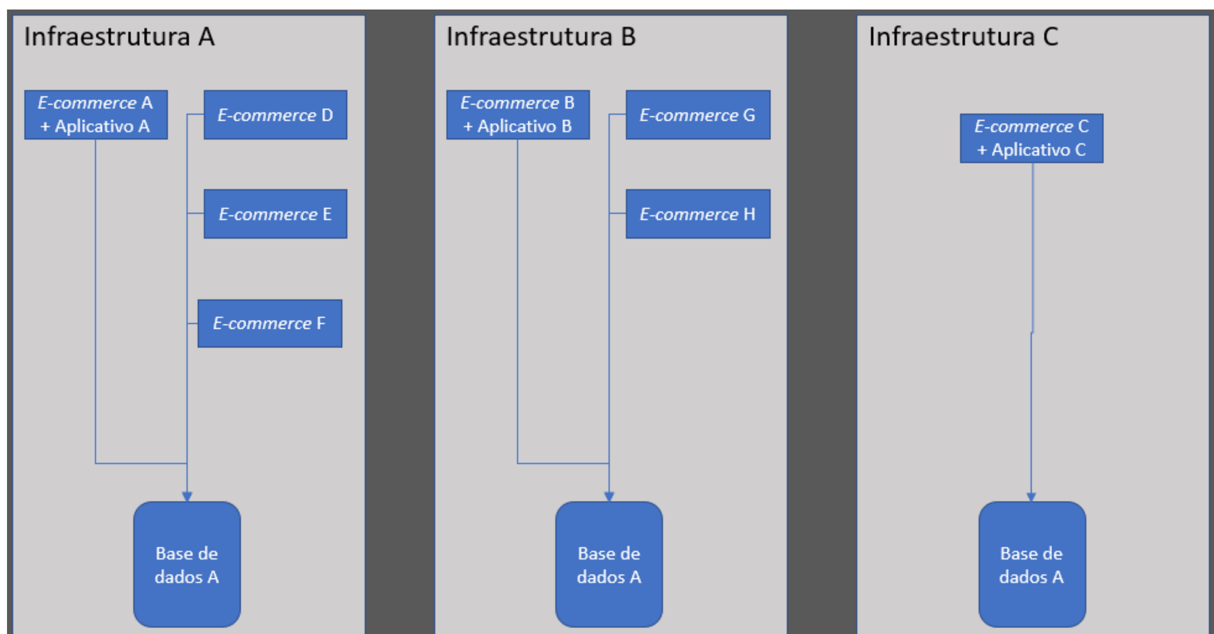
Fonte - criação própria

A organização arquitetural está representada pela Figura 4, onde mostra a separação das camadas que existem e o nível de dependência entre elas. Além disso, fica explícito que a

camada *frontend* dos aplicativos fica fora da estrutura, consumindo os *web services* que estão expostos na rede pública.

Em ambiente produtivo é utilizado três infraestruturas separadas, onde os sites estão distribuídos entre elas. Quatro sites e um aplicativo rodam na primeira, três sites e um aplicativo rodam na segunda e um site mais um aplicativo rodam na terceira infraestrutura. Cada infraestrutura funciona de forma independente, onde cada uma delas tem o seu banco de dados e os seus servidores específicos. Esta estratégia está ilustrada na Figura 5, a separação serve para não sobrecarregar os recursos e diminuir problemas se algo der errado em alguma das estruturas.

Figura 5 – Infraestrutura do ambiente produtivo.



Fonte - criação própria

4.2 TECNOLOGIAS EM USO

O código fonte das aplicações está inserido no GitHub, que auxilia para controlar as versões e alterações dos arquivos.

4.2.1 Backend

A camada *backend* faz uso da linguagem de programação Java na versão 11. Para auxiliar o desenvolvimento foi adicionado o *framework* Spring. O banco de dados utilizado é o Oracle 11g. Para executar as aplicações se utiliza o Apache Tomcat como servidor que já está embutido na plataforma. Como sistema de pesquisa, se faz uso da plataforma Apache Solr.

4.2.2 *Frontend*

Para exibir páginas dinâmicas dos sites é utilizado JSP (*Java Server Pages*), onde elas são renderizadas no lado do servidor e não no navegador como acontece em alguns sites. Páginas estáticas utilizam HTML (*HyperText Markup Language*). Para interações com usuários e validações de formulários se faz uso da linguagem JavaScript e o estilo é gerado pelo SASS (*Syntactically Awesome Style Sheets*).

4.3 MÓDULOS PRINCIPAIS EXISTENTES

Conforme descrito anteriormente, a arquitetura atual é composta de alguns módulos. Cada um contendo suas responsabilidades, a fim de deixar o projeto mais organizado. O nível de dependência entre eles é controlado via código fonte. Grande parte dos módulos existentes servem para integração com APIs externas. No momento de compilar a aplicação para realizar uma publicação em algum ambiente, é gerado um único pacote.

4.3.1 *Core* da aplicação

O *core* da aplicação reúne a maior quantidade de serviços, além de ser a parte mais complexa do sistema. Aqui estão implementados funcionalidades como: rotinas agendadas que rodam em segundo plano, principalmente para exportar dados; envio de e-mails; sistema de busca; gerenciador de conteúdo das páginas; painel de administração dos *e-commerces*; sistema de cache; serviço que separa o pedido em partes baseadas na localização dos depósitos; catálogo de produtos; dados do cliente.

Algumas funcionalidades, a própria plataforma já oferece nativamente, como por exemplo, o painel de administração, sistema para gerenciar conteúdo, o sistema de cache e o modelo básico do catálogo de produtos. O restante das funcionalidades do *core* está organizado em um módulo customizado, que é considerado o centro da aplicação. Praticamente todos os outros módulos têm alguma dependência com este *core*.

4.3.2 Lojas (sites *frontend*)

Cada site contém um módulo que agrupa os arquivos de *frontend*. Arquivos como JavaScript, CSS (*Cascading Style Sheets*) e as próprias páginas que pertencem ao site. As páginas são renderizadas no lado do servidor. Classes Java controladoras das páginas do site estão em um outro módulo a parte e esse único módulo serve todos os sites. A camada *backend* é unificada para servir qualquer *e-commerce* que a plataforma hospeda.

Para facilitar o desenvolvimento e manter código centralizado é utilizado o conceito de componentes. Partes do site como por exemplo, menus, banners, *footer*, logotipo, que são exibidos em mais de uma página, são normalmente implementados como componentes. Eles têm seu código implementado em arquivos separados e o *backend* fica responsável por renderizar nas partes certas das páginas.

4.3.3 Web service

Existem dois módulos que servem como *web service*, expondo serviços na internet para serem consumidos. Os consumidores desses *web services* são os aplicativos. Com exceção da camada *backend*, os aplicativos têm seu código separado de todo o *e-commerce* e precisam se comunicar através desta API para obter os dados necessários e funcionarem corretamente. A diferença dos dois módulos é que um é mais moderno que o outro, tendo algumas diferenças no gerenciamento de cache dos dados. Dois aplicativos que foram lançados nos anos de 2020 e 2021 estão utilizando a versão moderna.

A camada de segurança é feita utilizando o *Spring Security*, baseado em *tokens*. Uma *request* de autenticação fica disponível e espera alguns parâmetros no corpo da requisição. Parâmetros de identificação da API (client_id) e senha da identificação (client_secret). Esse modo da requisição retorna um *token* para um acesso anônimo, onde não existe um usuário logado. Esse *token* pode ser usado em transações que não precisam de dados do usuário. Para transações que necessitam de um usuário autenticado, são passados mais dois parâmetros na requisição, que é a identificação de usuário e senha. Com isso, é retornado um *token* que permite realizar transações restritas. O *token* retornado na requisição de autenticação é enviado no *header* das demais requisições realizadas nestas APIs. Com isso, nas chamadas realizadas, o *backend* consegue identificar quem está solicitando os dados e quais permissões tem.

4.3.4 Integração *marketplace out*

Uma funcionalidade existente na aplicação é o *marketplace out*. Nesta modalidade as empresas buscam *marketplaces* já estruturados, com grande número de audiência para ofertarem seus produtos e otimizarem as vendas (AZEVEDO, 2021). Neste módulo há uma integração com um parceiro externo que já está integrado com outros *marketplaces* de mercado. O catálogo de produtos é enviado para esse parceiro via integração e os produtos podem ser selecionados para venda em vários *marketplaces* externos, como por exemplo: Netshoes, Dafiti, Mercado Livre, entre outros. Além disso, existe uma integração de pedidos para o momento em

que o cliente realizar a compra de produtos da marca ofertada. A integração utiliza o padrão REST (*Representational State Transfer*).

REST trata-se de um conjunto de princípios e definições necessários para a criação de um projeto com interfaces bem definidas. A utilização deste padrão permite a comunicação entre aplicações (TOTVS, 2020).

4.3.5 Integração *marketplace in*

No *marketplace in* a empresa abre seu próprio *marketplace* e faz captação de *sellers* para que eles vendam em seu *e-commerce* (AZEVEDO, 2021). Para esse canal de vendas existe uma integração com um sistema que realiza o gerenciamento do *marketplace*, tendo configurações de *sellers*, moderação de produtos e os produtos que foram aprovados pela curadoria são recebidos desta integração. Para essa funcionalidade existem quatro módulos. Um responsável pela parte que se comunica com o sistema de gerenciamento, outro que armazena os serviços e regras de negócios, outro que expõe a API que recebe os dados do gerenciador e o último que é apenas uma fachada, fazendo a ligação entre a API e a camada de serviços.

4.3.6 Trocas e devoluções

Para produtos com defeito, tamanho errado ou não atendimento das expectativas, existe a possibilidade de realizar troca ou devolução de produtos comprados. A solicitação pode ser feita diretamente pelo site através de um formulário. Feita a solicitação, é realizada uma integração com a API dos Correios, para obter um código para coleta da mercadoria no endereço do cliente. Feito isso, é realizada uma nova integração com o sistema que a equipe do SAC (Serviço de Atendimento ao Consumidor) utiliza, enviando o que foi solicitado pelo cliente, juntamente com o código de coleta dos Correios. A solicitação do cliente entra em uma fila de atendimento para ser processada pelo SAC. Para essa integração existem dois módulos responsáveis. Um para os Correios e outro para o sistema do SAC. O tipo de integração é REST para ambas integrações.

4.3.7 Integração de canais

Os *e-commerces* têm duas modalidades de entrega além da tradicional. Essas modalidades servem para os franqueados terem uma participação nas vendas dos canais digitais. Uma das modalidades consiste no cliente retirar o produto na loja franqueada e a outra é a loja entregar o produto diretamente para o cliente. As duas modalidades são oferecidas quando a loja está entre alguns quilômetros do endereço do cliente, possibilitando ao cliente escolher qual modalidade ele prefere. Também existe um painel para as lojas realizarem o atendimento ao

cliente, podendo selecionar se a loja tem condições ou não de proceder com o atendimento. Para essas formas de venda existem três módulos, para o painel, para os serviços e regras de negócio e para a fachada que conecta o painel aos serviços.

4.3.8 Sistema de recomendação de produtos

No momento está em uso duas ferramentas para recomendação de produtos, que funcionam de maneiras diferentes. Uma delas monitora o acesso do cliente nas páginas de detalhes de produtos, compila os dados coletados, executa o motor de recomendação através de inteligência artificial, indexa os produtos e disponibiliza para o *e-commerce* requisitar para oferecer ao cliente. Essa é uma ferramenta própria da empresa. A outra ferramenta recebe todo dia um arquivo contendo todos os carrinhos montados pelos clientes, cada carrinho contém as informações do seu cliente dono. Feito isso, os dados são processados resultando em uma lista de produtos recomendados para o cliente. As duas ferramentas geram uma lista de produtos que são mostradas em algumas páginas dos sites.

4.3.9 Marketing digital

Nas páginas dos *e-commerces* existem lugares para captação de clientes que desejam receber novidades. O cliente apenas insere seu e-mail de contato. Essa informação é enviada através de integração para um sistema externo que serve para auxiliar na realização de campanhas de *marketing* e divulgação de marca. A integração acontece através do protocolo REST.

4.3.10 *Gateway* de pagamento

Os *e-commerces* possibilitam a utilização de várias formas de pagamento ao cliente, formas que necessitam de um sistema externo para interceptar o valor pago e em algum momento realizar a captura do montante. Algumas modalidades de venda realizam tratamentos especiais internamente, como por exemplo as modalidades de Retire na Loja e Entrega pela Loja, ou até mesmo uma venda feita no *marketplace*. Essas duas, em especial, têm necessidade de repassar uma fatia de valor da venda para os lojistas ou até mesmo para os *sellers*, que são donos dos produtos ofertados e vendidos. Além desse formato ainda existe a possibilidade da forma tradicional, onde não tem necessidade de realizar *split* de pagamento entre mais de um recebedor. Para esses dois tipos de captura de valores é utilizado duas integrações diferentes, cada uma sendo controlada por um módulo separado e ambas utilizam protocolo REST para comunicação. Existe ainda um terceiro módulo, que é a integração para cartões de presente. Nesta forma, o cliente compra ou ganha um cartão em uma quantia em dinheiro, junto a isso,

recebe um código e senha que podem ser usados no site para compra de produtos. Essa integração utiliza o padrão SOAP (*Simple Object Access Protocol*) para transferir os dados.

4.3.11 Cotações de frete

Para buscar os prazos e valores de entregas dos produtos integrados diretamente com as transportadoras parceiras, é utilizado uma integração com uma empresa que facilita esse processo. Essa empresa tem parceria com as transportadoras em que o *e-commerce* trabalha. Com isso é possível obter os valores e o prazo para entrega dos produtos.

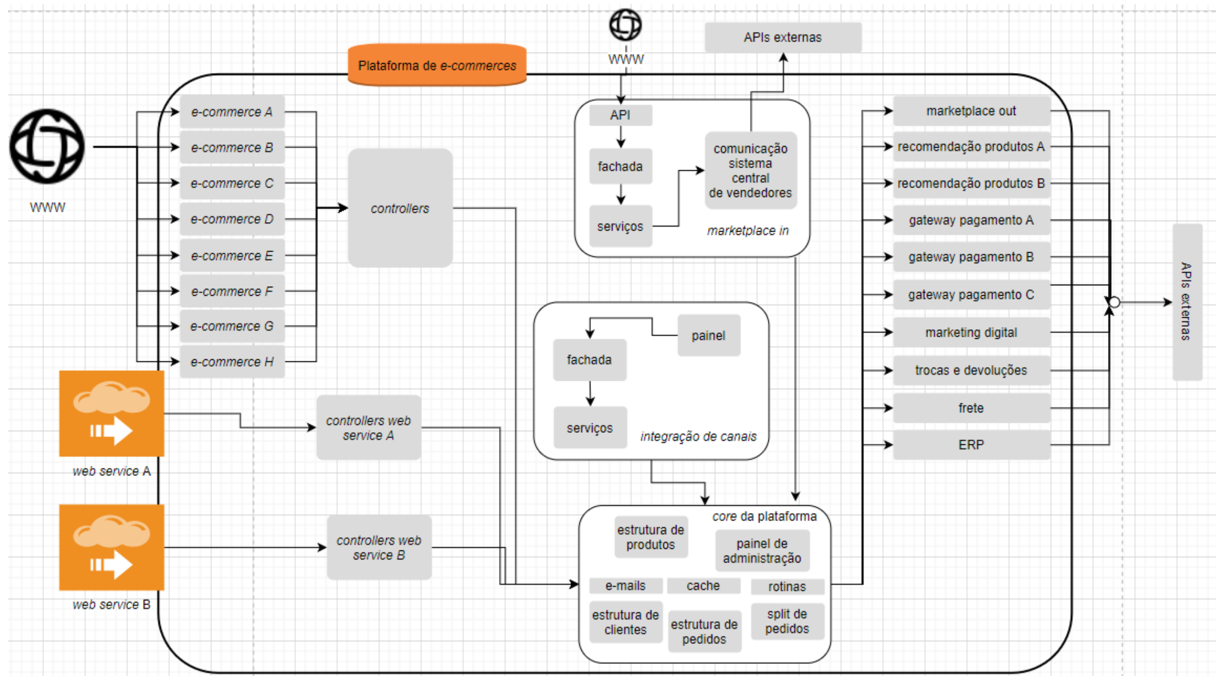
4.3.12 ERP (*Enterprise Resource Planning*)

Estoques e preços são controlados no sistema ERP; os *e-commerces* apenas armazenam um espelho do que a integração com este sistema envia. Cada alteração de preço, adição de desconto ou modificação de estoque é realizado no painel do ERP e replicado para os *e-commerces*. Além disso, os pedidos gerados são enviados para o ERP, chegando até a equipe do centro de distribuição para realizar a separação das mercadorias e encaminhar para as transportadoras entregarem até o endereço do cliente. Cada alteração de *status* dos pedidos é trafegada através desta integração. A integração utiliza o protocolo de comunicação SOAP e trabalha no conceito de filas, onde cada informação compartilhada fica disponível em uma fila até que ela seja consumida com sucesso.

SOAP é um formato de mensagem XML (*Extensible Markup Language*) usado nas interações de serviços da *web*. As mensagens SOAP são tipicamente enviadas através de HTTP ou JMS (*Java Message Service*), mas outros protocolos de transporte podem ser usados. O uso de SOAP em um serviço da *web* específico é descrito por uma definição WSDL (*Web Service Description Language*) (IMB, 2020b).

A Figura 6 representa o detalhamento dos módulos que foram descritos nos parágrafos anteriores. A ilustração mostra como está a separação no interior da aplicação.

Figura 6 – Detalhamento dos módulos da plataforma.



Fonte - criação própria

4.4 MODELO DE TRABALHO

A aplicação é sustentada e evoluída por uma equipe técnica de mais de sessenta pessoas distribuídas em oito *squads*. Cada *squad* contém um PO (*Product Owner*), desenvolvedores *backends* e *frontends* e ao menos um testador. A distribuição dos desenvolvedores é baseada na necessidade de cada *squad*, de acordo com o volume de modificações que são normalmente necessárias nas camadas do software. As *squads* existentes estão listadas nos tópicos abaixo:

- *Checkout*: responsável pelas páginas de carrinho, *checkout*, confirmação do pedido e integrações realizadas no fechamento dos pedidos;

- *Shopping*: responsável pelas páginas *homepage*, pesquisa, detalhe de produto, minha conta e integrações que envolvem o cadastro do cliente;

- Pós-venda: controla as rotinas realizadas após a conclusão da compra, por exemplo, e-mails transacionais, trocas e devoluções e integrações como checagem de fraude;

- Integração de canais: mantém e evolui modalidades de venda Retire na Loja e Entrega pela Loja, além do painel que as lojas utilizam para atendimento dos pedidos;

- Mobilidade: responsável pela manutenção dos aplicativos, inclusive a camada *backend*;
- *Marketplace*: cuida do site que é *marketplace* nas modalidades *in* e *out*;
- Painel *marketplace*: responsável por manter o painel de gerenciamento do *marketplace*, inclusive a integração com o *e-commerce*;
- Mobilidade *marketplace*: equipe que evolui o aplicativo que é *marketplace*.

Para o método de trabalho são utilizadas algumas práticas do Scrum. O desenvolvimento das atividades é realizado por *sprints* com duração de duas semanas. Antes de iniciar a *sprint* é realizada uma cerimônia de refinamento, onde o PO fornece todas modificações, juntamente com todos detalhes e regras de negócio das modificações que se tem desejo de realizar. Neste momento a equipe tira todas as dúvidas em relação ao negócio e já se inicia um planejamento da parte técnica. Em outro dia, acontece a apresentação do que foi desenvolvido na *sprint* passada para a equipe de negócios, logo após isso, a realização de uma retrospectiva, discutindo o que não funcionou e o que aconteceu bem na *sprint* anterior. Além disso, no mesmo dia, acontece o encontro de planejamento, onde se define tudo que é necessário para desenvolver as atividades, inclusive estimando-as em formato de tempo.

As entregas das *squads* são divididas em janelas semanais, onde ocorre um revezamento com dois grupos. Em uma semana é gerado um pacote de quatro *squads*, na outra semana outras quatro *squads* se organizam para realizar entregas. Os pacotes gerados são homologados através de testes regressivos, manuais e automatizados, e publicados em ambiente produtivo. O processo de homologação e publicação em produção leva ao menos três dias para ser realizado, que é executado a cada semana.

Existem três ambientes de apoio que o time faz uso:

- Ambiente local: cada desenvolvedor tem o seu e é usado para o desenvolvimento das atividades;
- Ambiente de integração: cada *squad* tem o seu e é utilizado para validar as atividades entregues pelos desenvolvedores;
- Ambiente de homologação: é o espelho do ambiente de produção. Aqui é reunido o pacote gerado pelas entregas das *squads* e ocorre os testes regressivos antes de realizar a entrega em produção.

Para junção dos pacotes entregues pelas *squads*, publicação do pacote em homologação, *deploy* em produção e suporte de problemas que ocorrem em produção, é montada uma equipe temporária, que trabalha durante trinta dias. A equipe é composta de pessoas retiradas das *squads*. Essa estratégia é utilizada para o processo não se tornar maçante para os integrantes.

4.5 PRINCIPAIS PROBLEMAS

Muitas mudanças ocorreram na equipe técnica que mantém e evolui os produtos, com crescimento de mais de 600% no número de pessoas nos últimos seis anos. Muitas funcionalidades foram agregadas nesses produtos e a audiência em acessos teve aumento significativo. O que não mudou foi a arquitetura do sistema. Uma arquitetura que funcionava bem com uma equipe pequena ou média, já não flui com estabilidade em uma equipe consideravelmente grande.

O alto acoplamento entre os módulos da aplicação causa problemas para as equipes. Muitas vezes uma alteração realizada por uma *squad* afeta a funcionalidade que é de responsabilidade de outra *squad* e, normalmente, isso é identificado apenas quando está publicado no ambiente de homologação ou em casos mais críticos, em produção. A alta complexidade da aplicação aumenta a recorrência desse tipo de problema. As alterações em alguns pontos do código são grandes desafios para as equipes. Não se tem conhecimento de todas variações que o método pode ter. Uma alteração comum corre grande risco de gerar problemas. Isso se agrava com a baixa cobertura de testes unitários que a camada *backend* tem.

Quando ocorrem problemas graves no ambiente produtivo, é necessário um *deploy* para correção. Esta não é uma publicação trivial de se fazer, é necessário realizar os procedimentos completos de deploy. Não existe a possibilidade de publicar apenas a correção na funcionalidade atingida, já que se trata de um monolito. O procedimento envolve a validação completa do sistema. Entre os meses de junho até novembro, no ano de 2021, foram necessárias oito publicações para correção de problemas, para aproximadamente vinte entregas de novos pacotes (publicações normais que são realizadas semanalmente).

As dificuldades descritas até aqui também aparecem entre as respostas da pesquisa (Apêndice A) aplicada entre os *stakeholders* da plataforma. Nela há vários comentários sobre a dificuldade que é realizar publicações em produção e da inflexibilidade que a plataforma aparenta com a atual arquitetura.

O volume de acessos dos *e-commerces* é bastante variado entre as páginas. Para se ter uma base, a partir de um dos *e-commerces* deste trabalho, foram obtidos dados referentes à quantidade de requisições realizadas em algumas páginas-chave. Uma comparação foi feita entre páginas de navegação do site, que são compostas por página de *homepage,* categoria, pesquisa e detalhe do produto, com páginas de compra, compostas pelas telas de carrinho, *checkout* e resumo do pedido. O resultado da comparação foi que as páginas de navegação detêm aproximadamente 92% das requisições e o restante (8%) fica para as páginas de compra. A quantidade de requisições foi obtida através de um sistema de monitoramento New Relic. Uma variação tão grande entre o grupo de páginas deixa claro que quando houver necessidade de escalar a aplicação, em caso de aumento do número de acessos, o caminho ideal é que seja necessário despejar o maior número de recursos nas páginas de navegação do site. As páginas de compra se sustentariam apenas com o básico da infraestrutura. Esse seria o caminho ideal para melhor aproveitamento de recursos e o menor custo possível. Porém em uma aplicação monolítica, quando existe necessidade de escalar, a única alternativa é escalar completamente o sistema, aumentando o número de servidores, onde cada servidor terá uma instância da aplicação instalada.

Os itens descritos nos parágrafos anteriores são os maiores problemas enfrentados pela equipe técnica e geram impactos de forma direta para a equipe de negócios e para a empresa dona da aplicação. Tais problemas são resolvidos através da utilização de uma arquitetura de microsserviços, construída de forma adequada para hospedar todas funcionalidades que hoje existem e ainda ter bom aproveitamento em longo prazo.

5 CONSTRUÇÃO ARQUITETURAL DA PLATAFORMA DE *E-COMMERCES*

Uma proposta de transformação da plataforma de *e-commerces*, utilizando a arquitetura de microsserviços, foi construída compondo a definição dos microsserviços necessários e a estratégia de transformação para executar um possível projeto. A proposta é criada com base nos trabalhos relacionados utilizados neste projeto, além de utilizar informações retiradas de uma pesquisa executada com profissionais-chave que atuam na plataforma atualmente, no formato de monolito.

5.1 DEFINIÇÃO DE MICROSSERVIÇOS COM DDD

O ponto de partida para realizar a quebra dos módulos em microsserviços foi a utilização da técnica DDD (*Domain-Driven Design*). Ela foi utilizada por alguns trabalhos relacionados obtidos durante o levantamento bibliográfico.

A técnica, através de princípios e padrões de projetos, visa ajudar equipes de desenvolvimento a entender melhor o contexto dos projetos, contribuindo para a construção do produto final com mais qualidade (DEVMEDIA, 2009). Segundo Evans (2004), a técnica possui três pilares: linguagem ubíqua, *bounded contexts* e *context maps*. O entendimento desses pontos principais já é uma boa base para aplicar a técnica.

Com base nos pilares da técnica DDD, um glossário foi construído com objetivo de obter termos utilizados para os principais domínios do sistema. A Tabela 5 é o resultado da construção.

Tabela 5 - Glossário dos domínios da plataforma.

Store	Site / loja *e-commerce* para navegar.
Product	Produto ofertado no *e-commerce*.
Catalog	Catálogo onde ficam agrupados os produtos de cada *e-commerce*.
Category	Categoria em que um produto pertence.
Customer	Cliente que acessa o site.
Cart	Carrinho da navegação do cliente, onde pode

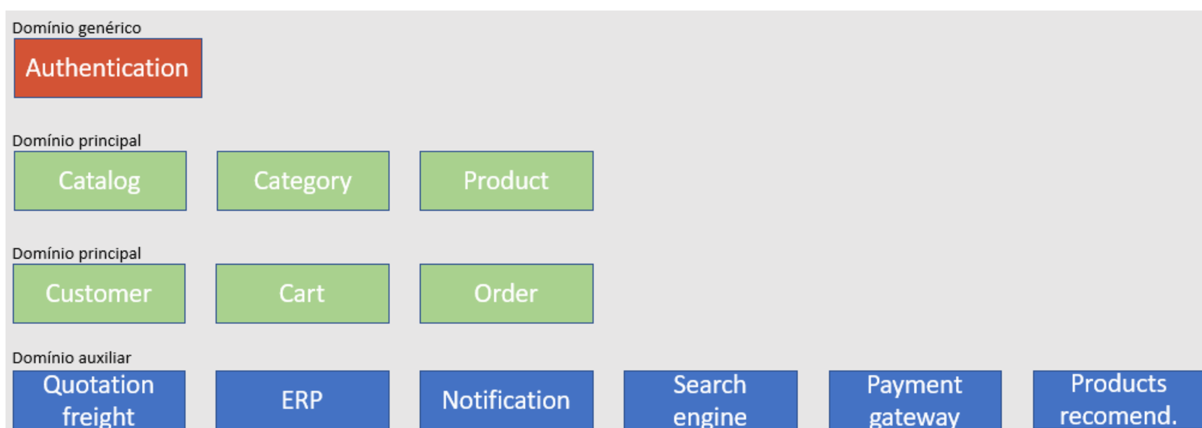
	ser adicionado itens para em algum momento realizar a compra.
Order	Pedido do cliente. O *cart* é transformado em *order* no momento da finalização de compra do cliente.
Promotion	Promoção ou cupom ofertado ao cliente.

A partir de então, pode-se construir os *bounded contexts*, onde cada contexto possui suas responsabilidades definidas, construído a partir de histórias de usuários. Abaixo estão os principais contextos para cada *e-commerce*:

1. Acessar uma loja;
2. Navegar no catálogo de produtos;
3. Realizar busca através de nome, categoria ou cor;
4. Adicionar produtos ao carrinho;
5. Criar uma conta de acesso;
6. Fazer login;
7. Acessar o carrinho criado;
8. Selecionar endereço de entrega;
9. Preencher pagamento;
10. Concluir compra;
11. Aguardar pós-processamento de pedido (integração com ERP, faturamento, despacho, notificações).

Com o contexto definido, é possível formar uma modelagem estratégica, como mostra a Figura 7.

Figura 7 - Modelagem estratégica utilizando DDD.



Fonte - criação própria

Authentication: é um domínio genérico pois pode ser utilizado por todos os domínios para identificação do cliente e garantir a segurança.

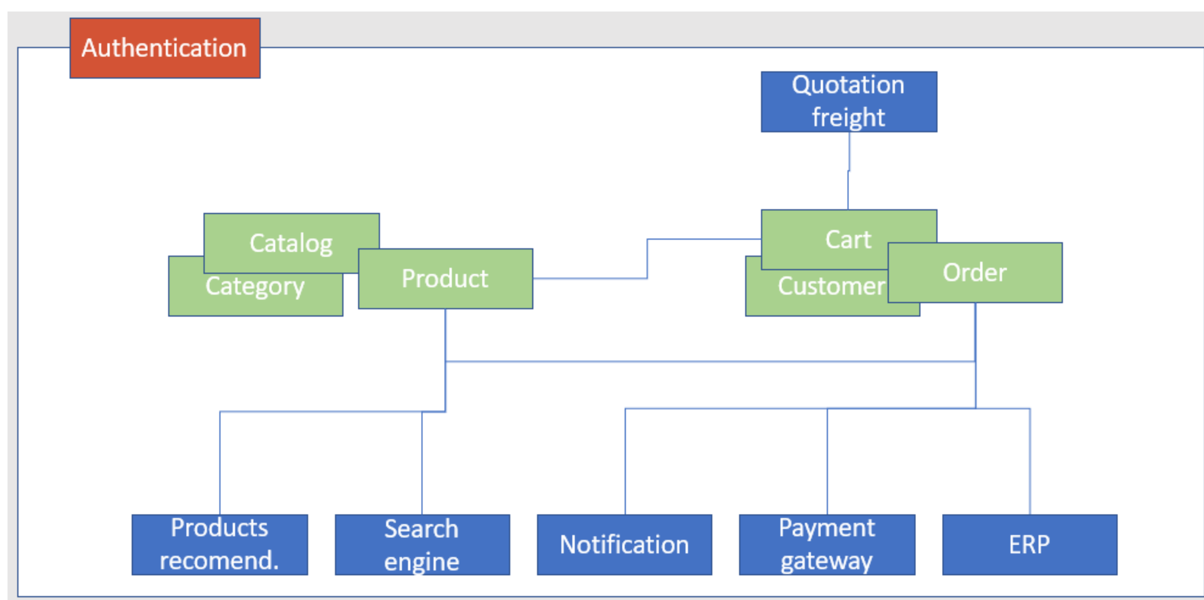
Catalog, *category* e *product*: são classificados como principal, pois sem eles o site não funciona e não tem sentido.

Customer, *cart* e *order*: são classificados como principal, pois sem eles o site também não faz sentido, não existe o principal objetivo de um *e-commerce* que é a venda.

Além dos domínios construídos a partir do contexto principal, existem domínios auxiliares, que servem como apoio aos outros domínios, contribuindo para o funcionamento do domínio principal.

O terceiro pilar é a construção do *context maps*, onde é mapeado o relacionamento dos *bounded contexts* identificados (FULLCYCLE, 2019). A Figura 8 mostra a estrutura.

Figura 8 - *Context maps.*



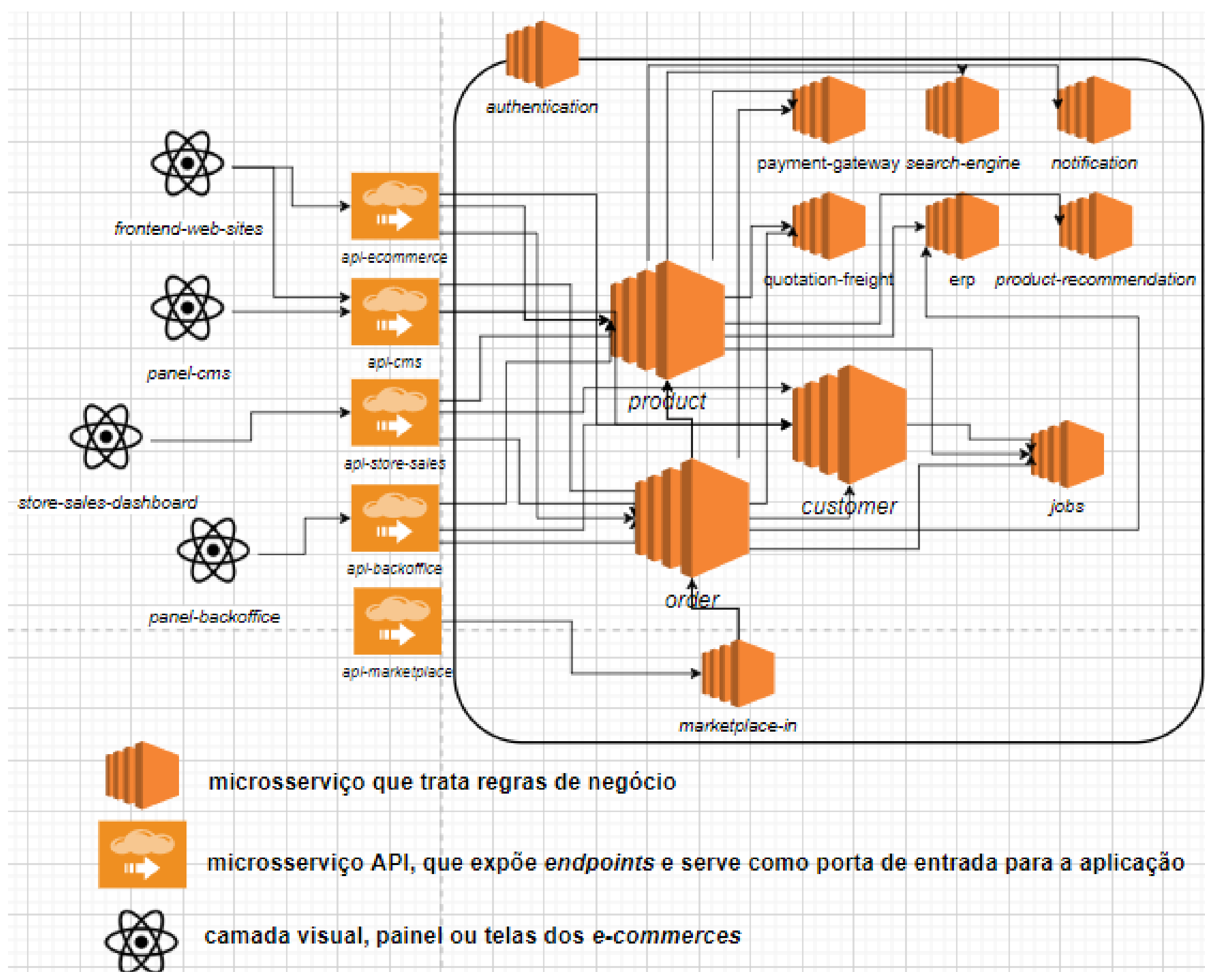
Fonte - criação própria

Na relação entre os domínios principais e auxiliares, os principais são sempre prioridade. Caso algo mude, a alteração deve ser feita no lado do domínio auxiliar, pois existem outros domínios consumindo o principal, como se fosse uma relação entre cliente/fornecedor (FULLCYCLE, 2019).

5.2 PROPOSTA DA ESTRUTURA DE MICROSSERVIÇOS

A partir da técnica DDD utilizada, foram mapeados os principais domínios para compor a plataforma. Essa construção contribuiu para um melhor entendimento e definição dos domínios, suas dependências e uma visão estrutural, possibilitando o levantamento de todos os microsserviços necessários para rodar a plataforma e seus *e-commerces*. A Figura 9 contém a lista de microsserviços e seus relacionamentos em forma de estrutura.

Figura 9 - Estrutura de microsserviços.



Fonte – criação própria

5.2.1 *Frontend web* sites

Neste microsserviço estarão contidos os códigos de todos os *e-commerces* hospedados na plataforma, cada loja existente terá seu código *frontend* agrupado nesta camada. O motivo para ter todas as lojas em um só microsserviço é para facilitar o reaproveitamento de código, utilização de componentes e padronização de código, já que existem telas com a mesma estrutura e funcionalidades. Nesses casos, as telas são tratadas como unificadas, mesmo que diferentes sites utilizem-nas.

5.2.2 *Store Sales dashboard*

Aqui estarão contidos os códigos que compõem o *dashboard* da ferramenta de *Store Sales*, onde entram compras realizadas nas lojas físicas. O acesso a esse *dashboard* se dará aos responsáveis de cada loja física.

5.2.3 *Panel* CMS

O gerenciamento de conteúdo das telas dos *e-commerces* estará neste microsserviço. Esta camada será acessada pelo time operacional que controla *banners*, vitrines de produtos, menus e outros componentes variados.

5.2.4 *Panel backoffice*

Este microsserviço será responsável pelo controle geral dos *e-commerces*, como promoções, organização de pedidos, produtos e controle de usuários.

5.2.5 API *e-commerce*

Esta API será o *web service* para servir as funcionalidades dos *e-commerces* e dos aplicativos. Todas funcionalidades que necessitam de algum dado existente no *backend* ou persistir informações em base de dados irão passar por aqui.

5.2.6 API CMS

Esta API será o *web servisse* para servir as funcionalidades do *Panel* CMS. Todas as manutenções nos conteúdos das telas que precisam de integração com o *backend* passarão por aqui. Além disso, o *frontend* dos *e-commerces* irá consumir desta API o conteúdo das telas (*banners*, menus, vitrines).

5.2.7 API *Store Sales*

A porta de entrada do *Store Sales dashboard* para a comunicação com o *backend* será esta API.

5.2.8 API *backoffice*

Este microsserviço será responsável por servir a integração do *Panel backoffice* com os serviços do *backend*.

5.2.9 API *marketplace*

Para venda de produtos em *marketplaces* externos é necessário ter uma porta de entrada para recebimento dos pedidos e suas atualizações. Além disso, para o recebimento dos produtos em que as empresas externas publicam no *marketplace* da plataforma em questão, também é necessário ter uma porta de entrada. Isso se deve tanto para novos produtos, quanto para atualizações de estoque e preço. Ainda para servir o *marketplace* da plataforma é necessário receber atualizações de *status* dos pedidos que vão sendo criados pelos clientes. Os pontos citados estarão disponíveis no microsserviço API *marketplace*.

5.2.10 *Marketplace in*

Aqui ficarão todos os serviços para servir o *marketplace* da plataforma. Chegada de novos produtos, novos vendedores e atualizações de pedidos aqui criados. Além disso, o serviço responsável por enviar novos pedidos criados ficará aqui.

5.2.11 *Marketplace out*

Para controlar a venda de produtos em *marketplaces* externos, existirá esse microsserviço. Compondo o envio do catálogo de produtos para esses *marketplaces* e o recebimento de novos pedidos.

5.2.12 *Order*

Agrupados aqui estarão os serviços para o funcionamento dos pedidos. Remoção e adição de produtos no carrinho, criação de pedido, trocas e devoluções, além do controle de toda a atualização dos pedidos.

5.2.13 *Customer*

Serviços para criação de novos clientes, atualização de seus dados, controle para *login*, endereços e dados para pagamentos serão agrupados nesta camada.

5.2.14 *Product*

Catálogo de produtos, categorias de produtos, controle de promoções e cupons de desconto, além de serviços para atualizações de imagens, preço e estoque dos produtos serão armazenados neste módulo.

5.2.15 *Jobs*

Este microsserviço será responsável por executar rotinas agendadas. Rotinas que rodam de tempo em tempo ou por algum disparo realizado. Essas rotinas terão comportamentos diversos, como por exemplo: exportação de dados via arquivo para outros sistemas, transações no banco de dados para atualização de registros, processos *background* em pedidos, entre outros.

5.2.16 *Payment gateway*

Aqui ficarão integrações com sistemas externos responsáveis por transacionar os valores pagos pelos clientes para todas formas de pagamento aceitas nos *e-commerces*. Transações como autorização de captura de pagamento e até mesmo o cancelamento de alguma transação.

5.2.17 *Search engine*

Serviços responsáveis pelos motores de busca dos e-commerces serão armazenados neste bloco. Esses motores serão responsáveis por minimizar o tempo de resposta nas buscas de produtos dos sites. A ideia é que na navegação do cliente, para buscar os produtos, o sistema não necessite acessar o banco de dados. Sendo assim, os dados ficarão indexados nesses motores de busca e prontos para serem entregues para o cliente acessar.

5.2.18 *Notification*

Os *e-commerces* têm várias rotinas de envio de e-mails para diversos destinatários, desde clientes até grupos de funcionários. Além disso, a ideia é que esse microsserviço receba qualquer integração que se refira a notificações.

5.2.19 *Quotation freight*

Serviços para cotar valor e prazo para a entrega dos produtos com base em uma localização de recebimento serão implementados neste bloco. Normalmente essas integrações fazem requisições em sistemas externos que têm parcerias com todas as transportadoras em que a empresa detentora do *e-commerce* tem contrato. Com isso, esses sistemas externos sabem informar preço e prazo para entregar os pacotes para todo o país.

5.2.20 ERP

Este módulo é responsável pela integração com o sistema responsável pelo ERP da plataforma. Essa integração consiste no envio e atualização dos pedidos, além do recebimento de produtos e atualizações de estoques e preços. O ERP é outro sistema, externo, onde é controlado toda a base de estoque, preço e faturamento dos pedidos.

5.2.21 Product recommendation

Os *e-commerces* ficam monitorando e registrando os produtos que o cliente acessa. A partir disso, são construídas vitrines de produtos relacionados com a navegação realizada. Essa é a recomendação de produtos. Essa ferramenta também está disposta em outro sistema externo e o módulo *product recommendation* é responsável por enviar os dados de navegação e consultar as vitrines para recomendar aos clientes.

5.3 PRINCIPAIS TECNOLOGIAS SELECIONADAS PARA A TRANSFORMAÇÃO

A Figura 10 apresenta as principais tecnologias para a camada *frontend*, onde ficam os painéis e sites.

Figura 10 – Tecnologias da camada *frontend.*

Fonte - criação própria

A utilização do React js se dá por ser uma ferramenta consolidada no mercado, onde grandes aplicações utilizam no momento. A ferramenta é uma biblioteca JavaScript com uma proposta de facilitar a criação de interfaces interativas. Além disso, traz a possibilidade de criação de componentes, contribuindo para a organização e reaproveitamento de código. Outro motivo que reforça a utilização dessa ferramenta é que ela já está em uso na empresa no momento em outras aplicações, então o conhecimento já circula no time.

Next js é um *framework* para utilização com React js. A ferramenta faz com que aplicações em React js sejam renderizadas no lado do servidor e não no navegador do cliente como é o padrão. O *framework* ainda traz facilidades para cachear as páginas, sendo possível tornar boa parte de uma página dinâmica em estática, trazendo muita velocidade no carregamento das telas.

Sass é uma linguagem de extensão do CSS, que compõe algumas funções que o CSS nativo não tem. O motivo de uso dessa ferramenta no projeto é a facilidade na utilização. Essa tecnologia também já é dominada pelo time técnico da empresa.

A Figura 11 apresenta a lista de tecnologias para a camada de *backend* da plataforma.

Figura 11 - Tecnologias da camada *backend.*

Fonte - criação própria

A linguagem base para todos os microsserviços *backend* será o Java. A definição é justamente por todo o time de *backend* que dará manutenção para a plataforma ser conhecedor e especialista da linguagem. Além disso, grande parte da comunidade está utilizando o conjunto de ferramentas Java + Spring Boot para rodar em microsserviços. Spring Boot torna muito fácil a inicialização de um projeto em Java.

OpenFeign vem junto com o Spring, que serve para facilitar integrações REST entre os microsserviços internos. Ele será utilizado em todos microsserviços que necessitam integrar dados com outros de forma síncrona.

JMS e ActiveMQ são soluções do próprio Java para troca de mensagens assíncronas através de filas. JMS é o servidor onde ficam as configurações de todas as filas de mensagens. Configurações como limite de retentativas em caso de erro na leitura da mensagem, envio da mensagem para uma fila de erros em caso de exceder o limite de retentativas, entre outras configurações. ActiveMQ é quem possibilita o Java de escutar as filas para processar as mensagens que vão sendo adicionadas e também para realizar o envio de alguma mensagem para as filas. Esse conjunto será utilizado em microsserviços que têm possibilidade de integrar dados de forma assíncrona.

Spring Security será quem garantirá a segurança da aplicação, servindo como base para os serviços de autenticação e para criptografar dados sensíveis salvos na aplicação. A escolha se deu por ser uma ferramenta do *framework* Spring e por ser utilizado por praticamente todas aplicações Java web da comunidade.

O Redis será utilizado para armazenar as sessões dos usuários que estarão logados nos sites e painéis. O armazenamento das sessões no Redis torna mais rápido o manuseio dos dados

pela aplicação. A alternativa para gerenciamento das sessões seria o armazenamento direto no banco de dados, porém isso traria problemas de performance.

PostgreSQL será o banco de dados para os microsserviços que necessitam armazenar dados. Microsserviços que terão essa necessidade são: *Order*, *Product*, *Customer*, *Jobs*, *Marketplace in*, *Marketplace out* e *Authentication*.

Apache Solr é um serviço que roda separado da aplicação. Neste serviço ficam indexados todos dados que são utilizados em consultas com respostas rápidas. Esse tipo de consulta se torna efetiva, pois não tem necessidade de ir ao banco de dados buscar as informações. A indexação dos dados ocorre em um determinado número de minutos, realizando a atualização dos dados no Apache Solr. O microsserviço que fará uso dessa ferramenta será o de *Product*. O motivo da utilização é que a ferramenta já está em uso no momento e atende muito bem as necessidades.

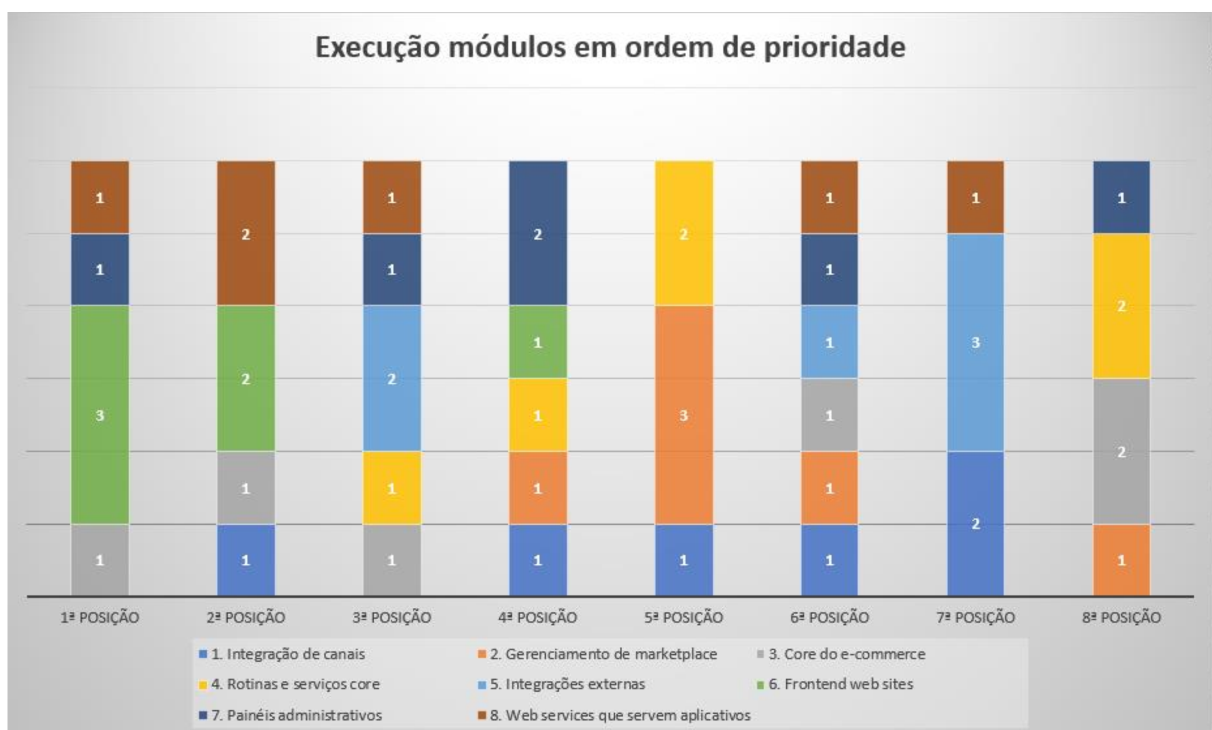
6 ESTRATÉGIA PARA TRANSFORMAÇÃO DA PLATAFORMA

A estratégia para transformação da plataforma de arquitetura monolítica para microsserviços consiste em uma execução em partes, realizando a transformação em módulos, implementando e publicando em produção cada um deles. Os módulos transformados estarão plugados na plataforma antiga, até o momento em que todos os módulos estarão rodando na nova arquitetura e o monolito seja extinto.

6.1 ORDEM DE IMPLEMENTAÇÃO

A primeira definição que se deve ter é a ordem de execução da transformação dos módulos, para que a transformação seja vantajosa desde o início, para clientes e operadores. Para apoiar essa definição, foi aplicada uma pesquisa (Apêndice A) entre onze *stakeholders*, onde apenas seis responderam, que atuam como referências na plataforma atualmente, onde uma das perguntas (número 6) questionou sobre a ordem de execução da transformação dos módulos com a atribuição de uma justificativa para tal decisão. Houve algumas respostas em comum que possibilitaram a geração de um gráfico para representação, exibido na Figura 12.

Figura 12 - Respostas para decidir sobre a estratégia de transformação dos módulos.



Fonte - criação própria

O gráfico da Figura 12 está organizado da seguinte maneira: o eixo horizontal representa a posição de prioridade na transformação. Cada cor representa um módulo e estão listados na parte de baixo do gráfico como legenda. Na faixa vertical são contabilizados de forma agrupada a quantidade de vezes em que o módulo recebeu voto para ser executado naquela posição.

Com base nas respostas da pesquisa, foi definida a seguinte ordem de transformação dos módulos:

1º - *Frontend web* sites. A estratégia para este módulo ser extraído primeiro, é por ser uma camada em que se recebe muitas customizações. Com isso, no momento em que houver um microsserviço com esta camada, haverá mais facilidade de publicar as alterações em produção, já que ela vai estar desacoplada e de uma forma mais independente do restante do monolito. Além disso, a maneira que está implementada hoje, não está com o reaproveitamento de código necessário, tornando toda alteração muito trabalhosa.

2º - *Web services* que servem aplicativos. O motivo deste módulo estar no topo é que frequentemente ele sofre modificações, muito parecido com o que acontece no módulo *frontend web* sites. Além disso, no momento está bem desorganizado, retornando muito mais dados que os aplicativos precisam, além de, em alguns casos, ter o mesmo dado em diferentes campos.

3º - Integrações externas. Nenhum software pode ser intimamente dependente de uma integração com fornecedor externo, sendo assim seria bom que esse módulo fosse removido o quanto antes do monolito. Desta forma, seria fornecida uma maneira de intercambiar os fornecedores e suas integrações de modo que não afetaria /dependeria do *core* do sistema.

4º - Painéis administrativos. Os painéis administrativos estão muito defasados no momento, não são intuitivos e são motivos de dificuldades para o time de operações da plataforma. A reescrita deste módulo é complexa, pois existem muitas frentes utilizando inúmeras funcionalidades, desde a configuração do conteúdo que é mostrado nos *e-commerces*, até o acompanhamento dos pedidos realizados. Pela sua alta complexidade não está no topo e pelo seu alto acoplamento com o *core* da plataforma, ele não está no final.

5º - Gerenciamento de *marketplace*. Este módulo está fortemente ligado ao *core* do *e-commerce*, utilizando serviços para criação e manuseio de pedidos e produtos. É interessante realizar neste momento, pois aqui existe uma grande variedade de funcionalidades e é mais um dos módulos que serão extraídos do *core* da ferramenta.

6º - Integração de canais. Nesta posição não tivemos nenhum módulo se destacando nas respostas da pesquisa, mas a integração de canais foi escolhida para esta ordem pelos mesmos motivos do módulo de gerenciamento de *marketplace*, além de deixarmos para as últimas posições os módulos que fazem parte do *core* da plataforma.

7º - Rotinas e serviços *core*. Este módulo ficou nesta posição por se tratar de funcionalidades que não são tão customizadas e a sua transformação não teria muitos ganhos se fizéssemos antes.

8º - *Core* do *e-commerce*. Aqui temos o último módulo e mais complexo de se transformar. Por ele estar na última posição, não teremos tanta dificuldade de realizar a transformação, já que todos os outros módulos já teriam sido extraídos do monolito e é exatamente por este motivo que está por último.

A partir deste momento, a estratégia para transformação está definida, as tecnologias macro e a ordem de execução foram analisadas e escolhidas baseadas em técnicas e na pesquisa elaborada.

6.2 MÓDULO SELECIONADO PARA IMPLEMENTAÇÃO

Até o atual momento a transformação de monolito para microsserviços considera a plataforma completa, incluindo todos os módulos em funcionamento no sistema monolítico. Porém, para este trabalho é inviável ser realizada a implementação completa da solução, já que é necessária uma grande equipe atuando em alguns anos no projeto. Com isso, um único módulo da plataforma foi selecionado para ser implementado na nova solução, respeitando a arquitetura proposta no capítulo anterior.

Para a seleção do módulo a ser implementado foram considerados dois fatores: tempo para execução e importância das funcionalidades contidas. Para parametrizar a importância de cada módulo foi utilizado a ordem construída neste capítulo (Figura 12) e a partir de cada módulo foi feita uma estimativa superficial de implementação.

Na classificação como primeiro módulo a ser implementado está a camada de *frontend* dos *web* sites. Para extrair esse módulo deve ser considerada a implementação da responsividade mobile dos sites, boas práticas para as páginas serem indexadas no Google, além de considerar a implementação de oito *e-commerces*. Por toda essa complexidade já pode-se descartar a escolha deste módulo.

Por segundo, chega o módulo *web services* que serve os aplicativos. Este módulo foi descartado pela sua utilização estar totalmente ligada aos aplicativos. Para a implementação ser dada como concluída é necessário a alteração em todos os aplicativos, considerando ter impacto em todas as funcionalidades que dependem da API.

O módulo “Integrações externas” está colocado em terceira posição e no momento existem dez integrações com APIs diferentes na plataforma. De forma superficial, estima-se que sejam necessárias 160 horas de um profissional *backend*. Desta forma, este módulo já é descartado por exceder o número de horas disponíveis para implementação neste trabalho.

O próximo colocado são os painéis de administração. Este já é descartado de início pela necessidade de ser aplicado um momento de análise de negócio para reestruturar os sistemas que atualmente são um grande problema no quesito UX (*User eXperience*) e UI (*User Interface*).

Em quinto colocado está o módulo que gerencia o *marketplace*. Ele tem importância significativa já que trata de todas as regras de negócio para recebimento e atualização dos produtos para *marketplace*, além de controlar a criação dos pedidos e realizar o envio dos mesmos ao sistema que notifica os vendedores. Baseado em experiências de implementações feitas, estima-se que 80 horas de um profissional *backend* seja suficiente para implementar todo esse módulo. Os detalhes da estimativa podem ser vistos na Tabela 6. Por se tratar de uma implementação viável para o projeto, este módulo foi escolhido para ser implementado na arquitetura de microsserviços.

Tabela 6 - Detalhamento da estimativa de tempo do módulo *marketplace*.

Tarefa	**Tempo gasto**
Criação dos projetos *marketplace-common*, *marketplace-in* e *marketplace-api* com inclusão de bibliotecas e ferramentas necessárias para funcionamento.	3 horas
Criação de classes DTO para integração de produtos.	3 horas
Criação de *endpoint* e serviço para integração de novos produtos.	15 horas
Criação de *endpoints* e serviços para atualizações de estoque, preço e dimensões.	3 horas
Adequação da plataforma de *e-commerces* para receber os novos produtos e suas atualizações	9 horas
Criação de classes DTO para integração de pedidos.	3 horas
Criação de *listener* e serviço para integração de novos pedidos.	6 horas

Criação de *endpoint*, serviço e *listener* para atualizações dos pedidos.	6 horas
Adequação da plataforma de *e-commerces* para enviar atualizações dos pedidos.	9 horas
Adequação da plataforma de *e-commerces* para receber atualizações dos pedidos.	11 horas
Criação de conta na AWS.	3 horas
Criação de *api Gateway*, *script* Lambda e filas para integração na AWS.	9 horas
TOTAL	**80 horas**

6.3 IMPLEMENTAÇÃO DO MÓDULO DE GERENCIAMENTO DE *MARKETPLACE*

Um dos *e-commerces* hospedados na plataforma atua como *marketplace*, que oferta produtos de diferentes vendedores com contrato fechado com a empresa dona do *marketplace*. Os demais sites hospedados na plataforma são *e-commerces* comuns, que ofertam apenas produtos da própria marca. A estrutura de dados das duas modalidades fica um pouco diferente, mas o código *core* da plataforma foi construído para funcionar em ambos os casos e as regras de negócio de uma modalidade e outra são com base nessa estrutura de dados. O catálogo de produtos do site que é *marketplace* contempla múltiplas representações de depósitos e cada um agrupa os estoques de um vendedor. Os produtos mantêm a estrutura padrão da plataforma, que têm um produto base, tendo dados como nome, descrição, foto e cor. Vinculado no produto base, ficam as variações, que agrupam dados como preço, estoque e dimensões.

Uma das responsabilidades do módulo de gestão de *marketplace* é adequar a estrutura dos produtos dos vendedores para a estrutura padrão da plataforma. Normalmente, o que difere as duas estruturas é a questão das variações, onde existem variações de cores em um mesmo produto base. O que o módulo de *marketplace* faz é readequar o agrupamento das variações, criando 1 produto base para cada grupo de cores que chega do vendedor externo. Exemplo: o produto X chega com 6 variações, nos tamanhos P, M e G, nas cores azul e vermelho. O módulo *marketplace* irá gerar a seguinte estrutura para a plataforma: produto X1 na cor vermelha, com 3 variações de tamanho P, M e G; produto X2 na cor azul, com 3 variações de tamanho P, M e G. A identificação dos produtos se mantém sem impacto, já que é isso que liga o produto da plataforma com o vendedor para recebimento de atualizações de preço, estoque e dimensões.

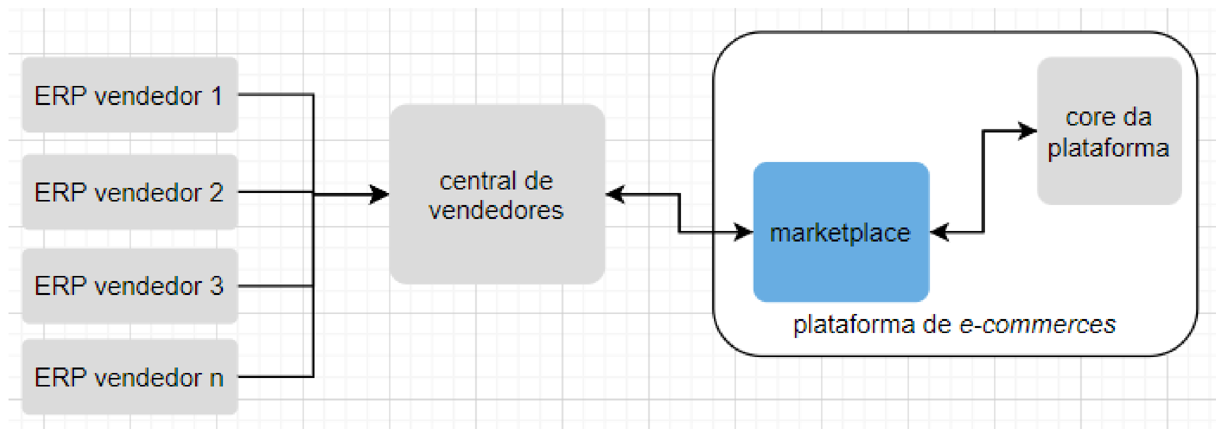
A camada que gerencia o *marketplace* consiste em receber dados, tratar os mesmos e enviá-los adiante. Pode-se dizer que fica localizado entre o *e-commerce* e o sistema de central de vendedores. Esse sistema faz a conexão com o ERP dos vendedores para recebimento de produtos, estoques e preços, além de reunir todos os pedidos criados no *marketplace* e servir para a moderação dos produtos pelo time operacional antes de serem inseridos na plataforma de *e-commerces*.

As integrações que seguem a direção do sistema de central de vendedores para a plataforma de *e-commerces* são: envio e atualização completa dos produtos, compondo atualização de preço, estoque e dimensões. Além dos produtos, existe o envio dos dados de vendedor e atualização da situação dos pedidos, como faturamento, despacho e sinalização de entrega.

Na direção contrária, em integrações que seguem da plataforma de *e-commerces* para o sistema de central de vendedores são: envio de novos pedidos e atualizações de cancelamento e confirmação de pagamento dos mesmos.

A Figura 13 ilustra como fica localizada a camada de *marketplace* inserida no monolito.

Figura 13 - Camada atual do *marketplace*.



Fonte - criação própria

A implementação do módulo para gerenciamento de *marketplace* não consiste apenas em criar os serviços novos com a nova arquitetura, mas também é necessário preparar a plataforma monolítica para possibilitar a integração dessas duas aplicações. Para tanto, a integração deve ser construída para ter o menor impacto possível no momento em que o módulo da plataforma a ser integrada for transformada para microsserviços também. Para isso, o protocolo de comunicação para enviar e receber os dados entre as duas partes deve seguir o

formato em que dois microsserviços em uma mesma aplicação se comunicam, dessa forma, na reescrita dos demais módulos a forma de comunicação não tem impacto.

6.3.1 Protocolo de comunicação entre microsserviço e monolito

Uma questão bastante importante de entender para definir o protocolo de comunicação é que eles possuem dois eixos. Microsoft (2022) descreve a definição deles:

> - Protocolo síncrono. HTTP é um protocolo síncrono. O cliente envia uma requisição e espera uma resposta do serviço.
> - Protocolo assíncrono. Outros protocolos como AMQP (um protocolo compatível com vários sistemas operacionais e ambientes de nuvem), usam mensagens assíncronas. O código do cliente ou o remetente da mensagem geralmente não espera uma resposta. Ele apenas envia a mensagem como ao enviar uma mensagem para uma fila do RabbitMQ ou para qualquer outro agente de mensagens.

Tendo ciência dessa classificação, fica simples definir qual protocolo utilizar. Para comunicações que necessitam de resposta, o protocolo HTTP foi utilizado. Caso não tenha necessidade de retorno, aí a utilização tradicional, através de mensageria.

6.3.2 Adaptações na plataforma de *e-commerces*

A plataforma de *e-commerces* precisa sofrer ajustes para possibilitar a integração com o novo microsserviço e desacoplar o antigo módulo.

Tendo em vista os dois eixos para comunicação, pode-se concluir que o recebimento de dados na plataforma se dará pelo protocolo síncrono. Isso porque o sistema de central de vendedores precisa do retorno dessas requisições, para identificar se os envios deram sucesso ou erro e comunicar os vendedores.

Para recebimento dos dados foram criados novos *endpoints* na estrutura do *core* da aplicação. Esses *endpoints* executam métodos responsáveis por armazenar os dados recebidos, métodos que precisaram ser criados novos também. Não foram aplicadas nenhuma regra de negócio e validações de *marketplace* nessas implementações, já que as regras de *marketplace* ficam fora da plataforma na nova estrutura. Com isso, os *endpoints* criados podem servir para qualquer *e-commerce* hospedado na plataforma, já que as regras aplicadas são a nível de *e-commerce* e não de *marketplace*.

O envio dos dados foi implementado utilizando o protocolo assíncrono, tendo em vista que a plataforma de *e-commerces* não precisa do retorno no envio dessas informações. Um exemplo dessa dinâmica é no envio de novos pedidos, onde a plataforma realiza o envio e aguarda o recebimento das próximas atualizações. Caso ocorra qualquer erro no processamento

de um registro da mensagem da fila, o erro deve ser sinalizado e esse procedimento é de responsabilidade do sistema de mensageria.

A implementação na plataforma de *e-commerces* não pode ser compartilhada por diretrizes da empresa dona da ferramenta.

6.3.3 Integração por mensageria

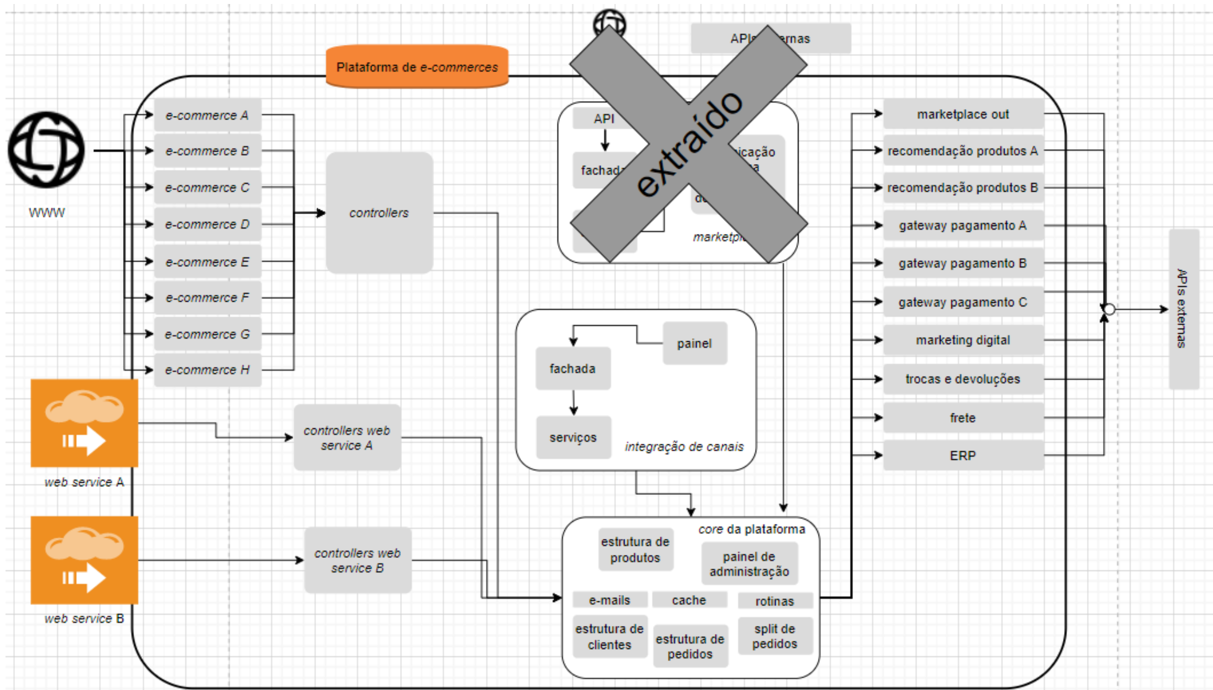
Para cada integração foi criada uma fila na AWS (*Amazon Web Services*) a fim de compartilhar os dados com um ouvinte no microsserviço. Para inserir as mensagens na fila, normalmente os desenvolvedores inserem na aplicação uma biblioteca da AWS, que trás métodos prontos para esse objetivo. Porém, neste cenário, é incluída uma dependência com a plataforma AWS, e se algum dia for necessário mudar de provedor de nuvem será necessário a manutenção dessa dependência. A maneira ideal para este cenário é utilizar um API Gateway que recebe requisições HTTP e executa um *script* Lambda responsável por se comunicar com as filas. Tanto o API Gateway quanto o Lambda estão hospedados no provedor AWS. Dessa forma, em caso de mudança de provedor, é necessário apenas recriar o API Gateway e o Lambda, sem que haja alterações em código.

Um grande benefício que a utilização de mensageria traz consigo é a independência entre as camadas. Caso o serviço ouvinte esteja indisponível em qualquer momento, o serviço que insere as mensagens na fila não sofre nenhum problema. O que acontece é que a mensagem cai na fila e fica parada até o serviço ouvinte voltar a ficar estável.

Definidos os formatos para transferência dos dados, foi necessário apenas alterar na plataforma lugares centrais onde é feito o cadastro de novos pedidos e onde se recebe atualizações de situação nesses pedidos. Antes da nova arquitetura, essas camadas se comunicavam com o módulo de *marketplace* interno, mas a partir de agora, realizam requisições para o API Gateway que foi criado.

Após os ajustes na plataforma, já é possível remover o módulo para gestão de *marketplace*, como ilustra a Figura 14.

Figura 14 – Detalhamento de módulos sem *marketplace.*



Fonte - criação própria

6.3.4 Implementação dos microsserviços para gestão de *marketplace*

Conforme construído no capítulo 5, foram criados dois microsserviços para organizar o módulo *marketplace*.

O microsserviço *marketplace* API, como foi descrito anteriormente, será a porta de entrada para recebimento dos dados. É onde fica localizada todas as requisições recebidas. Essa camada não tem inteligência nenhuma, apenas recebe os dados e direciona para o serviço seguinte. O motivo para deixar essa porta de entrada em um microsserviço só, é pela separação de responsabilidades, onde pode-se aplicar regras de segurança, tratamento de erros para retorno e fazer rastreio de todas as requisições que chegarem.

Internamente, no *marketplace* API, o código está organizado em 4 pacotes. *Controller*: agrupa todos *endpoints* da API, parâmetros recebidos, dados para respostas das requisições e caminhos de identificação; *Service*: essa camada normalmente recebe as regras de negócio da aplicação, mas como aqui não tem nenhuma contida, servirá apenas para boas práticas e se algum dia necessitar de validação, a camada já está posta; *Proxy*: aqui ficam apenas arquivos do tipo interface, marcadas por usarem anotações da ferramenta OpenFeign, que serve de apoio para integrações entre dois serviços que utilizam padrão REST. As interfaces são responsáveis por receber os dados e encaminhá-los para outro serviço; *Exception:* localiza-se aqui todos os

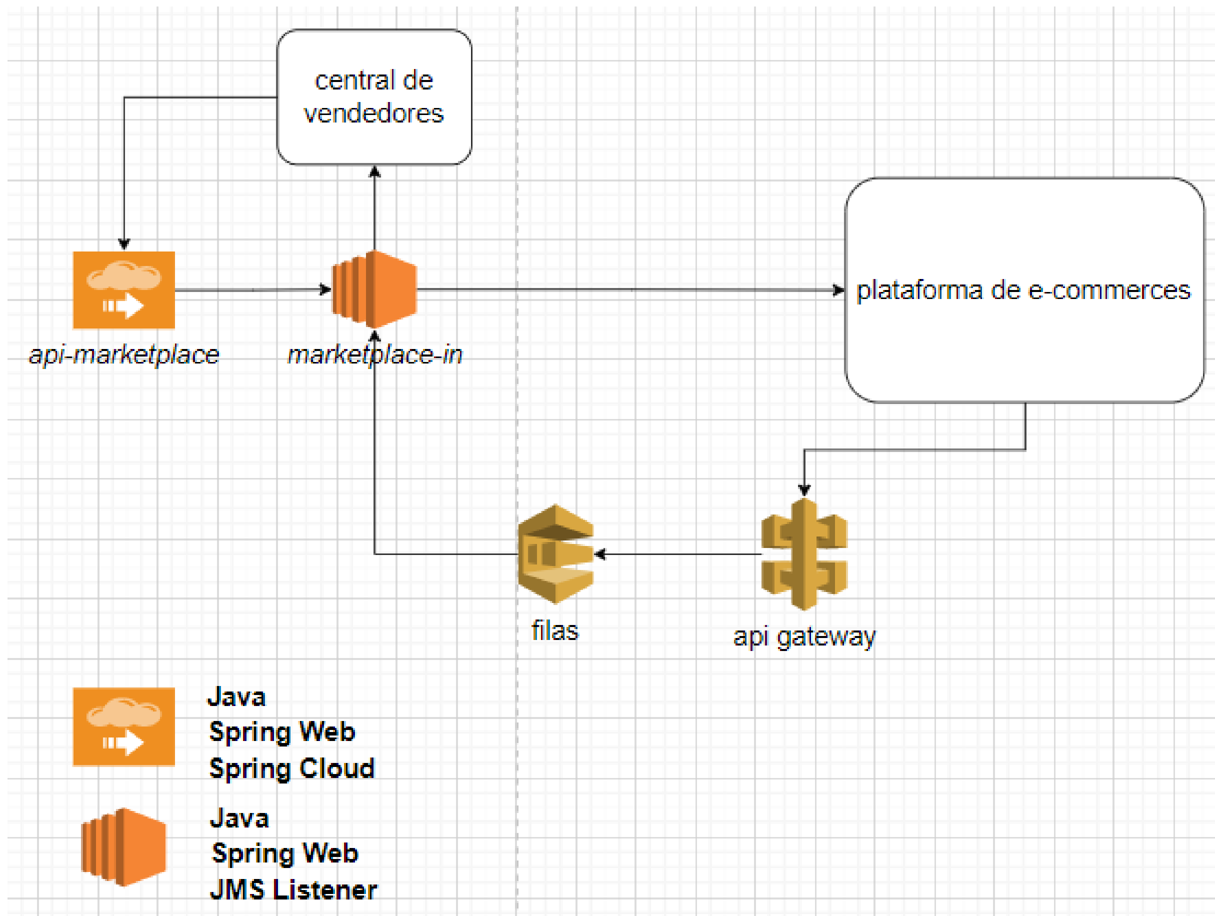
arquivos para interceptar os erros e tratá-los antes de mandar como resposta da requisição. Informações sensíveis da aplicação são removidas e somente dados como o tipo do erro e uma mensagem sucinta são retornados.

O segundo microsserviço, *marketplace in*, é onde ficam as regras de negócio, onde os dados são tratados para se adequarem ao modelo que a plataforma de *e-commerces* espera. Ainda nesse segundo serviço, ficaram os métodos que escutam as movimentações de registros nas filas de dados que a plataforma envia. Os métodos recebem os dados na estrutura da plataforma de *e-commerces* e tratam esses dados para serem enviados para o sistema de central de vendedores. Com isso, esta parte da implementação conta com duas integrações de sistemas: plataforma de *e-commerces* e sistema de central de vendedores.

O código do *marketplace in* está organizado com 6 pacotes. *Controller*: mesmos tipos de arquivos contidos no microsserviço *marketplace* API; *Service*: arquivos que tratam de regras de negócio. *Config*: arquivos de configuração. Aqui ficou apenas 1 arquivo, que serve para configurações de escuta das integrações de mensageria por filas; *Listener*: arquivos onde se inicia os processamentos de movimentações em filas. Qualquer registro adicionado em filas são escutados pelos arquivos postos nesse pacote; *Replication*: arquivos que fazem a conexão com o mundo externo. Aqui ficam implementações para envio de dados para a plataforma de *e-commerces* e para o sistema de vendedores. *Utils*: métodos utilitários de apoio como formatadores de dados, remoção de caracteres especiais, entre outros pequenos procedimentos.

O desenho da arquitetura da nova aplicação é exibido na Figura 15, onde pode ser visto como os dados trafegam entre as camadas.

Figura 15 – Desenho da arquitetura microsserviços *marketplace*.



Fonte - criação própria

Os dados que o *marketplace* recebe do sistema de central de vendedores, são recebidos no microsserviço API e processados no microsserviço que trata as regras. Os dados trafegam através de classes DTO (*Data Transfer Object*) entre um serviço e outro. Para não ser necessário ter essas classes duplicadas nas duas camadas, foi criado um projeto Java para servir como uma biblioteca entre elas. Esta biblioteca tem todas as classes DTO que seriam duplicadas e é importada como dependência entre os dois serviços através do Maven.

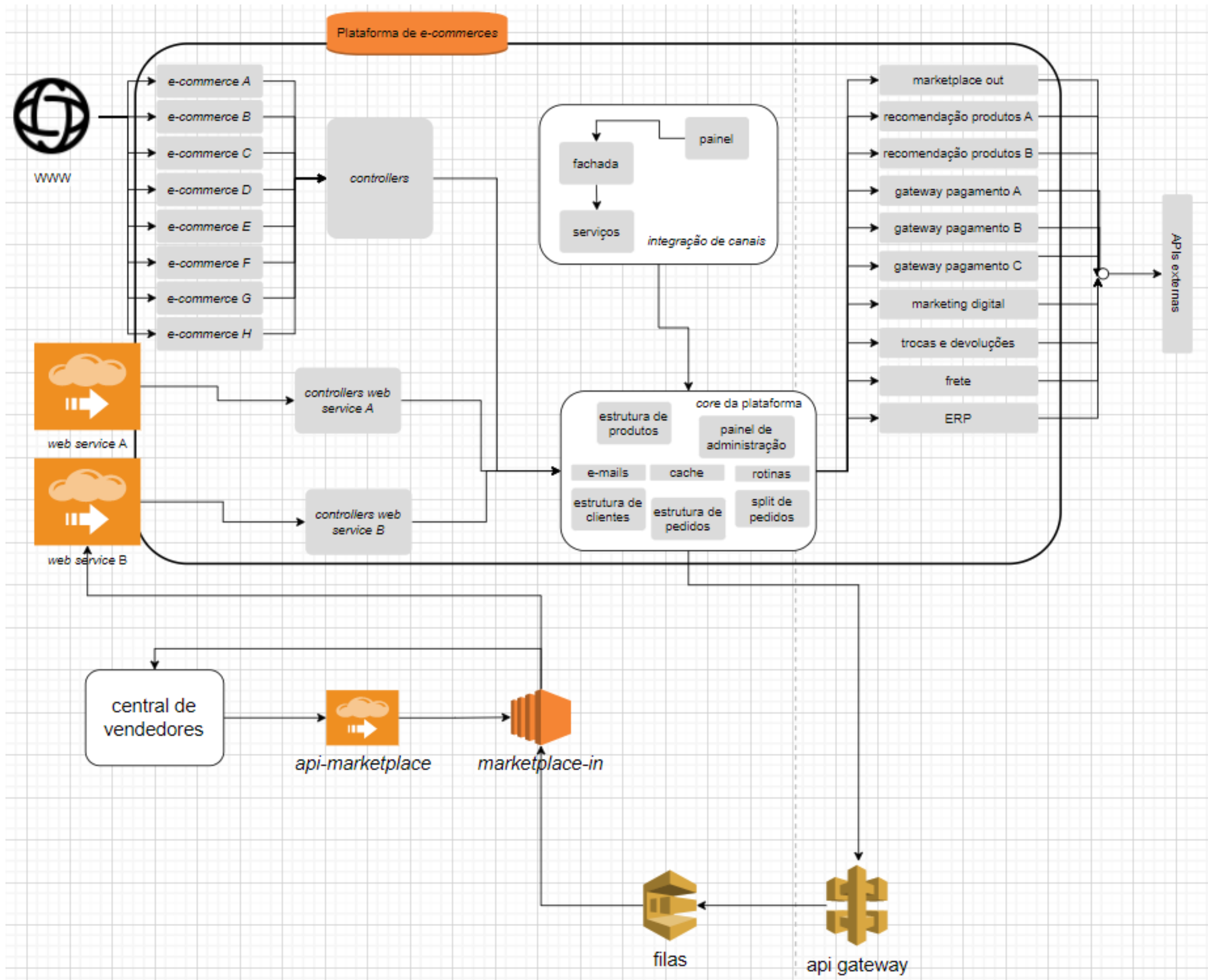
A estrutura foi criada utilizando Java na versão 17 e não teve necessidade de utilização de banco de dados. Para apoiar o Java foi aplicado o Spring na versão 2.6. O Spring ingressou no projeto através do Spring Boot, que serve como facilitador para configurar um projeto Java. Auxilia na configuração e na injeção de dependências entre as classes de serviço (DEVMEDIA, 2015). A utilização do Spring Boot, além de todo o framework Spring, traz consigo uma infinidade de ferramentas úteis para a implementação das aplicações. No projeto criado existem três exemplos dessas ferramentas sendo utilizadas:

- JMS *Listener:* ferramenta que tem capacidade de processar mensagens de filas. Serve como um ouvinte de mensagens (ORACLE, 2006, tradução nossa).
- Spring Web: permite a exposição de *endpoints* de API no padrão REST.
- Spring Cloud: permite a integração entre microsserviços internos simplificando o processo de integração de dois serviços. Para tal integração foi utilizado o OpenFeign que está incluído no Spring Cloud.

A nova estrutura de microsserviços para *marketplace* tem seu código armazenado no GitHub em repositórios públicos. O acesso para visualização do código construído está disponível no link https://github.com/jonascolling.

Por fim, a estrutura completa do novo microsserviço se comunicando com a plataforma de *e-commerces*, pode ser representada pela Figura 16. A entrada de informações para os *e-commerces* acontece pelo *web service* B para chegar até a camada de serviços no *core*. Para realizar o envio dos dados, o *core* da plataforma faz conexão com o API Gateway para os dados chegarem até a fila e consequentemente o microsserviço *marketplace in* irá processar os dados.

Figura 16 - Estrutura completa, *marketplace* e plataforma.



Fonte - criação própria

7 CONSIDERAÇÕES FINAIS

Pode-se compreender, através da literatura, as principais características dos tipos de arquitetura monolítica e microsserviços. Os benefícios e complexidades que elas trazem consigo na utilização em aplicações de software. Além disso, em trabalhos relacionados é possível perceber que a comunidade enfrenta limitações semelhantes às listadas neste trabalho. Limitações originadas por sistemas de arquitetura monolítica, de grande complexidade, com características parecidas às deste trabalho.

Foi possível compreender que as aplicações de arquitetura em microsserviços são mais complexas para trabalhar e necessitam de equipe qualificada para evoluir e manter o sistema. Por outro lado, a arquitetura de microsserviços chega para quebrar as limitações que os monolitos trazem. Baixo acoplamento de módulos, independência entre equipes, melhor aproveitamento de recursos e mais rapidez na publicação de novas funcionalidades são os principais pontos que a comunidade traz como sendo as vantagens da arquitetura moderna.

Com base no contexto de utilização da plataforma de *e-commerces*, é possível afirmar que a migração completa da plataforma, em apenas uma publicação, é inviável pela quantidade de mudanças que ocorrem em um curto período de tempo. Com apoio em uma pesquisa (Apêndice A) aplicada com *stakeholders*, foi possível construir uma estratégia de migração, onde a evolução da plataforma não para durante a execução e são feitas entregas para cada módulo extraído do monolito, sendo possível usufruir da nova arquitetura logo nos primeiros módulos reescritos. A partir da análise feita nas respostas obtidas na pesquisa, percebe-se algumas características em comum para propor a ordem de execução dos módulos: baseado nos possíveis resultados para a camada de negócio, defasagem técnica e simplificação no processo de reescrita, deixando para o final os módulos com nível de acoplamento mais alto.

Para implementação foi escolhido apenas 1 dos módulos presentes na plataforma para ser reescrito. Esta fase foi muito importante para expor pontos de atenção que se deve ter na implementação dos demais módulos. Itens como: adequação da plataforma para integrar com a nova estrutura. A complexidade aumenta conforme está o nível de acoplamento do módulo no monolito. Outro ponto é a forma de integração de dados que entram e saem da plataforma de *e-commerces*. Aqui o importante já é considerar como será a transferência de dados no momento em que o ponto de contato na plataforma já estiver com microsserviço, para minimizar retrabalhos nas reescritas dos módulos seguintes.

O desenvolvimento ocorreu entre o período do dia 04/04/2022 até 23/05/2022. Utilizou o planejamento elaborado no capítulo 4, onde foram definidos os microsserviços e as tecnologias macro a serem utilizadas. O esforço entre desenvolvimento e testes de desenvolvedor foi de 96 horas. Entre essas horas estão contabilizadas a criação do repositório no GitHub e a criação da conta na AWS.

Para trabalhos futuros a orientação é seguir com os próximos passos desta reescrita. A organização inicial se dá pelos seguintes tópicos:

1. Aplicar camada de segurança no *marketplace* API: definir isso logo no início é de suma importância pelo fato de diminuir a chance de retrabalho em alguma reestruturação. Além de que é importante ter a segurança reforçada antes de ter publicado essa nova estrutura em ambiente produtivo.
2. Criar infraestrutura para ambientes e automação nos processos de *deploy*: para ser possível usufruir da nova camada, é necessário criar toda a infraestrutura para hospedagem em ambientes produtivos e de testes. Em complemento a esta parte, construir a esteira de *deploys* para publicações automatizadas nos ambientes disponíveis.

O desenvolvimento deste trabalho tinha como proposta responder a seguinte questão: Com a aplicação da nova arquitetura, é possível diminuir a complexidade e o acoplamento entre *squads* nas entregas de novas funcionalidades no *e-commerce* já existente?

Como resposta, não é possível comprovar nenhuma melhoria nos processos diários na evolução da plataforma, já que não foi aplicado o uso da nova estrutura na empresa, por conta de ainda faltar a implementação de uma camada de segurança e ter os ambientes de teste juntamente com ferramenta para *deploy* disponíveis. Mas é possível observar que o módulo extraído do monolito já fica desprendido e a evolução desta parte tende a ser mais simples, pela utilização de ferramentas modernas e pelas publicações não terem impacto na plataforma de *e-commerces*. A organização da *squad* que daria manutenção a essa nova camada também deve ficar mais independente do restante dos times, já que está na estrutura isolada e a as regras de negócio para integrações de *marketplace* estão todas incluídas no microsserviço. A plataforma fica com a responsabilidade de controlar apenas questões que correspondem aos *e-commerces*.

REFERÊNCIAS BIBLIOGRÁFICAS

AL-DEBAGY, Omar; MARTINEK, Peter. ***A Comparative Review of Microservices and Monolithic Architectures***. IEEE *18th International Symposium on Computational Intelligence and Informatics*, 2018.

AWS. **O que são microsserviços?**. Disponível em: <https://aws.amazon.com/pt/microservices>. Acesso em: out/2021.

AZEVEDO, Yasmin. **Você conhece o ecossistema que engloba marketplace?**, 2021.Disponível em: <https://vtex.com/pt-br/blog/marketing/voce-conhece-todo-ecossistema-que-engloba-o-marketplace>. Acesso em nov/2021.

BALALAIE, Armin et al. ***Microservices migration patterns***, 2018.

BALALAIE, Armin; HEYDARNOORI, Abbas; JAMSHIDI, Pooyan. ***Microservices Architecture Enables DevOps****: An Experience Report on Migration to a Cloud-Native Architecture*, 2016.

BENNETT, K. H.; RAJLICH, V. T. ***Software maintenance and evolution***: *a roadmap. In: ACM. Proceedings of the Conference on the Future of Software Engineering*, 2000.

CARNELL, John. ***Spring Microservices in Action***, 2021.

CHAN, Mike. ***Microservices vs Monolithics***: *What's the Right Architecture for Your Software*, 2017. Disponível em: <https://www.thorntech.com/microservices-vs-monoliths-whats-right-architecture-software>. Acesso em: out/2021.

CHIKOFSKY, E. J.; CROSS, J. H. ***Reverse Engineering and Design Recovery***: *A taxonomy*. IEEE Software, 1990.

DEVMEDIA. **DDD**: *Domain-Driven Design* com .NET, 2009. Disponível em: < https://www.devmedia.com.br/ddd-domain-driven-design-com-net/14416 >. Acesso em mar/2022.

DEVMEDIA. **Projeto Jigsaw**: Desenvolvimento modularizado no Java 9, 2016. Disponível em: <https://www.devmedia.com.br/projeto-jigsaw-desenvolvimento-modularizado-no-java-9/34406>. Acesso em out/2021.

EVANS, Eric. ***Domain Driven Design***: Atacando as complexidades no coração do software, 2016.

FRITZSCH, Jonas et al. ***Microservices Migration in Industry****: Intentions, Strategies, and Challenges*, 2019.

FULLCYCLE. **O que é DDD**: *Domain Driven Design*, 2019. Disponível em: < https://fullcycle.com.br/domain-driven-design>. Acesso em mar/2022.

GRUBB, P.; TAKANG, A. A. ***Software maintenance: concepts and practice***, 2003

HASSAN, Sara; ALI, Nour; BAHSOON, Rami. ***Microservice Ambients****: An Architectural Meta-modelling Approach for Microservice Granularity*, 2017.

IBM. ***ESB (Enterprise Service Bus)***, 2021. Disponível em: https://www.ibm.com/cloud/learn/esb. Acesso em: out/2021

IBM. **O que É SOAP?**, 2020. Disponível em: <https://www.ibm.com/docs/pt-br/integration-bus/10.0?topic=ssmkhh-10-0-0-com-ibm-etools-mft-doc-ac55770--html>. Acesso em: nov/2021.

JUNIOR, Vanderlei *et al*. ***Design Science Research Methodology As Methodological Strategy for Technological Research***. Revista Espacios, 2017. vol. 38.

KAINZ, Alexander. ***Microservices vs. Monoliths***: *An Operational Comparison*. Disponível em: <https://thenewstack.io/microservices-vs-monoliths-an-operational-comparison>, 2020. Acesso em: out/2021.

KAZANAVICIUS, Justas; MAZEIKA, Dalius. ***Migrating Legacy Software to Microservices Architecture***. *Open Conference of Electrical, Electronic and Information Sciences*, 2019.

KEMPF, Rachel. ***Monolithic Applications x Microservices***, 2021. Disponível em: https://www.azion.com/en/blog/monolithic-vs-modern-applications. Acesso em: nov/2021.

KURYAZOV, Dilshodbek; JABBOROV, Dilshod; KHUJAMURATOV, Bekmurod. ***Towards Decomposing Monolithic Applications into Microservices***. IEEE *14th International Conference on Application of Information and Communication Technologies*, 2020.

LAURETIS, Lorenzo De, ***From Monolithic Architecture to Microservices Architecture***.

IEEE *International Symposium on Software Reliability Engineering Workshops*, 2019.

LI, Chia-yu; MA, Shang-Pin; LU, Tsung-Wen. ***Microservice Migration Using Strangler Fig Pattern****: A Case Study on the Green Button System*, 2020.

FOWLER, Martin. ***Strangler Fig Application***, 2004. Disponível em: https://martinfowler.com/bliki/StranglerFigApplication.html. Acesso em out/2021.

GOUIGOUX, Jean-Philippe; TAMZALIT, Dalila. ***From Monolith to Microservices***: *Lessons Learned on an Industrial Migration to a Web Oriented Architecture*, 2017.

GOUIGOUX, Jean-Philippe; TAMZALIT, Dalila. ***“Funcional-first” recommendations for beneficial microservices migration and integration***, 2019.

MAZZARA, Manuel et al. ***Microservices****: Migration of a Mission Critical System*, 2018.

MICROSOFT. **Comunicação em uma arquitetura de microsserviço**, 2022. Disponível em: < https://docs.microsoft.com/pt-br/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture>. Acesso em mai/2022.

MICROSOFT. ***Monolithic Applications***, 2021. Disponível em: <https://docs.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications>. Acesso em: out/2021.

MICROSOFT. **Monolítico para microsserviços usando o design controlado por domínio**, 2021. Disponível em: <https://docs.microsoft.com/pt-br/azure/architecture/microservices/migrate-monolith>. Acesso em out/2021.

ORACLE. **JMS Listener**, 2006. Disponível em: < https://docs.oracle.com/cd/E19944-01/819-4520/JmsListener.html>. Acesso em mai/2022.

OSBORNE, W.M.; CHIKOFSKY, E.J. ***Fitting Pieces to the Maintenance Puzzle***. IEEE Software, v.7, 1990.

MUNIZ, Antonio, **Jornada DevOps:** unindo cultura ágil, Lean e tecnologia para entrega de software de qualidade, 2019. vol. 1.

PRASANDY, Teguh *et al*., ***Migrating Application from Monolith to Microservices***. IEEE *International Conference on Information Management and Technology*, 2020.

PRESSMAN, Roger, **Engenharia de Software**. Uma Abordagem Profissional, 2016. 8º ed.

PRODANOV, Cleber Cristiano; FREITAS, Ernani Cesar de. **Metodologia do trabalho científico:** métodos e técnicas da pesquisa e do trabalho acadêmico. 2º ed. Novo Hamburgo: Feevale, 2013.

SARITA; SEBASTIAN, Sunil, ***Transform Monolith into Microservices using Docker***. IEEE *International Conference on Computing, Communication, Control and Automation*, 2017.

SOMMERVILLE, Ian, **Engenharia de Software**. Editora Pearson, 2019. 10º ed.

TOTVS. **Arquitetura REST**: Saiba o que é e seus diferenciais, 2020. Disponível em: <https://www.totvs.com/blog/developers/rest>. Acesso em nov/2020.

VALEPUCHA, Vitor; FLORES, Pamela. ***Monoliths to microservices - Migrations Problems and Challenges***: *A SMS*, 2021.

VILLAMIZAR, M. et al. ***Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube***. *10th Computing Colombian Conference*, 2015.

APÊNDICE A - PESQUISA SOBRE EVOLUÇÃO DA PLATAFORMA DE *E-COMMERCES*

Autor do questionário e do projeto acadêmico: Jonas Rafael Colling.

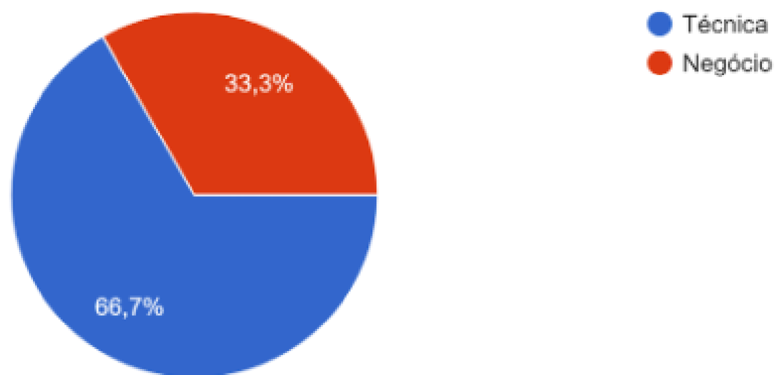
Este questionário tem como objetivo auxiliar nos próximos passos de um projeto acadêmico do curso de Ciência da Computação.

O projeto propõe a evolução tecnológica de uma plataforma de *e-commerces* já em uso por uma empresa de nome não mencionado, localizada no Brasil.

Perguntas:

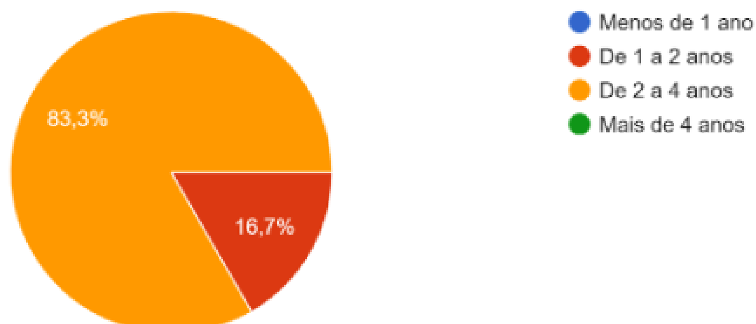
1. Qual sua área de atuação?

- Técnica.
- Negócio.



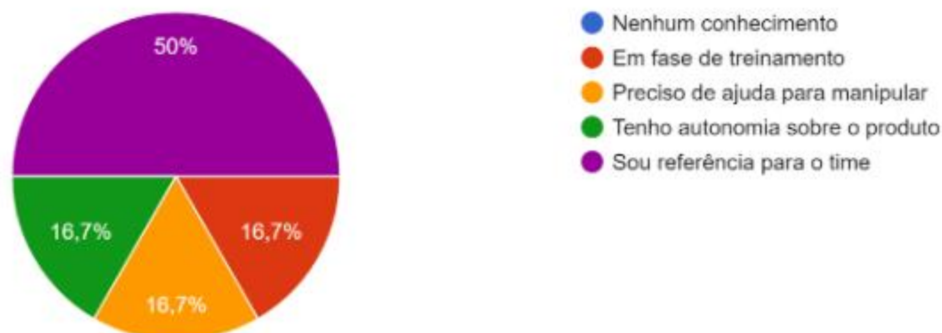
2. Você atua há quanto tempo diretamente na plataforma de *e-commerces*?

- Menos de 1 ano.
- De 1 a 2 anos.
- De 2 a 4 anos.
- Mais de 4 anos.



3. Na sua avaliação, qual seu domínio funcional/técnico no produto?

- Nenhum conhecimento.
- Em fase de treinamento.
- Preciso de ajuda para manipular.
- Tenho autonomia sobre o produto.
- Sou referência para o time.



4. Na sua opinião, a plataforma de gerenciamento dos *e-commerces* utilizada atualmente, de alguma forma, dificulta o lançamento de novas funcionalidades ou até mesmo o alcance dos objetivos da empresa para este canal de vendas? Se sim, comente algumas características e dificuldades que geram esse tipo de problema.

Respondente 1	Não lembro de alguma limitação que tivemos por conta da plataforma que nos impossibilitou a venda de produtos. Talvez não o formato ideal de venda, mas pelo que me recordo nunca tivemos problemas de não poder vender alguma coisa.
Respondente 2	Sim, penso que a plataforma atual, é vista de cima como um monólito preso a tecnologias, frameworks e regras pouco customizáveis. É muito robusta no que entrega, porém tem um grande e oneroso tempo de build para o projeto. Hoje em dia o tempo vem sendo muito necessário para gerar entregas de valor a todo o momento, a plataforma dificulta uma entrega contínua, visto sua robustez. Acredito também, que ao termos um suporte a plataforma que é muito distante do dia a dia do e-commerce, faz com que percamos alguns "*improvements*", "*fix*" e até "*features*" usáveis para o contexto.
Respondente 3	Sim, hoje o fato de o projeto ser um grande monolito torna todo o processo de testes e deployment mais lento e cansativo pois é necessário revalidar todo o projeto mesmo que não uma parte dele não tenha sido modificada. No que tange as novas funcionalidades, todo o processo de desenvolvimento é mais crítico pois é necessário mensurar quais os outros

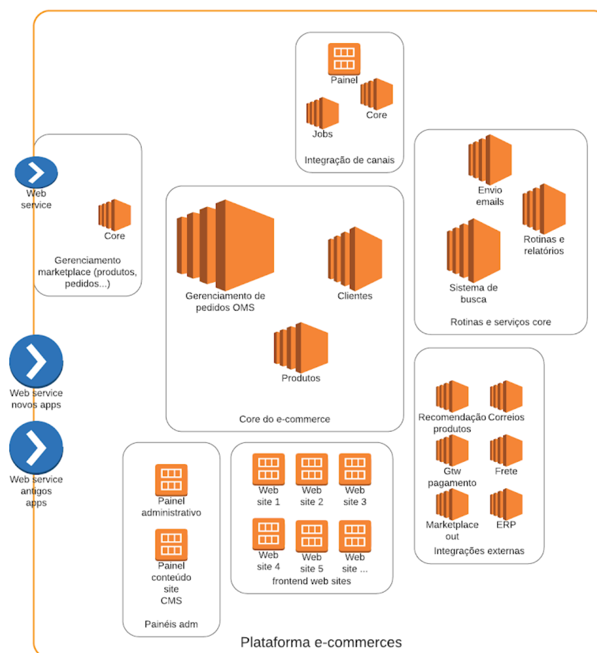
	pontos do código podem ser afetados pelo código novo, sem falar que algumas vezes é alterado algum código de um lado e acaba impactando outra parte do projeto de forma inesperada. Manutenção e atualização ficam mais difíceis também, pois é necessário atualizar o sistema inteiro e não apenas uma fração dele.
Respondente 4	Sim, devido a estrutura monolítica da aplicação são necessários muitos passos e processos que demandam tempo até que alguma funcionalidade possa ir para produção.
Respondente 5	Prazos apertados, e tempo mal gerido.
Respondente 6	Sim, pois o fluxo de release de uma nova versão da aplicação está atrelada diretamente às entregas de todos os *squads* presentes na empresa. Sendo assim, uma entrega do *squad* X com problemas faz com que a entrega do *squad* Y fique aguardando a solução desses problemas que não tem nenhum tipo de dependência. Ou seja, a entrega de N *squads* pode ser afetada por 1 único bug crítico.

5. Pensando na evolução da plataforma, a fim de subir um degrau na classificação de melhores plataformas de *e-commerces* do país, o que deveria ser modificado/transformado na plataforma atual?

Respondente 1	UX da plataforma em geral, algum módulo melhor para cadastro de produto e CMS.
Respondente 2	Penso que, evoluir mais a parte visual do Backoffice da plataforma. Hoje é uma parte muito customizável, porém, o "UX" para quem trabalha no dia a dia é muito prejudicado pela pouca autonomia e agilidade em alguns processos. Melhorar o desenvolvimento cloud e preparar melhor alguns módulos para serem desacoplados e customizados de uma melhor forma. Com toda certeza melhorar a capacidade de *build* da aplicação, o tempo é muito prejudicado para desenvolvimentos no *backend*.
Respondente 3	Uma mudança de arquitetura para fornecer uma maneira mais prática e eficaz de desenvolvimento e testes, seja mudando para micro serviços ou reestruturando o monolito.
Respondente 4	A automatização de processos e pipelines, *deploy* individualizado de seções/funcionalidades são bons exemplos.
Respondente 5	Melhorar performance, UX mais avançados, e usabilidade mais agradável.
Respondente 6	Deveríamos tentar, de alguma forma, separar as entregas dos diferentes *squads* fazendo com que uma entrega não dependa da outra. Dessa forma cada *squad* poderia criar a sua agenda de releases e o fluxo seria muito mais

	tranquilo para todos, tanto QAs quanto Devs, já que os pacotes subidos seriam menores. Além disso, a rastreabilidade de erros seria muito mais fácil, justamente por conta dos pacotes menores.

6. Na figura abaixo temos os módulos da plataforma atual agrupados por funcionalidades: 1. Integração de canais | 2. Gerenciamento de *marketplace* | 3. *Core* do *e-commerce* | 4. Rotinas e serviços *core* | 5. Integrações externas | 6. *Frontend web* sites | 7. Painéis administrativos | 8. *Web services* que servem aplicativos. Pensando em uma reescrita da plataforma de *e-commerces*, utilizando a estratégia de implementação e entrega parcial, com objetivo de priorizar os módulos mais importantes para o negócio e que mais são afetados em customizações e evoluções, qual seria a ordem dos módulos numerados que você consideraria ideal para ser atendida no projeto de reescrita? Por quê?



Respondente 1	1- Painéis administrativos 2- *Frontend web* sites 3- *Core* do *e-commerce* 4- Integração de Canais 5-Rotinas e serviços *core* 6-*Web services* que servem APPs 7-Integrações externas

<table>
<tr><td></td><td>8-Gerenciamento de marketplace
Acho que essa seria a ordem ideal de acordo com o que parece estar mais para o menos defasado hoje.</td></tr>
<tr><td>Respondente 2</td><td>| 1. Frontend web sites
| 2. Integração de canais
| 3. Painéis administrativos
| 4. Rotinas e serviços core
| 5. Gerenciamento de marketplace
| 6. Integrações externas
| 7. Web services que servem aplicativos.
| 8. Core do e-commerce
Eu pensaria nessa ordem, porque poderíamos ter entregas modulares rodando paralelamente, sem grande interferência de um módulo no outro. Ainda assim, causamos um impacto significativo para a entrega total até o core do e-commerce.</td></tr>
<tr><td>Respondente 3</td><td>Segui a ideia que o core é a parte mais crítica e que apresenta maior complexidade e risco, sendo assim considero ideal deixar esses módulos para o final pois todo o modelo já terá sido testado e valido bem como a equipe já terá adquirido experiência.
Seguiria essa ordem, sendo o número referente ao número do módulo
6 - Separar o front end e o backend é uma grande mudança e desacoplamento, mas que não apresenta grande complexidade comparada aos demais módulos.
8 - Serviços web não estão intimamente ligados ao core do monolito e são frequentemente modificados, sendo assim um bom candidato a ser desacoplado no início da migração.
5 - Nem um software pode ser intimamente dependente de uma integração com fornecedor externo, sendo assim seria bom que esse módulo fosse removido do quantos antes do monolito. Fornecendo assim uma maneira de intercambiar os fornecedores e suas integrações de maneira que não afetaria o/dependeria do core do sistema
7 - Painéis administrativos são em grande parte CRUDS, também não vejo complexidade na remoção desse módulo, porém deixo como quarto módulo pois há grande necessidade de removê-lo do core
Daqui para frente todos envolvem o núcleo do sistema, apenas ordenei de forma que achei ser menos arriscada para o negócio e desenvolvimento.
4 - Rotinas e serviços core podem ser migrados para outras ferramentas de rotina
Para Gerenciamento de marketplace e integração de canais não vejo diferença na prioridade ou complexidade deles
1 e 2 - Ambos estão mais ligados ao core porém ainda podem existir de maneira separada
3 - É o núcleo do projeto inteiro, a parte mais sensível e de alto valor. Creio que seja arriscado mover esse módulo primeiro (sem falar traumatizante) por isso deixaria por último. Assim todo o monolito já foi desmembrado e tornará o core mais enxuto e fácil migração.</td></tr>
<tr><td>Respondente 4</td><td>1. Frontend web sites</td></tr>
</table>

	2. *Web services* 3. Integrações externas 4. Painéis administrativos 5. Gerenciamento de *marketplace* 6. *Core* do *e-commerce* 7. Integração de canais 8. Rotinas e serviços core
Respondente 5	8 3 4 6 2 1 5 7
Respondente 6	*Core* do *e-commerce*, *Frontend web* sites, *Web services* que servem aplicativos, Gerenciamento de *marketplace*, Integração de canais, Painéis administrativos, integrações externas. Acredito que essa seria a ordem ideal pois fazendo os 3 primeiros o impacto para o cliente seria muito grande, pois a correção de bugs e entrega de novas funcionalidades seria muito mais rápida do que com o formato atual. Após isso poderíamos seguir com as outras frentes, que, apesar de importantes, não impactam de tal forma o usuário final.