Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Arquitetura Monolítica vs Microsserviços: uma análise comparativa

Autor: Iasmin Santos Mendes
Orientador: Dr. Renato Coral Sampaio

Brasília, DF
2021

Iasmin Santos Mendes

Arquitetura Monolítica vs Microsserviços: uma análise comparativa

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Dr. Renato Coral Sampaio

Brasília, DF

2021

Iasmin Santos Mendes
Arquitetura Monolítica vs Microsserviços: uma análise comparativa/ Iasmin Santos Mendes. – Brasília, DF, 2021-
84 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2021.

1. Arquitetura de Software. 2. Monolítico. 3. Microsserviços. I. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Arquitetura Monolítica vs Microsserviços: uma análise comparativa

CDU XX:XXX:XXX.X

Iasmin Santos Mendes

Arquitetura Monolítica vs Microsserviços: uma análise comparativa

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 11 de dezembro de 2019:

Dr. Renato Coral Sampaio
Orientador

Dr. Fernando William Cruz
Convidado 1

Dr. John Lenon Cardoso Gardenghi
Convidado 2

Brasília, DF
2021

Dedico este trabalho a minha mãe, por sonhar e me incentivar a chegar até aqui. E ao meu pai, por todo o apoio e suporte dado durante esta caminhada.

Agradecimentos

Aos meus pais pelo imensurável suporte ao longo de toda a minha trajetória.

À minha vó, Solange, por todo o apoio prestado mesmo nas coisas mais simples.

À Universidade de Brasília e ao seu corpo docente pela construção de um excelente curso de graduação.

Em especial, ao querido orientador Renato Sampaio por toda a atenção, paciência e contribuições.

Aos professores Fernando W. Cruz e John Lenon pelas significativas contribuições.

Ao Artur Bersan por todo incentivo, suporte e amizade.

Ao Erlan Cassiano pela disponibilidade e colabaoração.

Ao Paulo Fernandes por me permitir discursar sobre a arquitetura da RuaDois.

Aos professores Carla Rocha, Edson Alves, Fernando W. Cruz, casal Serrano (Milene e Maurício), Paulo Meirelles, Renato Sampaio, Sérgio Freitas e Tiago Alves - alguns já não mais presentes no corpo docente - por toda a dedicação que senti ao longo da minha trajetória acadêmica.

A deus por cada novo dia.

Faça o melhor que puder até saber mais. Então, quando souber mais, faça ainda melhor.

Maya Angelou

Resumo

A arquitura monolítica traz dentro da engenharia de software um histórico de sistemas legados e equipes frustradas com a complexidade de manutenção desses sistemas. Do outro lado a arquitetura de microsserviços é capaz de fornecer uma série de benefícios almejados por diversas empresas. Esse contexto leva essas empresas a optarem por adotar ou migrar seus sistemas para a arquitetura de microsserviços. Contudo, essa migração realizada em desacordo com os objetivos de negócio da empresa e sem o planejamento adequado, leva esses sistemas ao fracasso do modelo arquitetural. Tendo isso em vista, o presente estudo realizou uma pesquisa exploratória sobre os estilos arquiteturais monolítico e microsserviços, construindo um mapa mental com os fatores que afetam cada modelo arquitetural. Ao final, analisou-se esses fatores em quatro casos de estudo de empresas que optaram por migrar de um modelo arquitetural para o outro. Por fim, concluiu-se que a arquitetura monolítica é indicada para descoberta do domínio mas que essa arquitetura tende a perder manutenibilidade e evolucionabilidade a medida que a base de código cresce. Enquanto que a arquitetura de microsserviços tende a ser mais sustentável, porém apresenta um alto custo e exige domínio sobre a problemática e as tecnologias.

Palavras-chave: Arquitetura de Software. Microsserviços. Sistemas monolíticos.

Abstract

The monolithic architecture brings within software engineering a historic of legacy systems and teams frustrated with the complexity of maintaining these systems. On the other hand, the microservice architecture is capable of providing a series of benefits desired by several companies. This context leads these companies to choose to adopt or migrate their systems to the microservice architecture. However, this migration carried out in disagreement with the company's business objectives and without the proper planning, leads these systems to the failure of the architectural model. With this in mind, the present study carried out exploratory research on the monolithic and microservice architectural styles, building a mental map with the factors that affect each architectural model. In the end, these factors were analyzed in four case studies of companies that chose to migrate from one architectural model to another. Finally, it was concluded that the monolithic architecture is suitable for discovering the domain but this architecture tends to lose maintainability and evolutionability as the code base grows. While the microservice architecture tends to be more sustainable, however, it has a high cost and requires domain over the problematic and technologies.

Keywords: Microservices. Monolithic Systems. Software Architecture.

Lista de ilustrações

Figura 1 – Elementos que compõem a arquitetura de um software (RICHARDS; FORD, 2020) . . . 29
Figura 2 – Estrutura básica de um monolítico (RICHARDS; FORD, 2020) . . . 31
Figura 3 – Teoria vs Realidade de um monolítico (TILKOV, 2015) . . . 32
Figura 4 – Compartilhamento de objetos na arquitetura monolítica (RICHARDS; FORD, 2020) . . . 33
Figura 5 – Lei de Conway aplicada a uma arquitetura em camadas (LEWIS; FOWLER, 2014) . . . 39
Figura 6 – Lei de Conway aplicada a uma arquitetura de microsserviços (LEWIS; FOWLER, 2014) . . . 39
Figura 7 – Modelo de mapa mental utilizado para ilustrar o resultado das sínteses 44
Figura 8 – Mapa mental do processo de síntese realizado sobre a arquitetura monolítica . . . 51
Figura 9 – Mapa mental do processo de síntese realizado sobre a arquitetura de microsserviços . . . 55
Figura 10 – Fatores apresentados no caso de estudo da arquitetura monolítica do sistema KN Login . . . 59
Figura 11 – Fatores apresentados no caso de estudo da arquitetura monolítica da Otto . . . 63
Figura 12 – Fatores apresentados no caso de estudo da arquitetura de microsserviços da Otto . . . 64
Figura 13 – Fatores apresentados no caso de estudo da primeira arquitetura monolítica da Segment . . . 68
Figura 14 – Fatores apresentados no caso de estudo da arquitetura de microsserviços da Segment . . . 70
Figura 15 – Fatores apresentados no caso de estudo da arquitetura de microsserviços da RuaDois . . . 75
Figura 16 – Fatores apresentados no caso de estudo da arquitetura monolítica da RuaDois . . . 76
Figura 17 – Mapa mental do processo de síntese realizado sobre a arquitetura monolítica . . . 78
Figura 18 – Mapa mental do processo de síntese realizado sobre a arquitetura de microsserviços . . . 79

Lista de quadros

Quadro 1 – Definições adotadas 43
Quadro 2 – Quadro modelo aplicado na sintése dos aspectos arquiteturais elencados 44
Quadro 3 – Arquitetura monolítica - síntese sobre o domínio do problema 47
Quadro 4 – Arquitetura monolítica - síntese sobre o tamanho da aplicação 48
Quadro 5 – Arquitetura monolítica - síntese sobre os recursos financeiros 48
Quadro 6 – Arquitetura monolítica - síntese sobre os recursos humanos 48
Quadro 7 – Arquitetura monolítica - síntese sobre esforço e tempo inicial 48
Quadro 8 – Arquitetura monolítica - síntese sobre escalabilidade 48
Quadro 9 – Arquitetura monolítica - síntese sobre tolerância a falhas 49
Quadro 10 – Arquitetura monolítica - síntese sobre confiabilidade 49
Quadro 11 – Arquitetura monolítica - síntese sobre performance 49
Quadro 12 – Arquitetura monolítica - síntese sobre o processo de *deploy* 49
Quadro 13 – Arquitetura monolítica - síntese da testabilidade 50
Quadro 14 – Arquitetura monolítica - síntese da comunicação 50
Quadro 15 – Arquitetura monolítica - síntese da heterogeneidade das tecnologias . . 50
Quadro 16 – Arquitetura monolítica - síntese sobre complexidade do sistema 50
Quadro 17 – Arquitetura de microsserviços - síntese sobre o domínio do problema . 52
Quadro 18 – Arquitetura de microsserviços - síntese sobre o tamanho da aplicação . 52
Quadro 19 – Arquitetura de microsserviços - síntese sobre os recursos financeiros . . 52
Quadro 20 – Arquitetura de microsserviços - síntese sobre os recursos humanos . . 52
Quadro 21 – Arquitetura de microsserviços - síntese sobre esforço e tempo inicial . 53
Quadro 22 – Arquitetura de microsserviços - síntese sobre escalabilidade 53
Quadro 23 – Arquitetura de microsserviços - síntese sobre tolerância a falhas 53
Quadro 24 – Arquitetura de microsserviços - síntese sobre confiabilidade 53
Quadro 25 – Arquitetura de microsserviços - síntese sobre performance 53
Quadro 26 – Arquitetura de microsserviços - síntese sobre o processo de *deploy* . . 54
Quadro 27 – Arquitetura de microsserviços - síntese da testabilidade 54
Quadro 28 – Arquitetura de microsserviços - síntese da comunicação 54
Quadro 29 – Arquitetura de microsserviços - síntese da heterogeneidade das tecnologias 54
Quadro 30 – Arquitetura de microsserviços - síntese sobre complexidade do sistema 54

Lista de abreviaturas e siglas

API Application Programming Interface.

DDD Domain-driven Design.

FaaS Functions as a Service.

FIFO First in, First out.

MVC Model-View-Controller.

MVP Minimum Viable Product.

REST Representational State Transfer.

SOA Service Oriented-architecture.

TI Tecnologia da Informação.

UI User Interface.

Sumário

1 INTRODUÇÃO 23
1.1 Justificativa 24
1.2 Objetivos 24
1.2.1 Objetivo Geral 24
1.2.2 Objetivos Específicos 24

2 FUNDAMENTAÇÃO TEÓRICA 27
2.1 Arquitetura de Software 27
2.1.1 Leis da Arquitetura de Software 29
2.2 Sistemas monolíticos 30
2.2.1 Estrutura de um monolítico 31
2.3 Microsserviços 32
2.4 Perspectivas a respeito de cada estilo arquitetural 34
2.4.1 Problemática a ser resolvida 34
2.4.2 Recursos necessários 35
2.4.2.1 Recursos financeiros 35
2.4.2.2 Recursos humanos 35
2.4.2.3 Esforço e tempo inicial 36
2.4.3 Características arquiteturais 36
2.4.3.1 Escalabilidade 36
2.4.3.2 Tolerância a falhas 37
2.4.3.3 Confiabilidade 37
2.4.3.4 Performance 37
2.4.4 Manutenibilidade 37
2.4.4.1 *Deploy* 37
2.4.4.2 Testabilidade 38
2.4.4.3 Comunicação 38
2.4.5 Evolucionabilidade 40
2.4.5.1 Heterogeneidade das tecnologias 40
2.4.5.2 Complexidade 40

3 METODOLOGIA 41
3.1 Objetos de análise 41
3.2 Ferramentas 41
3.3 Descrição das etapas 42
3.3.1 Etapa 1 - Levantamento teórico 42

3.3.2 Etapa 2 - Perspectivas e aspectos . . . 42
3.3.3 Etapa 3 - Síntese . . . 43
3.3.4 Etapa 4 - Estudos de caso . . . 45
3.3.5 Etapa 5 - Análise comparativa final . . . 45

4 DESENVOLVIMENTO . . . 47
4.1 Síntese das perspectivas teóricas . . . 47
4.1.1 Síntese da arquitetura monolítica . . . 47
4.1.2 Síntese da arquitetura de microsserviços . . . 52
4.2 Análise dos casos de estudo . . . 56
4.2.1 KN Login . . . 56
4.2.1.1 Problemática da aplicação . . . 56
4.2.1.2 Solução adotada . . . 57
4.2.1.3 Panorama pós-adoção da solução . . . 58
4.2.1.4 Análise sobre o caso de estudo . . . 58
4.2.2 Otto . . . 58
4.2.2.1 Problemática da aplicação . . . 59
4.2.2.2 Solução adotada . . . 60
4.2.2.3 Panorama pós-adoção da solução . . . 61
4.2.2.4 Análise sobre o caso de estudo . . . 62
4.2.3 Segment . . . 63
4.2.3.1 Problemática da aplicação . . . 64
4.2.3.2 Solução adotada . . . 66
4.2.3.3 Panorama pós-adoção da solução . . . 67
4.2.3.4 Análise sobre o caso de estudo . . . 68
4.3 RuaDois . . . 69
4.3.0.1 Problemática da aplicação . . . 71
4.3.0.2 Solução adotada . . . 72
4.3.0.3 Panorama pós-adoção da solução . . . 73
4.3.0.4 Análise sobre o caso de estudo . . . 74

5 RESULTADOS . . . 77

6 CONSIDERAÇÕES FINAIS . . . 81

REFERÊNCIAS . . . 83

1 Introdução

Há cerca de 10 anos atrás, a Amazon e a Netflix se destacavam no mercado de software trazendo ao mundo uma tendência arquitetural: os microsserviços. A grande promessa era que essa arquitetura era capaz de prover escalabilidade, resiliência e diversos outros benefícios para os problemas comumente enfrentados pelas empresas com seus sistemas legados e monolíticos. Assim, dava-se início a uma nova corrida entre as empresas: Spotify, Coca Cola e muitos outros grandes nomes, começaram a migrar a sua arquitetura para esse novo padrão tão promissor (KWIECIEŃ, 2019).

10 anos após o início dessa grande tendência no mundo de software, têm se observado o movimento inverso: um crescente interesse em cima da arquitetura monolítica, em particular, dos monolíticos modulares (GUIMARÃES; BRYANT, 2020). A realidade é que muitas empresas têm se aventurado na arquitetura de microsserviços e se deparado com uma séries de desafios além do que estavam preparadas para enfrentar. Nas palavras de Fowler (2015b):

> Muitas equipes de desenvolvimento descobriram no estilo arquitetural dos microsserviços uma abordagem superior a uma arquitetura monolítica. Mas outras equipes descobriram que eles são um fardo que enfraquece a produtividade. [1] (FOWLER, 2015b, tradução nossa)

Assim, nos deparamos com uma realidade na qual existe uma frustração antiga das empresas em trabalhar com a arquitetura monolítica devido as dificuldades de manutenibilidade que eventualmente essa arquitetura acaba demonstrando, e que na tentativa de sair desses problemas e conquistar a modularidade e todos os outros benefícios oferecidos pelos microsserviços essas empresas acabam se frustrando novamente mediante os desafios desse estilo arquitetural.

Diante desta situação, o presente trabalho visa realizar uma análise comparativa entre os dois estilos arquiteturais, monolítico e microsserviços, com o intuito de compreender as diferenças entre cada estilo e quais são os fatores que influenciam, positivamente ou negativamente, cada uma dessas abordagens arquiteturais.

[1] Texto original: *Many development teams have found the microservices architectural style to be a superior approach to a monolithic architecture. But other teams have found them to be a productivity-sapping burden.*

1.1 Justificativa

No livro *Software Architeture in Pratice*, os autores Bass, Clements e Kazman (2015) apresentam a arquitetura de software como a ponte capaz de auxiliar as empresas a atingirem seus objetivos de negócio. Objetivos estes que por diversas vezes tendem a ser bastante abstratos, cabendo a arquitetura de software o grande papel de concretizar esses objetivos em sistemas (BASS; CLEMENTS; KAZMAN, 2015).

Na intenção de construir essa ponte, frequentemente times de software se encontram ansiosos para adotar uma arquitetura de microsserviços levando aos seus sistemas as grandes vantagens que essa arquitetura é capaz de trazer, contudo o grande prêmio oferecido vêm acompanhado de vários custos e riscos que os desenvolvedores tendem a não considerar quando optam pela utilização de microsserviços (FOWLER, 2015a).

Do outro lado, a arquitetura monolítica, comumente adotada por diversas aplicações, é inicialmente simples e agradável de manter, mas tende a crescer a sua complexidade mediante a sua inerente característica de ser acoplada (TILKOV, 2015).

Diante deste contexto, o presente trabalho justifica-se por buscar esclarecer as diferenças entre ambos os estilos arquiteturais, monolítico e microsserviços, visando fornecer embasamento para equipes de desenvolvimento de software analisarem de forma mais consciente o contexto em que os projetos estão inseridos e como a escolha de determinado tipo arquitetural pode influenciar, positivamente ou negativamente, o alcance dos objetivos de negócio.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho tem como objetivo geral realizar uma análise comparativa acerca dos estilos arquiteturais de software monolítico e microsserviços com o intuito de compreender as características de cada estilo e em quais contextos cada um é melhor aplicado. Para tal, pretende-se realizar uma pesquisa exploratória baseada em revisão bibliográfica e em relatos de casos reais nos quais empresas optaram por migrar de uma arquitetura monolítica para uma arquitetura de microsserviços ou vice-versa.

1.2.2 Objetivos Específicos

As seguintes metas foram levantadas visando atingir o objetivo geral do presente trabalho:

1. Compreender o que é arquitetura de software e como ela influência as características

dos sistemas de software;

2. Compreender o que é uma arquitetura monolítica e suas principais características;
3. Compreender o que é uma arquitetura de microsserviços e suas principais características;
4. Explorar diferentes visões apresentadas na bibliografia acerca de ambos os estilos arquiteturais;
5. Explorar relatos de casos reais nos quais empresas optaram por migrar de uma arquitetura para a outra, com o intuito de compreender as dificuldades e vantagens encontradas por elas em cada estilo arquitetural;
6. Entrevistar a *startup* RuaDois que optou por realizar a migração de microsserviços para um sistema monolítico, compreendendo o porquê dessa decisão e como esta impactou a *startup*;
7. Agrupar e classificar as informações obtidas por meio da revisão bibliográfica, dos relatos reais e da entrevista realizada, com o objetivo de analisar e comparar cada estilo arquitetural;
8. Elencar quais são as características de cada estilo arquitetural e com base nelas compreender em quais contextos cada estilo arquitetural é melhor aplicado.

2 Fundamentação Teórica

Neste capítulo serão apresentadas as bases teóricas que permeiam o escopo do presente trabalho. Partindo do entendimento sobre o conceito de arquitetura de software e como ela reflete nos sistemas. Em seguida, abordando estilos de arquitetura de software pertinentes para o contexto estudado. As seções estão dispostas em:

1. **Arquitetura de software:** definição do termo arquitetura dentro da área de software e suas características.

2. **Sistemas monolíticos:** descrição, vantagens e desvantagens da arquitetura monolítica.

3. **Microsserviços:** descrição, vantagens e desvantagens da arquitetura de microsserviços.

2.1 Arquitetura de Software

> Sistemas de software são construídos para satisfazer os objetivos de negócios das organizações. A arquitetura é a ponte entre esses objetivos (frequentemente abstratos) e o sistema resultante (concreto). Enquanto o caminho de objetivos abstratos para sistemas concretos possa ser complexo, a boa notícia é que a arquitetura de software pode ser projetada, analisada, documentada e implementada usando técnicas conhecidas que apoiarão a realização desses objetivos de negócios e missão. A complexidade pode ser domada, tornada tratável. [1] (BASS; CLEMENTS; KAZMAN, 2015, tradução nossa)

Como retratado na citação acima, arquitetura de software representa a ponte entre os sistemas construídos e os objetivos de cada organização. A publicação *What is your definition of software architecture?* do *Software Engineering Institute* da *Carnegie Mellon University* (SEI, 2017) apresenta uma variedade de diferentes conceitos a respeito de arquitetura de software abordados pela bibliografia, exemplificando a dificuldade existente

[1] Texto original: *Software systems are constructed to satisfy organizations' business goals. The architecture is a bridge between those (often abstract) business goals and the final (concrete) resulting system. While the path from abstract goals to concrete systems can be complex, the good news is that software architecture can be designed, analyzed, documented, and implemented using know techniques that will support the achievement of these business and mission goals. The complexity can be tamed, made tractable.*

em compreender o conceito de arquitetura dentro da área de Tecnologia da Informação (TI).

Segundo Vogel et al. (2011) arquitetura é um aspecto implícito com o qual os desenvolvedores são confrontados diariamente e que não pode ser ignorado ou eliminado, ainda que eles nem sempre tenham consciência sobre a sua constante presença.

Bass, Clements e Kazman (2015) traz a seguinte definição a respeito de arquitetura de software:

> A arquitetura de software de um sistema é o conjunto de estruturas necessárias para raciocinar sobre o sistema, que compreende os elementos do software, as relações entre eles e as propriedades de ambos. [2] (BASS; CLEMENTS; KAZMAN, 2015, tradução nossa)

Com base nessa abordagem, os autores defendem que os sistemas de software são compostos por diversas estruturas e elementos, por como essas estruturas se relacionam entre si e pelas propriedades que os caracterizam. Assim, a arquitetura se torna uma abstração do sistema, destacando aspectos desejados do mesmo e omitindo outros aspectos, com o intuito de minimizar a complexidade com a qual o compreendemos.

A Figura 1 apresenta uma segunda visão abordada por Richards e Ford (2020) a respeito de arquitetura. Nessa abordagem eles defendem quatro dimensões as quais definem o que é arquitetura de software. A primeira dimensão consiste nas estruturas: serviços, camadas e módulos são exemplos de conceitos abarcados por esse ponto, todavia, para eles descrever a arquitetura por meio de estruturas não é suficiente para contemplar todo o escopo acerca de arquitetura de software. Diante desta problemática, eles introduzem os conceitos de características, decisão e princípios de design referentes a arquitetura.

As características da arquitetura, por sua vez, referem-se aos critérios de sucesso do sistema, trazendo pontos como disponibilidade, confiabilidade, segurança, etc., os quais são anteriores e independentes ao conhecimento acerca das funcionalidades que o software implementará.

O terceiro aspecto apresentado pelos autores consiste nas decisões de arquitetura responsáveis por definir as regras sob as quais o sistema deve ser construído. Isto diz ao time de desenvolvimento o que ele pode ou não fazer. A exemplo, em uma arquitetura Model-View-Controller (MVC)[3], a *View* é impossibilitada de consumir dados da *Model*, e cabe aos desenvolvedores garantir que isso seja respeitado.

[2] Texto original: *The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*

[3] *Model-View-Controller* (MVC) consiste em um estilo arquitetural adotado por várias aplicações. A *Model* representa a camada de dados, a *View* a camada de apresentação e a *Controller* é a camada de lógica (MCGOVERN et al., 2004).

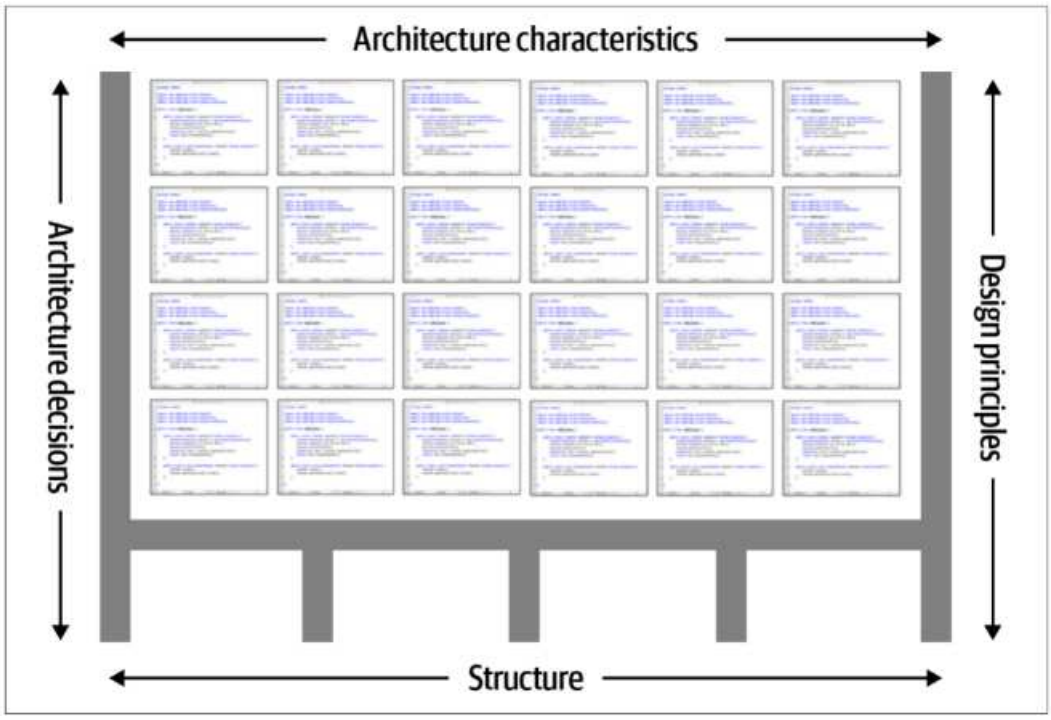

Figura 1 – Elementos que compõem a arquitetura de um software (RICHARDS; FORD, 2020)

Por fim, a última dimensão refere-se aos princípios de design. Estes consistem em um guia pelo o qual os desenvolvedores podem se orientar quando for necessário tomar alguma decisão. Diferentemente das decisões de arquitetura que devem ser sempre seguidas afim de garantir as características desejadas para a arquitetura, os princípios trazem maior flexibilidade e autonomia para o desenvolvedor, podendo este analisar o contexto do problema em mãos e decidir se seguir os princípios é a melhor abordagem para a funcionalidade desenvolvida, ou se deve seguir por um caminho diferente o qual ele julgue que os ganhos são maiores para a aplicação.

2.1.1 Leis da Arquitetura de Software

> **Tudo em arquitetura de software é um *trade-off*.** Primeira Lei da Arquitetura de Software
>
> ***Por que* é mais importantes do que *como*.** Segunda Lei da Arquitetura de Software
>
> [4] (RICHARDS; FORD, 2020, tradução nossa)

As leis acima foram apresentadas recentemente por Richards e Ford (2020) no livro *Fundamentals of Software Architecture: An Engineering Approach* e trazem consigo

[4] Texto original: *Everything in software architecture is a trade-off. First Law of Software Architecture. Why is more important than how. Second Law of Software Architecture.*

perspectivas importantes para compreender o que é arquitetura de software dentro do dia a dia dos engenheiros de software e como os mesmos devem lhe dar com ela.

A Primeira Lei da Arquitetura de Software descreve a realidade do arquiteto de software, o qual precisa lidar constantemente com conflitos de escolha. O papel do arquiteto está intrinsecamente ligado a análise da situação e a tomada de decisão sobre qual caminho seguir. Para tal, é necessário que o mesmo esteja alinhado com uma série de fatores, como práticas de engenharia, *DevOps*[5], processos, negócio, etc (RICHARDS; FORD, 2020).

A segunda lei traz uma perspectiva que por diversas vezes é ignorada, na qual tendemos a olhar para a topologia dos sistemas, observando suas estruturas e como estas se relacionam, e não damos a devida atenção ao porquê das decisões arquiteturais tomadas (RICHARDS; FORD, 2020).

Nesse sentindo, o ponto defendido pelos autores busca olhar o por quê certas decisões são tomadas mediante os *trade-offs* enfrentados.

2.2 Sistemas monolíticos

A visão inicial a respeito da Arquitetura Monolítica refere-se a um padrão de desenvolvimento de software no qual um aplicativo é criado com uma única base de código, um único sistema de compilação, um único binário executável e vários módulos para recursos técnicos ou de negócios. Seus componentes trabalham compartilhando o mesmo espaço de memória e recursos formando uma unidade de código coesa (SAKOVICH, 2017).

Newman (2019b) expande um pouco dessa visão, saindo da ideia de uma única base de código coesa e adotando a perspectiva de uma única unidade de *deploy*. Essa abordagem permite a ele distinguir os seguintes tipos de monolíticos:

Monolítico com um único processo é o tipo mais comum de monolíticos e refere-se fundamentalmente a existência de um único processo executando toda a base de código. Vale ressaltar que dentro desta classificação exite ainda um subconjunto de monolíticos modulares nos quais têm se cada módulo trabalhando de forma independente um do outro, mas ainda com a necessidade de implantar uma única unidade de código.

Monolítico distribuído são sistemas compostos por múltiplos serviços mas que precisam ser implantados juntos. Nas palavras de Newman (2019b):

[5] *DevOps* é um movimento cultural que altera como indivíduos pensam sobre o seu trabalho, dando suporte a processos que aceleram a entrega de valor pelas empresas ao desenvolverem práticas sustentáveis de trabalho (DAVIS; DANIELS, 2016).

> Um monolítico distribuído pode muito bem atender a definição de uma Service Oriented-architecture (SOA)[6], mas muitas vezes falha em cumprir as promessas do padrão. [7] (NEWMAN, 2019b, tradução nossa)

Sistemas caixa preta de terceiros também são abordados por Newman (2019b) como sendo monolíticos, uma vez que não é possível decompor os esforços referente a uma migração.

2.2.1 Estrutura de um monolítico

A arquitetura monolítica é um padrão de fácil compreensão que permite as empresas desfrutar de forma branda dos processos de construção e implantação de sistemas de software no início dos projetos. Esse modelo arquitetural é caracterizado por ser tecnicamente particionado (RICHARDS; FORD, 2020), ou seja, seus componentes (classes, métodos, etc.) estão agrupados por sua função técnica dentro do sistema. Na Figura 2, podemos ver essa organização na qual temos:

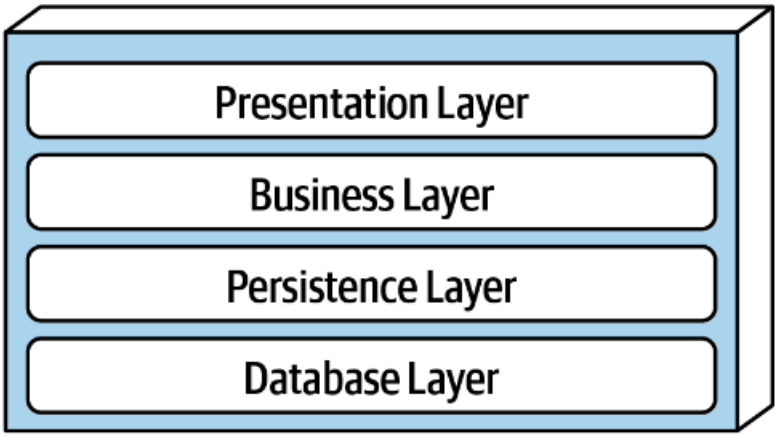

Figura 2 – Estrutura básica de um monolítico (RICHARDS; FORD, 2020)

Camada de apresentação destinada a apresentação dos dados;

Camada de negócios destinada a implementação das regras de negócio da aplicação;

Camada de persistência destinada a comunicação com o banco de dados;

Camada de dados destinada aos armazenamentos dos dados.

[6] A Arquitetura orientada a serviço - SOA é uma abordagem arquitetural na qual vários serviços trabalham juntos para entregar um conjunto de funcionalidades. (NEWMAN, 2015)

[7] Texto original: *A distributed monolithic may well meet the definition of a service oriented-architecture, but all too often fails to deliver on the promises of SOA.*

Tilkov (2015) defende que essa arquitetura é um modelo essencialmente acoplado, no qual as abstrações que compõe o sistema compartilham de todos os mesmos recursos da aplicação: bibliotecas, canais de comunicação dentro do próprio processo, modelos de persistência no banco de dados, etc. A Figura 3 exemplifica como mesmo no modelo modularizado é fácil ultrapassar os limites definidos para cada módulo visto que não há barreiras técnicas que impeçam os desenvolvedores de tal. Dessa forma, garantir as características de alta coesão e baixo acoplamento, tão desejadas na manutenibilidade dos softwares, se torna uma tarefa difícil dependendo exclusivamente da disciplina dos desenvolvedores (TILKOV, 2015; FOWLER, 2015b).

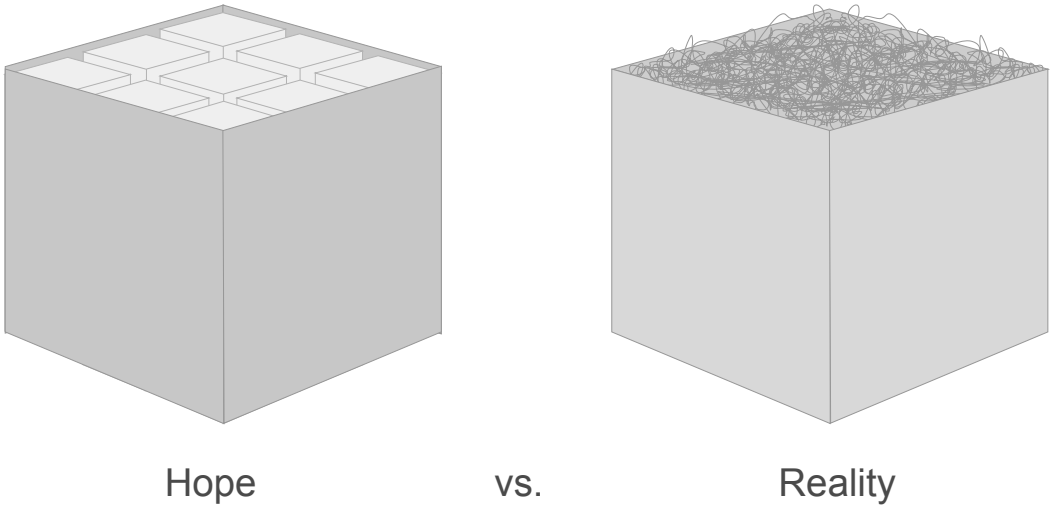

Figura 3 – Teoria vs Realidade de um monolítico (TILKOV, 2015)

A Figura 4 aprofunda ainda mais nesse contexto da arquitetura monolítica, mostrando a tendência de compartilhar componentes, a exemplo classes de *log*, classes com funções compartilhadas por todo o sistema, etc., transparecendo como a reusabilidade do código dentro de sistemas monolíticos tende a aumentar o acoplamento entre seus componentes (RICHARDS; FORD, 2020).

2.3 Microsserviços

Segundo Richards e Ford (2020), microsserviços é um estilo arquitetural fortemente inspirado nas ideias de Domain-driven Design (DDD)[8], trazendo deste design o conceito de limites de contexto. Esses limites referem-se a um domínio sob o qual estão entidades e comportamentos que podemos desacoplar do restante do sistema. Assim, diferentemente da arquitetura monolítica que é organizada por atribuições técnicas, os microsseviços trazem em sua estrutura um reflexo de como o contexto é organizado no mundo real.

[8] *Domain-Driven Design* é uma estratégia de modelagem de software que visa a resolução de problemas complexos se concentrando do domínio principal e explorando modelos colaborativos entre profissionais de diferentes áreas.

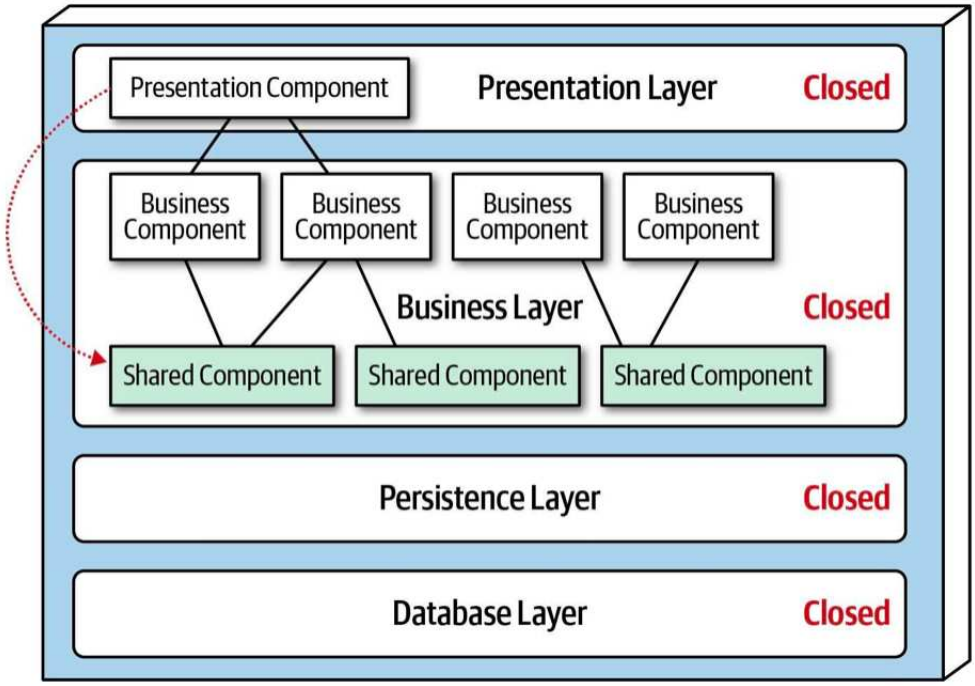

Figura 4 – Compartilhamento de objetos na arquitetura monolítica (RICHARDS; FORD, 2020)

Essa abordagem torna os microsserviços pequenas partes autônomas que trabalham juntas em cima de uma base de código coesa e focada em resolver as regras de negócio dentro do domínio especificado (NEWMAN, 2015). Dessa forma, a aplicação deixa de ser um único processo e se torna um conjunto de processos separados e independentes, se comunicando por meio de algum protocolo, comumente Representational State Transfer (REST)[9] (LEWIS; FOWLER, 2014).

Um dos grandes benefícios proporcionados por esta arquitetura são os limites impostos por cada serviço, os quais trazem maiores garantias sobre alta coesão, baixo acoplamento e modularização do sistema do que os modelos arquiteturais mais tradicionais (FOWLER, 2015b). Outra característica que contribui para tal é que nesse padrão a duplicação do código é, em diversas situações, preferível a reutilização do mesmo, visto que apesar do reuso de código ser algo benéfico, ele favorece o acoplamento por meio de herança e composição (RICHARDS; FORD, 2020).

Todas essas vantagens oferecidas pelos limites de domínio, fazem com que a escolha cuidadosa sobre quais são esses limites seja de grande importância para o sistema. Nas palavras de Newman (2015):

> Sem desacoplamento, tudo pode desabar. A regra de ouro: você pode alterar um serviço e implantá-lo sozinho sem alterar nada mais? Se a resposta for não, então será difícil pra você alcançar muitas das vantagens

[9] REST é um protocolo de comunicação entre *web services* que utiliza a semântica dos métodos HTTP.

> que discutiremos ao longo deste livro. [10] (NEWMAN, 2015, tradução nossa)

Fowler (2015b) ainda ressalta que os microsserviços utilizam de um sistema distribuído para prover toda essa modularidade, e como tal, acabam trazendo junto todas as desvantagens deste modelo, principalmente, a complexidade.

2.4 Perspectivas a respeito de cada estilo arquitetural

Esta seção visa aprofundar no entendimento sobre as vantagens e desafios encontrados na adoção de cada um dos estilos arquiteturais abordados nesse trabalho. Para tal, será explorado as seguintes perspectivas[11] com o intuito de compreender como a escolha da arquitetura impacta ou é impactada pelas mesmas:

1. Problemática a ser resolvida;
2. Recursos necessários;
3. Características arquiteturais;
4. Manutenibilidade;
5. Evolucionabilidade.

2.4.1 Problemática a ser resolvida

Ao iniciar um projeto é comum se ter um baixo domínio sobre o problema a ser resolvido. A exemplo, as denominadas *startups* estão continuamente surgindo no mercado com o intuito de descobrir e validar suas ideias. Contextos como este trazem as empresas a necessidade de realizar constantes alterações de forma extremamente rápida no sistema visando atingir um curto ciclo de *feedback*.

Fowler (2015c) defende que construir uma versão simplista do seu software é a melhor forma de descobrir o sistema e validar se ele é realmente útil para os seus usuários. Assim, ele sugere a estratégia "monolítico primeiro", uma vez que a arquitetura monolítica permite um desenvolvimento fácil por ser de conhecimento da maioria dos desenvolvedores de software e apresentar baixa complexidade para a execução de tarefas como *deploy*, confecção de testes e compartilhamento do código.

[10] Texto original: *Without decoupling, everything breaks down for us. The golden rule: can you make a change to a service and deploy it by itself without changing anything else? If the answer is no, then many of the advantages we discuss throughout this book will be hard for you to achieve.*

[11] Na subseção 3.3.2 a proposta de cada perspectiva adotada é explicada.

Fowler (2015c) e Newman (2019a) fortalecem essa visão apontando que conhecer os limites do seu software é de extrema importância antes de iniciar uma arquitetura de microsserviços. De forma que os custos e impactos de errar os limites de cada serviço nessa arquitetura tendem a ser muito maiores do que em um software monolítico mal projetado por não conhecer muito bem o domínio do problema.

Um outro ponto a ser considerado na problemática da aplicação refere-se ao tamanho que essa aplicação terá. Richards e Ford (2020) apontam que a arquitetura monolítica é uma arquitetura de menor custo portanto costuma ser uma boa escolha para sistemas pequenos.

2.4.2 Recursos necessários

2.4.2.1 Recursos financeiros

De acordo com Richards e Ford (2020), a simplicidade da arquitetura monolítica reflete em seu baixo custo de desenvolvimento e manutenção. Diferentemente, de uma arquitetura de microsserviços que exige uma série de fatores: monitoramento, replicação, expertise sobre as ferramentas, etc., fazendo com o que o custo para desenvolver e manter tal arquitetura seja elevado. Mediante essa visão, os autores indicam o uso desse modelo para empresas de grande porte, enquanto que empresas pequenas tendem a lhe dar melhor com monolíticos.

2.4.2.2 Recursos humanos

Na seção subseção 2.4.1, nós enfatizamos a importância de conhecer os limites da sua aplicação antes de optar por uma arquitetura de microsserviços. Isso nos leva a uma outra necessidade desse estilo arquitetural, referente a relevância de possuir alguém na equipe que detenha esse conhecimento aprofundado acerca do negócio e da problemática abordada. No texto *Microservices For Greenfield?*, Newman (2019a) relata um projeto no qual eles estavam reimplementando um sistema que já existia há muitos anos, mas com uma equipe relativamente nova, ou seja, apesar do domínio de negócio ter sido explorado por um bom tempo, eles não possuíam a expertise necessária para a construção da aplicação de forma distribuída.

Nesse contexto, Newman (2019a) indica a adoção de uma arquitetura monolítica como uma melhor opção uma vez que, como citado na subseção 2.4.1, este estilo arquitetural é indicado para contextos nos quais não se possui domínio e, para completar, apresenta a vantagem de ser uma arquitetura familiar para boa parte dos desenvolvedores (RICHARDS; FORD, 2020), dessa forma, ele se diferencia de uma arquitetura de microsserviços por não precisar de um especialista sobre a tecnologia e sobre o contexto.

2.4.2.3 Esforço e tempo inicial

O tempo de lançamento no mercado costuma ser um fator prioritário para diversas empresas, em geral, elas desejam lançar o produto no menor tempo possível, seja para validar a ideia, como vimos na subseção 2.4.1, ou por uma questão estratégica de mercado.

Nesse sentido, Fowler (2015a) aponta que a arquitetura de microsserviços consegue te entregar uma séries de recompensas (escalabilidade, tolerância a falhas, etc.) contudo, essas recompensas vêem acompanhada do alto custo que é inerente dos sistemas distribuídos. Esse modelo arquitetural necessita de *deploy* automatizado, monitoramento, lidar com eventuais inconsistências dos dados e outros fatores que acabam contribuindo para aumentar o tempo e o esforço inicial necessário para lançar um produto no mercado. Nas palavras de Richards e Ford (2020):

> Microsserviços não poderiam existir sem a revolução *DevOps* e a implacável marcha em direção à automação de questões operacionais.[12] (RICHARDS; FORD, 2020, tradução nossa)

Por outro lado, a simplicidade da arquitetura monolítica permite que funcionalidades cheguem muito mais rápido no mercado, uma vez que nesse modelo não há tais necessidades e processos como *deploy* podem ser realizados de maneira operacional. Contudo, vale ressaltar que essa alta produtividade inicial tende a decrescer com o tempo, a medida que o monolítico vai ganhando mais funcionalidades e, consequentemente, aumentando a sua complexidade.

2.4.3 Características arquiteturais

2.4.3.1 Escalabilidade

A arquitetura monolítica pode ser escalada horizontalmente utilizando de um balanceador de carga para executar várias instâncias do sistema (LEWIS; FOWLER, 2014). Entretanto, a escalabilidade oferecida por essa arquitetura é bastante limitada. A falta de modularidade exige que toda a aplicação seja replicada ao invés de replicarmos somente as funções que apresentam maior demanda. Existe a possibilidade de aplicar técnicas que amenizam esse problema, contudo, isto requer um esforço considerável para adaptar o modelo arquitetural (RICHARDS; FORD, 2020).

No caso dos microsserviços, a escalabilidade é um fator predominante mediante a sua estrutura altamente desacoplada que favorece a escalabilidade em um nível incremental. Essa característica é ainda apoiada pelas técnicas modernas de provisionamento de recursos que beneficiam a elasticidade do estilo arquitetural (RICHARDS; FORD, 2020).

[12] Texto original: *Microservices couldn't exist without the DevOps revolution and the relentless march toward automating operational concerns.*

2.4.3.2 Tolerância a falhas

O padrão arquitetural monolítico não suporta tolerância a falhas visto que se algo inesperado acontece em uma determinada parte do código, como uma leitura de memória inválida, essa falha se propaga até matar o processo no qual roda todo o código monolítico, indisponibilizando a aplicação por inteiro, a qual pode levar em média, dependendo do tamanho do monolítico, de de 2 a 15 minutos para ser reinicializada (RICHARDS; FORD, 2020).

Do outro lado, os microsserviços naturalmente apresentam aspectos positivos em relação a tolerância a falhas uma vez que uma falha em um serviço tende a impactar somente uma parte da aplicação. Richards e Ford (2020) fazem uma ressalva que essa tolerância pode decair se houver muita dependência comunicativa entre os serviços, por isso definir cuidadosamente a granularidade do serviço é um ponto que deve ser bem avaliado.

2.4.3.3 Confiabilidade

A arquitetura monolítica quando comparada a uma arquitetura distribuída, como microsserviços, tende a apresentar maior confiabilidade uma vez que esta não possui os problemas de conexão da rede, largura de banda e latência que os mesmos apresentam. Contudo, como será abordado na subseção 2.4.4.2, a testabilidade é um problema desse modelo arquitetural e consequentemente isso decresce a confiabilidade do sistema (RICHARDS; FORD, 2020).

2.4.3.4 Performance

A característica desacoplada dos microsserviços traz consigo um enorme número de requisições sendo disparadas na rede juntamente com várias camadas adicionais de segurança na comunicação entre os serviços, o que faz com que o tempo de processamento cresça, tornando este um modelo arquitetural de baixa performance (RICHARDS; FORD, 2020).

A característica simplista dos monolíticos também não contribui nesse quesito. Segundo Richards e Ford (2020), a arquitetura monolítica não é por si mesma uma arquitetura de alta performance, exigindo esforço dos engenheiros para proporcionar tal fator para a aplicação por meio de desenvolvimento paralelo, *multithreading*, etc.

2.4.4 Manutenibilidade

2.4.4.1 *Deploy*

Na seção 2.2 foi apresentado como monolíticos são em sua essência uma unidade de *deploy*. No início do projeto, esta característica contribui com a simplicidade e a velocidade

em lançar o sistema, como relatado na subseção 2.4.2.3. Porém, a medida que a base de código do monolítico cresce, compilar e implantar toda a base de código a fim de disponibilizar uma alteração tende a se tornar um processo árduo, fazendo com que seja difícil e demorado disponibilizar as mudanças realizadas no sistema (LEWIS; FOWLER, 2014). Para completar o quadro, o *deploy* de uma versão da aplicação monolítica com um *bug*, por exemplo, acresce os riscos dessa alteração atingir o banco de dados ou alguma outra parte do sistema, tornando a reversão do impacto do processo de implantação mais difícil (RICHARDS; FORD, 2020).

Os microsserviços apresentam o caminho inverso: inicialmente eles exigem mais esforços para implantar, visto que é necessário automatizar todas as partes do fluxo de implantação, todavia, uma vez que os processos estão automatizados, disponibilizar mudanças no sistema se torna uma tarefa fácil e rápida. Essa abordagem se caracteriza por somente uma parte do código ser enviada para produção, minimizando os impactos que possam ocorrer ao disponibilizar a funcionalidade.

2.4.4.2 Testabilidade

A testabilidade de uma arquitetura monolítica é apontada por Richards e Ford (2020) como um ponto fraco nesse estilo arquitetural. Os autores defendem que este é um aspecto que decresce a medida que a complexidade do sistema aumenta e, dessa forma, desenvolvedores ao fazerem uma pequena modificação tendem a não executar toda a *suite* de testes devido ao alto custo de tempo para tal.

Já para uma arquitetura de microsserviços, eles consideram a testabilidade um ponto positivo, visto que cada serviço pode ser testado facilmente dentro do seu domínio. Lewis e Fowler (2014) ressaltam que a base desse modelo arquitetural são os limites de domínio de cada serviço, e para tal é importante coordenar qualquer mudança nas interfaces de comunicação adicionando camadas extras de compatibilidade de versões, o que acaba tornando o processo de testes mais difícil.

2.4.4.3 Comunicação

> Qualquer organização que projeta um sistema (definido de forma mais ampla aqui do que apenas sistemas de informação) produzirá inevitavelmente um projeto cuja estrutura é uma cópia da estrutura de comunicação da organização.[13] (CONWAY, 1968, tradução nossa)

Lewis e Fowler (2014) faz uma menção a Lei de Conway apresentada acima, exemplificando-a na Figura 5 como essa arquitetura em camadas reflete a organização

[13] Texto original: *Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.*

das empresas. Eles apontam que essa estrutura deixa a comunicação dentro do projeto mais lenta, de forma que mudanças simples no sistema podem demorar bastante devido a dependência de uma equipe externa.

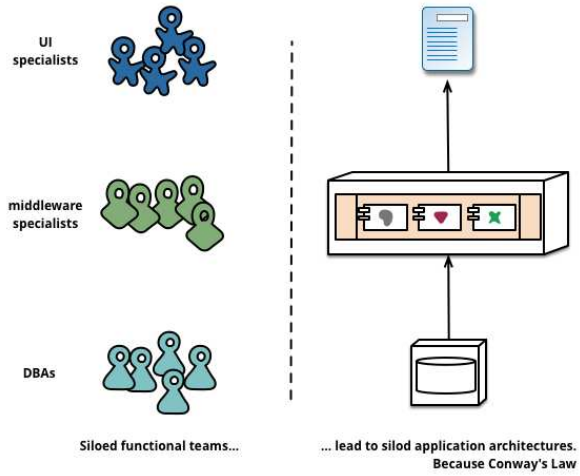

Figura 5 – Lei de Conway aplicada a uma arquitetura em camadas (LEWIS; FOWLER, 2014)

Quando olhamos para essa estrutura organizacional dentro da arquitetura de microsserviços, Figura 6, a essência desta arquitetura de ser particionada por domínio do negócio e não por suas características técnicas é refletida gerando times multifuncionais, de forma que a equipe possua toda a gama de habilidades necessárias (LEWIS; FOWLER, 2014).

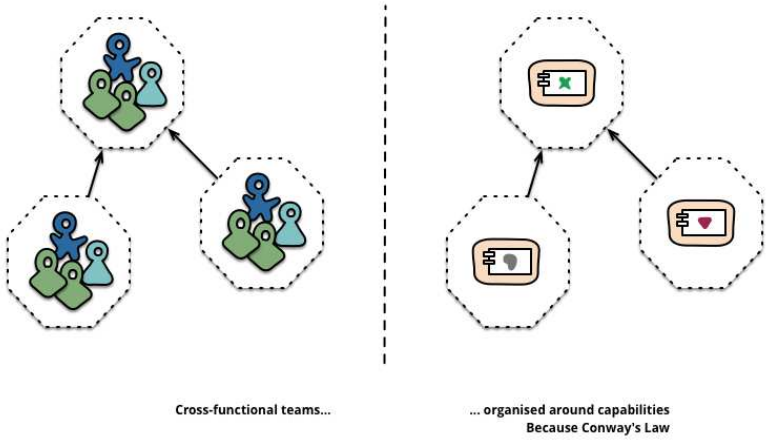

Figura 6 – Lei de Conway aplicada a uma arquitetura de microsserviços (LEWIS; FOWLER, 2014)

2.4.5 Evolucionabilidade

2.4.5.1 Heterogeneidade das tecnologias

Tradicionalmente, as empresas padronizam a *stack* de tecnologias com a qual trabalham. Na arquitetura monolítica esse fator acaba se tornando algo enraizado no sistema devido a sua característica inerente de ser uma arquitetura acoplada, fazendo com que a adoção de tecnologias diferentes para cada problema, ou mesmo a migração do sistema para uma tecnologia mais atual seja um processo bastante árduo.

Os microsserviços já apresentam a característica oposta de serem em sua essência altamente desacoplados. Isso permite aos engenheiros de software escolherem a melhor tecnologia para escalar a solução de acordo com problema abordado (RICHARDS; FORD, 2020).

2.4.5.2 Complexidade

Como abordado na seção 2.2, a arquitetura monolítica é naturalmente uma arquitetura simplista, de conhecimento da grande maioria de desenvolvedores, o que faz com que a sua complexidade no início de um projeto seja extremamente baixa. Contudo, existe uma tendência nessa arquitetura de promover o acoplamento e consequentemente aumentar a complexidade do sistema. Mesmo em monolíticos modularizados, é extremamente difícil evitar a criação de conexões entre os módulos (TILKOV, 2015), fazendo com o que a medida que cresce a base de código o sistema vá perdendo os benefícios de ser uma arquitetura simplista.

Do outro lado, os microsserviços apresentam uma arquitetura distribuída e naturalmente isso vêm acompanhado da complexidade de coordenar todas essas partes independentes que o compõem para prover a funcionalidade final. Juntamente com esse contexto vêm as necessidades de monitorar e automatizar os processos que tornam ainda mais árdua tal tarefa.

Apesar dessa perspectiva global da complexidade dos microsserviços, existe também uma perspectiva local, referente ao domínio de um único serviço, na qual nota-se uma baixa complexidade mediante a as características de alta coesão, baixo acoplamento e limite de domínio sobre as quais cada serviço é imposto. Isso faz dos microsserviços complexos quando olhamos globalmente para eles, mas simplistas quando olhamos localmente, de forma que é fácil para os desenvolvedores lidar com os *trade-offs* de cada serviço.

3 Metodologia

Com o propósito de atingir o objetivo descrito na subseção 1.2.1, o presente trabalho realizou uma pesquisa exploratória acerca do tema abordado juntamente com a análise de alguns casos práticos de empresas que fizeram a migração de uma arquitetura monolítica para uma arquitetura de microsserviços, ou vice e versa. Assim, o desenvolvimento deste estudo baseou-se nas seguintes etapas:

Etapa 1 Levantamento teórico inicial acerca dos objetos de análise diante dos referências teóricos adotados;

Etapa 2 Fichamento e agrupamento dos pontos de vista descobertos na Etapa 1, afim de encontrar as perspectivas comuns e divergentes entre os autores estudados;

Etapa 3 Síntese acerca dos aspectos levantados sobre cada estilo arquitetural;

Etapa 4 Descrição e análise dos casos de estudo relevantes para o tema;

Etapa 5 Análise final comparando os dois modelos arquiteturais com base nos referências teóricos adotados e nos casos práticos estudados.

A seguir serão apresentados os objetos de análise, as ferramentas utilizadas e a descrição de cada etapa adotada. Vale ressaltar que deste ponto em diante alguns termos, como perspectivas ou impactos, estarão destacados na presente seção com o intuito de chamar a devida atenção para o seu significado, mas a frente será dada a definição tomada para cada um deles.

3.1 Objetos de análise

Os objetos de análise pertinentes para o presente estudo consistem em:

Arquitetura monolítica: estilo arquitetural tradicionalmente adotado em diversas empresas, no qual se detém uma base única de código.

Arquitetura de microsserviços: estilo arquitetural distribuído, largamente adotado nos últimos anos por diversas empresas.

3.2 Ferramentas

As ferramentas utilizadas para auxiliar na confecção do presente estudo foram:

Miro editor gráfico online que permite a construção de diagramas, mapa mentais, entre outros recursos gráficos relevantes ao conteúdo apresentado;

Google Planilhas ferramenta para construção de tabelas.

Trello quadro interativo que auxilia na organização de projetos utilizando do modelo Kanban.

3.3 Descrição das etapas

3.3.1 Etapa 1 - Levantamento teórico

A fundamentação teórica deste trabalho visou explorar os temas de arquitetura de software, estilo arquitetural monolítico e estilo arquitetural de microsserviços por meio de livros, Google Scholar e outras fontes de informação. Os tópicos abordados foram selecionados com base na sua respectiva relevância dentro do assunto estudado e tiveram embasamento principalmente nos autores Martin Fowler, Sam Newman, Mark Richards e Neal Ford, dentre outros.

3.3.2 Etapa 2 - Perspectivas e aspectos

Mediante a fundamentação teórica construída na Etapa 1, levantou-se os principais pontos acerca dos estilos arquiteturais abordados que eram tratados pelos autores estudados, para tal utilizou-se da ferramenta Trello com o intuito de organizar os referências teóricos e destacar o ponto de vista de cada autor organizando as referências em *cards*. A partir daí, utilizou-se *post-its* para agrupar pontos de vista em comum entre os autores e classificar quais eram os **aspectos** positivos e negativos de cada arquitetura.

Diante dos agrupamentos formados, levantou-se a definição de **perspectivas** para o presente estudo, as quais referem-se a formas diferentes de olhar para a arquitetura. Assim, elencou-se as seguintes perspectivas:

Problemática a ser resolvida: fatores que influenciam o sistema sob a perspectiva do problema e do contexto no qual a aplicação está sendo construída;

Recursos necessários: fatores que influenciam o sistema sob a perspectiva de quais recursos: financeiro, humano e tempo, são necessários para o desenvolvimento de uma aplicação nos modelos arquiteturais analisados;

Características arquiteturais: fatores que influenciam o sistema sob a perspectiva de capacidade da arquitetura de prover algum aspecto que possa ser relevante para a empresa, como escalabilidade;

Manutenibilidade: fatores que influenciam o sistema sob a perspectiva cotidiana de manutenção de cada modelo arquitetural, como processos rotineiros de *deploy* e integração contínua e comunicação das equipes;

Evolucionabilidade: fatores que influenciam o sistema sob a perspectiva de tomada de decisão dentro de cada modelo arquitetural.

Dentro de cada perspectiva foram listados **aspectos** levantados pelos autores. Esses aspectos são responsáveis por gerar ou sofrer algum **impacto** dentro do estilo arquitetural, a exemplo: sob o aspecto financeiro a arquitetura monolítica gera o impacto de ser de baixo custo.

Nota-se que a fundamentação teórica foi apresentada sob a organização dessas perspectivas e aspectos com o intuito de facilitar a compreensão do presente estudo.

3.3.3 Etapa 3 - Síntese

Neste ponto, as perspectivas teóricas sobre a arquitetura estavam definidas e iniciou-se uma nova etapa de sintetizar as informações discutidas na fundamentação teórica com o intuito de padronizar e classificar os aspectos levantados. O Quadro 1 visa apresentar as definições tomadas nessa etapa do trabalho com o intuito de realizar a classificação dos aspectos elencados.

Quadro 1 – Definições adotadas

Estilo arquitetural	Estilo arquitetural em análise.
Perspectiva	Ponto de vista sob o qual a arquitetura está sendo analisada.
Aspecto	Fator que impacta ou é impactado pela arquitetura dentro daquela perspectiva.
Impacto	Efeito gerado mediante determinado aspecto.
Causa	característica presente na natureza daquela arquitetura que é responsável por gerar o dado impacto.
Fatores adversos	Fatores que podem mudar a polaridade do impacto
Constância	Tendência do impacto a aumentar ou diminuir a medida que a base de código cresce.
Embasamento	Seção de referência na fundamentação teórica.

Mediante tais definições, construiu-se o Quadro 2 como o modelo adotado na sintetização do referencial teórico. Na seção seção 4.1 este modelo é aplicado, fazendo a análise sobre o referêncial teórico, e ao todo foram levantados 14 aspectos que de alguma forma impactam ou são impactados pelo estilo arquitetural. Vale ressaltar que, nem todos os aspectos possuem todos os pontos elencados no Quadro 2.

Quadro 2 – Quadro modelo aplicado na sintése dos aspectos arquiteturais elencados

A escolha de um	estilo arquitetural
sob a perspectiva	analisada
mediante o aspecto	levantado no referencial teórico
gera um impacto	positivo ou negativo sob a arquitetura
devido à	uma característica do estilo arquitetural que predispõem aquele impacto
contudo	é preciso lidar com alguns fatores adversos
consciente de que este aspecto tende a	decair ou a aumentar mediante o fator adverso mencionado
com base na	seção de referência na fundamentação teórica

Ao final, o resultado dessa análise é resumido em um mapa mental, semelhante a Figura 7, com o intuito de facilitar a compreensão sobre os aspectos analisados. Assim, mediante a análise desses aspectos sintetizados, foi possível classifica-los em:

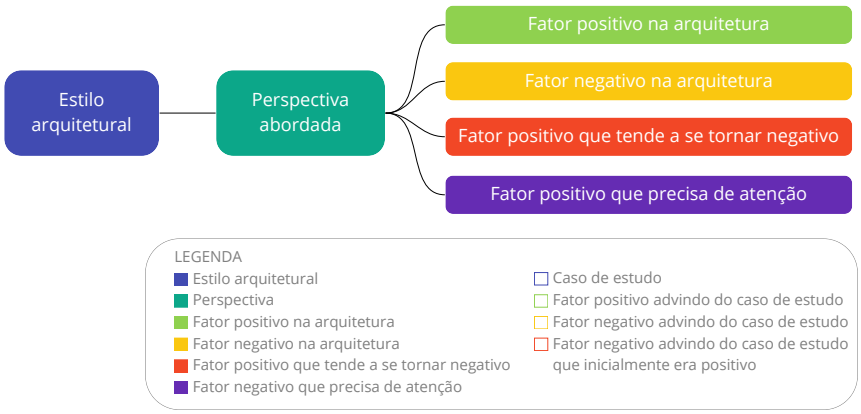

Figura 7 – Modelo de mapa mental utilizado para ilustrar o resultado das sínteses

Fatores positivos: fatores que beneficiam o sistema diante um dado contexto;

Fatores negativos: fatores que o modelo arquitetural não provê ou possui alguma limitação;

Fatores positivos que precisam de atenção: fatores que são positivos no modelo arquitetural mas que precisam de cuidados adicionais para não se tornarem um problema

Fatores positivos que tendem a se tornar negativos: fatores que são positivos no modelo arquitetural mas que tendem a se tornar um problema a medida que a base de código cresce.

3.3.4 Etapa 4 - Estudos de caso

Com o propósito de enriquecer o presente trabalho e avaliar os aspectos levantados nas etapas anteriores sob uma concepção prática, nesta etapa visou-se analisar casos reais de empresas que passaram pelo conflito de escolha arquitetural entre microsserviços e monolíticos. Foram analisados quatro casos:

KN Login: um sistema monolítico inflado de funcionalidades, no qual a equipe se encontra impossibilitada de migrar para uma arquitetura de microsserviços mediante a complexidade para fazê-lo;

Otto: um sistema monolítico legado que passa pela migração para a arquitetura de microsserviços;

Segment: um sistema monolítico relativamente novo, que passa pela migração para a arquitetura de microsserviços e depois passa por uma nova migração para a arquitetura monolítica novamente.

RuaDois: uma *startup* que começou a sua arquitetura em microsserviços e depois optou por mudar para uma arquitetura monolítica.

Para cada caso foi apresentado o contexto no qual o sistema abordado está inserido, seguido dos seguintes tópicos:

1. **Problemática da aplicação:** compreensão dos problemas e dificuldades enfrentados pela empresa na arquitetura inicialmente escolhida;
2. **Solução adotada:** compreensão sobre as decisões tomadas pela empresa mediante os problemas enfrentados;
3. **Panorama pós-adoção da solução:** compreensão sobre como a solução escolhida impactou o desenvolvimento de software dentro da empresa;
4. **Análise sobre o caso de estudo:** análise sobre o caso estudado mediante as perspectivas e aspectos levantados nas etapas anteriores.

3.3.5 Etapa 5 - Análise comparativa final

Por fim, mediante a base de informações reunidas sobre ambos os estilos arquiteturais nas etapas anteriores, fez-se uma análise final comparando as duas arquiteturas, passando por suas características mais básicas até a sua evolução no dia a dia dos engenheiros de software.

4 Desenvolvimento

Esta seção tem por propósito apresentar a pesquisa elaborada a respeito dos estilos arquiteturais abordados no presente trabalho. O escopo deste capítulo abrangerá o estudo feito sobre os objetos de análise referidos na seção 3.1, os aspectos levantados sobre cada arquitetura e análise comparativa realizada.

As seções estão dispostas em:

1. **Síntese das perspectivas teóricas** síntese a respeitos das informações coletadas nos referenciais teóricos sobre cada arquitetura;
2. **Análise dos casos de estudo** detalhamento sobre as experiências de algumas empresas que optaram por fazer a migração de uma arquitetura monolítica para uma arquitetura de microsserviços ou vice-versa.

4.1 Síntese das perspectivas teóricas

O presente tópico visa apresentar a síntese das perspectivas abordadas na seção 2.4 da fundamentação teórica com o intuito de esclarecer os pontos chaves presentes em cada arquitetura. Para tal, organizou-se os temas discutidos na fundamentação teórica no modelo do Quadro 2 apresentado na seção metodológica.

4.1.1 Síntese da arquitetura monolítica

Quadro 3 – Arquitetura monolítica - síntese sobre o domínio do problema

A escolha de uma	arquitetura monolítica
sob a perspectiva	da problemática a ser resolvida
mediante o aspecto	de domínio sobre o problema
gera um impacto	positivo se o domínio sobre o contexto é baixo
devido à	arquitetura simplista
com base na	subseção 2.4.1

Quadro 4 – Arquitetura monolítica - síntese sobre o tamanho da aplicação

A escolha de uma	arquitetura monolítica
sob a perspectiva	da problemática a ser resolvida
mediante o aspecto	do tamanho planejado para a aplicação
gera um impacto	positivo se a aplicação é pequena
devido à	arquitetura simplista e a simplicidade dos processos operacionais
com base na	subseção 2.4.1

Quadro 5 – Arquitetura monolítica - síntese sobre os recursos financeiros

A escolha de uma	arquitetura monolítica
sob a perspectiva	recursos necessários
mediante o aspecto	de recursos financeiros
gera um impacto	baixo custo
devido à	arquitetura simplista e a simplicidade dos processos operacionais
com base na	subseção 2.4.2.1

Quadro 6 – Arquitetura monolítica - síntese sobre os recursos humanos

A escolha de uma	arquitetura monolítica
sob a perspectiva	recursos necessários
mediante o aspecto	de recursos humanos
gera um impacto	não necessitar de expertise sobre as tecnologias
devido à	arquitetura simplista e a simplicidade dos processos operacionais
com base na	subseção 2.4.2.2

Quadro 7 – Arquitetura monolítica - síntese sobre esforço e tempo inicial

A escolha de uma	arquitetura monolítica
sob a perspectiva	recursos necessários
mediante o aspecto	esforço e tempo inicial
gera um impacto	baixo esforço inicial e rápido lançamento no mercado
devido à	arquitetura simplista e a simplicidade dos processos operacionais
com base na	subseção 2.4.2.3

Quadro 8 – Arquitetura monolítica - síntese sobre escalabilidade

A escolha de uma	arquitetura monolítica
sob a perspectiva	de características arquiteturais
mediante o aspecto	da escalabilidade
gera um impacto	de escalabilidade limitada
devido à	baixa modularidade da arquitetura
com base na	subseção 2.4.3.1

Quadro 9 – Arquitetura monolítica - síntese sobre tolerância a falhas

A escolha de uma	arquitetura monolítica
sob a perspectiva	de características arquiteturais
mediante o aspecto	de tolerância a falhas
gera um impacto	não possui tolerância a falhas
devido à	a existência de um único ponto de falha em toda a arquitetura
com base na	subseção 2.4.3.2

Quadro 10 – Arquitetura monolítica - síntese sobre confiabilidade

A escolha de uma	arquitetura monolítica
sob a perspectiva	de características arquiteturais
mediante o aspecto	de confiabilidade do sistema
gera um impacto	alta confiabilidade
contudo	é preciso lidar com a testabilidade do sistema
consciente de que este aspecto tende a	decair a medida que a base de código aumenta
com base na	subseção 2.4.3.3

Quadro 11 – Arquitetura monolítica - síntese sobre performance

A escolha de uma	arquitetura monolítica
sob a perspectiva	de características arquiteturais
mediante o aspecto	de performance do sistema
gera um impacto	baixa performance
devido à	arquitetura simplista
com base na	subseção 2.4.3.4

Quadro 12 – Arquitetura monolítica - síntese sobre o processo de *deploy*

A escolha de uma	arquitetura monolítica
sob a perspectiva	de manutenibilidade
mediante o aspecto	de implantação do sistema
gera um impacto	de fácil implantação
devido à	arquitetura simplista
contudo	é preciso lidar com a crescente base de código
consciente de que este aspecto tende a	decair a medida que a base de código fica maior
com base na	subseção 2.4.4.1

Quadro 13 – Arquitetura monolítica - síntese da testabilidade

A escolha de uma	arquitetura monolítica
sob a perspectiva	de manutenibilidade
mediante o aspecto	de testabilidade do sistema
gera um impacto	de facilidade para realizar testes
devido à	arquitetura simplista
contudo	é preciso lidar com a crescente complexidade
consciente de que este aspecto tende a	decair a medida que a complexidade do sistema aumenta
com base na	subseção 2.4.4.2

Quadro 14 – Arquitetura monolítica - síntese da comunicação

A escolha de uma	arquitetura monolítica
sob a perspectiva	de manutenibilidade
mediante o aspecto	de comunicação entre os times
gera um impacto	negativo
devido aos	times segregados por domínio tecnológico
com base na	subseção 2.4.4.3

Quadro 15 – Arquitetura monolítica - síntese da heterogeneidade das tecnologias

A escolha de uma	arquitetura monolítica
sob a perspectiva	de evolucionabilidade
mediante o aspecto	de heterogeneidade das tecnologias
gera um impacto	de heterogeneidade limitada
devido ao	modelo arquitetural com alto acoplamento
com base na	subseção 2.4.5.1

Quadro 16 – Arquitetura monolítica - síntese sobre complexidade do sistema

A escolha de uma	arquitetura monolítica
sob a perspectiva	de evolucionabilidade
mediante o aspecto	da complexidade do sistema
gera um impacto	de baixa complexidade
devido ao	arquitetura simplista
contudo	é preciso lidar com o acoplamento do código
consciente de que este aspecto tende a	crescer a medida que ao crescimento da base de código
com base na	subseção 2.4.5.1

A Figura 8 visa ilustrar um resumo dos pontos destacados com base na síntese do referêncial teórico. Nela podemos observar de verde os fatores positivos da arquitetura, de amarelo os fatores negativos e de vermelho os fatores que inicialmente são positivos dentro do modelo arquitetural mas que tendem a decair a medida que a base de código aumenta.

É importante ter em mente que os fatores marcados como negativo, cor amarela, são características da arquitetura monolítica, com as quais a equipe de software ao optar por este modelo arquitetural vai ter que lhe dar conscientemente que estes problemas existem. Já os fatores marcados com a cor vermelha, que tendem a cair, são mais traiçoeiros, de forma que inicialmente eles não são um problema, pelo contrário, eles são algo positivo quando se tem um monolítico novo, porém a medida que a base de código cresce eles tendem a se tornar um problema que se instala na arquitetura sem a equipe de TI perceba.

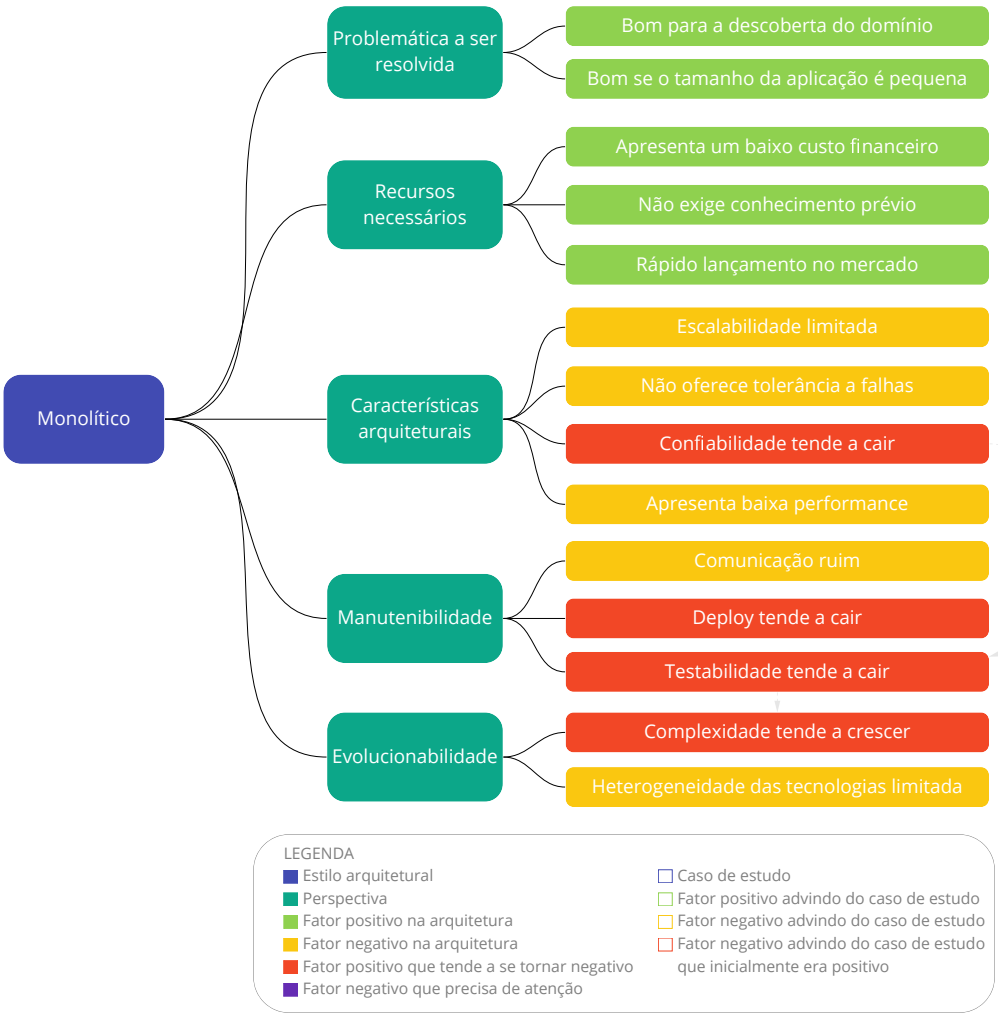

Figura 8 – Mapa mental do processo de síntese realizado sobre a arquitetura monolítica

4.1.2 Síntese da arquitetura de microsserviços

Quadro 17 – Arquitetura de microsserviços - síntese sobre o domínio do problema

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	da problemática a ser resolvida
mediante o aspecto	de domínio sobre o problema
gera um impacto	de necessidade de domínio sobre o contexto
devido à	definição dos limites que impacta toda a arquitetura
com base na	subseção 2.4.1

Quadro 18 – Arquitetura de microsserviços - síntese sobre o tamanho da aplicação

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	da problemática a ser resolvida
mediante o aspecto	do tamanho planejado para a aplicação
gera um impacto	de custo muito alto para aplicações pequenas
devido ao	custo muito alto de implementação e manutenção
com base na	subseção 2.4.1

Quadro 19 – Arquitetura de microsserviços - síntese sobre os recursos financeiros

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	recursos necessários
mediante o aspecto	de recursos financeiros
gera um impacto	alto custo
devido à	necessidade de mão de obra especializada, monitoramento, replicação...
com base na	subseção 2.4.2.1

Quadro 20 – Arquitetura de microsserviços - síntese sobre os recursos humanos

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	recursos necessários
mediante o aspecto	de recursos humanos
gera um impacto	necessita de expertise sobre as tecnologias
devido à	complexidade da arquitetura
com base na	subseção 2.4.2.2

Quadro 21 – Arquitetura de microsserviços - síntese sobre esforço e tempo inicial

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	recursos necessários
mediante o aspecto	esforço e tempo inicial
gera um impacto	alto esforço inicial e demorado lançamento no mercado
devido à	complexidade da arquitetura e a necessidade de automatizar todos os processos operacionais
com base na	subseção 2.4.2.3

Quadro 22 – Arquitetura de microsserviços - síntese sobre escalabilidade

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de características arquiteturais
mediante o aspecto	da escalabilidade
gera um impacto	de ser altamente escalável
devido à	alta modularidade e automatização dos processos
com base na	subseção 2.4.3.1

Quadro 23 – Arquitetura de microsserviços - síntese sobre tolerância a falhas

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de características arquiteturais
mediante o aspecto	de tolerância a falhas
gera um impacto	alta tolerância
devido à	presença de vários pontos de falha, comprometendo apenas parcialmente a aplicação
contudo	é preciso lidar com a escalabilidade da rede
com base na	subseção 2.4.3.2

Quadro 24 – Arquitetura de microsserviços - síntese sobre confiabilidade

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de características arquiteturais
mediante o aspecto	de confiabilidade do sistema
gera um impacto	alta confiabilidade
contudo	é preciso lidar com eventuais inconsistências e escalabilidade da rede
com base na	subseção 2.4.3.3

Quadro 25 – Arquitetura de microsserviços - síntese sobre performance

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de características arquiteturais
mediante o aspecto	de performance do sistema
gera um impacto	baixa performance
devido à	dependência da rede e da adição de várias camadas de segurança na comunicação entre os serviços
com base na	subseção 2.4.3.4

Quadro 26 – Arquitetura de microsserviços - síntese sobre o processo de *deploy*

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de manutenibilidade
mediante o aspecto	de implantação do sistema
gera um impacto	de fácil e rápida de implantação
devido	aos processos automatizados
contudo	é preciso automatizar os processos operacionais
com base na	subseção 2.4.4.1

Quadro 27 – Arquitetura de microsserviços - síntese da testabilidade

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de manutenibilidade
mediante o aspecto	de testabilidade do sistema
gera um impacto	fácil de uma perspectiva local e difícil de uma perspectiva global
devido à	modularidade local e a complexidade global respectivamente
com base na	subseção 2.4.4.2

Quadro 28 – Arquitetura de microsserviços - síntese da comunicação

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de manutenibilidade
mediante o aspecto	de comunicação entre os times
gera um impacto	positivo
devido aos	times multifuncionais
com base na	subseção 2.4.4.3

Quadro 29 – Arquitetura de microsserviços - síntese da heterogeneidade das tecnologias

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de evolucionabilidade
mediante o aspecto	de heterogeneidade das tecnologias
gera um impacto	de alta heterogeneidade
devido ao	modelo arquitetural com baixo acoplamento
com base na	subseção 2.4.5.1

Quadro 30 – Arquitetura de microsserviços - síntese sobre complexidade do sistema

A escolha de uma	arquitetura de microsserviços
sob a perspectiva	de evolucionabilidade
mediante o aspecto	da complexidade do sistema
gera um impacto	de baixa complexidade de uma perspectiva local e alta complexidade de uma perspectiva global
devido ao	característica distribuída da arquitetura
com base na	subseção 2.4.5.1

A Figura 9 visa ilustrar um resumo dos pontos destacados com base na síntese elaborada a partir do referêncial teórico. Nela podemos observar de verde os fatores positivos da arquitetura, de amarelo os fatores negativos e de roxo os fatores que em geral são positivos na arquitetura mas que exigem que o time de desenvolvimento lide com algum fator adverso. Por exemplo, o *deploy* é apontado como um fator positivo dentro da arquitetura de microsserviços, visto que é fácil e rápido atualizar um serviço, contudo existe o fator adverso de que todo o processo deve estar automatizado, caso a equipe não automatize esse processo, provavelmente eles terão algum problema pra lidar com o *deploy* dentro dessa arquitetura.

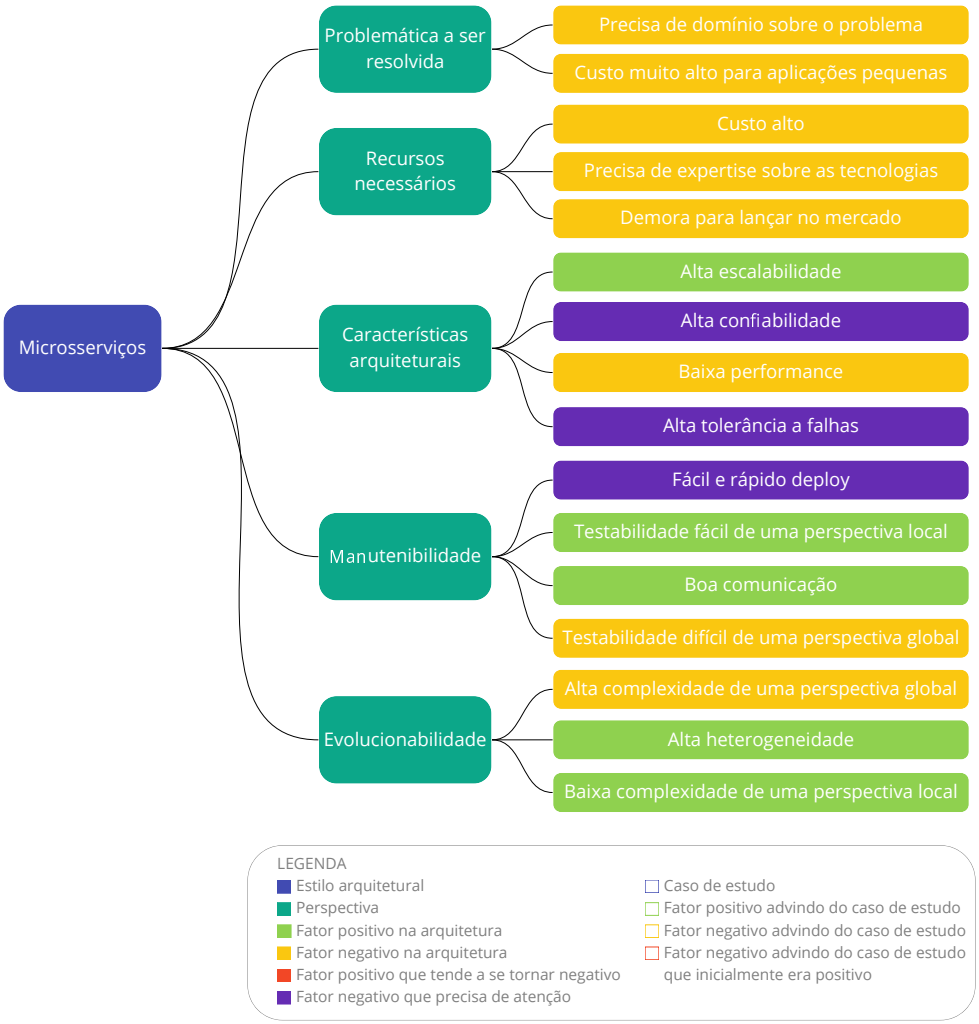

Figura 9 – Mapa mental do processo de síntese realizado sobre a arquitetura de microsserviços

4.2 Análise dos casos de estudo

A presente seção visa relatar casos reais de empresas que vivenciaram os estilos arquiteturais estudados. A proposta é entender o contexto dessas empresas, as decisões tomadas e qual a percepção que elas obtiveram sobre seus sistemas ao experimentar diferentes arquiteturas.

Serão abordados três casos:

1. **KN Login:** um monolítico legado transformado em uma série de sistemas autocontidos, com o intuito de caminhar em direção a uma arquitetura de microsserviços;
2. **Otto:** a reimplementação do zero de um monolítico em uma arquitetura de sistemas autocontidos que posteriormente é evoluída para microsserviços;
3. **Segment**: a transição de um monolítico para uma arquitetura de microsserviços, seguido do retorno para a arquitetura monolítica após as várias dificuldades encontradas.

4.2.1 KN Login

O caso apresentado nessa seção é um relato de Annenko (2016) a respeito da transição realizada por Björn Kimminich[1] e sua equipe na empresa Kuehne + Nagel[2] de um sistema monolítico para sistemas auto-contidos[3].

A Kuehne + Nagel é uma empresa especializada em transportes e logística que presta serviços em uma escala global. Dentre as aplicações de software utilizadas na empresa para entrega dos seus produtos, existe um serviço chamado KN Login, o qual proporciona diversas funcionalidades importantes para a empresa, como gerenciamento dos contêineres transportados, auxilia no controle da integridade e eficiência da cadeia de suprimentos de seus clientes, rastreio dos contêineres com análise das condições ambientais, etc.

4.2.1.1 Problemática da aplicação

O KN Login é um sistema monolítico iniciado em 2007 construído em cima do *framework* Java Web[4]. Inicialmente foi construído como uma aplicação simples mas com o passar do tempo se tornou um imenso monolítico com mais de um milhão de linhas de

[1] Gerente sênior de arquitetura de TI na empresa Kuehne + Nagel. Mais informações em: <https://www.linkedin.com/in/bkimminich>
[2] Vide <https://home.kuehne-nagel.com>
[3] O relato original e completo da autora sobre o caso pode ser encontrado no link <https://www.elastic.io/breaking-down-monolith-microservices-and-self-contained-systems/>
[4] Vide <https://docs.oracle.com/javase/7/docs/technotes/guides/javaws/developersguide/overview.html>

código e com uma grande variedade de atribuições e responsabilidades dentro da empresa. Esse contexto trouxe a equipe de software da Kuehne + Nagel as seguintes dificuldades:

- A impossibilidade de trocar o *framework* Java Web para outra tecnologia que se adeque melhor as necessidades da empresa, uma vez que o *framework* se encontra enraizado no sistema;
- A manutenção de dois *frameworks* em paralelo na mesma aplicação, visto que, a equipe de TI deles recomenda o uso do Spring Framework[5] em detrimento do Java Web Framework sempre que possível;
- Várias tecnologias diferentes de User Interface (UI) utilizadas para conseguir atender as diferentes problemáticas enfrentadas pela empresa;
- O sistema se tornou bastante instável mediante a vasta variedade de tecnologias aplicadas, tornando difícil a manutenção do mesmo pela equipe;
- Alta complexidade que dificulta a entrada de novos membros no time de desenvolvimento.

4.2.1.2 Solução adotada

Mediante as dificuldades enfrentadas, a Kuehne + Nagel optou por seguir os seguintes passos:

- Parar de adicionar quaisquer funcionalidades ou modificações no sistema por mais importante que estas sejam para o escopo do negócio;
- Identificar os módulos e limites do monolítico, de forma que fosse possível criar uma separação ou até mesmo cortar tais partes do monolítico;
- Ter ciência de quão dependentes são cada módulo dentro do monolítico, de tal forma que alguns sejam aparentemente impossíveis de separar;
- Começar a separação de cada módulo aos poucos, de maneira que pedaços possam ser desativados no monolítico a medida que cada novo módulo é liberado;

Após analisar esses quatro fatores, a equipe técnica da Kuehne + Nagel chegou ao ponto de que a mudança diretamente para microsserviços seria inviável mediante a grande complexidade do monolítico, além de que esta mudança não iria contribuir para deixar

[5] Vide <https://spring.io>

mais fácil a visualização que eles tinham do sistema. Diante desta situação a equipe optou por abordar uma arquitetura de sistemas autocontidos[6].

Sistemas autocontidos segundo Annenko (2016), diference dos microsserviços por serem aplicações web autônomas e substituíveis as quais tendem a ser maiores do que um microsserviço e possuem uma unidade de UI própria. Assim, a Kuehne + Nagel construiu o seu primeiro sistema autocontido chamado KN Freightnet, o qual contou com a duplicação de algumas funcionalidades presentes no KN Login, representando o primeiro passo para minimizar a complexidade do KN Login.

4.2.1.3 Panorama pós-adoção da solução

O relato apresentado por Annenko (2016), não traz informações a respeito de quais foram os resultados efetivos da adoção de sistemas autocontidos mediante as dificuldades que a equipe enfrentava dentro no monolítico KN Login, mas a autora traz a perspectiva de utilizar sistemas autocontidos como um passo intermediário em direção aos microsserviços quando o monolítico em mãos tem uma complexidade muito alta para ser quebrado.

4.2.1.4 Análise sobre o caso de estudo

O caso da KN Login traz a típica história dos monolíticos: inicialmente uma aplicação que deveria ser simples mas que com o tempo o sistema foi inflando com várias funcionalidades. Nota-se que a equipe do KN Login se deparou com um contexto complexo: rastreio dos contêineres, análise climática, etc. E mediante esse contexto, eles tentaram lidar com toda a problemática dentro do monolítico sem considerar a limitação que este estilo arquitetural oferece referente a heterogeneidade das tecnologias envolvidas.

Diante da combinação: contexto complexo e heterogeneidade de tecnologias, o monolítico KN Login resultou em uma aplicação de alta complexidade e baixa confiabilidade, se tornando um sistema instável e difícil de manter. A Figura 10 visa ilustrar os pontos destacados na arquitetura monolítica do KN Login mediante aos fatores levantados sobre essa arquitetura na subseção 4.1.1.

4.2.2 Otto

Nesta seção será apresentado o caso relatado por Steinacker (2015), a respeito da construção do novo *e-commerce* Otto[7].O Grupo Otto[8] é uma empresa de origem alemã

[6] Abordagem arquitetural que consiste em quebrar o sistema em vários sistemas independentes, cada um com sua propría UI, camada de lógica e persistência de dados.

[7] O relato original e completo do autor sobre o caso pode ser encontrado nos links <https://www.otto.de/jobs/technology/techblog/artikel/on-monoliths-and-microservices_2015-09-30.php> e <https://www.otto.de/jobs/technology/techblog/artikel/why-microservices_2016-03-20.php>

[8] Vide: <https://www.otto.de>

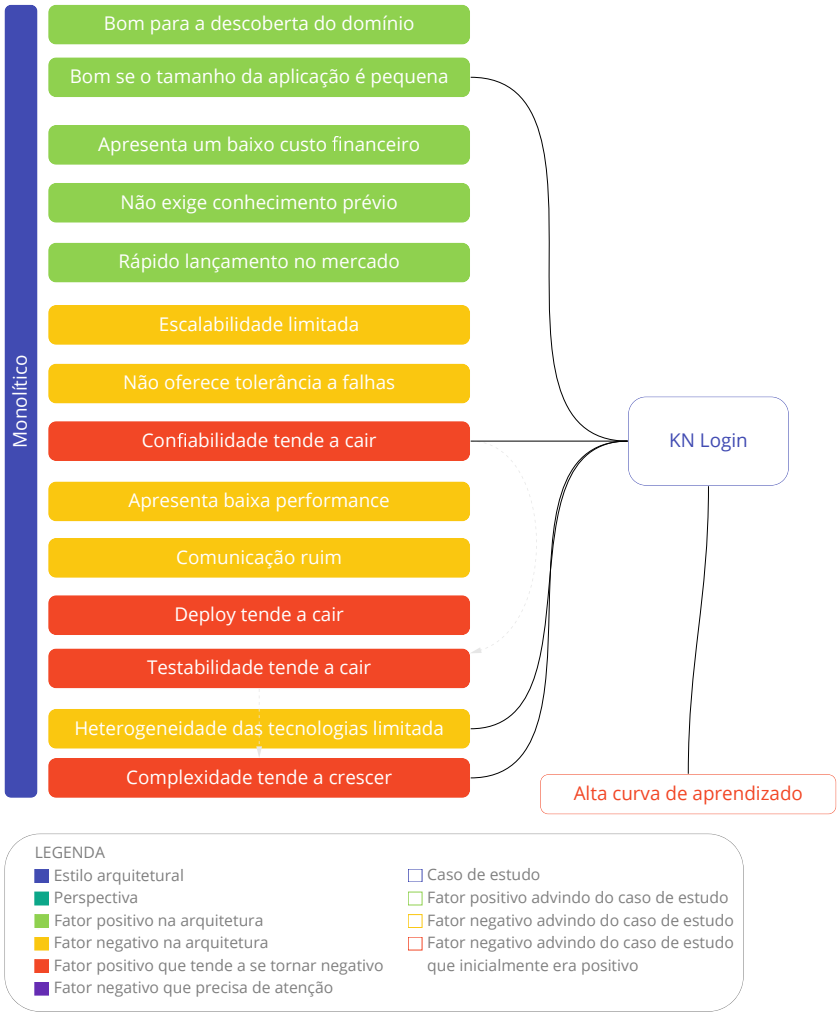

Figura 10 – Fatores apresentados no caso de estudo da arquitetura monolítica do sistema KN Login

que trabalha em diversos países desenvolvendo soluções de negócios voltadas a necessidade do setor de *e-commerce*.

4.2.2.1 Problemática da aplicação

O sistema da Otto originou-se inicialmente em um projeto que deveria ser simples, no qual implicitamente a equipe adotou uma macroarquitetura monolítica sem considerar ou até mesmo enxergar que tal decisão estava sendo tomada. E mediante tal decisão, a equipe passou a ter dificuldades com os problemas apresentados abaixo:

- Escalabilidade limitada ao balanceamento de carga;
- Dificuldade de manutenção crescia juntamente com o crescimento da aplicação;
- Dificuldade de contratar desenvolvedores dispostos a trabalhar em um sistema de alta complexidade;

- Dificuldades ao realizar o *deploy* principalmente no caso de aplicações sem períodos de inatividade;
- Dificuldades para trabalhar com várias equipes diferentes em uma mesma aplicação.

Steinacker (2015) enfatiza três pontos que para ele são importantes:

- O desenvolvimento começa em equipe e no início a aplicação é bastante compreensível, necessitando apenas de uma única aplicação;
- Quando se tem um único aplicativo, o custo inicial é relativamente baixo: construção do repositório, automatização da *build* e do processo de *deploy*, configurar ferramentas de monitoramento, etc. Ativar e operar um novo aplicativo é relativamente fácil;
- É mais complexo operar grandes sistemas distribuídos do que um *cluster* balanceador de carga.

4.2.2.2 Solução adotada

Diante do contexto apresentado, a Otto decidiu por começar um novo sistema de *e-commerce*, adotando dessa vez uma arquitetura semelhante a apresentada na subseção 4.2.1 de sistemas autocontidos. Para tal, eles estabeleceram quatro equipes funcionais dando origem a quatro aplicações. Os custos iniciais de cada aplicação foram minimizados padronizando o processo de automação.

A medida que o sistema da Otto evoluía foram criados novos sistemas autocontidos, cada um com sua própria equipe de desenvolvimento, seu próprio *frontend* e seu próprio banco de dados, tudo dentro de um escopo de negócio muito bem definido a respeito das suas atribuições.

A Otto definiu aspectos macro e microarquiteturais. Os aspectos microarquiteturais estavam voltados ao contexto local de cada sistema autocontido construído, enquanto que nos aspectos macro foram definidos questões mais globais a respeito da arquitetura, como por exemplo:

- A comunicação entre os sistemas autocotidos deveria ser feita sempre por meio do modelo REST;
- Não poderia existir estado mutável compartilhado entre as aplicações;
- Os dados devem ser compartilhados por meio de uma Application Programming Interface (API) REST, havendo somente uma aplicação responsável pelo dado e as outras unicamente com a permissão de leitura sobre o mesmo.

Nessa abordagem arquitetural escolhida, a Otto se deparou com dificuldades no compartilhamento de dados entre as aplicações, o que os levou a adotar a estratégia de replicação de dados, na qual, eventualmente eles precisam lhe dar com inconsistências temporárias entre os sistemas.

Contudo, alguns sistemas autocontidos ficaram bastante volumosos após cerca de três anos trabalhando em cima dessa arquitetura. Essa nova situação fez com que a Otto adota-se uma nova medida em direção a uma arquitetura de microsserviços. Decisão esta que segundo Steinacker (2016) não gerou um processo tão árduo dentro da empresa uma vez que eles já vinham trabalhando em cima de uma arquitetura bastante modularizada e já trabalhavam de forma concisa as questões de limites de cada aplicação.

4.2.2.3 Panorama pós-adoção da solução

Após a adoção de uma arquitetura autocontida e a migração para uma arquitetura de microsserviços, Steinacker (2016) traz como pontos positivos de ambas as arquiteturas os seguintes pontos:

- Facilidade em estabelecer uma pirâmide de testes que funcione de forma eficiente e rápida, auxiliando os desenvolvedores no lançamento de novas versões do código;
- Facilidade em gerenciar e integrar o trabalho de diferentes equipes uma vez que cada uma está trabalhando em seu próprio serviço;
- Aumento na velocidade de implantação de novas funcionalidades, sendo mais fácil testar e coletar rapidamente *feedback*;
- Cada aplicativo pode ser implementado de forma independente;
- Não há necessidade de coordenar o processo de implantação, cada equipe possui autonomia para gerenciar o seu processo de implantação sem afetar as demais equipes;
- Baixa curva de aprendizado, cada serviço é de fácil compreensão pelos desenvolvedores;
- Escalabilidade em relação a aplicação e aos próprios times de desenvolvimento;
- A complexidade de um serviço é extremamente baixa quando comparada a uma arquitetura monolítica;
- O sistema não está limitado a uma tecnologia específica;
- Exige que todos os processo estejam automatizados;
- Facilidade em reverter a versão do projeto quando algum *bug* é introduzido;

- A ocorrência de falhas não afeta o sistema por inteiro, somente uma parte dele;
- Permite manter com facilidade a alta coesão e o baixo acoplamento do sistema;

Segundo a visão do autor, os microserviços possibilitam a construção de um sistema de software muito mais sustentável ao longo dos anos do que um sistema monolítico, de forma que nessa arquitetura é possível substituir pequenos pedaços da aplicação quando for necessário, enquanto que em sistemas monolíticos essa abordagem se torna muito mais complexa.

A adoção de sistemas autocontidos é apresentada por ele como uma porta de entrada na qual é possível resolver problemas semelhante aos microsserviços de uma forma mais pragmática, deixando ainda a possibilidade de migrar futuramente para uma arquitetura de microsserviços de uma forma não tão árdua (STEINACKER, 2016).

4.2.2.4 Análise sobre o caso de estudo

A figura Figura 11 ilustra os pontos da arquitetura monolítica levantados pelo caso de estudo da Otto. Percebe-se que a experiência deles atesta os pontos referentes ao início do projeto quando a aplicação é pequena, de fácil compreensão por todos, tem um custo baixo, etc. Também atesta os problemas de complexidade e de implantação que tendem nesse modelo arquitetural a crescer juntamente com a base de código.

A tomada de decisão por migrar para uma arquitetura de sistemas auto-contidos auxilia eles como um passo intermediário antes de aderir toda a complexidade da arquitetura de microsserviços. Nessa abordagem percebe-se que eles vivenciam características dos dois estilos arquiteturais. Olhando da perspectiva dos microsserviços, eles definem protocolos de comunicação e compartilhamento de dados e lidam com eventuais inconsistência de dados (um problema comum da arquitetura de microsserviços). Por outro lado, eles ainda precisam lidar com a crescente base de código dos monolíticos dentro de cada sistema auto-contido.

A figura Figura 12 ressalta os pontos da arquitetura de microsserviços adotada pela Otto, na qual nota-se alta escalabilidade, alta tolerância a falhas, facilidade no *deploy*, testabilidade e alta heterogeneidade de tecnologias observados no caso de estudo. A experiência da Otto também traz outros pontos dessa arquitetura, não presentes no mapa mental construído, como baixa curva de aprendizado e a necessidade de automatizar todos os processos.

Um ponto que se diferencia do esperado, de acordo com a fundamentação teórica levantada para esse trabalho, é que a Otto apresenta a coleta de *feedbacks* como um ponto positivo da arquitetura de microsserviços enquanto o referencial teórico aponta a arquitetura monolítica como mais indicada para tal. Nesse sentido, percebe-se que existe o

Figura 11 – Fatores apresentados no caso de estudo da arquitetura monolítica da Otto

fator tempo de projeto envolvido. Como relatado na subseção 2.4.1 e na subseção 2.4.2.3, ao iniciar um projeto a arquitetura monolítica é melhor para a rápida coleta de *feedbacks* por ser mais simplista, contudo no caso da Otto, eles já tinham um sistema monolítico com uma base de código grande e devido aos problemas de complexidade já discutidos evoluir e testar rápido esse sistema com os usuários já não deveria ser um processo tão fácil. Já na arquitetura de microsserviços, existe o esforço inicial de automatizar todos os processos e de planejar cuidadosamente os limites de domínio de cada aplicação, mas após essas etapas estarem bem definidas alterar e testar novos serviços tende a ser fácil.

4.2.3 Segment

O caso apresentado a seguir foi retratado por Noonan (2020) na conferência QCon London. Ela trata a respeito do sistema da Segment[9], o qual foi construído inicialmente em uma arquitetura monolítica, passou por uma migração para uma arquitetura de mi-

[9] Vide: <https://segment.com>

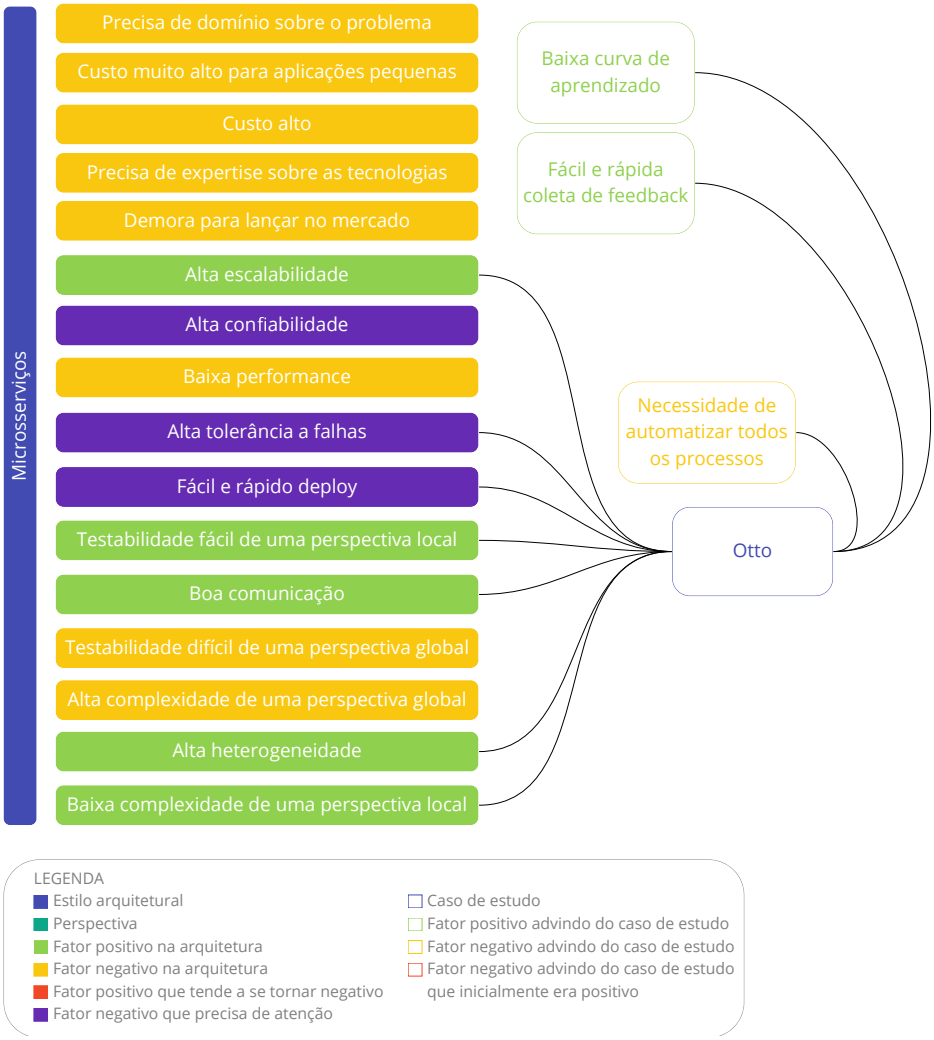

Figura 12 – Fatores apresentados no caso de estudo da arquitetura de microsserviços da Otto

crosserviços e após 3 anos em cima desta arquitetura, decidiram por retornar para a arquitetura monolítica.

A Segment é uma empresa responsável por fazer a integração entre as aplicações de software de seus clientes com diversas ferramentas de análise de dados, atuando como um *pipeline* facilitador na integração entre as partes citadas.

4.2.3.1 Problemática da aplicação

Em 2013, quando a empresa foi criada havia necessidade de construir um sistema que possuísse uma baixa sobrecarga operacional, sendo simples de gerenciar e fácil de iterar, assim, diante de tais necessidades a primeira versão do sistema da Segment foi construído em cima de uma arquitetura monolítica.

A arquitetura inicial consistia em uma API de entrada que recebia os dados enviados pelas aplicações dos clientes da Segment, e colocava esses dados em uma fila de

processamento. A partir deste ponto, havia uma monolítico responsável por consumir a fila e enviar os dados para as aplicações de destino (Google Analytics, Salesforce...). O consumo dessa fila ocorria no modelo First in, First out (FIFO)[10]. A dificuldade enfrentada pela equipe da Segment nessa arquitetura estava na tentativa de reenviar algum dado quando por algum motivo a primeira tentativa tinha falhado. De acordo com Noonan (2020), cerca de 10% das solicitações para as aplicações de destino falhavam e nesses casos, o monolítico fazia de 2 a 10 tentativas de reenvio.

Nessa abordagem, a Segment passou a enfrentar um bloqueio frontal na fila. Sempre que uma aplicação de destino estava muito instável ou ficava fora do ar, o monolítico travava a fila tentando refazer o envio, de forma que o problema se propagava: ao invés de ter uma integração X fora do ar, todas as outras integrações também paravam de funcionar visto que a fila estava bloqueada.

Após um ano trabalhando em cima dessa arquitetura, o time de desenvolvimento da Segment decidiu por mudar para uma arquitetura de microsserviços visando o isolamento do ambiente, de forma que problemas com uma determinada integração não impactasse na outra. Para tal, eles segregaram cada integração em um serviço próprio acompanhado de sua própria fila e adicionaram antes das filas um roteador capaz de receber os dados da API de entrada e direcioná-los para as filas necessárias. Essa abordagem trouxe as seguintes vantagens para a Segment:

- Diminui a necessidade de *backup* da fila, uma vez que ao falhar uma aplicação de destino eles não precisam mais salvar a fila toda, mas somente a parte referente a integração problemática;
- A falha em uma aplicação de destino não afetava mais as outras integrações;
- Facilitou a inserção de novas aplicações de destino ao sistema da empresa, permitindo um desenvolvimento mais rápido;
- Facilidade em tratar as peculiaridades de cada integração dentro do seu próprio serviço, sem precisar compatibilizar os dados com todas as aplicações de uma única vez;
- Agregou a visibilidade de toda a pilha do processo, permitindo identificar mais facilmente as falhas e a qual serviço específico elas eram referentes.

Em 2016, a Segment possui 50 serviços integrando diferentes aplicações. Com o aumento na quantidade de serviços para gerenciar eles começaram a ter algumas dificuldades, entre elas:

[10] Vide <https://pt.wikipedia.org/wiki/FIFO>

- Não possuíam processos de *build* e *deploy* automatizados, o que acaba exigindo bastante tempo da equipe mediante a quantidade de serviços no ar;
- Cada sistema realizava diferentes tratamentos sobre o dado antes de enviar para a aplicação de destino e como cada cliente podia enviar os dados de diferentes formas, acabou se tornando árduo entender o que estava se passando dentro de cada serviço. Nesse caso, eles optaram por criar bibliotecas comuns entre os serviços para fazer tal tratamento. Inicialmente, isto se refletiu positivamente dentro da equipe, mas sem os processos de implantação automatizados, a atualização da versão das bibliotecas em cada serviço se tornou um problema de tal forma que eles passaram a preferir não corrigir um *bug* dentro da biblioteca do que fazer a atualização;
- Administrar o escalonamento da arquitetura não era uma tarefa fácil. Alguns clientes geravam um tráfego de milhares de requisições por segundo, e frequentemente acontecia de algum desses clientes ativarem alguma integração com o intuito de experimentar, e apesar das regras automáticas de escalonamento e dimensionamento de recursos, o time de TI da Segment não conseguia encontrar uma regra de escalabilidade que se adequasse bem ao contexto deles. Diante desse caso, eles pensaram em soluções de supervisionamento da arquitetura, manter sempre alguns trabalhadores mínimos, etc., mas chegaram a conclusão de que essas abordagens teriam um custo muito alto para a empresa;
- Mesmo com todas as dificuldades, eles continuavam adicionando cerca de três aplicações de destino por mês. O que tornava cada vez mais complexa a base de código e fazia a carga operacional, diante dos processos não automatizados, crescer linearmente ainda que o tamanho da equipe fosse constante;
- A empresa chegou a parar de desenvolver novas funcionalidades visto que a complexidade e a dificuldade para manter os microsserviços funcionando estava altíssima;
- A carga encaminhada dos clientes havia aumentado consideravelmente e eles ainda continuavam com o mesmo problema de limitação do tráfego pelo lado das aplicações de destino, o mesmo bloqueio frontal mencionado anteriormente;

4.2.3.2 Solução adotada

Em 2017, Noonan (2020) relata que eles chegaram ao ponto de ruptura: haviam 140 serviços rodando, uma equipe pequena mantendo uma série de processos operacionais e uma base de código complexa fazendo com que a arquitetura que inicialmente parecia ideal e que os auxiliaria a escalar, se torna-se o fator paralisador de todo o sistema. Nesse ponto, eles reunirão todas as experiências que haviam reunido desde 2013, e tomaram a decisão de retornar para uma arquitetura monolítica, com um sistema único que eles fossem capaz de gerenciar e escalar.

A solução adotada reuniu todas as 140 integrações no mesmo repositório novamente, ao invés de ter uma fila para cada aplicação eles optaram por salvar os dados em um banco MySQL[11], com as condições de que:

- As linhas do banco deveriam ser imutáveis, devendo o monolítico sempre inserir o novo estado mas nunca atualizar o antigo;
- Operações de JOIN não deveriam ser realizadas. No modelo desenvolvido por eles, o monolítico deveria apenas ler informações básicas sobre os dados a serem tratados;
- A escrita deve ser a operação predominante. Nessa nova arquitetura, a maioria dos dados eram tratados em cache e tinham apenas o seu estado registrado no banco de dados.

Dessa forma, os dados chegavam ao monolítico, o qual registrava os mesmos no banco mas ainda os mantinham em cache. A medida que os dados eram tratados, o monolítico removia os dados do cache e registrava o novo estado no banco. Quando alguma integração falhava, o estado de falha era registrado no banco para que fosse tratado posteriormente por meio da estratégia de *backoff* definida por eles.

Olhando a perspectiva da escalabilidade, o monolítico podia ser replicado sempre que houvesse a necessidade. Para tal, havia um serviço responsável por gerenciar as instâncias de banco de dados. Sempre que um novo monolítico era colocado no ar, ele recebia por meio desse serviço o acesso a um banco de dados exclusivamente seu. Dessa forma, eles conseguiram automatizar o processo de escalabilidade de acordo com o uso da CPU (FRENCH-OWEN, 2018)[12].

4.2.3.3 Panorama pós-adoção da solução

A solução adotada pela Segment permitiu a eles:

- Escalar melhor a sua aplicação, lidando de forma mais eficiente com as limitações de capacidade das aplicações destino e permitindo a adição de novos serviços sem a necessidade de aumentar os custos operacionais da equipe;
- Aumentou consideravelmente a produtividade do time;
- Solucionou o problema de compatibilidade de diferentes versões de bibliotecas compartilhadas entre os serviços;
- Facilitou o desenvolvimento de novas funcionalidades para o sistema;

[11] Vide <https://www.mysql.com>

[12] A descrição completa da solução adotada pela Segment pode ser encontrada no link <https://segment.com/blog/introducing-centrifuge/>

Os problemas que eles tinham na primeira arquitetura monolítica relacionados às dificuldades de isolamento de ambiente continuaram presentes nessa nova arquitetura, contudo, agora o time de desenvolvimento da Segment soube lhe dar de forma muito mais madura com essa situação, procurando outras soluções diferente de microsserviços para tratar os obstáculos com os quais eles se deparavam.

4.2.3.4 Análise sobre o caso de estudo

O caso da Segment conta com três fases que precisam ser avaliadas: o monolítico inicial, os microsserviços e o monolítico final. A figura Figura 13 representa o primeiro monolítico construído pela empresa. A escolha inicial de uma arquitetura monolítica condiz com o contexto da empresa, a qual era composta por uma equipe pequena de engenheiros de software e estava buscando validar sua ideia e descobrir o domínio no qual estavam imersos.

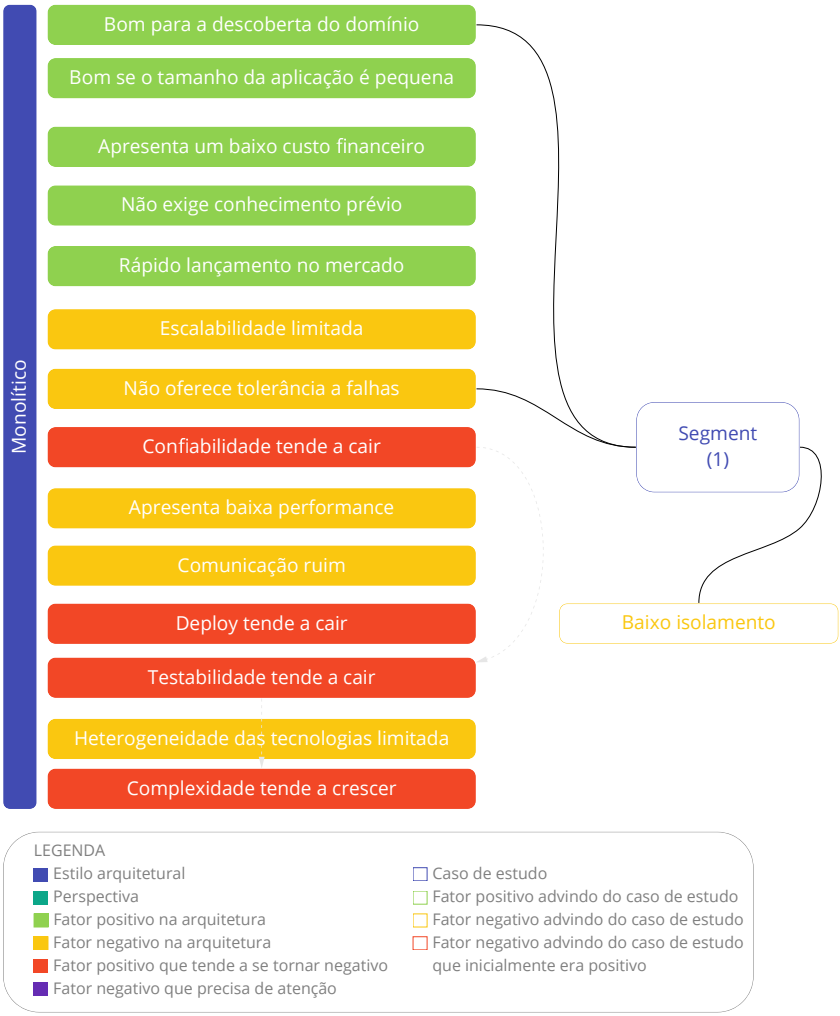

Figura 13 – Fatores apresentados no caso de estudo da primeira arquitetura monolítica da Segment

No caso, a Segment se depara logo com a limitação de tolerância a falhas dessa

arquitetura e decidem mudar para uma arquitetura de microsserviços. Visto que o sistema da Segment era um software novo, com um ano de funcionamento quando eles optaram por fazer a migração, percebe-se que outros problemas da arquitetura monolítica não aparecem no relato, como alta complexidade, dificuldades de manutenibilidade, etc., evidenciando o ponto levantado na subseção 4.1.1 de que essas características são inicialmente positivas no sistema e estão diretamente relacionadas ao tamanho da base de código.

Ao passar o sistema para a arquitetura de microsserviços, o time consegue perceber os pontos destacados na Figura 14 como a tolerância a falhas, a baixa complexidade local, etc. Vale ressaltar que a Segment também relata, semelhante a Otto, a facilidade e velocidade de adicionar novas funcionalidades nessa arquitetura. Contudo, percebe-se também a ausência da expertise, comentada na subseção 2.4.2.2, sobre este modelo arquitetural visto que os processos operacionais não são automatizados pela equipe da Segment. Este fator somado ao crescente número de serviços funcionando faz com que a equipe se afogue na grande demanda de processos operacionais exigida por essa arquitetura. Assim, eles param todo o desenvolvimento mediante a inviabilidade de gerenciar essa arquitetura nas proporções que ela tomou.

No meio desta situação, a Segment opta por voltar para a arquitetura monolítica mas de uma forma mais consciente sobre quais eram as dificuldades do contexto deles, planejando estratégias para lidar com as adversidades que eles já conheciam. Assim, eles conseguem construir um sistema que possa prover escalabilidade, tolerância a falhas e produtividade para a equipe em cima de uma arquitetura monolítica. Vale salientar que, a respeito da produtividade da equipe, a Segment se encontra com um monolítico relativamente novo, que foi planejado e assim ele apresenta o perfil de um monolítico em sua fase inicial: baixa complexidade, módulos bem definidos, etc., contudo, é necessário que a equipe se empenhe para conseguir manter essas características a medida que sua base de código cresce.

4.3 RuaDois

O caso apresentado nesta seção é o resultado de uma entrevista com Erlan Cassiano[13], *Header of Product* da *startup* RuaDois[14], sobre a experiência que eles passaram dentro da empresa ao iniciar com uma arquitetura de microsserviços e optar por migrar para uma arquitetura monolítica. Este relato se diferencia dos outros casos estudados, por ser uma migração na qual a presente autora deste documento fez parte do time de desenvolvimento acompanhado de perto as dificuldades da equipe as quais trouxeram a inspiração para realizar o presente estudo.

[13] Vide <https://www.linkedin.com/in/erlan-cassiano-56370231/>
[14] Vide <https://www.ruadois.com.br/pt-br/home>

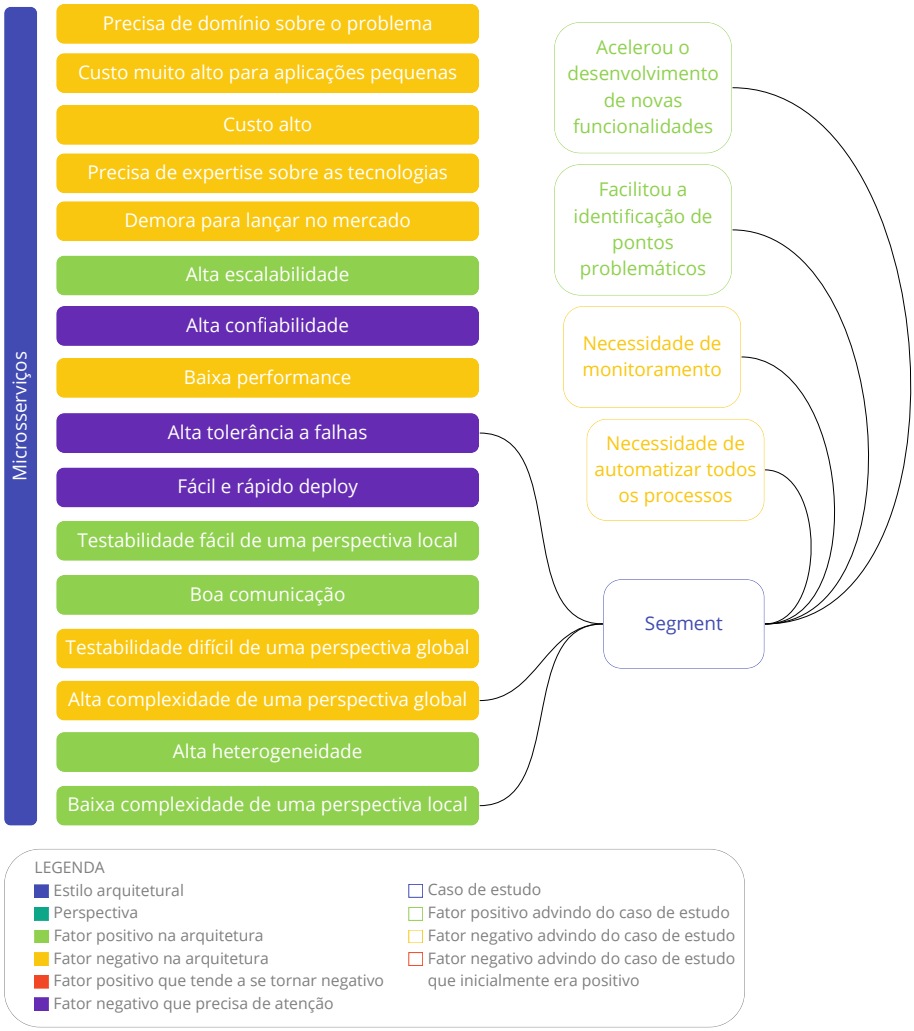

Figura 14 – Fatores apresentados no caso de estudo da arquitetura de microsserviços da Segment

A RuaDois é uma empresa brasileira que surgiu em outubro de 2018 com o propósito de solucionar as dificuldades enfrentadas tanto por parte das imobiliárias quanto por parte dos locatários no processo de locação de imóveis. A proposta[15] da empresa consiste em atuar como um facilitador para as imobiliárias ingressarem no meio digital e consequentemente propiciar seu crescimento. Para tal, eles buscam aumentar a eficiência do processo de locação por meio da digitalização, proporcionando melhor integração entre operação e tecnologias. Assim, com um processo mais digital, os recursos humanos podem ser melhores empregados no atendimento do cliente favorecendo a experiência do mesmo nesse mercado tão burocrático.

[15] Mais informações a respeito da proposta e visão da RuaDois podem ser encontradas em <https://open.spotify.com/episode/2jYKPCPLCIdDWwxpR0theT?si=dT6WUG7JSEGJH3mVVIJitQ>

4.3.0.1 Problemática da aplicação

O início do desenvolvimento do software da RuaDois se deu com o intuito de validar as ideias que os fundadores tinham de melhorar a experiência dos locatários e da própria imobiliária com o processo de visitação dos imóveis. O Cassiano relata que inicialmente eles construíram um Minimum Viable Product (MVP) visando validar a ideia, e que naturalmente o desenvolvimento da aplicação caminhou para uma arquitetura de microsserviços, composta neste começo por uma pequena aplicação *frontend* denominada *widget*, algumas funções rodando em um modelo *serverless*[16] e pelos serviços:

r2service serviço responsável por gerenciar os imóveis e as propostas recebidas pelos imóveis;

r2visit serviço responsável por gerenciar as visitas aos imóveis.

O *Header of product* da RuaDois conta que eles objetivavam com a arquitetura de microsserviços obter maior estabilidade, independência entre os sistemas e maior tolerância a falhas, principalmente em relação ao módulo de visitas visto que falhas nesse módulo geravam uma série de problemas operacionais para as imobiliárias que utilizavam o sistema. Cassiano aponta os seguintes pontos positivos que eles obtiveram nesse modelo arquitetural:

- Facilidade em validar novas ideias dentro do escopo restrito de cada serviço;
- Facilidade e rapidez em realizar o *deploy* de cada serviço;
- A possibilidade de testar diferentes abordagens de resolução do problema;
- A lógica interna de cada serviço podia ser alterada com facilidade.

Contudo, o time de desenvolvimento, composto naquele período por quatro membros, começou a enfrentar algumas dificuldades, entre elas estão:

- Um modelo rígido de comunicação foi definido entre os serviços, e mudar esses protocolos de comunicação garantindo a integração se tornou uma tarefa árdua;
- Gerenciar cada serviço, chaves de integração, acesso ao banco, etc., era mais complicado tendo vários serviços separados;
- Evoluir os microsserviços com uma equipe pequena e com pouca experiência sobre o modelo arquitetural e tecnologias era difícil;

[16] Modelo de execução no qual o provedor *cloud* é responsável por realizar alocação dinâmica dos recursos necessários para executar, em geral, as denominadas Functions as a Service (FaaS).

- A demanda inicial de clientes da *startup* era baixa para justificar o custo de manter quatro instâncias dos serviços, contando ambiente de homologação, rodando sem consumir tantos recursos;
- Dificuldade em realizar testes de integração.

4.3.0.2 Solução adotada

Mediante a equipe pequena, as dificuldades encontradas e os custos de manter os serviços a equipe técnica da RuaDois optou por migrar a arquitetura para um sistema monolítico, visando ganhar testabilidade, confiabilidade e estabilidade nos módulos cuja a ideia base da solução já tinha sido validada. Assim, eles optarem pela construção de um monolítico, contudo de forma modularizada visando que no futuro pudesse ser simples a migração de volta para uma arquitetura de microsserviços.

A primeira dificuldade da equipe foi a de convencer as outras áreas da empresa de que havia a necessidade de migrar todo o sistema para uma arquitetura monolítica e quais seriam as vantagens dessa arquitetura. Após a migração, que durou cerca de quatro meses, notaram-se as seguintes melhorias:

- A base do código de *backend* centralizada em um único lugar facilitou as dificuldades que haviam antes em relação a gerência dos microsserviços;
- Facilidade em testar todo o fluxo uma vez que todo o código se encontrava na mesma aplicação;
- Ganho de velocidade na evolução das funcionalidades já existentes;
- Ganho de estabilidade dos serviços;

Por outro lado, as seguintes dificuldades foram encontradas nesse modelo arquitetural:

- Perda de velocidade na criação de novas funcionalidades;
- Ao fazer uma pequena alteração em determinada parte do sistema é necessário rodar a *pipeline* inteira de teste tornando o processo de desenvolvimento mais lento;
- A dependência entre os módulos trazia a necessidade de alterar toda a aplicação para conseguir lançar algumas funcionalidades;
- Demandas pontuais da empresa, como a importação de XML que é executada apenas uma vez ao dia, consumiam muito da CPU e eles possuíam dificuldade para equilibrar custo e escalabilidade da instância de uma forma saudável para a empresa;

- Dificuldade de imersão de novos desenvolvedores, uma vez que havia a necessidade de compreender toda a base de código do monolítico;
- Demora reiniciar o monolítico quando acontece alguma falha no sistema;
- Estão limitados pela tecnologia. O monolítico foi construído usando do *framework* Node.js[17], e atualmente eles desejam utilizar algumas bibliotecas do *python* para processamento de imagens mas estão impossibilitados de fazer isso no monolítico;
- A limitação da tecnologia vale para o *frontend* também, o qual foi construído em Vue.js[18], mas eles desejam aplicar React[19] em algumas partes do sistema que eles julgam mais adequado.

4.3.0.3 Panorama pós-adoção da solução

A visão do Erlan Cassiano após passar por ambos os estilos arquiteturais dentro da RuaDois é que microsserviços são mais fáceis para validação de ideias uma vez que você é capaz de fechar os escopos de cada serviço e testar diferentes abordagens para um mesmo problema sem precisar alterar na aplicação inteira, tornando esse processo de validação e coleta de *feedbacks* muito mais rápido. Por outro lado, esse modelo arquitetural apresenta um custo alto de manutenção que não se paga no início da empresa com poucos clientes. Na arquitetura monolítica o custo de manter esse sistema é mais sustentável nos primeiros anos da *startup*, contudo tende a crescer junto com a camada de testes que é executada a cada *deploy*, uma vez que as ferramentas de integração contínua cobram por hora de execução.

Existe um ponto enfatizado por Cassiano que são os recursos humanos e como eles afetam a escolha da arquitetura. Pelas experiências dele, microsserviços é um modelo arquitetural que exige mais conhecimento e mais desenvolvedores compondo a empresa com o propósito de manter essa arquitetura funcionando bem. Enquanto que a arquitetura monolítica se dá melhor com times pequenos contudo tende a ser mais difícil quanto a inserção de novos desenvolvedores.

Por fim, ele relata que é essencial o sistema se adaptar aos objetivos de negócio e ao tempo da empresa. Assim, ele sugere validar a ideia usando de serviços separados objetivando uma rápida coleta de *feedback*, após essa solução validada, estabilizar o sistema em uma arquitetura monolítica visando ser financeiramente mais sustentável nesses primeiros anos da empresa, e por último avaliar a mudança para uma arquitetura de microsserviços ou outro modelo arquitetural. Cassiano aponta este como um caminho mais estratégico quando pensamos na evolução da *startup*.

[17] Vide <https://nodejs.org/>
[18] Vide <https://vuejs.org>
[19] Vide <https://pt-br.reactjs.org>

4.3.0.4 Análise sobre o caso de estudo

Diante do caso de estudo da RuaDois, percebe-se que, semelhante ao caso da Otto, eles afirmam que a arquitetura de microsserviços é mais fácil para coletar *feedbacks* dos clientes e testar novas funcionalidades. Este é um reflexo da modularidade natural deste estilo arquitetural, que torna fácil e rápida a concepção dos serviços de uma perspectiva local porém a RuaDois, nessa fase inicial em que experimentaram essa arquitetura, possuía apenas dois serviços *backends*, três aplicações de *frontend* (*web*, *mobile* e o *widget*) e uma série de funções *serverless*, os quais não tinham ou apresentavam baixa automatização na maioria dos processos de *deploy* e integração contínua desses serviços. Assim percebe-se no relato uma dificuldade de lhe dar com a gerência desses serviços quando olhamos para uma perspectiva mais global a respeito do caso.

O relato da RuaDois também enfatiza a importância do conhecimento sobre as tecnologias e sobre o modelo arquitetural nessa abordagem de microsserviços, sendo fatores apresentados como fundamentais para a evolução da mesma. A Figura 15 resume os pontos observados neste estudo de caso.

Quando olhamos para a arquitetura monolítica da RuaDois, percebe-se que o custo da aplicação é mais sustentável mediante o tamanho da empresa, também nota-se que a testabilidade do sistema por inteiro é mais fácil visto que não há necessidade de integrar diferentes sistemas, entretanto essa testabilidade posterga o processo de *deploy* e gera um custo extra ao rodar um conjunto de testes que não estão relacionados a funcionalidade alterada. Por fim, vemos que eles conseguem aumentar a estabilidade da aplicação na arquitetura monolítica mas em contrapartida, a complexidade de desenvolver novas funcionalidades nessa base acoplada tende a aumentar. A Figura 16 resume os pontos observados.

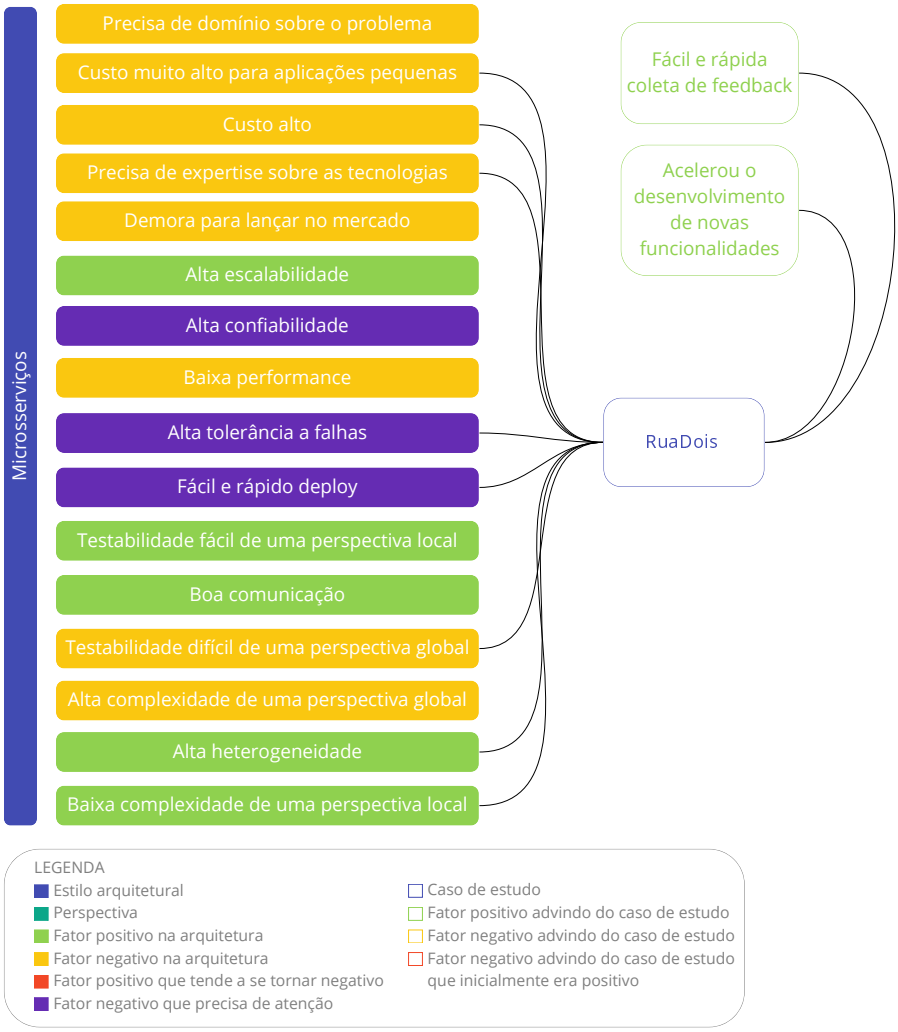

Figura 15 – Fatores apresentados no caso de estudo da arquitetura de microsserviços da RuaDois

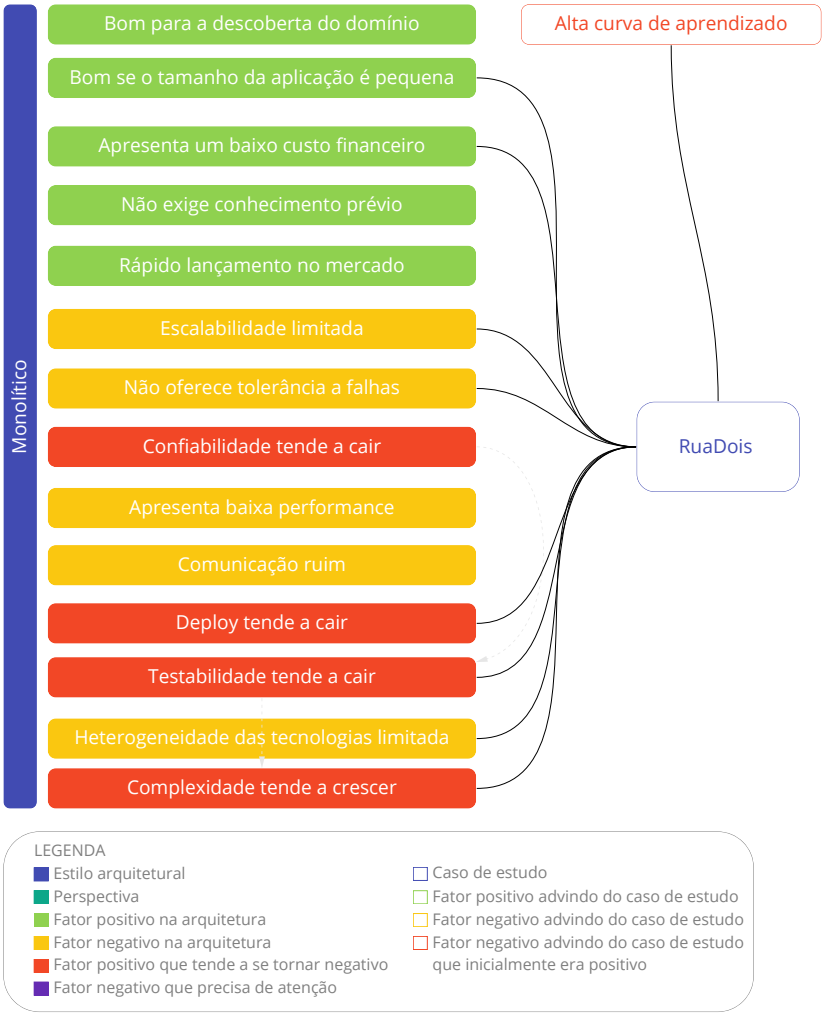

Figura 16 – Fatores apresentados no caso de estudo da arquitetura monolítica da RuaDois

5 Resultados

O presente estudo reuniu uma série de aspectos que caracterizam ambos os estilos arquiteturais com base no que é apresentado nos referenciais teóricos adotados. Dentro dessa base construída de informação, notou-se níveis diferentes dentre essas características, as quais foram classificadas em aspectos, impactos, causa e constância na seção 4.1 de acordo com as perspectivas e fatores adversos identificados para os mesmos.

Nessa abordagem, notou-se nas causas elencadas as características mais básicas de cada estilo arquitetural. Tais características são para a arquitetura monolítica a simplicidade arquitetural e a essência de ser um modelo altamente acoplado. É importante compreender que a natureza tecnicamente estruturada do modelo e a prática de reutilização de código comumente aplicada a esta arquitetura contribuem para a baixa coesão e o alto acoplamento geralmente encontrado neste estilo.

Do outro lado, os microsserviços têm como características básicas a alta complexidade e o baixo acoplamento oriundos da sua natureza distribuída e dos conceitos de DDD. Para completar, neste modelo é preferível a duplicação de código entre os serviços, o que contribui para manter a alta coesão e o baixo acoplamento.

A Figura 17 e a Figura 18 apresentam os diagramas elaborados na seção 4.1. Ao comparar os dois diagramas, vemos que a arquitetura monolítica é favorecida em um contexto inicial da aplicação, quando olhamos as perspectivas de problemática a ser resolvida e recursos necessários, essa arquitetura favorece contextos onde não se têm domínio sobre o problema, nem grandes recursos financeiros e humanos.

Já a arquitetura de microsserviços, Figura 18, quando analisada diante desses aspectos, é uma arquitetura que exige mais tempo, planejamento e recursos ao se iniciar um projeto, visto que é preciso domínio sobre o problema, mão de obra especializada e automatizar uma série de processos para que a arquitetura seja realmente viável.

Todavia, ao olhar para as perspectivas de características arquiteturais, manutenibilidade e evolucionabilidade, a arquitetura de microsserviços se destaca em relação a arquitetura monolítica, apresentando boa escalabilidade, boa comunicação entre os times, etc. Vale destacar que o modelo arquitetural de microsserviços apresenta uma dualidade que não existe na arquitetura monolítica, que é a percepção local e global dessa arquitetura. Assim, nota-se aspectos como a testabilidade e a complexidade que olhando localmente para cada serviço são fatores positivos de lhe dar, porém ao olhar os mesmos fatores de uma perspectiva global de todos os serviços estes se tornam aspectos negativos da arquitetura.

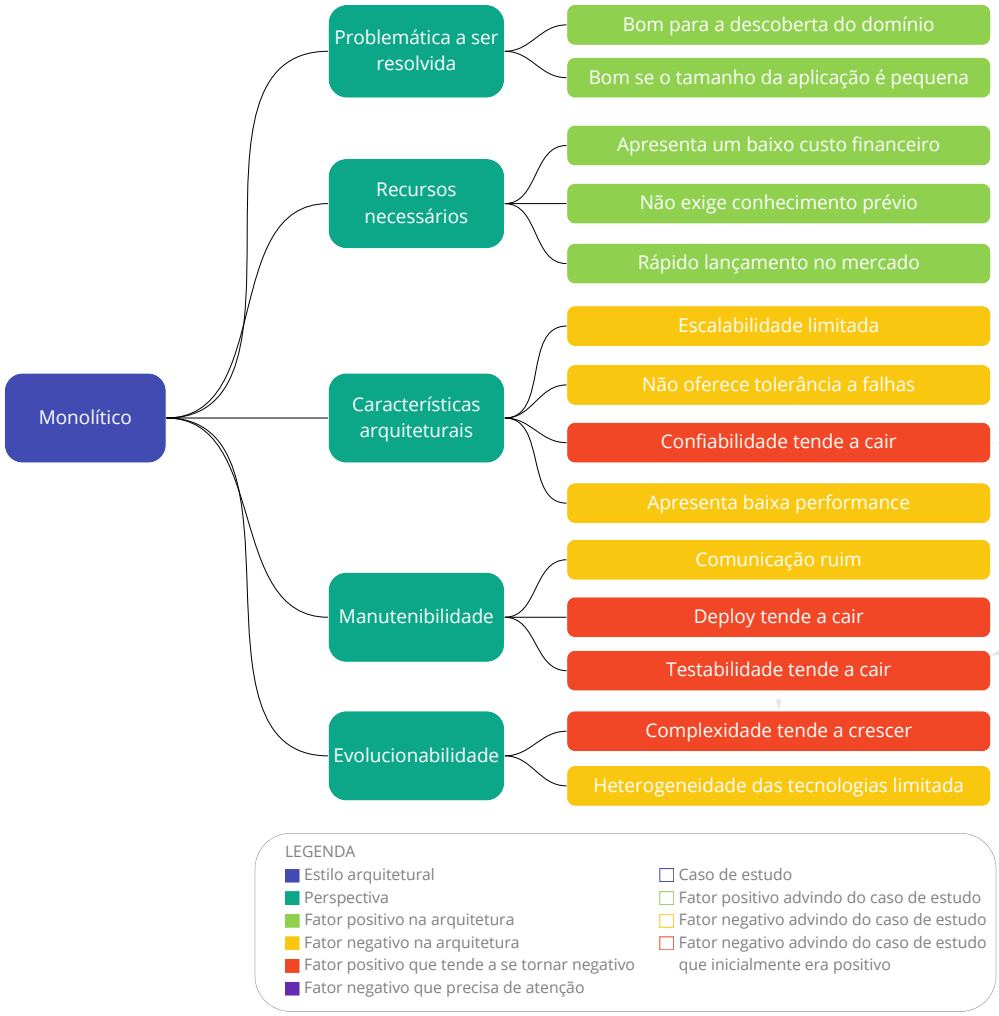

Figura 17 – Mapa mental do processo de síntese realizado sobre a arquitetura monolítica

Por último, temos os fatores que necessitam de atenção em cada uma das arquiteturas (destacados com as cores vermelha e roxo nos diagramas). A primeira diferença que se nota ao olhar estes fatores é que a arquitetura monolítica só possui fatores positivos que tendem a se tornar negativos (vermelhos), enquanto que a arquitetura de microsserviços só possui fatores positivos que precisam de atenção (roxo). Esta distinção é essencial para entender o crescimento de cada uma das arquiteturas. Quando olhamos para os monolíticos, fatores como confiabilidade, testabilidade, são inicialmente bons e providos pelo modelo arquitetural, contudo eles tendem a decair e não há nada nesse estilo arquitetural que contribua para minimizar tal aspecto, dependendo unicamente dos desenvolvedores de controlar a crescente complexidade do sistema.

Na arquitetura de microsserviços, vemos os fatores de confiabilidade, tolerância a falhas e *deploy* como os fatores que precisam de atenção. Neste caso, entende-se que existem meios de contornar essas questões que não dependem unicamente do dia-a-dia dos desenvolvedores. Por exemplo, a confiabilidade e a tolerância a falhas podem ser

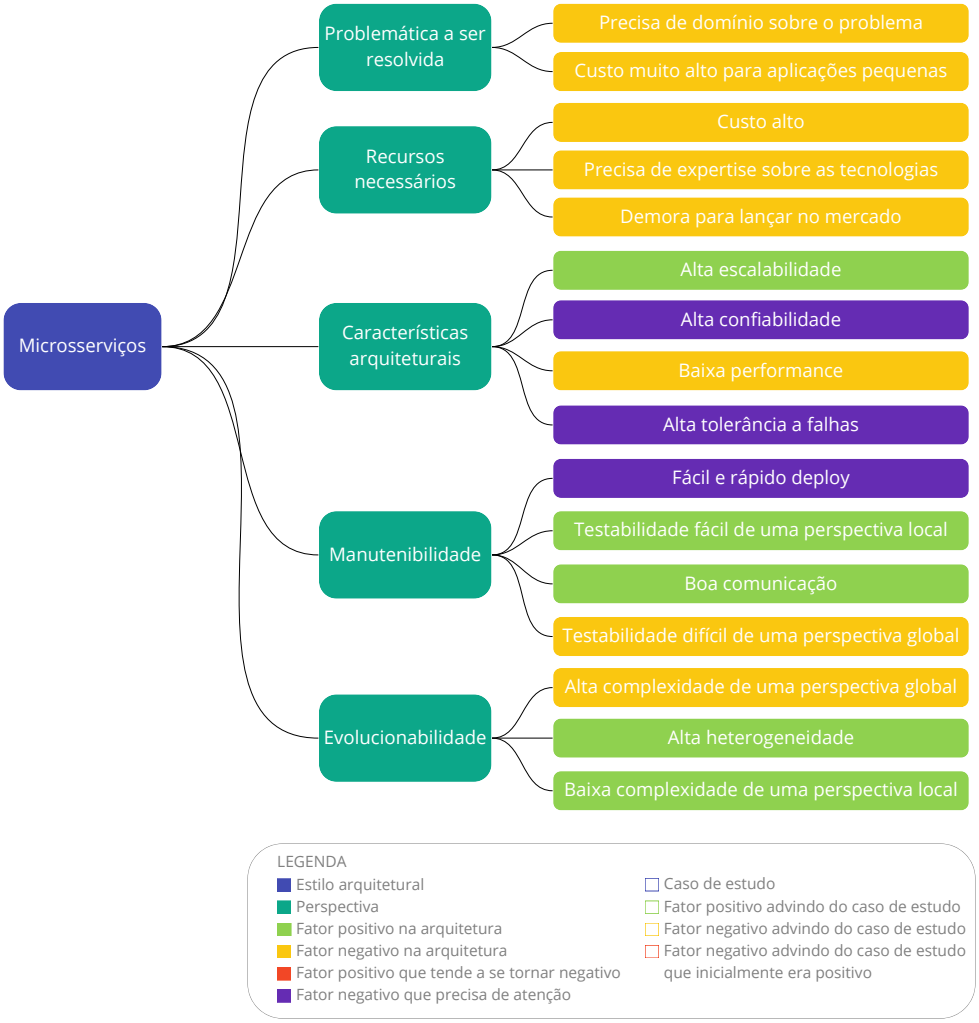

Figura 18 – Mapa mental do processo de síntese realizado sobre a arquitetura de microsserviços

garantidas mediante estratégias para lhe dar com a sobrecarga da rede, já a facilidade do *deploy* é garantida se você automatizar os processos.

Ao olharmos para os casos práticos, percebe-se experiências que atestam as características elencadas pelos referenciais teóricos, como a complexidade dos monolíticos quando a base de código se torna volumosa, relatado pela Otto e pelo KN Login, ou a alta tolerância a falha dos microsserviços apresentada nos casos da Otto e da Segment. Percebe-se também, outros aspectos comuns da arquitetura abordados pelos casos práticos como a curva de aprendizado sobre o sistema que tende a ser maior na arquitetura monolítica, contribuindo para a dificuldade de contratar novos desenvolvedores, ou a facilidade de receber *feedbacks* e testar novas funcionalidades na arquitetura de microsserviços após o período inicial de planejar e automatizar todos os processos.

Por fim, voltamos as Leis da Arquitetura de Software abordadas na subseção 2.1.1, as quais dizem que tudo na arquitetura de software é um conflito de escolha e que "por

que"é mais importante do que "como". A escolha de adotar um estilo arquitetural monolítico traz uma série de benefícios no começo da aplicação, principalmente o tempo de lançamento no mercado que é mais curto e a facilidade de alterar as funcionalidades nesse início no qual se está descobrindo o domínio da aplicação com os usuários, contudo, a medida que a base de código cresce essas características vão se perdendo. Mediante tal situação entra uma análise sobre os objetivos de negócio da empresa e os recursos disponíveis, se o planejado é manter uma aplicação pequena e simples ou se a empresa não possui muitos recursos a arquitetura monolítica é a abordagem recomenda, mas se o almejado é ter uma aplicação maior em um contexto mais complexo ou se a aplicação monolítica está crescendo além do planejado, o mais recomendado é migrar para outro modelo arquitetural. Nas palavras de Martin Fowler:

> Não tenha medo de construir um monolítico que você irá descartar, especialmente se o monolítico puder levar você ao mercado rapidamente.[1] (FOWLER, 2015c, tradução nossa)

A arquitetura de microsserviços provê um sistema mais sutentável ao longos dos anos (STEINACKER, 2016), porém ela apresenta um alto custo de construção e manutenção além de exigir que se tenha domínio sobre o contexto e sobre as tecnologias. Desta forma voltamos ao ponto, defendido na justificativa do presente trabalho, seção 1.1, de que é preciso analisar o contexto e os objetivos de negócio da empresa de forma que o modelo arquitetural seja capaz de auxliar essas empresas a alcançarem seus objetivos.

[1] Texto original: *Don't be afraid of building a monolith that you will discard, particularly if a monolith can get you to market quickly.*

6 Considerações Finais

O presente trabalho dedicou-se a realizar uma comparação entre os estilos arquiteturais monolítico e microsserviços, por meio de uma pesquisa exploratória com base principalmente nos autores Martin Fowler, Sam Newman, Mark Richards e Neal Ford, dentre outros. Traçando um paralelo com os casos de estudo do KN Login, Otto, Segment e RuaDois objetivando abordar juntamente uma visão prática sobre os modelos arquiteturais.

Notou-se que, em geral, os autores estudados apresentam uma concepção parecida a respeito do uso de cada estilo arquitetural mendiante o contexto da aplicação e que os casos de estudo corroboram com essa visão exposta pelos autores.

A perspectiva final sobre os estilos arquiteturais analisados é que a arquitetura monolítica tende a ter um menor custo e por isso é mais recomendada para fase inicial do sistema como um meio de descobrir a aplicação, porém é uma arquitetura que tende a perder a sua manutenibilidade e evolucionabilidade a medida que a base de código cresce e por esta razão alguns autores a indicam como uma arquitetura de sacrifício, de forma que após explorar a problemática da aplicação essa arquitetura dê espaço para um modelo arquitetural mais sustentável.

No caso dos microsserviços, este já tende a ser um modelo arquitetural mais sustentável, porém exige um custo maior de construção e manutenção além de domínio sobre o problema, o que faz com que nem todas as empresas estejam preparadas para adotar tal estilo arquitetural ou alcançar todos os benefícios que este modelo pode proporcionar.

Diante das limitações de tempo do presente estudo, focou-se bastante na análise dos aspectos arquiteturais abordados pelos referenciais teóricos, entretanto, existem outros pontos levantados pelos casos práticos, como a curva de aprendizado dentro de cada estilo arquitetural, ou mesmo pontos que não foram explorados nesta análise e são pouco explorados na bibliografia a respeito do tema, a exemplo, a influência do tamanho do time na escolha do estilo, questões de alocação de recursos, disponibilidade da aplicação, aspectos de segurança, processo de *debugger*, documentação, etc., pontos que abrem espaço para futuras pesquisas e aprofundamento no tema.

Por fim, deseja-se que engeheiros de software ao lerem o presente estudo obtenham maior embasamento para analisar o contexto no qual estão inseridos no momento de optar por utilizar uma arquitetura de microsserviços ou monolítica, tomando uma decisão muito mais sólida e alinhada as expectivas e objetivos de negócio de cada empresa.

Referências

ANNENKO, O. *Breaking Down a Monolithic Software: A Case for Microservices vs. Self-Contained Systems*. elastic.io, 2016. Disponível em: <https://www.elastic.io/breaking-down-monolith-microservices-and-self-contained-systems/>. Acesso em: 10.04.2021. Citado 2 vezes nas páginas 56 e 58.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Pratic*. 3. ed. Massachusetts, USA: Pearson Education, Inc., 2015. ISBN 978-0-321-81573-6. Citado 3 vezes nas páginas 24, 27 e 28.

CONWAY, M. E. How do committees invent? Datamation magazine, USA, Apr 1968. Disponível em: <http://www.melconway.com/Home/pdf/committees.pdf>. Citado na página 38.

DAVIS, J.; DANIELS, R. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*. O'Reilly Media, 2016. ISBN 9781491926437. Disponível em: <https://books.google.com.br/books?id=6e5FDAAAQBAJ>. Citado na página 30.

FOWLER, M. *Microservice Premium*. Martin Fowler, 2015. Disponível em: <https://martinfowler.com/bliki/MicroservicePremium.html>. Acesso em: 10.04.2021. Citado 2 vezes nas páginas 24 e 36.

FOWLER, M. *Microservice Trade-Offs*. Martin Fowler, 2015. Disponível em: <https://martinfowler.com/articles/microservice-trade-offs.html#ops>. Acesso em: 10.04.2021. Citado 4 vezes nas páginas 23, 32, 33 e 34.

FOWLER, M. *Monolith First*. Martin Fowler, 2015. Disponível em: <https://martinfowler.com/bliki/MonolithFirst.html>. Acesso em: 10.04.2021. Citado 3 vezes nas páginas 34, 35 e 80.

FRENCH-OWEN, C. *Centrifuge: a reliable system for delivering billions of events per day*. Otto, 2018. Disponível em: <https://segment.com/blog/introducing-centrifuge/>. Acesso em: 10.04.2021. Citado na página 67.

GUIMARÃES, L.; BRYANT, D. *QTrends Arquitetura - A ascensão e queda dos microservices*. InfoQ, 2020. Disponível em: <https://www.infoq.com/br/news/2020/06/qtrends-arquitetura-microservice/>. Acesso em: 24.04.2021. Citado na página 23.

KWIECIEŃ, A. *10 companies that implemented the microservice architecture and paved the way for others*. Divante, 2019. Disponível em: <https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/>. Acesso em: 24.04.2021. Citado na página 23.

LEWIS, J.; FOWLER, M. *Microservice Premium*. Martin Fowler, 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 21.04.2021. Citado 5 vezes nas páginas 15, 33, 36, 38 e 39.

MCGOVERN, J. et al. *A Practical Guide to Enterprise Architecture*. Prentice Hall/Professional Technical Reference, 2004. (The Coad series). ISBN 9780131412750. Disponível em: <https://books.google.com.br/books?id=YGCNnnzeov0C>. Citado na página 28.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. USA: O'Reilly Media, Inc., 2015. Citado 3 vezes nas páginas 31, 33 e 34.

NEWMAN, S. *Microservices For Greenfield?* 2019. Disponível em: <https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/>. Acesso em: 10.04.2021. Citado na página 35.

NEWMAN, S. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. [S.l.]: O'Reilly Media, 2019. ISBN 9781492047810. Citado 2 vezes nas páginas 30 e 31.

NOONAN, A. To microservices and back again. In: . London: QCon London, 2020. Citado 3 vezes nas páginas 63, 65 e 66.

RICHARDS, M.; FORD, N. *Fundamentals of Software Architecture: An Engineering Approach*. USA: O'Reilly Media, 2020. ISBN 9781492043423. Citado 12 vezes nas páginas 15, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38 e 40.

SAKOVICH, N. *Monolithic vs. Microservices: Real Business Examples*. 2017. Disponível em: <https://www.sam-solutions.com/blog/microservices-vs-monolithic-real-business-examples/>. Acesso em: 07.11.2019. Citado na página 30.

SEI, S. E. I. What is your definition of software architecture? Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, n. 3854, Jan 2017. Citado na página 27.

STEINACKER, G. *On Monoliths and Microservices*. Otto, 2015. Disponível em: <https://www.otto.de/jobs/technology/techblog/artikel/on-monoliths-and-microservices_2015-09-30.php>. Acesso em: 10.04.2021. Citado 2 vezes nas páginas 58 e 60.

STEINACKER, G. *Why Microservices?* Otto, 2016. Disponível em: <https://www.otto.de/jobs/technology/techblog/artikel/why-microservices_2016-03-20.php>. Acesso em: 10.04.2021. Citado 3 vezes nas páginas 61, 62 e 80.

TILKOV, S. *Don't start with a monolith*. Martin Fowler, 2015. Disponível em: <https://martinfowler.com/articles/dont-start-monolith.html>. Acesso em: 10.04.2021. Citado 4 vezes nas páginas 15, 24, 32 e 40.

VOGEL, O. et al. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. [S.l.]: Springer Berlin Heidelberg, 2011. ISBN 9783642197369. Citado na página 28.