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

Arquitetura Monolítica vs Microsserviços: qual a melhor solução?

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

Brasília, DF
2019

Iasmin Santos Mendes

Arquitetura Monolítica vs Microsserviços: qual a melhor solução?

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

2019

Iasmin Santos Mendes

Arquitetura Monolítica vs Microsserviços: qual a melhor solução?/ Iasmin Santos Mendes. – Brasília, DF, 2019-

54 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2019.

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: qual a melhor solução?

CDU 02:141:005.6

Iasmin Santos Mendes

Arquitetura Monolítica vs Microsserviços: qual a melhor solução?

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
2019

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.

Acreditamos no sonho e
construímos a realidade.

Roberto Marinho

Resumo

Palavras-chave: Desenvolvimento de software. Escalabilidade. Estudo de caso. *Startup*.

Abstract

Key-words: Case study. Scalability. Software development. Startup.

Lista de ilustrações

Lista de quadros

Lista de abreviaturas e siglas

API Application Programming Interface.

DDD Domain-driven Design.

FIFO First in, First out.

MVC Model-View-Controller.

REST Representational State Tranfer.

SOA Service Oriented-architecture.

TI Tecnologia da Informação.

UI User Interface.

Sumário

1 Introdução

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 mencionados.

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 uma *startup* brasileira X 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).

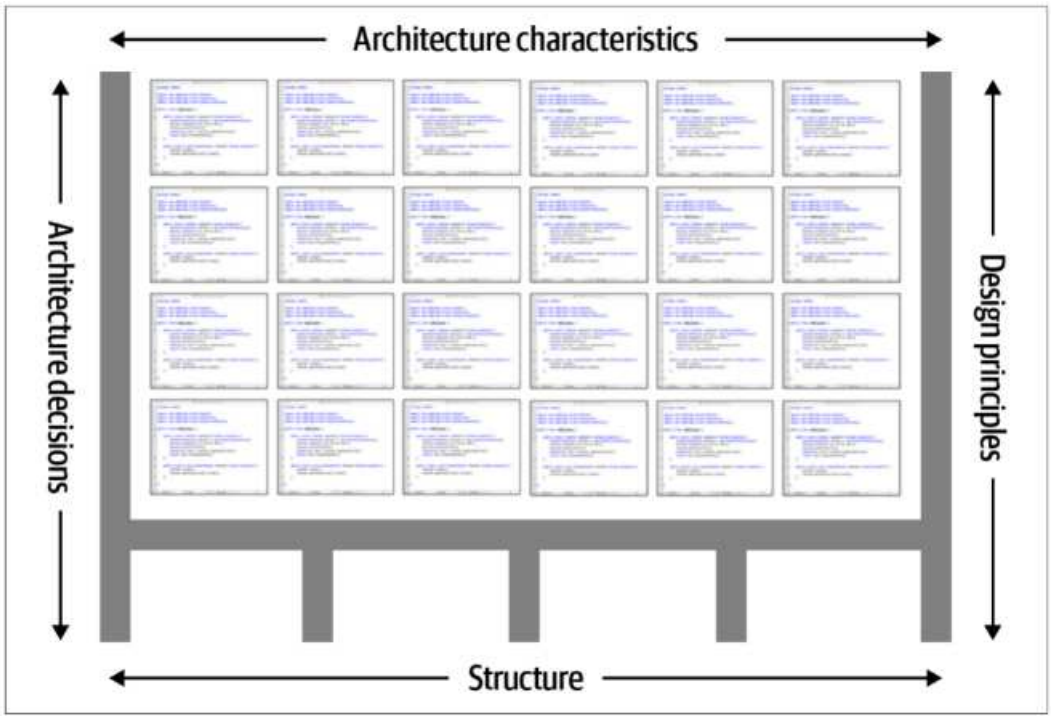

Figura 1 – Elementos que compõem a arquitetura de um software (RICHARDS; FORD, 2020)

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 técninca dentro do sistema. Na Figura 2, podemos ver essa organização na qual temos:

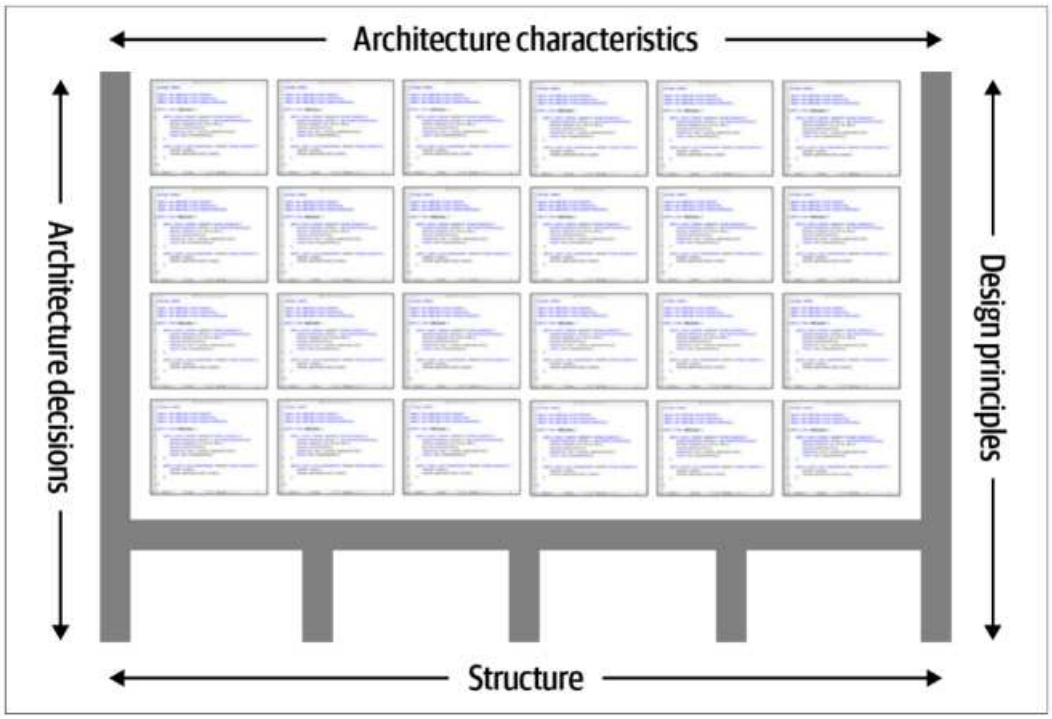

Figura 2 – Estrutura básica de um monolítico (RICHARDS; FORD, 2020)

Camada de apresentação destinada a apresentação dos dados;

[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.*

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.

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 impeçãm 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).

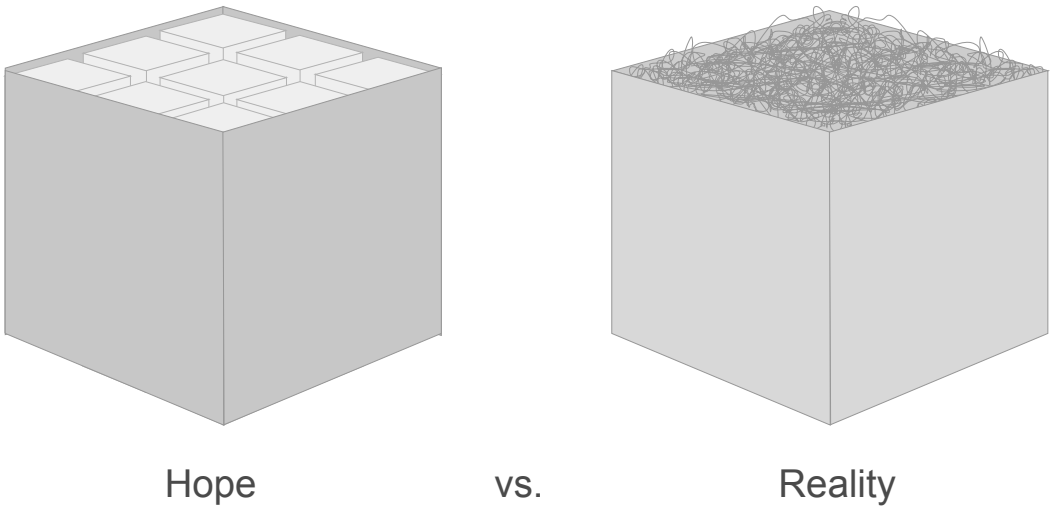

Figura 3 – Teoria vs Realidade de um monolítico (TILKOV, 2015)

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 reausabilidade 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), 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

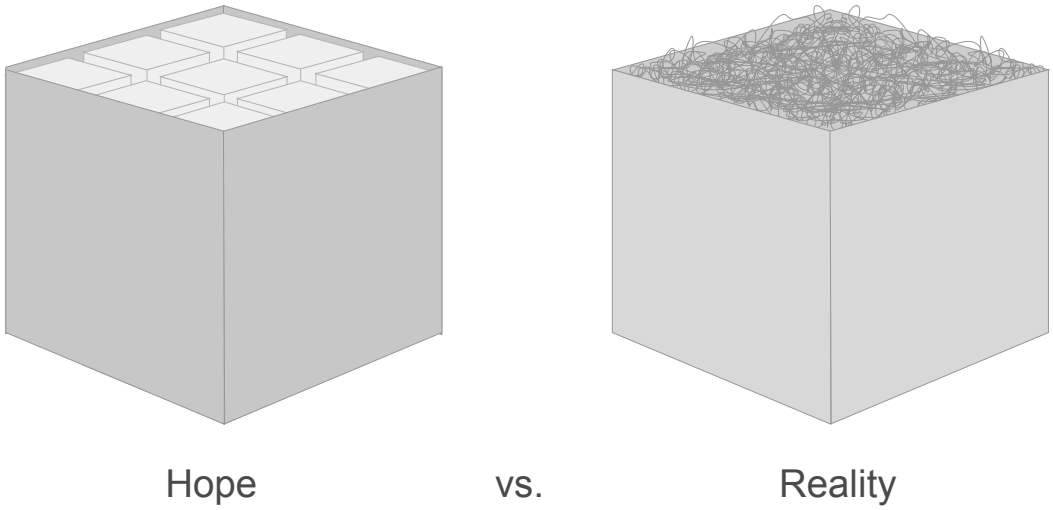

Figura 4 – PAGINA 189 (**??**)

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.

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 Tranfer (REST) (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 arquitetura mais tradicionais (FOWLER, 2015b). Outra característica que contríbui para tal é que nesse padrão a duplicação do código é, em diversas situacões, preferível a reutilização do mesmo, visto que apesar do reuso de código ser algo benéfico, ele favore 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 que discutiremos ao longo deste livro. [8] (NEWMAN, 2015, tradução nossa)

[8] 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.*

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 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.

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 distri-

buídos. 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.[9] (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 *load balancer*[10] 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).

2.4.3.2 Volume de dados

2.4.3.3 Disponibilidade

2.4.3.4 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

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

[10] DESCREVER

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 bem avaliado.

2.4.3.5 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).

2.4.3.6 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, torando 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 *Debugger*

2.4.4.4 Curva de aprendizado

2.4.4.5 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.[11] (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 das empresas. Eles apontam que essa estrutura deixa a comunicação dentro do projeto

[11] 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.*

mais lenta, de forma que mudanças simples no sistema podem demorar bastante devido a dependência de uma equipe externa.

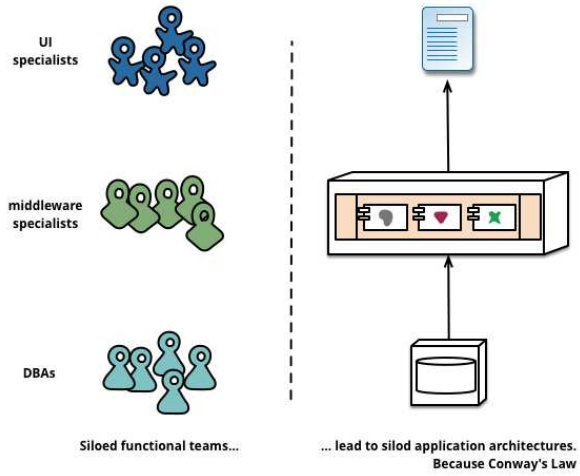

Figura 5 – Lei de Conway aplicada a uma arquitetura em camadas (LEWIS; FOWLER, 2014)

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).

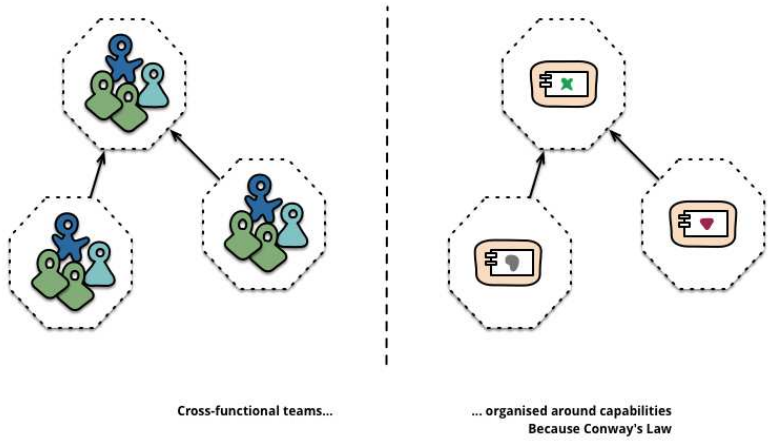

Figura 6 – Lei de Conway aplicada a uma arquitetura de microsserviços (LEWIS; FOWLER, 2014)

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

3 Metodologia

3.1 Levantamento bibliográfico

Para referenciamento deste trabalho foram realizados estudos acerca de *startups*, desenvolvimento de software e aspectos relacionados a escalabilidade de um sistema por meio de livros, Google Scholar, dentre outras fontes. Os tópicos abordados dentro de cada tema foram selecionados com base na sua respectiva importância dentro do assunto e no seu envolvimento com o contexto aprensentado. Individualmente, os temas citados são bem explorados e abordados pela comunidade acadêmica, contudo, quando relacionados os artigos e estudos a respeito tendem a ser mais escassos.

Publicações e palestras de empresas de referencia voltadas ao desenvolvimento de software no contexto de *startups* também foram utilizadas com o intuito de contribuir com perspectivas mais atuais a respeito do tema.

3.2 Metodologia de pesquisa

A proposta deste trabalho é correlacionar técnicas de desenvolvimento de software escalável com as fases de vida de uma *startup*. Para tal, foi aplicado o METODO X, visando estabelecer essa correlação. Esta seção visa descrever os objetos de análise, o método aplicado e as etapas envolvidas no desenvolvimento de toda a pesquisa.

3.2.1 Objetos de análise

Os objetos de análise do presente estudo consistem em:

Ciclo de vida de uma *startup* Fases pelas quais uma *startup* passa desde o seu nascimento até a sua estabilização no mercado.

Técnicas de escalabilidade Técnicas e boas práticas aplicadas no mercado em busca da construção de sistemas escaláveis e/ou com a capacidade para tal.

3.2.2 Método de correlação

3.2.3 Etapas do desenvolvimento

A etapas adotadas na construção da seguinte pesquisa foram:

Etapa 1 Definição do método de análise a ser aplicado;

Etapa 2 Coleta de dados a respeito dos objetos de análise;

Etapa 3 Aplicação do método de análise sobre os dados coletados;

Etapa 4 Análise dos resultados obtidos;

Etapa 5 Classificação das técnicas de escalabilidade;

Etapa 6 Validação ??.

A seguir segue a descrição de cada etapa.

3.2.3.1 Etapa 1: Definição do método de análise a ser aplicado

Nesta etapa, foi definido o método de análise utilizado, o qual tem como objetivo de estabelecer a correlação entre entre as técnicas de escalabilidade elencadas e as fases do cicla de vida de uma *startup*.

3.2.3.2 Etapa 2: Coleta de dados a respeito dos objetos de análise

Mediante o método de análise escolhido, foram coletados por meio de pesquisa bibliográfica as informações necessárias acerca dos objetos de análise definidos na subseção 3.2.1. Dessa forma obteve-se as características sobre os objetos de análise pertinentes a aplicação do método.

3.2.3.3 Etapa 3: Aplicação do método de análise sobre os dados coletados

Com os dados em mãos, nessa etapa aplicou-se o méotodo de análise proposto visando obter a correlação entre os objetos analisados.

3.2.3.4 Etapa 4: Análise dos resultados obtidos

A partir dos resultados obtidos na etapa anterior, realizou-se uma análise afim de compreender os mesmos e analisar a coerência com o que era esperado.

3.2.3.5 Etapa 5: Classificação das técnicas de escalabilidade

Por fim, classificou-se as técnicas de escalabilidade elencadas em relação as fases da *startup* usando como base os resultados obtidos nas etapas anteriores. Obtendo assim, as recomendações almejadas no objetivo geral deste trabalho, o qual está disposto na subseção 1.2.1.

3.3 Planejamento das atividades

Esta seção destina-se a descrever o planejamento realizado visando atingir as etapas definidas na subseção 3.2.3 e com base na metodologia de pesquisa descrita na seção 3.2.

Mediante o escopo da pesquisa a ser realizada, planejou-se a execução para um total de 7 semanas, iniciando na metade do mês de Março/2020 e indo até a primeira semana de Maio/2020. No Quadro 1 estão dispostas as atividades que pretende-se cumprir ao longo deste período.

Quadro 1 – Cronograma das atividades

Semana	**Atividades**
1	Definição do método de análise a ser aplicado
2	Coleta de dados a respeito dos objetos de análise
3	Aplicação do método de análise sobre os dados coletados
4	Análise dos resultados obtidos e classificação das técnicas de escalabilidade
6	Revisão e escrita do documento
7	Apresentação

3.4 Ferramentas

As ferramentas que pretende-se usar para a execução deste trabalho são:

Draw.io editor gráfico online que permite a construção de processos e desenhos relevantes ao conteúdo apresentado;

Google Planilhas ferramenta para construção de tabelas e gráficos. A qual pretende-se usar para realizar análises sobre os dados coletados.

Trello quadro interativo que auxilia na organização de projetos. O mesmo foi utilizado para gerenciar as etapas e tarefas alocadas para cada semana de execução do projeto, permitindo acompanhar o andamento do mesmo.

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 SECAO DA METODOLOGIA AQUI, os aspectos levantados sobre cada arquitetura e análise comparativa realizada.

As seções estão dispostas em:

1. **Referencias bibliográficas ?** ainda não sei como abordar isso;
2. **Relatos de alguns casos reais:** 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.
3. **Análise comparativa**

4.1 Relatos de casos reais

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.

Para cada caso será 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. **Perspectiva pós-adoção da solução:** compreensão sobre como a solução escolhida impactou o desenvolvimento de software dentro da empresa.

4.1.1 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.

4.1.1.1 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 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;

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

- 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.

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

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.

4.1.1.3 Perspectiva 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.

4.1.2 Otto.de

Nesta seção será apresentado o caso relatado por Steinacker (2015), a respeito da construção do novo *e-commerce* Otto.de[6].O Grupo Otto[7] é uma empresa de origem alemã que trabalha em diversos países desenvolvendo soluções de negócios voltadas a necessidade do setor de *e-commerce*.

4.1.2.1 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.

[6] 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>

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

4.1.2.2 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 4.1.1 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.

4.1.2.3 Perspectiva 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, os microsserviços de 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).

4.1.3 Segment

O caso apresentado a seguir foi retratado por Noonan (2020) na conferência QCon London. Ela trata a respeito do sistema da Segment[8], o qual foi construído inicialmente em uma arquitetura monolítica, passou por uma migração para uma arquitetura de microsserviços 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.

4.1.3.1 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)[9]. 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

[8] Vide: <https://segment.com>
[9] Vide <https://pt.wikipedia.org/wiki/FIFO>

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:

- 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;

4.1.3.2 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[10], 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.

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

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)[11].

4.1.3.3 Perspectiva 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;

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.

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

5 Considerações Finais

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 42 e 43.

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 21, 23 e 24.

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 34.

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 26.

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 21 e 31.

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 3 vezes nas páginas 28, 29 e 30.

FOWLER, M. *Monolith First*. Martin Fowler, 2015. Disponível em: <https://martinfowler.com/bliki/MonolithFirst.html>. Acesso em: 10.04.2021. Citado na página 30.

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 50.

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 13, 29, 32, 34 e 35.

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 24.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. USA: O'Reilly Media, Inc., 2015. Citado 2 vezes nas páginas 27 e 29.

NEWMAN, S. *Microservices For Greenfield?* 2019. Disponível em: <https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/>. Acesso em: 10.04.2021. Citado 2 vezes nas páginas 30 e 31.

NEWMAN, S. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. [S.l.]: O'Reilly Media, 2019. ISBN 9781492047810. Citado 2 vezes nas páginas 26 e 27.

NOONAN, A. To microservices and back again. In: . London: QCon London, 2020. Citado 2 vezes nas páginas 47 e 49.

RICHARDS, M.; FORD, N. *Fundamentals of Software Architecture: An Engineering Approach*. USA: O'Reilly Media, 2020. ISBN 9781492043423. Citado 13 vezes nas páginas 13, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34 e 36.

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 26.

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 23.

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 na página 44.

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 2 vezes nas páginas 45 e 46.

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 3 vezes nas páginas 13, 21 e 28.

VOGEL, O. et al. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. [S.l.]: Springer Berlin Heidelberg, 2011. ISBN 9783642197369. Citado na página 24.