UNIVERSIDADE FEDERAL DE ALAGOAS – UFAL
CAMPUS DE ARAPIRACA
CIÊNCIA DA COMPUTAÇÃO - BACHARELADO

ALAN DUDA DOS SANTOS

ANÁLISE COMPARATIVA ENTRE ARQUITETURA TRADICIONAL E ARQUITETURA DE MICROSSERVIÇOS

ARAPIRACA
2023

Alan Duda dos Santos

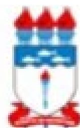
Análise comparativa entre arquitetura tradicional e arquitetura de microsserviços

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal de Alagoas - UFAL, Campus de Arapiraca.

Orientador: Prof. Dr. Patrick Henrique da Silva Brito

Arapiraca

2023



Universidade Federal de Alagoas – UFAL
Campus Arapiraca
Biblioteca Setorial *Campus* Arapiraca - BSCA

S237a Santos, Alan Duda dos
Análise comparativa entre arquitetura tradicional e arquitetura de microsserviços [recurso eletrônico] / Alan Duda dos Santos. – Arapiraca, 2023.
42 f.: il.

Orientador: Prof. Dr. Patrick Henrique da Silva Brito.
Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal de Alagoas, *Campus* Arapiraca, Arapiraca, 2023.
Disponível em: Universidade Digital (UD) / RD- BSCA– UFAL (*Campus* Arapiraca).
Referências: f. 42.

1. Arquitetura de software. 2. Arquitetura tradicional. 3. Arquitetura de microsserviços. 4. Arquiteturas - Comparação. 5. Características arquiteturais. I. Brito, Patrick Henrique da Silva. II. Título.

CDU 004

Bibliotecário responsável: Nestor Antonio Alves Junior
CRB-4 / 1557

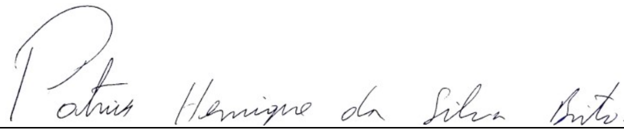
Alan Duda dos Santos

Análise comparativa entre arquitetura tradicional e arquitetura de microsserviços

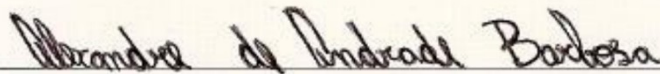
Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal de Alagoas - UFAL, Campus de Arapiraca.

Data de Aprovação: 27/07/2023.

Banca Examinadora



Prof. Dr. Patrick Henrique da Silva Brito
Universidade Federal de Alagoas – UFAL
Campus de Arapiraca
(Orientador)



Prof. Dr. Alexandre de Andrade Barbosa
Universidade Federal de Alagoas – UFAL
Campus de Arapiraca
(Examinador)

Prof. Dr. Ricardo Alexandre Afonso
Universidade Federal de Alagoas – UFAL
Campus de Arapiraca
(Examinador)

AGRADECIMENTOS

Em primeiro lugar, quero expressar minha gratidão à minha família, que esteve ao meu lado ao longo de toda a jornada acadêmica. Quero destacar especialmente o papel crucial de minha mãe, Sandra Maria do Santos, que não apenas trabalhou incansavelmente, mas também me incentivou, apoiou e compreendeu os desafios que enfrentei. Sua dedicação e amor inabaláveis foram a âncora que me manteve focado e determinado a alcançar meus objetivos.

Também sou imensamente grato aos meu orientador Patrick Henrique da Silva Brito por seu apoio incansável, orientação valiosa e paciência ao longo deste processo. Seus insights e feedbacks foram fundamentais para o sucesso deste projeto.

Ao Ytalo Ramon, a Crislâine Costa e o Ivillys Gomes por terem sido os meus principais parceiros nos incontáveis trabalhos acadêmicos ocorridos durante essa caminhada. Aos meus amigos e colegas de classe Hiran Júnior e José Adilson, que compartilharam suas ideias, conhecimentos e experiências, meu sincero agradecimento. Suas discussões e debates enriqueceram o meu conhecimento.

E não posso deixar de mencionar meus amigos José Carlos e Silvio Pinto, cujas presenças tornaram as minhas viagens à universidade verdadeiramente memoráveis. Suas risadas e histórias transformaram as longas jornadas de ida e volta em momentos satisfatórios e menos cansativos.

Por fim, mas não menos importante, agradeço também aos professores Elthon Allex e Thiago Sales, que desempenharam um papel crucial em minha formação acadêmica, não só como inspiração, mas como guia e fonte inestimável de conhecimento.

RESUMO

Neste trabalho de conclusão de curso é abordado a comparação entre a arquitetura tradicional e a arquitetura de microsserviços, tendo em vista a relevância atual destes padrões arquiteturais no mercado de desenvolvimento de software. O objetivo deste trabalho é demonstrar as principais características de ambas abordagens, suas vantagens e desvantagens com base nas principais características e subcaracterísticas definidas pela norma ISO/IEC 9126, além de mostrar cenários onde cada uma é mais adequada. O estudo foi baseado em uma revisão bibliográfica de livros, artigos científicos e análise de conteúdo de empresas líderes no mercado de arquitetura de software. O estudo concluiu que não existe uma abordagem única que seja adequada para todos os casos, e a escolha da arquitetura deve ser baseada nos requisitos específicos do sistema, tendo em vista que se verificou que a arquitetura tradicional é adequada para a fase inicial de descoberta do domínio, porém, à medida que o código cresce, sua manutenibilidade e evolucionabilidade são comprometidas, ao contrário da arquitetura de microsserviços que é mais sustentável a longo prazo, embora apresente um custo mais elevados e demande conhecimento aprofundado tanto da problemática quanto das tecnologias envolvidas. É essencial considerar esses aspectos ao decidir pela arquitetura mais adequada para um projeto, levando em conta o tamanho, a complexidade e as necessidades específicas do sistema em questão.

Palavras-chave: arquitetura de software; arquitetura tradicional; arquitetura de microsserviços; comparação entre arquiteturas; características arquiteturais.

ABSTRACT

This undergraduate thesis addresses the comparison between traditional architecture and microservices architecture, considering the current relevance of these architectural patterns in the software development market. The objective of this study is to demonstrate the main characteristics of both approaches, their advantages and disadvantages based on the key characteristics and sub-characteristics defined by the ISO/IEC 9126 standard, as well as to highlight scenarios where each approach is more suitable. The study was based on a literature review of books, scientific articles, and content analysis from leading companies in the software architecture market. The study concludes that there is no single approach that is suitable for all cases, and the choice of architecture should be based on the specific requirements of the system. It was found that the traditional architecture is suitable for the initial phase of domain discovery; however, as the code grows, its maintainability and evolvability are compromised. In contrast, microservices architecture is more sustainable in the long run, albeit at a higher cost and requiring in-depth knowledge of both the problem domain and the technologies involved. It is essential to consider these aspects when deciding on the most appropriate architecture for a project, taking into account the size, complexity, and specific needs of the system at hand.

Keywords: software architecture; traditional architecture; microservices architecture. comparison between architectures; architectural feature.

LISTA DE FIGURAS

Figura 1 - Arquitetura de software como uma ponte..13
Figura 2 - Arquitetura Cliente-Servidor com servidor único (a) ou replicado (b)....................15
Figura 3 - Estrutura da arquitetura tradicional..16
Figura 4 - Microsserviços podem dispensar componentes quando possuem pouco código.....17

SUMÁRIO

1 **INTRODUÇÃO**....9
1.1 APRESENTAÇÃO DO PROBLEMA....9
1.2 JUSTIFICATIVA DA ESCOLHA DO TEMA....10
1.3 OBJETIVO GERAL....11
1.4 OBJETIVOS ESPECÍFICOS....11
1.5 MÉTODO UTILIZADO E ESTRUTURA DO DOCUMENTO....11
2 **FUNDAMENTAÇÃO TEÓRICA**....13
2.1 ARQUITETURA DE SOFTWARE....13
2.1.1 Arquitetura Tradicional....15
2.1.2 Arquitetura de Microsserviços....17
2.2 ESCALABILIDADE VERTICAL E HORIZONTAL....18
2.3 ISO/IEC 9126....20
3 **METODOLOGIA**....22
4 **ANÁLISE COMPARATIVA**....23
4.1 MANUTENIBILIDADE....23
4.1.1 Atualizações de software....23
4.1.2 Adição de funcionalidades....24
4.2 MODULARIDADE....25
4.2.1 Divisão em componentes....25
4.2.2 Testabilidade....27
4.3 TOLERÂNCIA A FALHAS....28
4.3.1 Recuperação de erros....28
4.3.2 Isolamento de falhas....29
4.3.3 Disponibilidade....30
4.4 ESCALABILIDADE....32
4.4.1 Adição de recursos....32
4.5 GERENCIAMENTO DE DADOS....33
4.5.1 Armazenamento de dados....33
4.5.2 Governança de dados....35
5 **ESTUDOS DE CASOS**....37
5.1 NETFLIX: MIGRAÇÃO DE MONOLITO PARA MICROSSERVIÇOS....37
5.2 AMAZON PRIME VIDEO: MIGRAÇÃO PARCIAL DE MICROSSERVIÇOS PARA MONOLITO....38

6 CONCLUSÕES..40
REFERÊNCIAS..42

1 INTRODUÇÃO

1.1 APRESENTAÇÃO DO PROBLEMA

Nos últimos anos, temos testemunhado um cenário de aumento significativo na complexidade e na demanda por qualidade dos sistemas de software. À medida que a tecnologia avança e as expectativas e demandas dos usuários e clientes se tornam mais sofisticadas, os softwares precisam acompanhar esse ritmo e oferecer soluções cada vez mais abrangentes. No entanto, esse cenário apresenta desafios significativos para os desenvolvedores.

Um bom planejamento inicial é fundamental para lidar com a complexidade crescente dos softwares atuais. Antes mesmo de começar a escrever o primeiro código, é essencial compreender claramente os requisitos do projeto, identificar os objetivos e estabelecer uma estratégia de desenvolvimento sólida. Um planejamento adequado permite que os desenvolvedores possam antecipar e mitigar potenciais problemas antes que eles se tornem obstáculos significativos. Isso inclui a identificação de requisitos conflitantes ou ambíguos, a consideração de restrições de desempenho e escalabilidade, bem como a antecipação de possíveis atualizações e expansões futuras do software. Além disso, um bom planejamento auxilia na definição de metas realistas e na criação de um cronograma viável para o projeto.

Um dos artefatos de software mais estratégicos em relação aos atributos de qualidade do software é a arquitetura de software (BASS; CLEMENTS; KAZMAN, 2012) (GARLAN, 2008). Isso se dá pelo fato da arquitetura representar, estrategicamente, a forma como as partes do software estão dispostas e interagem entre si. Além do planejamento dos requisitos de qualidade em si, a arquitetura de software desempenha um papel crucial na garantia de que as empresas alcancem seus objetivos (BASS; CLEMENTS; KAZMAN, 2012). Uma arquitetura adequada define a estrutura e a organização do software, permitindo que ele evolua de maneira sustentável e seja fácil de entender, manter e modificar. Por outro lado, uma arquitetura mal projetada ou negligenciada pode resultar em problemas como código duplicado, baixa flexibilidade, dificuldades na implantação e obstáculos na implementação de novos recursos. Nesse contexto, a escolha de uma abordagem arquitetural adequada tornou-se um fator crítico para o desenvolvimento bem-sucedido de software (GARLAN, 2008).

Além da tendência que pode ser observada em relação à valorização do projeto arquitetural do software, o cenário de desenvolvimento de software atual apresenta um grande crescimento na utilização da computação em nuvem, isto é, softwares disponibilizados em

servidores acessados via Internet. Em um cenário que envolve altas demandas de acesso, a preocupação com a estruturação apropriada dos componentes do software, realizada pela arquitetura de software, torna-se uma preocupação ainda mais estratégica. Entre os diversos estilos arquiteturais disponíveis, dois têm recebido considerável atenção na indústria e na academia: a arquitetura tradicional, derivada diretamente do estilo “cliente-servidor” e a arquitetura de microsserviços.

A arquitetura tradicional é uma abordagem amplamente utilizada há décadas no mercado de software, motivada principalmente pela sua simplicidade e facilidade de implementação, na qual todas as funcionalidades do sistema são desenvolvidas e implantadas em um módulo servidor e a comunicação é realizada por conectores arquiteturais, normalmente capazes de distribuir carga entre múltiplas instâncias de servidores. No entanto, à medida que os sistemas crescem em tamanho e complexidade, a manutenção e a evolução tornam-se desafiadoras nessa abordagem, dada a potencial complexidade do servidor.

Por outro lado, a arquitetura de microsserviços tem ganhado popularidade nos últimos anos como uma alternativa à arquitetura tradicional, principalmente devido à disseminação promovida pela Amazon e Netflix após a adoção dessa abordagem (BOGNER *et al.*, 2021). Nesse modelo, o módulo servidor é dividido em serviços independentes, cada um responsável por uma funcionalidade específica. Esses serviços são implantados e configurados separadamente, permitindo maior flexibilidade e modularidade.

1.2 JUSTIFICATIVA DA ESCOLHA DO TEMA

A escolha deste tema para o presente estudo justifica-se pela sua relevância e atualidade. A arquitetura de software desempenha um papel crucial no desenvolvimento de sistemas de computação modernos. Muitas organizações estão considerando a transição da arquitetura tradicional para a arquitetura de microsserviços como uma estratégia para alcançar maior flexibilidade, escalabilidade e agilidade no desenvolvimento de software. No entanto, essa transição requer uma compreensão aprofundada das implicações e dos desafios envolvidos.

Compreender as diferenças, os benefícios e as limitações desses dois estilos arquiteturais são de extrema importância para os profissionais de software e pesquisadores da área. Ao analisar criticamente essas duas abordagens, podemos identificar as melhores práticas e orientar a tomada de decisões na escolha de uma arquitetura adequada a um determinado contexto.

Tal importância fica ainda mais evidente ao se observar grandes empresas da área, tais como Netflix e Amazon, que estão entre aquelas que têm se destacado no segmento da computação em nuvem, possuindo atuação global e altas demandas de acesso. Porém, ao analisar a evolução desses sistemas, percebe-se que enquanto a Netflix relata ganhos ao migrar de uma arquitetura tradicional para uma arquitetura de microsserviços, a Amazon relata ganhos ao percorrer o caminho inverso, migrando de uma arquitetura de microsserviços para uma arquitetura tradicional.

1.3 OBJETIVO GERAL

O objetivo principal deste estudo é realizar uma análise comparativa entre a arquitetura tradicional e a arquitetura de microsserviços, com base nas principais características e subcaracterísticas definidas pela ISO/IEC 9126. O estudo visa fornecer embasamento teórico para profissionais da área de desenvolvimento de software, ao identificar os desafios, benefícios e a adequação de cada abordagem em diferentes cenários. O objetivo é oferecer uma visão simplificada para auxiliar na tomada de decisões arquiteturais.

1.4 OBJETIVOS ESPECÍFICOS

Para atingir o objetivo geral do presente trabalho, serão investigadas as seguintes questões:

1. Quais são as principais características da arquitetura tradicional e da arquitetura de microsserviços?
2. Quais são as vantagens e desvantagens de cada abordagem?
3. Em quais cenários cada arquitetura é mais adequada?

1.5 MÉTODO UTILIZADO E ESTRUTURA DO DOCUMENTO

O estudo foi baseado em uma revisão bibliográfica da literatura, incluindo a análise de conteúdos disponibilizados pelas empresas Netflix e Amazon, relacionadas ao seu processo de migração de arquitetura. De maneira geral, o estudo concluiu que não existe uma abordagem única que seja adequada para todos os casos, e a escolha da arquitetura deve ser baseada nos requisitos específicos de cada sistema, tendo em vista que se verificou que a arquitetura tradicional apresenta vantagens relacionadas ao menor custo de comunicação, enquanto a

manutenibilidade do software é comprometida, ao contrário da arquitetura de microsserviços que é mais sustentável a longo prazo, embora apresente um custo de comunicação mais elevado e demande conhecimento aprofundado tanto da problemática quanto das tecnologias envolvidas. É essencial considerar esses aspectos ao decidir pela arquitetura mais adequada para um projeto, levando em conta o tamanho, a complexidade e as necessidades específicas de cada sistema.

O restante do documento está estruturado como segue. O Capítulo 2 apresenta a fundamentação teórica que aborda a temática principal do trabalho: arquitetura de software, os estilos arquiteturais tradicional e de microsserviços e alguns requisitos de qualidade influenciados pela adoção de diferentes estilos arquiteturais. O Capítulo 3 apresenta a metodologia de trabalho utilizada na condução da análise comparativa. O Capítulo 4 apresenta o resultado da análise comparativa entre os estilos arquiteturais tradicional e de microsserviços. O Capítulo 5 apresenta detalhes dos cenários relatados pelas empresas Netflix e Amazon, em vista de entender melhor as particularidades de cada cenário de uso, que motivaram a adoção de estilos arquiteturais distintos. Finalmente, o Capítulo 6 apresenta algumas considerações finais e direcionamentos para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

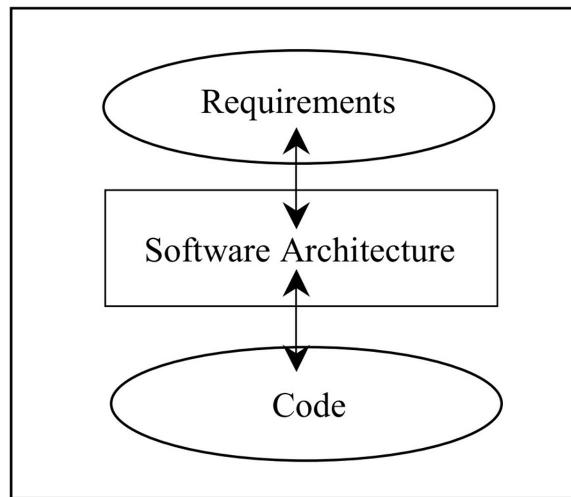
2.1 ARQUITETURA DE SOFTWARE

A arquitetura de software desempenha um papel crucial no desenvolvimento de sistemas de software, pois é responsável por definir a estrutura global do sistema, garantindo que ele seja escalável, flexível, sustentável e capaz de atender às necessidades dos usuários. Ela abrange todos os aspectos da aplicação, desde a organização dos componentes até a definição das interações entre eles, com o objetivo de alcançar uma solução coesa e de alta qualidade.

De acordo com Bass, Clements e Kazman (2015), a arquitetura de software de um sistema é o conjunto de estruturas necessárias para raciocinar sobre o sistema, que compreende elementos de software, relações entre eles e propriedades de ambos. Essa definição destaca a importância da arquitetura como uma forma de representar e compreender a estrutura essencial do sistema, permitindo que os desenvolvedores e as partes interessadas tenham uma visão clara das partes constituintes e de como elas se relacionam.

Ao projetar a arquitetura de um sistema de software, é necessário considerar uma série de fatores, como os requisitos do sistema, as restrições técnicas e orçamentárias, a interoperabilidade com outros sistemas, as necessidades de desempenho e segurança, e as características específicas do domínio de aplicação. Essas considerações são fundamentais para guiar as decisões arquiteturais e garantir que o sistema atenda às expectativas dos usuários e das partes interessadas. De acordo com Garlan (2008), a arquitetura de software geralmente desempenha um papel fundamental como uma ponte entre os requisitos e o código, conforme ilustrado na Figura 1.

Figura 1 - Arquitetura de software como uma ponte



Fonte: Garlan (2008).

Uma das principais metas da arquitetura de software é promover a modularidade e a reutilização de componentes. Isso é alcançado por meio da decomposição do sistema em módulos independentes, cada um com uma responsabilidade claramente definida. Esses módulos podem ser desenvolvidos separadamente e combinados de forma a formar o sistema completo. A modularidade não apenas facilita o desenvolvimento, mas também permite a manutenção e a evolução do sistema de forma mais eficiente, uma vez que alterações em um módulo não afetarão necessariamente os outros.

Outro aspecto importante é a definição das APIs (*Application Programming Interface*). As APIs estabelecem como os componentes e módulos do software devem interagir e se comunicar uns com os outros, além atuar como uma camada de abstração entre diferentes sistemas (BASS; CLEMENTS; KAZMAN, 2012), fornecendo uma interface consistente e padronizada para que os desenvolvedores possam acessar os recursos e funcionalidades de um determinado software ou serviço, sem precisar conhecer os detalhes internos de sua implementação.

A escolha do estilo arquitetural também desempenha um papel significativo na definição da arquitetura de software. Existem vários estilos arquiteturais amplamente utilizados, como a arquitetura em camadas, a arquitetura orientada a serviços, a arquitetura baseada em eventos e a arquitetura orientada a microsserviços. Cada estilo possui suas próprias características e benefícios, e a seleção do estilo adequado depende das necessidades e requisitos específicos do sistema em questão.

Além disso, a mesma também lida com aspectos de gerenciamento de dados, como o projeto do banco de dados e a definição das estratégias de persistência. Ela define como os

dados serão armazenados, acessados e manipulados pelo sistema, levando em consideração aspectos como desempenho, integridade, segurança e privacidade (GARLAN, 2008).

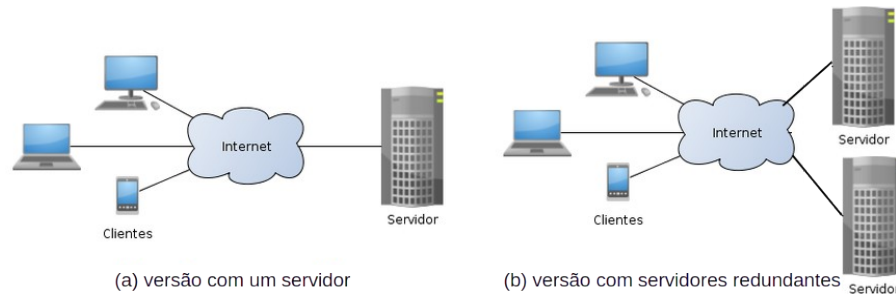
Vale ressaltar que essa disciplina não é um artefato estático, mas sim um elemento vivo e iterativo do processo de desenvolvimento. À medida que o sistema evolui, novos requisitos podem surgir, e a arquitetura pode precisar ser adaptada e aprimorada para atender a essas mudanças. Portanto, a manutenção contínua da arquitetura é fundamental para garantir que o sistema permaneça robusto, atualizado e capaz de se adaptar às demandas do ambiente em que está inserido.

Em resumo, a arquitetura de software é uma disciplina abrangente e complexa que envolve a definição da estrutura global, dos componentes, das interações e das decisões fundamentais de um sistema de software. Ela visa criar sistemas bem organizados, sustentáveis e de alta qualidade, capazes de atender às necessidades dos usuários e das organizações ao longo do tempo. Através da aplicação de princípios arquiteturais e considerações cuidadosas dos requisitos e restrições, é possível criar soluções de software eficientes, escaláveis e adaptáveis, que impulsionam o sucesso dos projetos de desenvolvimento de software.

2.1.1 Arquitetura Tradicional

A arquitetura tradicional, no contexto de sistemas Web, é um estilo arquitetural fortemente influenciado pelo estilo Cliente-Servidor (BASS; CLEMENTS; KAZMAN, 2012), onde as funcionalidades da aplicação são centralizadas em um único módulo coeso, denominado servidor. A Figura 2 ilustra os elementos arquiteturais de uma arquitetura Cliente-Servidor. Nesse tipo de arquitetura, os clientes são caracterizados por serem consumidores de serviços, que por sua vez são providos pelos componentes servidores. Além disso, todos os componentes e funcionalidades do sistema são agrupados e executados em um número controlado de processos, geralmente em um único servidor (SAKOVICH, 2017). Outra característica marcante das arquiteturas tradicionais é o fato de, mesmo nos casos onde há replicação de servidores Figura 2 (b), todos os servidores tendem a ser uniformes e autônomos, isto é, cada servidor executa todos os serviços necessários para compor a aplicação. Quanto à estrutura de código, o código-fonte da aplicação é geralmente organizado em projetos localizados em um único repositório. Todos os componentes relacionados à lógica de negócios e acesso a dados ficam contidos em um único código-base.

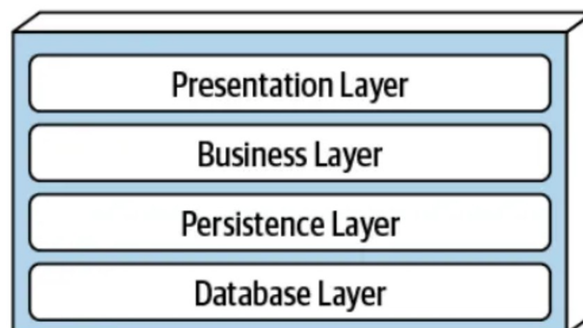
Figura 2 - Arquitetura Cliente-Servidor com servidor único (a) ou replicado (b)



Fonte: Fundação Wikimedia (2022). Adaptado (2023).

Dentro de uma aplicação tradicional, os diferentes componentes ou módulos são geralmente organizados em camadas lógicas, onde o número de camadas pode variar. No modelo de quatro camadas, apresentado na Figura 3, a camada de apresentação (*Presentation Layer*) é responsável pela interface de usuário e é a única localizada no lado "cliente"; todas as demais camadas estão centralizadas no servidor. A camada de negócios (*Business Layer*) contém as regras e a lógica de negócios, enquanto a camada de persistência de dados (*Persistence Layer*) é destinada à comunicação com o banco de dados. A camada de acesso a dados (*Database Layer*), por sua vez, é responsável pelo gerenciamento do banco de dados (RICHARDS; FORD, 2020). A estruturação do servidor em camadas internas facilita o desenvolvimento e a manutenção do sistema.

Figura 3 - Estrutura da arquitetura tradicional



Fonte: Richards; Ford (2020)

Uma das características da arquitetura tradicional é o maior acoplamento entre os componentes, quando comparada com a arquitetura de microsserviços, que será apresentada na Seção 2.1.2. Isso significa que na arquitetura tradicional, os diferentes módulos da aplicação estão intimamente ligados e dependem uns dos outros. O acoplamento ocorre principalmente por meio da comunicação direta, que ocorre entre os componentes do sistema.

Em termos de implantação e escalabilidade, a arquitetura tradicional exige que a aplicação seja implantada como uma única unidade física de execução (servidor). Isso significa que, quando uma parte da aplicação requer mais recursos ou atualizações, toda a aplicação precisa ser implantada novamente. A escalabilidade também é limitada, pois toda a aplicação precisa ser escalada de uma vez, em vez de permitir a escalabilidade granular por componente.

A arquitetura tradicional geralmente utiliza uma única linguagem de programação e conjunto de tecnologias para todo o sistema. Isso ocorre porque todas as partes da aplicação são interdependentes e interagem diretamente. Mas essa característica também pode ser vantajosa, pois pode facilitar a comunicação entre os desenvolvedores e manter a consistência no código.

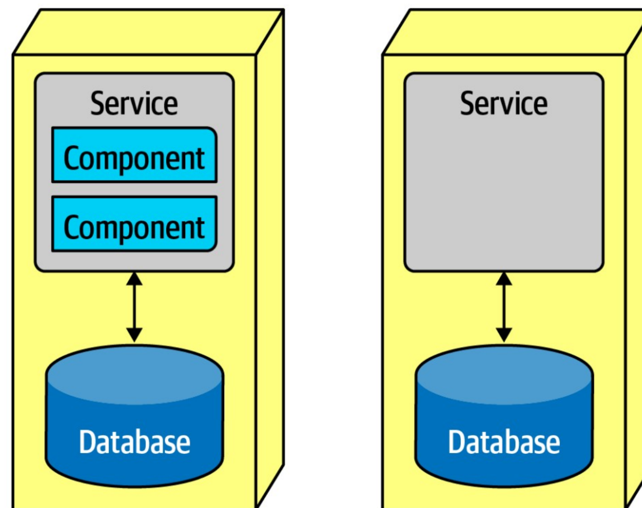
No que diz respeito ao banco de dados, em uma arquitetura tradicional, geralmente há um único banco de dados compartilhado por todos os componentes da aplicação. Isso significa que o esquema e o modelo de dados são aplicados globalmente, e qualquer mudança nesses aspectos afeta todos os módulos da aplicação.

À medida que a aplicação tradicional cresce, a complexidade tende a aumentar. A adição de novos recursos e funcionalidades pode se tornar mais desafiadora, uma vez que todas as partes da aplicação estão interligadas e mudanças em uma área podem afetar outras áreas do sistema. A manutenção e a depuração também podem ser mais complexas devido à interdependência dos componentes.

2.1.2 Arquitetura de Microsserviços

A arquitetura de software de microsserviços é um estilo arquitetural em que uma aplicação é decomposta em uma coleção de serviços independentes e autônomos, chamados de microsserviços. Trata-se de um estilo de arquitetura que se inspira fortemente nas ideias de *Design Orientado a Domínio* (RICHARDS; FORD, 2020). Nessa arquitetura, cada microsserviço é uma unidade de implementação que encapsula uma única funcionalidade ou um conjunto coeso de funcionalidades. Richards e Ford (2020) cita que, em arquiteturas de microsserviços, a simplicidade é um dos princípios arquiteturais. Assim, um serviço pode consistir em código suficiente para justificar componentes ou pode ser simples o suficiente para conter apenas um pequeno trecho de código, como ilustrado na Figura 4.

Figura 4 - Microsserviços podem dispensar componentes quando possuem pouco código



Fonte: Richards; Ford (2020).

Os microsserviços são projetados para serem independentes e se comunicam por meio de mecanismos de comunicação, como APIs. Essas interfaces definem as operações suportadas pelos microsserviços e as estruturas de dados que podem ser trocadas entre eles. Cada microsserviço é desenvolvido, implantado e dimensionado de forma independente dos outros. Isso significa que equipes de desenvolvimento diferentes podem trabalhar em microsserviços diferentes usando tecnologias e linguagens de programação adequadas para a tarefa específica.

Uma característica importante dos microsserviços é a autonomia, onde as partes trabalhando juntas em cima de uma base de código coesa, com foco em resolver as regras de negócio dento do domínio especificado (NEWMAN, 2015). Cada microsserviço possui sua própria base de código, banco de dados e possivelmente até mesmo seu próprio modelo de dados. Isso garante que cada microsserviço possa ser desenvolvido, testado, implantado e atualizado de forma independente dos demais, desde que as especificações continuem sendo seguidas.

A comunicação entre microsserviços pode ocorrer de forma síncrona ou assíncrona NEWMAN, 2019). Na comunicação síncrona, um microsserviço faz uma solicitação a outro microsserviço e aguarda a resposta antes de continuar sua execução. Já na comunicação assíncrona, os microsserviços se comunicam por meio de mensagens, onde um microsserviço pode publicar eventos ou mensagens que outros microsserviços podem consumir e processar quando estiverem prontos.

Em resumo, a arquitetura de software de microsserviços é baseada na decomposição de um sistema em serviços independentes que se comunicam por meio de interfaces bem definidas. Cada microsserviço é desenvolvido, implantado e dimensionado de forma independente, permitindo flexibilidade, agilidade e escalabilidade mais refinada (em menor granularidade), quando comparada com arquiteturas tradicionais.

2.2 ESCALABILIDADE VERTICAL E HORIZONTAL

A escalabilidade é um conceito fundamental em arquitetura de software, pois se refere à capacidade de um sistema lidar com um aumento na demanda, seja em termos de quantidade de usuários, volume de dados ou carga de processamento, de forma eficiente e sem comprometer o desempenho. Existem duas abordagens principais para a escalabilidade: vertical e horizontal.

A escalabilidade vertical, também conhecida como escalabilidade "scale-up", envolve aumentar a capacidade de um sistema adicionando mais recursos a uma única instância. Esses recursos podem incluir mais CPU, memória, armazenamento ou largura de banda de rede. Em outras palavras, é como adicionar mais poder de processamento a um único servidor ou máquina, tornando-a capaz de lidar com uma carga maior.

Esse tipo de escalabilidade é adequado quando o sistema está limitado por recursos específicos de uma única máquina e quando é possível aumentar esses recursos de forma significativa. No entanto, há um limite para a escalabilidade vertical, uma vez que os recursos de hardware têm um custo associado e podem atingir seu máximo suportado. Além disso, a escalabilidade vertical pode levar a um único ponto de falha, já que todo o sistema depende de uma única instância.

Por outro lado, a escalabilidade horizontal, também conhecida como escalabilidade "*scale-out*", envolve adicionar mais instâncias do sistema, distribuindo a carga entre elas. Em vez de aumentar os recursos de uma única máquina, o sistema é projetado para operar em um ambiente distribuído, com várias máquinas trabalhando em conjunto.

Nesse caso, cada instância do sistema é responsável por uma parte da carga de trabalho e pode ser replicada ou dividida em várias máquinas. Isso permite que o sistema cresça de forma linear, adicionando mais máquinas conforme necessário. Além disso, a escalabilidade horizontal oferece maior redundância e tolerância a falhas, uma vez que várias instâncias podem compensar o mau funcionamento de uma ou mais delas.

No entanto, para obter uma escalabilidade horizontal efetiva, é necessário projetar o sistema de forma a permitir a distribuição da carga e a coordenação entre as instâncias. Isso pode envolver a divisão das funcionalidades em serviços independentes, o uso de balanceadores de carga para direcionar as solicitações entre as instâncias e a implementação de mecanismos de comunicação e sincronização entre elas.

A escalabilidade horizontal é especialmente adequada para sistemas distribuídos e ambientes em nuvem, onde é possível provisionar e gerenciar instâncias conforme necessário. Além disso, ela oferece uma abordagem mais econômica em comparação com a escalabilidade vertical, uma vez que máquinas menos poderosas podem ser utilizadas, e os recursos podem ser ajustados de acordo com a demanda.

Ambas as abordagens têm seus benefícios e desafios, e a escolha entre elas depende das características específicas do sistema, dos requisitos de desempenho, das restrições de recursos e da disponibilidade de tecnologias e infraestrutura adequadas.

2.3 ISO/IEC 9126

O ISO/IEC 9126 é uma norma internacional que estabelece um modelo de qualidade de software. Essa norma foi desenvolvida pela Organização Internacional de Normalização (ISO) e pela Comissão Eletrotécnica Internacional (IEC) para fornecer diretrizes e critérios para avaliar a qualidade do software em termos de suas características e subcaracterísticas.

O ISO/IEC 9126 especifica as características e subcaracterísticas de qualidade do produto de software e propõe métricas para sua avaliação. É genérico e pode ser aplicado a qualquer tipo de produto de software, sendo adaptado para uma finalidade específica (KANELLOPOULOS *et al*., 2010).

Essa norma define seis características principais de qualidade do software, cada uma delas subdividida em várias subcaracterísticas. Essas características são as seguintes:

1. **Funcionalidade**: refere-se à capacidade do software de fornecer as funcionalidades declaradas e atender aos requisitos especificados. As subcaracterísticas incluem adequação funcional, precisão, interoperabilidade e conformidade.
2. **Confiabilidade**: diz respeito à capacidade do software de manter um nível adequado de desempenho em condições específicas ao longo do tempo. Isso inclui subcaracterísticas como maturidade, tolerância a falhas, capacidade de recuperação e conformidade com padrões de segurança.

3. **Usabilidade**: abrange a facilidade de uso e a capacidade do software de ser compreendido, aprendido e operado pelos usuários. Isso inclui subcaracterísticas como compreensibilidade, facilidade de aprendizado, operacionalidade e atratividade.
4. **Eficiência**: refere-se ao desempenho do software em relação aos recursos utilizados. Inclui subcaracterísticas como tempo de resposta, utilização de recursos, comportamento em relação à carga e eficiência operacional.
5. **Manutenibilidade**: diz respeito à facilidade de manutenção e modificação do software. Inclui subcaracterísticas como analisabilidade, modificabilidade, estabilidade e testabilidade.
6. **Portabilidade**: abrange a capacidade do software de ser transferido de um ambiente para outro. Isso inclui subcaracterísticas como adaptabilidade, facilidade de instalação, interoperabilidade e capacidade de substituição.

Cada subcaracterística é definida em termos de atributos mensuráveis e critérios específicos, fornecendo uma base objetiva para avaliar e medir a qualidade do software. Esses critérios podem ser aplicados tanto durante o desenvolvimento do software quanto na avaliação de produtos de software já existentes.

A aplicação da norma ISO/IEC 9126 requer a definição de métricas e a realização de avaliações sistemáticas para cada uma das características e subcaracterísticas. Essas métricas podem variar dependendo do contexto do projeto e dos requisitos específicos do software em questão.

É importante ressaltar que a norma ISO/IEC 9126 fornece um modelo abrangente de qualidade do software, mas sua aplicação efetiva requer experiência e conhecimento especializado para interpretar e aplicar os critérios de acordo com as necessidades e objetivos do projeto.

3 METODOLOGIA

Esta monografia baseia-se em uma revisão bibliográfica de livros e artigos científicos relevantes sobre o tema "arquitetura tradicional e arquitetura de microsserviços". A identificação e seleção das fontes bibliográficas inicialmente realizou-se a partir de uma busca sistemática por livros e artigos científicos em bases de dados acadêmicas, como Google Scholar, IEEE Xplore, ACM Digital Library e Periódicos CAPES. Utilizaram-se palavras-chave como "arquitetura de software", "arquitetura de microsserviços", "arquitetura tradicional", "comparação de arquiteturas", "vantagens e desvantagens de arquiteturas de software", "decomposição de sistemas de software", entre outras, para garantir a abrangência da pesquisa.

Além da pesquisa bibliográfica, foram explorados também recursos disponíveis em sites e portais de empresas líderes no mercado de arquitetura de software, como Netflix, Amazon Web Services (AWS) e Google Cloud Platform. Essas empresas são reconhecidas por suas práticas inovadoras em arquitetura de software e fornecem estudos de caso, artigos técnicos e documentação que contribuem significativamente para a compreensão das arquiteturas e suas aplicações práticas.

As fontes selecionadas foram submetidas a critérios de inclusão e exclusão para garantir a relevância e a qualidade dos materiais utilizados. Foram incluídos livros e artigos que abordassem diretamente a explanação e comparação entre arquitetura tradicional e arquitetura de microsserviços. Fontes duplicadas, não relacionadas ao tema ou que não apresentassem rigor científico adequado foram excluídas.

Durante a leitura crítica das fontes selecionadas, foram feitos anotações e resumos para auxiliar na organização das informações. Os principais conceitos, ideias, teorias e resultados relacionados à comparação entre as duas arquiteturas foram extraídos e organizados em categorias e tópicos relevantes. Temas comuns entre as abordagens foram identificados para facilitar a análise comparativa entre as mesmas.

Essa revisão e organização da literatura permitiu uma análise preliminar, porém embasada, das arquiteturas tradicional e de microsserviços, bem como uma avaliação dos atributos de qualidade com base nas métricas da ISO/IEC 9126. A combinação da pesquisa bibliográfica com a análise de conteúdo disponível em sites e portais de empresas líderes no setor contribuiu para uma análise abrangente e atualizada das arquiteturas e suas implicações práticas.

4 ANÁLISE COMPARATIVA

4.1 MANUTENIBILIDADE

A manutenibilidade de um sistema de software é um aspecto fundamental que influencia sua capacidade de ser mantido, atualizado e expandido ao longo do tempo. Nesta seção, vamos examinar minuciosamente como a arquitetura tradicional e a arquitetura de microsserviços diferem em termos de manutenibilidade.

4.1.1 Atualizações de software

Na arquitetura tradicional, as atualizações de software geralmente envolvem a modificação do código-fonte. Isso significa que qualquer alteração em uma parte do sistema pode exigir a recompilação, o teste e a implantação de toda a aplicação.

Uma das principais desvantagens dessa abordagem é a necessidade de implantar a aplicação como um todo, mesmo que apenas uma pequena parte tenha sido modificada (SAKOVICH, 2017). Isso pode resultar em tempo de inatividade desnecessário e aumentar o risco de interrupções durante o processo de atualização.

Além disso, a interdependência dos componentes no código-fonte da aplicação tradicional pode dificultar a realização de atualizações de software (SAKOVICH, 2017). Alterações em uma área específica podem ter efeitos colaterais indesejados em outras áreas do código, exigindo um cuidadoso planejamento e testes abrangentes para garantir que a aplicação continue funcionando corretamente após a atualização.

Já na arquitetura de microsserviços, as atualizações de software são mais granulares e podem ser realizadas em microsserviços individuais. Cada microsserviço é uma unidade independente e pode ser desenvolvido, testado e implantado separadamente.

Essa abordagem oferece uma série de benefícios em termos de atualizações de software. Primeiramente, como os microsserviços são independentes uns dos outros, é possível implantar uma nova versão de um microsserviço específico sem afetar a disponibilidade global do sistema (NEWMAN, 2015). Isso permite atualizações contínuas e progressivas, reduzindo o tempo de inatividade e os riscos associados à implantação.

Além de que, a modularidade dos microsserviços facilita a implementação de atualizações de software em partes específicas do sistema. Os desenvolvedores podem se concentrar em um único microsserviço, testá-lo adequadamente e implantar as alterações, sem

afetar outros componentes ou funcionalidades do sistema. Isso agiliza o processo de atualização e minimiza o impacto em outras partes do sistema.

No entanto, é importante ressaltar que a comunicação e a coordenação entre os microsserviços devem ser gerenciadas de forma adequada durante as atualizações (RICHARDS; FORD, 2020). É necessário garantir a compatibilidade das interfaces de comunicação entre os microsserviços e estabelecer mecanismos de versionamento para lidar com diferentes versões em execução.

Além disso, a implementação eficaz de atualizações de software em uma arquitetura de microsserviços exige uma infraestrutura de suporte robusta (NEWMAN, 2015). É necessário ter ferramentas de gerenciamento de configuração e implantação, bem como sistemas de monitoramento e registro centralizados para facilitar o controle e a governança dos microsserviços.

4.1.2 Adição de funcionalidades

Na arquitetura tradicional, a adição de novas funcionalidades geralmente também requer a modificação do código-fonte. Dependendo da complexidade do sistema, isso pode exigir mudanças em várias partes do código, afetando potencialmente outras funcionalidades existentes (RICHARDS; FORD, 2020).

Um dos principais desafios na adição de funcionalidades em uma arquitetura tradicional é a interdependência dos componentes. Alterações em uma área específica do código podem ter impacto em outras áreas, exigindo testes extensivos para garantir que as alterações não causem efeitos colaterais indesejados.

Além disso, a necessidade de recompilar e implantar toda a aplicação para adicionar uma nova funcionalidade pode ser demorada e aumentar o risco de erros. Isso pode resultar em um ciclo de desenvolvimento mais lento e menos ágil, especialmente em sistemas complexos com muitas funcionalidades interconectadas.

Na arquitetura de microsserviços, a adição de funcionalidades é facilitada pela modularidade dos microsserviços. Essa abordagem traz vantagens significativas quando se trata de adicionar novas funcionalidades (RICHARDS; FORD, 2020). Os desenvolvedores podem se concentrar em um único microsserviço e implementar a nova funcionalidade sem afetar outros microsserviços ou a funcionalidade existente do sistema como um todo.

Essa independência dos microsserviços agiliza o processo de desenvolvimento, teste e implantação de novas funcionalidades. Cada equipe pode trabalhar de forma autônoma,

desenvolvendo a nova funcionalidade em paralelo com outras equipes, o que reduz o tempo necessário para trazer a nova funcionalidade ao mercado.

No entanto, é fundamental garantir a coordenação adequada entre os microsserviços para que a nova funcionalidade seja efetivamente disponibilizada ao usuário final. Isso pode envolver o gerenciamento de dependências e a definição clara das interfaces entre os microsserviços para garantir uma comunicação adequada e a integração correta das novas funcionalidades.

Além de tudo, adição de funcionalidades em uma arquitetura de microsserviços requer uma abordagem cuidadosa em relação ao dimensionamento e à escalabilidade (NEWMAN, 2015). À medida que novos microsserviços são adicionados, é importante considerar o impacto no desempenho e na capacidade do sistema como um todo, garantindo que a infraestrutura seja dimensionada adequadamente para atender à demanda crescente.

4.2 MODULARIDADE

A modularidade é um aspecto importante a ser considerado ao comparar ambas abordagens, A modularidade na arquitetura de software refere-se à organização de um sistema em módulos independentes e interconectados, onde cada módulo é responsável por uma funcionalidade específica. Nesta seção, discutiremos como essas abordagens diferem em termos de modularidade e exploraremos duas perspectivas específicas: divisão em componentes e testabilidade.

4.2.1 Divisão em componentes

A divisão em componentes é um aspecto fundamental da modularidade na arquitetura de software. Na arquitetura tradicional, essa divisão é frequentemente realizada por meio de camadas lógicas, onde diferentes partes do sistema são agrupadas em camadas distintas, cada uma com uma responsabilidade específica (RICHARDS; FORD, 2020). Geralmente, as camadas comuns incluem a camada de apresentação, a camada de negócios e a camada de acesso a dados.

A camada de apresentação é responsável por lidar com a interface do usuário e a interação com o sistema. Ela inclui componentes como telas, formulários, botões e outros elementos de interação com o usuário. Essa camada é responsável por receber as entradas do usuário, exibir informações relevantes e encaminhar as solicitações para a camada de negócios.

A camada de negócios contém a lógica e as regras de negócios do sistema. Ela processa as solicitações recebidas da camada de apresentação, executa as operações necessárias e coordena a interação com outros componentes do sistema. Essa camada é responsável por validar as entradas, aplicar as regras de negócios, executar cálculos e tomar decisões com base nas regras definidas.

A camada de acesso a dados é responsável pelo gerenciamento do acesso e da persistência dos dados. Ela interage com o banco de dados ou outros sistemas de armazenamento de dados para recuperar e persistir as informações necessárias. Essa camada lida com a busca, a atualização e a remoção de dados, garantindo a integridade e a consistência dos dados no sistema.

A divisão em camadas facilita a organização e a estruturação do código, permitindo que diferentes aspectos do sistema sejam tratados de forma separada. Isso ajuda a melhorar a legibilidade, a manutenibilidade e a reutilização do código. Além disso, essa divisão promove a separação de preocupações, onde cada camada é responsável por uma área específica do sistema, facilitando a compreensão e o gerenciamento das funcionalidades (RICHARDS; FORD, 2020).

No entanto, a divisão em camadas também pode resultar em um alto acoplamento entre os componentes dentro de cada camada. Isso ocorre porque as camadas são interdependentes e qualquer alteração em uma camada pode afetar as outras. Por exemplo, uma mudança na camada de acesso a dados pode exigir modificações na camada de negócios e na camada de apresentação.

Além disso, a divisão em camadas pode levar a uma comunicação excessiva entre os componentes (RICHARDS; FORD, 2020). Por exemplo, para realizar uma determinada funcionalidade, pode ser necessário que a camada de apresentação se comunique com a camada de negócios, que por sua vez comunica-se com a camada de acesso a dados. Essa comunicação indireta pode aumentar a complexidade e a sobrecarga do sistema.

Por outro lado, na arquitetura de microsserviços, a divisão em componentes é baseada nos próprios microsserviços. Cada microsserviço é uma unidade autônoma que encapsula uma funcionalidade específica do sistema. Essa abordagem promove uma divisão clara das responsabilidades e permite que cada microsserviço seja desenvolvido de forma independente.

Essa divisão em microsserviços oferece várias vantagens em relação à divisão em camadas. Primeiramente, a divisão em microsserviços promove um baixo acoplamento entre os componentes, uma vez que cada microsserviço é independente e não possui dependências diretas com outros microsserviços (NEWMAN, 2015). Isso facilita a manutenção e a

evolução do sistema, pois as alterações em um microsserviço não afetam diretamente os outros.

Além disso, a divisão em microsserviços permite que equipes de desenvolvimento diferentes trabalhem em microsserviços específicos usando tecnologias e linguagens de programação adequadas para cada tarefa. Isso promove a flexibilidade e a especialização, pois cada equipe pode se concentrar em uma funcionalidade específica e adotar as melhores práticas e ferramentas para aquela área específica.

4.2.2 Testabilidade

Na arquitetura tradicional, a testabilidade pode ser um desafio, especialmente quando se trata de testes unitários. Devido à natureza monolítica da arquitetura tradicional, em que todos os componentes estão intimamente acoplados e compartilham o mesmo código-base, pode ser difícil isolar um componente específico para testá-lo independentemente. Isso ocorre porque a dependência de outros componentes e a complexidade do código-fonte podem dificultar a criação de testes unitários eficazes (RICHARDS; FORD, 2020).

Além disso, a configuração e o gerenciamento de ambientes de teste podem ser complicados na arquitetura tradicional. Como todos os componentes estão agrupados e executados em um único processo, pode ser necessário configurar um ambiente completo para realizar testes funcionais ou de integração. Isso pode exigir a configuração de bancos de dados, serviços externos e outros recursos, o que pode ser demorado e propenso a erros.

Por outro lado, a arquitetura de microsserviços favorece a testabilidade devido à modularidade dos microsserviços (NEWMAN, 2015). Cada microsserviço pode ser testado individualmente, permitindo a criação de casos de teste específicos e a execução de testes unitários de forma isolada. Isso facilita a identificação e a resolução de problemas, uma vez que é possível isolar e corrigir erros em um microsserviço sem afetar os outros.

Além disso, a testabilidade na arquitetura de microsserviços é impulsionada pela facilidade de implantação e dimensionamento dos microsserviços. A capacidade de implantar e escalar microsserviços independentemente permite a criação de ambientes de teste controlados e a execução de testes em escala reduzida (LEWIS; FOWLER, 2014). Isso possibilita a realização de testes de carga, estresse e desempenho mais eficazes, fornecendo uma visão abrangente do comportamento do sistema.

Entretanto, é importante destacar que a testabilidade na arquitetura de microsserviços também traz desafios. A comunicação e a coordenação entre os microsserviços durante os

testes devem ser gerenciadas adequadamente. É necessário garantir a consistência dos dados e o comportamento correto dos microsserviços em diferentes cenários de teste. Além disso, a complexidade de configurar e orquestrar os ambientes de teste para múltiplos microsserviços pode exigir ferramentas e práticas específicas para garantir a eficácia dos testes.

4.3 TOLERÂNCIA A FALHAS

A tolerância a falhas é um aspecto crítico em arquitetura de software, pois um sistema robusto deve ser capaz de lidar com erros e falhas de forma adequada, minimizando seu impacto e garantindo a continuidade do serviço. Nesta seção, exploraremos como a arquitetura tradicional e a arquitetura de microsserviços abordam a tolerância a falhas.

4.3.1 Recuperação de erros

Na arquitetura tradicional, a recuperação de erros é uma etapa crucial para garantir a estabilidade do sistema. Ela envolve a identificação, tratamento e correção de erros que possam ocorrer durante a execução do software. Esses erros podem ser causados por falhas de hardware, falhas de rede, erros de programação ou situações excepcionais inesperadas.

Para lidar com a recuperação de erros na arquitetura tradicional, são adotadas várias estratégias e técnicas. Uma delas é o tratamento de exceções, que permite capturar e manipular erros em tempo de execução. Por meio do uso de blocos try-catch, é possível identificar exceções específicas e tomar ações apropriadas para lidar com elas (SAKOVICH, 2017). Isso ajuda a evitar a interrupção abrupta do sistema e a fornecer uma resposta adequada ao erro.

Outra técnica comumente utilizada é o registro de erros, onde informações sobre erros são armazenadas em logs para análise posterior. Os logs podem conter detalhes sobre o erro, como a data e hora em que ocorreu, o local onde ocorreu e quaisquer informações relevantes para ajudar na depuração. Esses registros são úteis para identificar padrões de erros recorrentes, investigar a causa raiz de problemas e realizar melhorias futuras no sistema.

Além disso, a arquitetura tradicional pode adotar mecanismos de resiliência, como o retry automático de operações falhas. Isso significa que, quando uma operação falha, o sistema tenta executá-la novamente em um determinado intervalo de tempo. Essa abordagem visa lidar com erros temporários, como falhas de rede ou timeouts, permitindo que a operação seja concluída com sucesso em tentativas subsequentes.

No entanto, é importante notar que a recuperação de erros na arquitetura tradicional pode ser desafiadora em alguns casos. O alto acoplamento entre os componentes significa que um erro em um componente pode afetar outros componentes dependentes, aumentando a complexidade da recuperação (RICHARDS; FORD, 2020). Além disso, a necessidade de recompilar e implantar toda a aplicação para corrigir um erro pode ser demorada e aumentar o risco de introduzir novos problemas.

Por outro lado, na arquitetura de microsserviços, a recuperação de erros é abordada de forma descentralizada e granular. Cada microsserviço é responsável por sua própria recuperação de erros, o que significa que uma falha em um microsserviço específico não afeta diretamente os outros. Essa abordagem permite uma resposta rápida e eficaz aos erros, minimizando seu impacto no sistema como um todo (RICHARDS; FORD, 2020).

Os microsserviços podem adotar estratégias específicas para a recuperação de erros. Isso pode incluir o uso de circuit breakers, que monitoram o estado de um microsserviço e interrompem temporariamente as chamadas para ele em caso de falha contínua. Essa abordagem evita sobrecarregar o sistema com chamadas malsucedidas e permite que o microsserviço se recupere e volte a funcionar corretamente antes de retomar as chamadas.

Outra técnica utilizada na arquitetura de microsserviços é a implementação de estratégias de retry e fallback em nível de microsserviço. Isso permite que um microsserviço tente novamente executar uma operação após uma falha temporária ou redirecione a chamada para outro microsserviço alternativo em caso de falha permanente (NEWMAN, 2015). Essas estratégias ajudam a garantir a continuidade do serviço e a minimizar o impacto das falhas no sistema como um todo.

4.3.2 Isolamento de falhas

O isolamento de falhas é uma importante consideração na arquitetura de software, especialmente quando se trata de garantir a tolerância a falhas em um sistema. O objetivo do isolamento de falhas é limitar o impacto de uma falha em um componente ou microsserviço específico, evitando que ela se propague para outros componentes e afete o funcionamento geral do sistema.

Na arquitetura tradicional, o isolamento de falhas pode ser mais desafiador devido ao alto acoplamento entre os componentes (RICHARDS; FORD, 2020). Uma falha em um componente pode se espalhar rapidamente para outros componentes dependentes, resultando em uma interrupção geral do sistema. Além disso, a necessidade de recompilar e implantar

toda a aplicação para corrigir uma falha pode ser demorada e, também como no tópico anterior, aumentar o risco de introduzir novos problemas.

No entanto, existem algumas técnicas e estratégias que podem ser aplicadas na arquitetura tradicional para mitigar os efeitos de uma falha e melhorar o isolamento de falhas. Uma delas é a implementação de mecanismos de resiliência, como os já citados, circuit breakers e fallback.

Já se tratando da arquitetura de microsserviços, o isolamento de falhas é abordado de forma mais granular e descentralizada (NEWMAN, 2015). Cada microsserviço é projetado para ser autônomo e isolado, o que significa que uma falha em um microsserviço não afeta diretamente os outros. Isso permite um isolamento mais efetivo e uma melhor capacidade de lidar com falhas em um sistema distribuído.

Além disso, a arquitetura de microsserviços pode adotar práticas de design como o uso de padrões de resiliência, como o Circuit Breaker Pattern e o Bulkhead Pattern (RICHARDS; FORD, 2020). Esses padrões ajudam a controlar a propagação de falhas e a limitar o impacto em todo o sistema. O Circuit Breaker Pattern monitora a comunicação entre microsserviços e interrompe as chamadas para um microsserviço em caso de falha contínua, permitindo que o sistema se recupere antes de retomar as chamadas. O Bulkhead Pattern isola cada microsserviço em seu próprio ambiente de execução, garantindo que uma falha em um serviço não afete negativamente os outros.

Além das estratégias de isolamento de falhas, a essa abordagem também pode se beneficiar da implantação em contêineres (RICHARDS; FORD, 2020). Os contêineres fornecem uma camada adicional de isolamento entre os microsserviços, garantindo que uma falha em um contêiner específico não afete os outros. Isso permite que os microsserviços sejam dimensionados e atualizados de forma independente, contribuindo para a tolerância a falhas e a disponibilidade contínua do sistema.

4.3.3 Disponibilidade

A disponibilidade é um atributo de qualidade crucial em sistemas de software, pois se refere à capacidade de um sistema estar sempre disponível e operacional para os usuários, mesmo diante de falhas ou interrupções. Na arquitetura de software, a disponibilidade pode ser afetada por vários fatores, como falhas de hardware, falhas de rede, atualizações de software ou aumento repentino de demanda.

Na arquitetura tradicional, garantir alta disponibilidade pode ser desafiador devido à natureza monolítica do sistema. Uma falha em um componente crítico pode levar à interrupção de todo o sistema, resultando em tempo de inatividade para os usuários (RICHARDS; FORD, 2020). Além disso, a necessidade de implantar a aplicação como um todo para realizar atualizações ou corrigir problemas pode resultar em períodos de inatividade planejados e impactar negativamente a disponibilidade.

No entanto, existem estratégias e práticas que podem ser adotadas para melhorar a disponibilidade em uma arquitetura tradicional. Uma delas é a implementação de técnicas de redundância (RICHARDS; FORD, 2020). Por exemplo, ao ter servidores redundantes, se um servidor falhar, outro pode assumir suas funções e manter o sistema em funcionamento. Isso pode ser alcançado por meio de técnicas como a replicação de servidores e o balanceamento de carga, distribuindo a carga entre vários servidores para evitar pontos únicos de falha.

Além disso, a arquitetura tradicional pode se beneficiar do uso de sistemas de monitoramento e gerenciamento de falhas. Esses sistemas podem detectar problemas e alertar a equipe de operações para tomar medidas corretivas rapidamente. Isso pode reduzir o tempo de inatividade e minimizar o impacto nos usuários finais.

Por outro lado, a arquitetura de microsserviços oferece oportunidades adicionais para melhorar a disponibilidade do sistema. Como cada microsserviço é autônomo e independente, uma falha em um microsserviço específico não afeta diretamente os outros (NEWMAN, 2015). Isso significa que, mesmo que um microsserviço fique indisponível, o restante do sistema pode continuar operando normalmente. Isso contribui para uma maior tolerância a falhas e melhor disponibilidade.

Outra prática comum na arquitetura de microsserviços é a implementação de estratégias de resiliência, como o uso de filas e mensageria assíncrona (NEWMAN, 2015). Essas abordagens permitem que os microsserviços processem solicitações de forma assíncrona, o que significa que eles podem continuar funcionando mesmo quando um componente está temporariamente indisponível. Isso ajuda a garantir a disponibilidade contínua do sistema, mesmo em situações de falha.

No entanto, é importante ressaltar que a disponibilidade em uma arquitetura de microsserviços também requer atenção especial. A comunicação e a coordenação entre os microsserviços devem ser gerenciadas adequadamente para garantir que a disponibilidade seja mantida. O uso de estratégias de monitoramento, gerenciamento de falhas e recuperação automática de falhas é fundamental para garantir que os microsserviços estejam operacionais e disponíveis para os usuários.

4.4 ESCALABILIDADE

Escalabilidade em arquitetura de software refere-se à capacidade de um sistema lidar com um aumento de carga ou demanda, mantendo o desempenho e a disponibilidade. É a capacidade de dimensionar o sistema para lidar com maior volume de dados, tráfego, usuários ou processamento sem comprometer sua eficiência ou qualidade. Neste tópico trataremos da principal questão nesse tema que é a adição de recursos.

Um aspecto relevante a ser ressaltado é o fato da arquitetura de microsserviços incentivar a escalabilidade horizontal do sistema (ver Seção 2.2), em que múltiplas instâncias de um mesmo microsserviço são executadas para atender à demanda. Isso pode ser alcançado por meio de implantações em contêineres ou em ambientes de nuvem, onde as instâncias podem ser criadas ou encerradas sob demanda, conforme a necessidade de uso.

Já no caso das arquiteturas tradicionais, a escalabilidade ocorre em um nível de granularidade maior, envolvendo a replicação de todos os serviços do servidor. Uma estratégia comum nesse caso é a utilização de máquinas virtuais para os servidores, que podem ser executadas sob demanda. Porém, é importante ressaltar que nesse caso o custo da escalabilidade é maior, uma vez que todo o servidor de aplicação precisa ser replicado, em contraste com a replicação de serviços isolados, utilizada uma arquitetura de microsserviços.

4.4.1 Adição de recursos

Na arquitetura tradicional, a escalabilidade é geralmente alcançada por meio da adição de recursos em nível de infraestrutura, como servidores mais potentes, aumento de capacidade de armazenamento ou dimensionamento vertical (scale-up). Isso implica em aumentar a capacidade de processamento e recursos de todo o sistema para lidar com um aumento na demanda. No entanto, essa abordagem tem algumas limitações (LEWIS; FOWLER, 2014).

Um dos desafios da escalabilidade na arquitetura tradicional é a necessidade de dimensionar todo o sistema, mesmo que apenas um componente específico ou uma funcionalidade isolada exija recursos adicionais. Isso pode levar a um uso ineficiente de recursos, já que nem todos os componentes podem precisar ser escalados proporcionalmente. Além disso, a escalabilidade vertical pode ter limitações físicas e financeiras, uma vez que a adição de recursos em um único servidor tem um limite máximo.

Já na arquitetura de microsserviços, a escalabilidade é mais granular e flexível. Cada microsserviço é um componente independente e pode ser dimensionado individualmente, permitindo a adição de recursos apenas onde é necessário. Essa abordagem possibilita uma

melhor utilização dos recursos disponíveis, pois apenas os microsserviços relevantes para atender à demanda podem ser escalados. Isso resulta em um uso mais eficiente dos recursos de hardware e infraestrutura.

Além disso, na arquitetura de microsserviços, a escalabilidade pode ser alcançada por meio da adição de instâncias de microsserviços (dimensionamento horizontal ou scale-out) (RICHARDS; FORD, 2020). Isso significa que, em vez de aumentar a capacidade de um único componente, é possível adicionar mais instâncias dos microsserviços necessários para lidar com a carga crescente. Dessa forma, é possível distribuir a carga de trabalho de maneira mais equilibrada e escalonar horizontalmente, conforme a demanda exigir.

A adição de recursos na arquitetura de microsserviços pode ser realizada de forma mais ágil e automatizada. Com o uso de ferramentas de orquestração, como o Kubernetes, e conceitos como implantação de contêineres, é possível adicionar e remover instâncias de microsserviços dinamicamente com base em métricas e políticas predefinidas. Isso permite uma resposta rápida e elástica às flutuações de demanda, garantindo que os recursos sejam alocados conforme necessário.

No entanto, é importante considerar que a escalabilidade na arquitetura de microsserviços também apresenta desafios. A coordenação entre os microsserviços, o gerenciamento das dependências e a garantia de que a infraestrutura de suporte esteja dimensionada adequadamente são aspectos críticos a serem considerados. Além disso, é fundamental ter uma estratégia eficaz de monitoramento e gerenciamento de recursos para garantir um escalonamento adequado e evitar gargalos de desempenho.

4.5 GERENCIAMENTO DE DADOS

O gerenciamento de dados desempenha um papel crucial no desenvolvimento e na operação de sistemas de software, independentemente da arquitetura adotada. Neste tópico, serão abordados aspectos relacionados ao armazenamento e à governança de dados nas arquiteturas tradicional e de microsserviços.

4.5.1 Armazenamento de dados

No contexto do gerenciamento de dados, o armazenamento de dados é um aspecto fundamental a ser considerado no desenvolvimento e na operação de sistemas de software. Tanto na arquitetura tradicional quanto na arquitetura de microsserviços, existem diferentes

abordagens para armazenar os dados, cada uma com suas características e considerações específicas.

Na arquitetura tradicional, é comum utilizar um banco de dados centralizado, conhecido como Sistema de Gerenciamento de Banco de Dados Relacional (SGBDR). Esses bancos de dados são projetados para armazenar grandes volumes de dados e fornece mecanismos avançados de consulta. Eles usam tabelas para organizar e estruturar os dados, garantindo a integridade e a consistência dos mesmos. Essa abordagem é amplamente adotada em ambientes corporativos, onde a confiabilidade e a precisão dos dados são fundamentais.

No entanto, a abordagem centralizada de armazenamento de dados pode apresentar desafios em termos de escalabilidade e flexibilidade como já citado no tópico 6.4. À medida que o volume de dados cresce ou a demanda por acesso aos dados aumenta, o banco de dados centralizado pode se tornar um gargalo de desempenho. Além disso, qualquer alteração no esquema do banco de dados pode exigir atualizações e migrações complexas em todo o sistema.

Na arquitetura de microsserviços, a abordagem de armazenamento de dados é mais descentralizada (NEWMAN, 2015). Cada microsserviço pode ter seu próprio banco de dados dedicado ou utilizar diferentes tecnologias de armazenamento, como bancos de dados NoSQL, armazenamento em nuvem ou armazenamento em memória. Essa abordagem descentralizada permite que cada microsserviço tenha autonomia sobre seus próprios dados e escolha a tecnologia de armazenamento mais adequada às suas necessidades específicas (NEWMAN, 2015).

Os bancos de dados NoSQL, por exemplo, são conhecidos por sua capacidade de lidar com grandes volumes de dados e oferecer escalabilidade e desempenho superiores. Eles são especialmente úteis em casos onde a flexibilidade do esquema de dados e a capacidade de escalabilidade horizontal são prioridades.

Além disso, a arquitetura de microsserviços também pode adotar abordagens assíncronas de armazenamento de dados, onde cada microsserviço armazena seus próprios eventos e mensagens em filas ou sistemas de mensagens distribuídos (RICHARDS; FORD, 2020). Essa abordagem é útil em cenários que exigem alta disponibilidade e tolerância a falhas, garantindo a consistência dos dados por meio de processos assíncronos de sincronização e replicação.

Outra opção de armazenamento de dados é o uso de serviços de armazenamento em nuvem. Esses serviços permitem armazenar e recuperar grandes volumes de dados de forma escalável e eficiente, sem a necessidade de gerenciar a infraestrutura subjacente. Eles são

particularmente úteis para cenários onde a escalabilidade, a durabilidade e a disponibilidade dos dados são essenciais.

4.5.2 Governança de dados

A governança de dados é um aspecto fundamental do gerenciamento de dados em qualquer arquitetura de software. Ela envolve a definição e implementação de políticas, procedimentos e processos para garantir a qualidade, a integridade, a segurança e o uso adequado dos dados em toda a aplicação.

Na arquitetura tradicional, a governança de dados é geralmente realizada de forma centralizada (RICHARDS; FORD, 2020), onde há um conjunto de regras e controles definidos para garantir a consistência e a conformidade dos dados. Isso é frequentemente implementado por meio de sistemas de gerenciamento de banco de dados robustos, políticas de segurança rigorosas e procedimentos de acesso controlado aos dados.

Nesse contexto, a governança de dados na arquitetura tradicional envolve atividades como a definição de esquemas de banco de dados, a especificação de políticas de segurança e acesso, a implementação de mecanismos de backup e recuperação, a criação de regras de integridade dos dados e a garantia da conformidade com regulamentações e normas aplicáveis (RICHARDS; FORD, 2020). Essas atividades visam manter a qualidade e a consistência dos dados, além de garantir sua disponibilidade, confidencialidade e integridade.

Na arquitetura de microsserviços, a governança de dados assume uma abordagem mais distribuída devido à natureza descentralizada da arquitetura. Cada microsserviço pode ter suas próprias regras e controles de governança de dados, responsáveis por definir as políticas de uso, acesso, segurança e qualidade dos dados específicos ao microsserviço.

Essa abordagem descentralizada da governança de dados em microsserviços exige uma coordenação eficiente entre os diferentes microsserviços para garantir a consistência e a conformidade dos dados em todo o sistema (NEWMAN, 2015). Isso pode ser alcançado por meio do estabelecimento de acordos de interface e contratos de comunicação entre os microsserviços, que definem as regras e os padrões para a troca de dados.

Além disso, a governança de dados em arquiteturas de microsserviços também pode envolver a implementação de mecanismos de rastreamento e auditoria dos dados, a fim de garantir a rastreabilidade e a responsabilidade no uso e manipulação dos dados. Isso pode incluir a utilização de logs, trilhas de auditoria e registros de alterações para registrar as atividades relacionadas aos dados em cada microsserviço.

Outro aspecto importante da governança de dados em microsserviços é a adoção de práticas de gerenciamento de metadados (NEWMAN, 2019). Os metadados são informações descritivas sobre os dados, como sua origem, formato, significado e relacionamentos. O gerenciamento eficiente dos metadados permite um melhor entendimento e controle dos dados em um ambiente distribuído, facilitando a descoberta, a integração e a governança dos dados.

5 ESTUDOS DE CASOS

5.1 NETFLIX: MIGRAÇÃO DE MONOLITO PARA MICROSSERVIÇOS

A transição da Netflix de uma arquitetura monolítica para uma arquitetura de microsserviços é um caso de estudo interessante que exemplifica os benefícios e os desafios enfrentados durante essa migração. A Netflix, um dos maiores provedores de streaming de conteúdo, precisou lidar com um aumento exponencial no número de usuários, conteúdo e complexidade do sistema, levando-os a adotar uma arquitetura mais flexível e escalável.

Anteriormente, a Netflix operava um aplicativo monolítico, onde todas as funcionalidades estavam agrupadas em um único código-base. Embora essa abordagem tenha sido viável no início, conforme a popularidade da plataforma cresceu, o monolito começou a apresentar limitações. A escalabilidade se tornou um desafio, pois qualquer mudança ou atualização exigia implantar e dimensionar todo o aplicativo, mesmo que apenas uma pequena parte fosse afetada.

A decisão estratégica de migrar para uma arquitetura de microsserviços foi tomada para enfrentar esses desafios. A Netflix optou por decompor seu sistema monolítico em uma coleção de microsserviços independentes e autônomos. Cada microsserviço foi projetado para tratar de uma funcionalidade específica, como recomendações de conteúdo, gerenciamento de usuários, processamento de pagamentos e reprodução de vídeo.

A migração permitiu à Netflix obter diversos benefícios. Em relação à escalabilidade, os microsserviços podem ser dimensionados individualmente, o que significa que recursos adicionais podem ser alocados para os microsserviços que exigem maior capacidade, enquanto outros permanecem inalterados. Isso resultou em um melhor aproveitamento dos recursos e na capacidade de lidar com picos de demanda sem afetar o desempenho global do sistema.

Além disso, a arquitetura de microsserviços possibilitou maior agilidade no desenvolvimento e na entrega de novos recursos. Equipes de desenvolvimento independentes foram formadas para trabalhar em microsserviços específicos, com cada equipe sendo responsável por desenvolver, testar e implantar seu próprio microsserviço. Essa abordagem descentralizada acelerou o ciclo de desenvolvimento, permitindo que a Netflix lançasse atualizações e novas funcionalidades com mais rapidez, mantendo-se à frente da concorrência.

A modularidade dos microsserviços também facilitou a manutenção contínua do sistema. Como cada microsserviço é isolado e independente, alterações em um microsserviço não

afetam diretamente os outros. Isso significa que atualizações, correções de bugs e aprimoramentos podem ser aplicados de forma mais rápida e segura, sem a necessidade de testar e implantar o sistema inteiro. Além disso, a detecção e a resolução de problemas se tornaram mais fáceis, uma vez que os microsserviços isolados simplificam a identificação de áreas problemáticas específicas.

No entanto, a migração para a arquitetura de microsserviços também apresentou desafios significativos para a Netflix. A complexidade do ecossistema de microsserviços aumentou, exigindo o desenvolvimento e a manutenção de uma infraestrutura robusta para lidar com questões como monitoramento, gerenciamento de configuração, descoberta de serviços e comunicação entre os microsserviços. Garantir a consistência dos dados entre os microsserviços também se tornou uma tarefa crítica, exigindo a implementação de padrões e práticas adequadas.

Em resumo, o caso da Netflix demonstra como a migração de uma arquitetura monolítica para uma arquitetura de microsserviços pode trazer inúmeros benefícios para uma empresa em rápido crescimento. A flexibilidade, a escalabilidade e a agilidade obtidas por meio da modularidade dos microsserviços permitiram à Netflix continuar inovando, melhorando a experiência do usuário e se adaptando às demandas do mercado em constante evolução. No entanto, a migração também exigiu investimento em infraestrutura e processos de gerenciamento para lidar com a complexidade inerente à arquitetura de microsserviços.

5.2 AMAZON PRIME VIDEO: MIGRAÇÃO PARCIAL DE MICROSSERVIÇOS PARA MONOLITO

O Amazon Prime Video também é um dos maiores serviços de streaming do mundo, atendendo a milhões de clientes anualmente. Para fornecer uma experiência de visualização perfeita para seus usuários, a empresa criou uma ferramenta para monitorar todos os fluxos visualizados pelos clientes, que analisa a qualidade dos fluxos de áudio e vídeo em tempo real, construído inicialmente na infraestrutura de microsserviços.

Com aumento significativo do número de fluxos na plataforma, esse infra enfrentou desafios significativos em relação à escalabilidade e aos custos operacionais em sua arquitetura de microsserviços distribuídos. Diante disso, a empresa tomou a decisão de migrar para um aplicativo monolítico em seu serviço de monitoramento de áudio/vídeo, buscando obter maior eficiência e redução de custos (KOLNY, 2023).

Um dos principais problemas identificados na arquitetura de microsserviços distribuídos era a complexidade e o alto custo da orquestração dos componentes. O uso do AWS Step Functions (serviço de orquestração sem servidor) para a orquestração do serviço resultava em transações de estado frequentes, atingindo rapidamente os limites de dimensionamento e gerando custos operacionais significativos.

Além disso, a transferência de dados entre os componentes distribuídos, especialmente em relação aos quadros de vídeo, envolvia chamadas de alto nível para o serviço de armazenamento, aumentando ainda mais os custos operacionais. Esses desafios impactavam negativamente a escalabilidade do serviço e tornavam sua operação financeiramente inviável em larga escala.

Diante desses problemas, a equipe responsável pelo serviço de monitoramento de áudio/vídeo do Amazon Prime Video decidiu migrar para um aplicativo monolítico. Essa transição envolveu a consolidação de todos os componentes do serviço em um único processo, eliminando a necessidade de orquestração distribuída e simplificando a transferência de dados.

Com a migração para o monolito, o Amazon Prime Video obteve melhorias significativas em termos de escalabilidade e redução de custos operacionais. A consolidação dos componentes em um único processo permitiu uma melhor utilização dos recursos de computação, reduzindo a necessidade de redundância e minimizando o desperdício de recursos (KOLNY, 2023). Além disso, a simplificação da lógica de orquestração resultou em maior eficiência operacional e redução dos custos associados às transações de estado.

A migração de microsserviços para um aplicativo monolítico no serviço de monitoramento de áudio/vídeo do Amazon Prime Video demonstra a busca contínua por soluções arquiteturais que melhor atendam às necessidades de escalabilidade, eficiência e custo operacional das empresas de streaming. Essa transição reflete a importância de avaliar cuidadosamente os trade-offs entre diferentes arquiteturas e adotar abordagens que ofereçam maior eficiência e sustentabilidade financeira.

Em conclusão, a migração bem-sucedida do Amazon Prime Video para um aplicativo monolítico em seu serviço de monitoramento de áudio/vídeo permitiu à empresa superar os desafios de escalabilidade e custos operacionais enfrentados na arquitetura de microsserviços distribuídos. Essa transição ilustra a importância de adaptar as arquiteturas de software às necessidades específicas de cada serviço, buscando sempre melhorias na eficiência, escalabilidade e custo operacional.

6 CONCLUSÕES

Com base na análise realizada neste estudo, podemos chegar a várias conclusões importantes sobre a arquitetura tradicional e a arquitetura de microsserviços.

Não existe uma abordagem única que seja adequada para todos os casos, tanto a arquitetura tradicional quanto a arquitetura de microsserviços têm suas vantagens e desvantagens. A escolha entre elas dependerá das necessidades específicas do sistema, dos requisitos do negócio e das restrições técnicas.

A arquitetura tradicional é mais adequada para sistemas de menor complexidade. Se o sistema não requer escalabilidade extrema, flexibilidade ou entrega contínua, a arquitetura tradicional pode ser uma opção viável. Ela oferece simplicidade de desenvolvimento, implantação e manutenção, sendo mais adequada para projetos menores com requisitos estáveis.

Já arquitetura de microsserviços é indicada para sistemas complexos e escaláveis. Se o sistema precisa lidar com alta escalabilidade, flexibilidade e entrega contínua, a arquitetura de microsserviços pode ser a melhor escolha. Ela permite o desenvolvimento, implantação e dimensionamento independentes de cada serviço, facilitando a adaptação a mudanças e acomodando um maior volume de tráfego. O dimensionamento adequado dos microsserviços é essencial, embora essa abordagem ofereça escalabilidade, é importante dimensionar corretamente cada serviço para evitar sobrecarga e ineficiências. Um planejamento cuidadoso e uma estratégia eficaz de gerenciamento de recursos são fundamentais para garantir um desempenho ótimo do sistema.

A gestão da complexidade é um desafio em ambas as abordagens. Tanto a arquitetura tradicional quanto a de microsserviços exigem um cuidadoso gerenciamento da complexidade. Na arquitetura tradicional, a complexidade pode aumentar à medida que o sistema cresce, tornando a manutenção mais difícil. Já na arquitetura de microsserviços, a complexidade está relacionada à coordenação e comunicação entre os serviços.

A escolha da arquitetura deve ser orientada pelos requisitos do negócio. A decisão sobre a arquitetura a ser adotada deve ser baseada nos objetivos e requisitos do negócio. É fundamental entender as necessidades do sistema, as expectativas dos usuários, a escalabilidade esperada e a capacidade de manutenção a longo prazo.

Em resumo, não há uma resposta definitiva sobre qual abordagem arquitetural é a melhor. Cada projeto requer uma avaliação cuidadosa das necessidades e requisitos específicos, levando em consideração os trade-offs envolvidos. É essencial considerar fatores

como escalabilidade, flexibilidade, complexidade e custo para tomar uma decisão informada sobre a arquitetura mais adequada.

É importante ressaltar que o campo da arquitetura de software está em constante evolução, e novas abordagens e tecnologias continuam surgindo. Portanto, é recomendável acompanhar as tendências e melhores práticas da indústria para tomar decisões informadas e atualizadas sobre arquitetura de software.

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice.** 3rd ed. London: Addison-Wesley, 2012. ISBN: 978-0-321-81573-6.

BOGNER, J. *et al*. Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review. Empirical Software Engineering. **Empirical Software Engineering**, Springer, v. 26, n. 104, p. 1-39, 22 july 2021.

FUNDAÇÃO WIKIMEDIA. Modelo cliente–servidor. **Wikipedia**, São Petersburgo, Flórida, EUA, 2022. Disponível em: https://pt.wikipedia.org/wiki/Modelo_cliente %E2%80%93servidor. Acesso em: 05 set. 2023.

GARLAN, D. **Software architecture**. Pittsburgh, PA, USA: Carnegie Mellon University, 2008.

KANELLOPOULOS, Y. *et al*. Code quality evaluation methodology using the ISO/IEC 9126 standard. **International Journal of Software Engineering & Applications (IJSEA)**, v.1, n.3, July 2010 Disponível em: https://www.airccse.org/journal/ijsea/papers/0710ijsea2.pdf . Acesso em: 05 set. 2023.

KOLNY, M. Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. **Prime video**, 22 mar. 2023. Disponível em: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90/. Acesso em: 10 jul. 2023.

LEWIS, J.; FOWLER, M. Microservice Premium. **Martin Fowler**. 25 mar. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 15 jul. 2023.

NEWMAN, S. **Building microservices**: designing fine-grained systems. Sebastopol, Califórnia, USA: O'Reilly Media, Inc., 2015.

NEWMAN, S. **Monolith to microservices:** evolutionary patterns to transform your monolith. Sebastopol, Califórnia, USA: O'Reilly Media, 2019. ISBN 9781492047810.

RICHARDS, M.; FORD, N.; SHOUP, D. **Fundamentals of software architecture**: an engineering approach. Sebastopol, Califórnia, USA: O'Reilly Media, 2020. ISBN: 9781492043423.

SAKOVICH, N. **Microservices vs. Monolithic Architecture Comparison**. 2017. Disponível em: https://www.sam-solutions.com/blog/microservices-vs-monolithic-real-business-examples/. Acesso em: 11 jul. 2023.