CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

JOÃO VITOR DE OLIVEIRA MENDONÇA

MICROSSERVIÇOS COMO SOLUÇÃO ARQUITETURAL PARA O DESENVOLVIMENTO DE BACK-END

Guarulhos

2024

JOÃO VITOR DE OLIVEIRA MENDONÇA

MICROSSERVIÇOS COMO SOLUÇÃO ARQUITETURAL PARA O DESENVOLVIMENTO DE BACK-END

Trabalho de Graduação do Curso de Análise e Desenvolvimento de Sistemas, apresentado como requisito parcial para obtenção do Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Jadir Custódio Mendonça Junior

Guarulhos

2024

JOÃO VITOR DE OLIVEIRA MENDONÇA

MICROSSERVIÇOS COMO SOLUÇÃO ARQUITETURAL PARA O DESENVOLVIMENTO DE BACK-END

Trabalho de Graduação apresentado ao Curso de Análise e Desenvolvimento de Sistemas como requisito parcial para obtenção do **Título de Tecnólogo em Análise e Desenvolvimento de Sistemas.**

Banca Examinadora

Orientador:____

Prof. Me. Jadir Custódio Mendonça Junior

Fatec Guarulhos

Banca:____

Titulação + nome completo

Fatec Guarulhos

Banca:____

Titulação + nome completo

Fatec Guarulhos

Guarulhos, 21/11/2024

RESUMO

MENDONCA, João Vitor de Oliveira. **Microsserviços como solução arquitetural para o desenvolvimento de back-end**. 2024. 58 p. Trabalho de Graduação – Faculdade de Tecnologia de Guarulhos, Guarulhos.

A arquitetura de microsserviços tem se destacado como uma abordagem moderna para o desenvolvimento de sistemas, permitindo que equipes de desenvolvimento trabalhem de maneira simultânea em diferentes partes de uma aplicação. Isso resulta em entregas mais rápidas e seguras para os clientes. Embora apresente desafios, como a complexidade na definição dos microsserviços e a comunicação entre eles, essa abordagem oferece vantagens significativas, especialmente quando comparada a arquiteturas mais tradicionais.

Aplicações modernas exigem constante atualização e evolução, e a escolha de uma arquitetura adequada impacta diretamente a agilidade dessas modificações. Optar por uma arquitetura de microsserviços facilita a implementação de novas funcionalidades e a sua entrega contínua ao cliente. Tecnologias como Kafka para mensageria, Spring Boot para a construção de APIs, MySQL como banco de dados, e Java como linguagem de programação, permitem que diferentes partes do sistema sejam desenvolvidas, escaladas e mantidas de forma independente.

A transição para uma arquitetura de microsserviços deve ser planejada cuidadosamente, identificando partes do código que possam ser extraídas sem causar grandes impactos. Iniciar com componentes menos dependentes reduz a probabilidade de erros durante o processo. Além disso, a comunicação entre microsserviços pode ser otimizada com ferramentas como Kafka, que possibilitam a troca de mensagens de forma eficiente e segura.

A adoção de microsserviços também permite uma maior flexibilidade tecnológica. Além disso, o escalonamento independente de cada serviço aumenta a robustez do sistema, permitindo que uma falha em um serviço específico não comprometa o funcionamento de toda a aplicação.

Palavras-chave: Arquitetura de Microsserviços, Spring Boot, Kafka, MySQL, API, Java, Padrões de Microsserviços.

ABSTRACT

Microservices architecture has stood out as a modern approach to systems development, allowing development teams to work simultaneously on different parts of an application. This results in faster and safer deliveries for customers. Although it presents challenges, such as the complexity in defining microservices and communicating between them, this approach offers significant advantages, especially when compared to more traditional architectures.

Modern applications require constant updating and evolution, and choosing an appropriate architecture directly impacts the agility of these modifications. Choosing a microservices architecture facilitates the implementation of new functionalities and their continuous delivery to the customer. Technologies such as Kafka for messaging, Spring Boot for building APIs, MySQL as a database, and Java as a programming language, allow different parts of the system to be developed, scaled and maintained independently.

The transition to a microservices architecture must be carefully planned, identifying parts of the code that can be extracted without causing major impacts. Starting with less dependent components reduces the likelihood of errors during the process. Furthermore, communication between microservices can be optimized with tools like Kafka, which enable the exchange of messages efficiently and securely.

The adoption of microservices also allows for greater technological flexibility. Furthermore, the independent scaling of each service increases the robustness of the system, allowing a failure in a specific service not to compromise the functioning of the entire application.

Keywords: *Microservices Architecture, Spring Boot, Kafka, MySQL, API, Java, Microservices Patterns.*

LISTA DE FIGURAS

Figura 1 – Arquitetura monolítica. 15
Figura 2 - Monolítico num único processso. 16
Figura 3 - Monolítico Modular. 16
Figura 4 – Escalonamento de uma Arquitetura Monolítica. 18
Figura 5 - Arquitetura de Microsserviços. 20
Figura 6: Escalonamento de uma arquitetura monolítica 27
Figura 7: Arquitetura de microsserviços sem escalonamento. 28
Figura 8: Diagrama de Caso de Uso. 35
Figura 9: Modelo de dados. 36
Figura 10: Definição da base de dados. 37
Figura 11: Repositório de agendamento. 38
Figura 12: Repositório de agendamento. 39
Figura 13: Função para criação de um novo agendamento. 40
Figura 14: Containers no Docker 42
Figura 15: Página do GitHub. 43
Figura 16: Endpoints da página Agenda. 48
Figura 17: Endpoints da página Dashboards 48
Figura 18: Endpoints da página Estoque. 49
Figura 19: Endpoints da página Login. 49
Figura 20: Endpoints da página Paciente. 50
Figura 21: Endpoints da Página Serviços. 51
Figura 22- Endpoints da Página Usuários. 51

SUMÁRIO

1. INTRODUÇÃO ... 9
1.1. Objetivos ... 10
1.1.1. Gerais ... 10
1.1.2. Específicos ... 10
1.2. Justificativa ... 11
1.3. Metodologia ... 12
2. ESTADO DA ARTE ... 14
2.1. Arquitetura Monolítica ... 14
2.1.1. Tipos de monolítico ... 15
2.1.2. Vantagens ... 17
2.1.3. Desvantagens ... 18
2.1.4. Decisão arquitetural ... 19
2.2. Arquitetura de Microsserviços ... 20
2.2.1. Definição de um Microsserviço ... 20
2.2.2. Características de um microsserviço ... 22
2.2.3. Acoplamento e Coesão ... 23
2.2.4. Comunicação entre microsserviços ... 23
2.2.5. Vantagens sobre uma arquitetura monolítica ... 26
2.2.6. Desvantagens ... 30
2.2.7. Desafios na Adoção de Microsserviços ... 30
3. DESENVOLVIMENTO DA APLICAÇÃO COM MICROSSERVIÇOS ... 32
3.1. Migração Incremental ... 32
3.2. Definição da Plataforma ... 33
3.3. Usuários ... 34
3.3.1. Usuários Diretos ... 34
3.3.2. Requisitos ... 34
3.3.3. Tecnologia ... 34
3.3.4. Modelo de Domínio ... 35
3.3.5. Diagramação ... 35
3.3.6. Modelo de Dados ... 35
3.3.7. Desenvolvimento do Backend ... 36
3.3.8. Método de desenvolvimento ... 36
3.3.9. Conexão à base de dados ... 37

3.3.10. Acesso à base de dados 37
3.3.11. Funcionamento de um agendamento 38
4. DESENVOLVIMENTO BACKEND 41
4.1. Docker 41
4.2. GitHub e GitLab: Ferramentas Fundamentais para Desenvolvimento Colaborativo 42
4.3. Apache Kafka 44
4.4. MySQL 44
4.5. Modelagem do Software 44
4.5.1. Requisitos Funcionais 44
4.5.2. Requisitos Não Funcionais 45
4.6. Codificação 45
5. RESULTADOS 47
5.1. Agenda 47
5.2. Dashboards 48
5.3. Estoque 49
5.4. Login 49
5.5. Paciente 50
5.6. Serviços 50
5.7. Usuários 51
6. CONCLUSÃO E TRABALHO FUTURO 52
6.1. Conclusão 52
6.2. Trabalho Futuro 53
REFERÊNCIAS BIBLIOGRÁFICAS 56

1. INTRODUÇÃO

O avanço da tecnologia e a crescente demanda por sistemas que possam ser modificados e escalados rapidamente impulsionaram o desenvolvimento de novas arquiteturas de software. Nesse contexto, a arquitetura de microsserviços tem se destacado como uma alternativa eficiente em relação às arquiteturas monolíticas tradicionais. Diferente do monolito, em que toda a aplicação é desenvolvida e executada como uma única unidade, os microsserviços dividem a aplicação em pequenos serviços independentes, que podem ser desenvolvidos, implantados e escalados separadamente [Newman, 2015].

Uma das principais vantagens dessa abordagem é a possibilidade de dividir o trabalho entre diferentes equipes, permitindo o desenvolvimento simultâneo de novos recursos, o que acelera a entrega de funcionalidades ao cliente. Essa característica é especialmente relevante em ambientes de desenvolvimento ágeis, onde a entrega contínua é essencial para manter a competitividade do produto [Fowler, 2019]. No entanto, a adoção de microsserviços também traz desafios significativos, como a necessidade de coordenação entre serviços, a definição clara de responsabilidades e a garantia de uma comunicação eficiente entre eles.

Para mitigar esses desafios, diversas tecnologias têm sido amplamente adotadas em sistemas baseados em microsserviços. O Spring Boot, por exemplo, facilita a criação de APIs e a integração com outras ferramentas e serviços, sendo amplamente utilizado em sistemas de back-end escritos em Java [Walls, 2021]. Em conjunto, bancos de dados relacionais como o MySQL são escolhidos para armazenar dados de forma estruturada, enquanto ferramentas de mensageria como o Kafka permitem a comunicação assíncrona entre microsserviços, garantindo escalabilidade e resiliência [Kreps, 2014].

Essa flexibilidade tecnológica é outro benefício fundamental da arquitetura de microsserviços. A independência entre serviços permite que diferentes tecnologias sejam usadas conforme as necessidades específicas de cada módulo, otimizando o desempenho e a eficiência do sistema como um todo [Dragoni et al., 2017]. Além disso, o escalonamento independente de cada microsserviço possibilita maior

robustez, uma vez que a falha de um serviço não afeta o funcionamento dos demais, aumentando a resiliência da aplicação.

Contudo, para que a migração ou o desenvolvimento inicial em microsserviços seja bem-sucedido, é crucial que haja uma estratégia bem definida desde o início. A identificação de quais partes do sistema devem ser transformadas em microsserviços deve ser cuidadosa, partindo de componentes que possuam menos dependências e, portanto, sejam mais fáceis de extrair e escalar separadamente [Thönes, 2015]. Uma análise prévia da estrutura do código por meio de diagramas e ferramentas de visualização é recomendada, pois oferece uma visão clara da complexidade da migração e ajuda a minimizar riscos.

Portanto, este estudo visa explorar os benefícios e desafios da implementação de uma arquitetura de microsserviços, utilizando tecnologias amplamente aceitas como Spring Boot, MySQL, Kafka e Java. A proposta é apresentar como essas ferramentas podem ser integradas para formar uma base sólida de desenvolvimento e garantir uma transição eficiente e bem-sucedida para microsserviços.

1.1. Objetivos

1.1.1. Gerais

Desenvolver uma arquitetura de microsserviços para uma aplicação de back-end utilizando as tecnologias mais presentes no mercado, a fim de garantir escalabilidade, flexibilidade e eficiência no desenvolvimento e na manutenção do sistema.

1.1.2. Específicos

- Definir a arquitetura de microsserviços para o projeto, identificando as funcionalidades que serão transformadas em serviços independentes.
- Implementar a comunicação assíncrona entre microsserviços utilizando Kafka para garantir a troca de mensagens de maneira eficiente e resiliente.
- Criar APIs RESTful com Spring Boot para a integração entre os microsserviços e o front-end ou outros sistemas externos.
- Utilizar o banco de dados relacional MySQL para armazenar os dados dos serviços, garantindo integridade e consistência das informações.

- Avaliar a performance e a escalabilidade da aplicação, aplicando técnicas de monitoramento para identificar e resolver possíveis gargalos.
- Desenvolver uma estratégia de teste e monitoramento contínuo, garantindo que os microsserviços possam ser atualizados e implantados sem interrupções no funcionamento global do sistema.

1.2. Justificativa

A evolução das necessidades de sistemas modernos, que exigem rápida resposta a mudanças, alta disponibilidade e escalabilidade, torna a adoção da arquitetura de microsserviços uma escolha estratégica para muitos projetos. Diferente das arquiteturas monolíticas, os microsserviços oferecem flexibilidade e independência entre os componentes da aplicação, permitindo que diferentes equipes trabalhem em paralelo e que cada serviço seja escalado individualmente conforme a demanda.

Esta abordagem é especialmente relevante em um cenário em que a rápida entrega de novas funcionalidades se tornou um diferencial competitivo. Utilizando Spring Boot e Java para a criação de APIs RESTful, a arquitetura de microsserviços não apenas facilita a comunicação entre os componentes do sistema, mas também melhora a integração com sistemas externos. Além disso, a utilização de Kafka como solução de mensageria possibilita uma comunicação assíncrona e eficiente, essencial para garantir que os serviços funcionem de maneira desacoplada e escalável. O MySQL fornece uma solução robusta e confiável para o gerenciamento de dados, garantindo a integridade das informações.

Além disso, o desenvolvimento e manutenção de sistemas monolíticos apresentam desafios relacionados à complexidade do código, dificuldades em escalar e integrar novas tecnologias e dependências, e o risco de falhas que podem afetar toda a aplicação. A adoção de uma arquitetura de microsserviços endereça esses problemas, permitindo atualizações e manutenção contínua sem a necessidade de interromper todo o sistema. A possibilidade de escalar independentemente cada serviço, bem como a maior resiliência do sistema em caso de falhas, faz com que essa arquitetura seja altamente recomendada para projetos que visam crescimento sustentável e inovação constante.

Portanto, a escolha por desenvolver um sistema utilizando a arquitetura de microsserviços justifica-se pela necessidade de entregar um produto de alta qualidade, que seja escalável, resiliente e adaptável às mudanças constantes do mercado, ao mesmo tempo em que utiliza tecnologias de ponta, para garantir um desenvolvimento ágil e eficiente.

1.3. Metodologia

A metodologia de pesquisa, conforme descrito por Pinheiro (2010, p. 33), é "[...] o conjunto de técnicas e processos empregados pela ciência para formular e resolver problemas relacionados à obtenção objetiva do conhecimento de maneira sistemática". O autor destaca a relevância do método científico, pois sua aplicação possibilita organizar as etapas necessárias para conduzir uma investigação de forma estruturada.

A pesquisa desenvolvida neste trabalho é classificada, quanto à sua natureza, como pesquisa aplicada. Isso porque o guia elaborado, juntamente com a arquitetura de referência, pode ser implementado em projetos onde foram identificadas demandas por uma arquitetura de microsserviços.

Em relação aos objetivos, trata-se de uma pesquisa descritiva, uma vez que foram analisadas as principais características de uma arquitetura de microsserviços, além de estabelecer conexões entre essas características.

As etapas principais do desenvolvimento foram organizadas conforme segue:

1. **Levantamento de Requisitos**

Foi realizado um levantamento inicial junto aos stakeholders do sistema, incluindo profissionais de saúde e administradores de clínicas odontológicas, para identificar as necessidades funcionais e não funcionais do software. Este levantamento resultou na definição dos requisitos apresentados na seção de modelagem.

2. **Modelagem do Sistema**

A partir dos requisitos, foram elaborados diagramas de caso de uso, entidade-relacionamento e a definição de uma arquitetura inicial. A modelagem focou em segmentar as funcionalidades principais em serviços independentes, de acordo com os princípios de microsserviços.

3. **Implementação**

A implementação seguiu as boas práticas de desenvolvimento em Java, utilizando o framework Spring Boot para criação de APIs RESTful. O banco de dados MySQL foi configurado para garantir a persistência de dados, enquanto o Apache Kafka foi integrado para gerenciar a comunicação assíncrona entre os microsserviços.

4. **Testes Unitários e de Integração**

Foram desenvolvidos testes unitários para validar o funcionamento de cada serviço de forma isolada, utilizando frameworks como JUnit e Mockito. Testes de integração foram realizados para verificar a interação entre os microsserviços e a comunicação via Kafka.

5. **Implantação e Monitoramento**

O sistema foi containerizado utilizando Docker, garantindo portabilidade e facilidade de implantação em diferentes ambientes. A orquestração foi realizada com Kubernetes, permitindo escalabilidade e monitoramento contínuo da aplicação.

Ferramentas e Tecnologias

As ferramentas e tecnologias utilizadas incluem:

- **Spring Boot**: Para criação de APIs RESTful e integração com o banco de dados.
- **MySQL**: Banco de dados relacional para armazenar informações críticas do sistema.
- **Apache Kafka**: Para comunicação assíncrona entre os microsserviços.
- **Docker e Kubernetes**: Para gerenciamento de contêineres e escalabilidade da aplicação.
- **GitHub**: Controle de versão e colaboração no desenvolvimento.

Limitações e Desafios

Durante o desenvolvimento, alguns desafios foram enfrentados, como a configuração inicial do Apache Kafka e a definição precisa dos limites de cada microsserviço. A adoção de boas práticas de design e a realização de testes extensivos ajudaram a mitigar esses desafios.

2. ESTADO DA ARTE

Neste capítulo, será explorado o estudo comparativo entre as arquiteturas monolítica e de microsserviços. A análise incluirá uma comparação das duas abordagens, discutindo suas respectivas vantagens e desvantagens, bem como os critérios para escolher entre uma e outra.

2.1. Arquitetura Monolítica

A arquitetura monolítica refere-se a um modelo onde um produto de software é projetado para funcionar como uma única unidade, operando em um único processo. Nesse tipo de arquitetura, os componentes estão interconectados e dependem uns dos outros, resultando em um código mais acoplado e exigindo que toda a aplicação seja instalada como um todo [Newman, 2021].

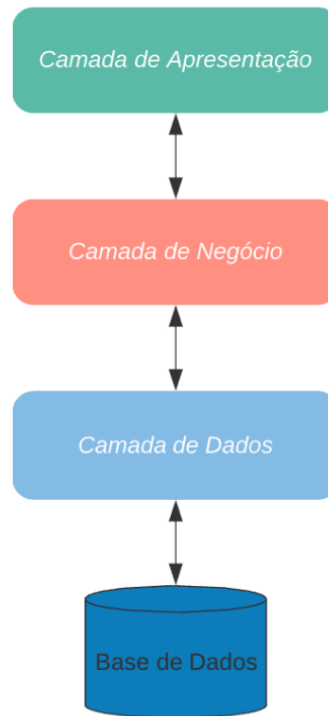
Em geral, uma aplicação monolítica é composta por três camadas principais: a camada de apresentação, a camada de negócio e a camada de dados. A Figura 1 ilustra um diagrama representativo dessa arquitetura, mostrando a presença dessas três camadas. Todo o código, componentes e lógica formam uma única entidade [Javed, 2019].

Esse modelo é frequentemente adotado devido à sua familiaridade e simplicidade. A organização das equipes de desenvolvimento geralmente segue as competências específicas dos programadores, como administração de banco de dados, backend ou frontend. Portanto, o desenvolvimento é estruturado de acordo com a organização da equipe [Conway, 1968].

Na arquitetura monolítica, é comum que haja uma única base de dados atendendo a toda a aplicação [Javed, 2019]. A base de dados pode ser replicada por motivos de desempenho e tolerância a falhas, com o objetivo de prevenir a perda de dados e melhorar o tempo de acesso. As aplicações com essa arquitetura costumam ter um código extenso, altamente acoplado e, por vezes, pouco modular.

É possível executar várias instâncias de uma aplicação monolítica para aumentar a robustez e escalabilidade, embora a aplicação continue funcionando como uma única unidade [Newman, 2021].

Figura 1 – Arquitetura monolítica.



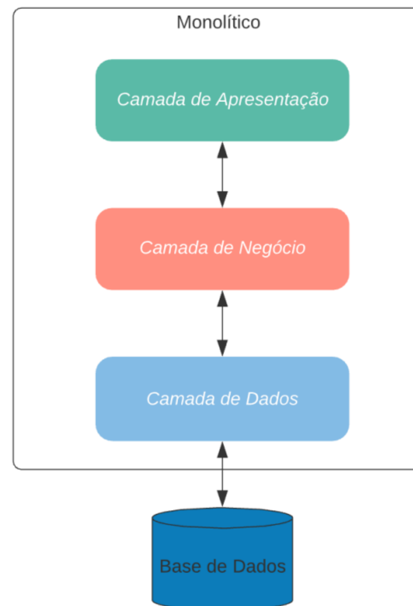
Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

2.1.1. Tipos de monolítico

Monolítico em um Único Processo

O exemplo mais popular dessa abordagem é um sistema onde todo o código opera dentro de um único processo. É possível ter múltiplas instâncias para maior robustez e resiliência, mas o código permanece em um único processo [Newman, 2021].

Figura 2 - Monolítico num único processso.

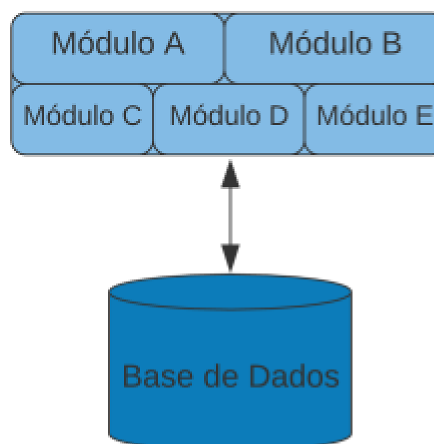


Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

Monolítico Modular

Essa variação é semelhante ao monolítico em um único processo, mas organiza o sistema em módulos independentes, que ainda precisam estar juntos para a instalação da aplicação [Newman, 2021].

Figura 3 - Monolítico Modular.



Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

A modularização permite trabalho paralelo em diferentes módulos por várias equipes, sem lidar com as complexidades dos sistemas distribuídos. Contudo, a base

de dados pode não ter a mesma modularidade, complicando a separação do monolítico [Newman, 2021].

Monolítico Distribuído

Um sistema composto por vários serviços que devem ser instalados juntos, mesmo que não possam ser instalados de forma independente [Newman, 2021].

2.1.2. Vantagens

A arquitetura monolítica possui vantagens como as apresentadas:

- **Desenvolvimento Facilitado**

  A arquitetura monolítica simplifica o desenvolvimento inicial, pois não há necessidade de gerenciar a comunicação entre componentes através de rede [Richardson, 2018].

- **Resolução de Problemas Simplificada**

  Problemas em funcionalidades ou na aplicação podem ser diagnosticados e resolvidos mais facilmente devido à centralização [Richardson, 2018].

- **Testes Simplificados**

  Testar a aplicação e a integração entre seus componentes é mais direto quando tudo está em um único artefato [Richardson, 2018].

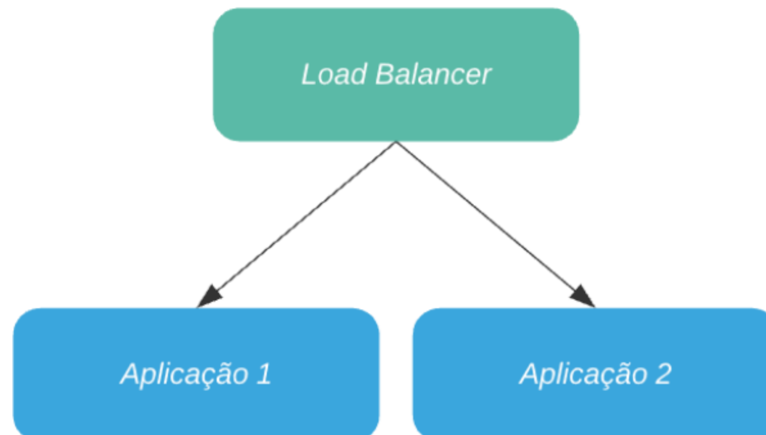
- **Instalação Direta**

  A aplicação é um único artefato que pode ser facilmente copiado e executado em um servidor, sem a necessidade de configurar múltiplos componentes [Richardson, 2018].

- **Escalabilidade Simples**

  O escalonamento horizontal é direto, com várias máquinas rodando a mesma aplicação por trás de um balanceador de carga [Richardson, 2018].

Figura 4 – Escalonamento de uma Arquitetura Monolítica.



Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

- **Desempenho**

A comunicação entre componentes no mesmo processo é mais eficiente do que a comunicação via rede, reduzindo latências e problemas de inconsistência [Newman, 2021].

2.1.3. Desvantagens

Apesar de vantagens, uma aplicação monolítica possui também desvantagens, como as apresentadas:

- **Código Extenso e Complexo**

O código monolítico pode ser extenso e difícil de compreender, o que pode complicar a manutenção e a integração de novos desenvolvedores [Richardson, 2018].

- **Compilação e Instalação Requerem Recompilação Completa**

Alterações em um componente requerem recompilação e reinstalação da aplicação inteira [Richardson, 2018].

- **Dificuldade para Equipes Múltiplas**

O desenvolvimento simultâneo por várias equipes pode levar a conflitos e interferências no código [Newman, 2021].

- **Falhas em Componentes Afetam a Aplicação Toda**

Problemas em um componente podem impactar toda a aplicação, mesmo em um ambiente escalado [Javed, 2019].

- **Complexidade na Adoção de Novas Tecnologias**

  Integrar novas linguagens ou frameworks pode exigir uma reescrita significativa do código [Newman, 2021].
- **Escalabilidade Limitada**

A escalabilidade deve abranger a aplicação inteira, não permitindo escalonamento individual de componentes [Richardson, 2018].

2.1.4. Decisão arquitetural

Optar por uma arquitetura monolítica no desenvolvimento de software pode simplificar e agilizar as etapas iniciais do projeto. Ao não haver a necessidade de segmentar os microsserviços, o desenvolvimento do produto ou MVP (produto mínimo viável) tende a ser mais rápido [Fowler, 2019].

A escolha por uma arquitetura monolítica é recomendada nas seguintes situações:

- **Equipe de desenvolvimento reduzida.**

Quando uma equipe pequena se dedica a uma aplicação baseada em microsserviços, grande parte do tempo e dos recursos pode ser consumida pelas complexidades dessa abordagem. O modelo monolítico, nesse caso, oferece uma alternativa mais rápida e eficiente.

- **Aplicações de baixa complexidade.**

Para projetos simples, com poucas funcionalidades, uma arquitetura monolítica é mais fácil de compreender e implementar [Fowler, 2019].

- **Falta de experiência com microsserviços.**

Iniciar o desenvolvimento em uma arquitetura de microsserviços sem o devido conhecimento pode tornar o processo caro e desgastante. A gestão dos serviços, suas interações, e a resolução de conflitos se tornam mais complicadas devido à natureza distribuída dos microsserviços [Fowler, 2019].

- **Necessidade de instalação rápida.**

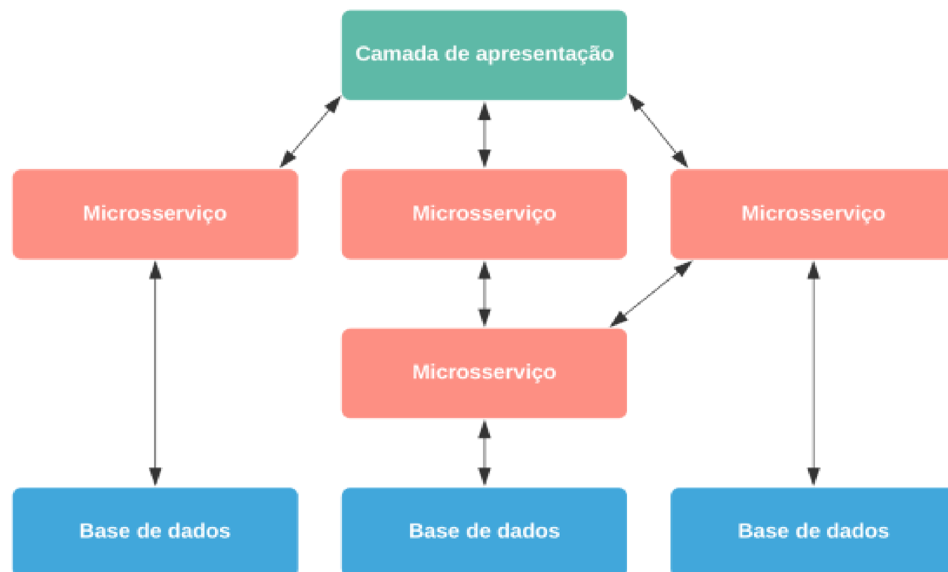
Aplicações monolíticas são mais fáceis de implantar, pois tudo está contido em um único artefato, que pode ser colocado em um servidor e executado com rapidez.

2.2. Arquitetura de Microsserviços

A arquitetura de microsserviços é uma abordagem para o desenvolvimento de software que se caracteriza por dividir o sistema em múltiplos microsserviços autônomos. Esses microsserviços operam de forma independente, mas cooperam e se comunicam entre si para garantir o funcionamento do produto de software [Fowler e Lewis, 2014].

Diferentemente da arquitetura monolítica mostrada na Figura 3 da seção 2.1.1, a Figura 5 ilustra a separação da camada de negócios em vários microsserviços independentes, que se comunicam entre si e mantêm suas próprias bases de dados.

Figura 5 - Arquitetura de Microsserviços.



Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

2.2.1. Definição de um Microsserviço

Na arquitetura de microsserviços, cada microsserviço é uma unidade de software autônoma, que pode ser substituída ou atualizada sem impactar os outros serviços [Newman, 2021].

Cada microsserviço opera em seu próprio processo, encapsulando funcionalidades e comunicando-se com outros microsserviços pela rede [Newman, 2021]. Conforme mostrado na Figura 5, cada microsserviço gerencia seus próprios dados e, se precisar de informações de outro serviço, faz uma solicitação a ele. Externamente, um microsserviço é tratado como uma "caixa preta", onde outros

serviços não acessam diretamente sua base de dados. No entanto, alguns microsserviços podem não precisar de uma base de dados própria [Newman, 2021].

Por exemplo, um microsserviço pode gerenciar o inventário, outro o processamento de pedidos, e outro o envio das encomendas, trabalhando juntos para formar uma aplicação completa. A definição clara e as fronteiras bem estabelecidas dos microsserviços, que permanecem inalteradas mesmo com mudanças internas, resultam em um sistema de baixo acoplamento e alta coesão [Newman, 2021], como será discutido na seção 2.2.3.

Agregados

Uma das maiores dificuldades ao desenvolver uma aplicação baseada em microsserviços é determinar o que deve constituir um microsserviço, quais funcionalidades ele deve abranger, e onde estabelecer suas fronteiras.

Eric Evans introduziu o conceito de "agregado" em seu livro *Domain-Driven Design* [Evans, 2003], definindo-o como um conjunto de objetos de domínio que podem ser tratados como uma unidade única. Esses agregados representam domínios reais, como um pedido e os produtos relacionados a ele. Apesar de serem objetos distintos, pode ser vantajoso tratá-los como um único agregado [Fowler, 2013].

Normalmente, um agregado possui um ciclo de vida, permitindo que seja implementado como uma máquina de estados. A ideia é tratar agregados como unidades independentes, com todo o código relacionado ao estado e às transações de estado agrupado em um só lugar. Os agregados podem ter relacionamentos com outros agregados [Newman, 2021].

Mapeando esse conceito para a arquitetura de microsserviços, um único microsserviço pode gerenciar o ciclo de vida e o armazenamento de dados de um ou mais agregados. Se outro microsserviço precisar modificar o estado de um agregado, deve fazer uma solicitação ao serviço responsável por ele, garantindo que as mudanças de estado ocorram de forma controlada e válida [Newman, 2021].

2.2.2. Características de um microsserviço

Instalação Independente

Cada microsserviço deve ser capaz de ser modificado e implementado sem causar impacto nos outros. Isso elimina a necessidade de ajustar os outros serviços quando um é alterado, reduzindo o número de implantações necessárias. Para que isso seja possível, é fundamental que os microsserviços sejam fracamente acoplados, permitindo mudanças sem exigir alterações em outros serviços [Newman, 2019].

Modularidade em Torno de um Domínio

Fazer alterações ou adicionar funcionalidades que envolvem vários serviços é uma tarefa complexa e onerosa. Quando uma funcionalidade requer mudanças em mais de um microsserviço, é necessário compilar e implementar todos eles, dobrando o trabalho em comparação com a alteração de um único serviço. Desenvolver uma funcionalidade que envolve múltiplos microsserviços demanda coordenação entre as equipes e um planejamento cuidadoso da ordem de implementação, resultando em um esforço maior do que o necessário para realizar a mudança em apenas um serviço [Newman, 2021].

Propriedade dos Dados

Cada microsserviço deve ter sua própria base de dados, sem compartilhar bases de dados com outros serviços. Se um microsserviço precisar de informações de outro, ele deve solicitar os dados diretamente ao serviço responsável, em vez de acessar sua base de dados [Newman, 2021]. Isso permite que cada microsserviço controle quais dados são compartilhados e quais permanecem ocultos, diminuindo o acoplamento entre os serviços.

Independência Tecnológica

Cada microsserviço pode ser implementado usando a tecnologia mais adequada para seu caso específico, permitindo a escolha da melhor ferramenta para cada situação [Newman, 2021].

Ocultação de Informações

Os microsserviços permitem ocultar o máximo de informações possível dentro do serviço, expondo apenas o mínimo necessário. Essa prática assegura uma separação clara entre o que pode ser alterado facilmente e o que deve ser manuseado com cuidado. Modificações nas implementações ocultas podem ser feitas sem

preocupações, pois não impactam outros serviços que não têm acesso a esses detalhes [Newman, 2021].

2.2.3. Acoplamento e Coesão

Ao desenvolver microsserviços, é crucial considerar o acoplamento e a coesão ao definir as fronteiras de cada serviço [Newman, 2021]. O acoplamento refere-se ao grau de dependência entre sistemas; quanto mais acoplados estiverem os microsserviços, mais mudanças serão necessárias quando uma modificação for feita [Newman, 2021].

A coesão, por outro lado, implica que o código que precisa ser alterado em uma modificação deve estar agrupado em um único serviço [Newman, 2021]. O objetivo dos microsserviços é que, ao realizar uma alteração, apenas um serviço precise ser modificado. Isso é alcançado com microsserviços altamente coesos e fracamente acoplados [Newman, 2021].

Microsserviços que são pouco coesos e fortemente acoplados tornam o processo de mudança mais caro, pois é necessário modificar e implementar serviços em processos separados. Isso aumenta o esforço de implantação [Newman, 2021]

O ideal em uma arquitetura de microsserviços é ter serviços coesos e estáveis, que possam ser implantados de forma independente. Para isso, cada microsserviço deve fornecer uma interface estável, que não mude constantemente [Newman, 2021].

2.2.4. Comunicação entre microsserviços

James Lewis e Martin Fowler introduziram o conceito “smart endpoints and dumb pipes” [Fowler e Lewis, 2014], em tradução livre: "endpoints inteligentes e comunicação simples", sugerindo que a complexidade da comunicação entre microsserviços deve ser minimizada. A lógica de tratamento das mensagens deve estar contida dentro de cada serviço, enquanto a comunicação entre eles deve ser o mais simples possível.

REST

Os microsserviços podem se comunicar usando diferentes protocolos, entre os quais o mais comum é o REST (Representational State Transfer) [Fielding e Taylor, 2000]. REST é um estilo arquitetural que define um conjunto de padrões aplicados

aos componentes e dados de um sistema, permitindo a disponibilização de funcionalidades de forma independente [Doyle,2021].

No REST, todos os componentes têm uma interface uniforme com operações como GET, PUT, POST, DELETE, que possibilitam a interação entre sistemas de maneira independente, sem necessidade de estabelecer protocolos específicos [Doyle, 2021]. O REST é sem estado, o que significa que um sistema não precisa saber nada sobre o outro que faz a solicitação, e vice-versa [Fielding e Taylor, 2000].

Cada sistema que utiliza REST é composto por recursos, que representam qualquer objeto ou informação armazenada e que pode ser acessada por outros. Os recursos têm identificadores únicos e são acessados por meio da interface REST fornecida pelo sistema. Normalmente, são retornadas representações dos recursos em formatos como JSON (JavaScript Object Notation), que é popular por sua simplicidade e independência de linguagem.

Uma desvantagem do REST é que sua comunicação é síncrona, o que significa que o sistema que faz a solicitação precisa aguardar a resposta, o que pode ser problemático em caso de falha no sistema invocado ou lentidão na rede [Doyle, 2021].

Message Brokers

Um message broker é um middleware que facilita a comunicação e troca de informações entre aplicações. Ele recebe mensagens de um remetente e as transforma em um formato compreensível para o destinatário, permitindo que as duas partes possam ser desenvolvidas em linguagens e tecnologias diferentes. O message broker atua como um intermediário, validando, armazenando, roteando e entregando mensagens ao destino apropriado [IBM, 2020].

Uma das principais vantagens dos message brokers é a comunicação assíncrona, que permite ao sistema remetente enviar uma mensagem sem precisar aguardar uma resposta imediata. Isso garante que o sistema remetente não fique bloqueado e aumenta o desacoplamento entre sistemas. O remetente não precisa conhecer a localização ou o status do destinatário, apenas o message broker, que gerencia todo o processo de envio e recebimento de mensagens. Isso também permite a substituição do broker por uma nova versão sem impactar os sistemas conectados [IBM, 2020].

Os message brokers geralmente utilizam message queues para armazenar e ordenar mensagens até que os destinatários estejam prontos para processá-las. Existem diferentes modelos de message brokers, como:

- **Ponto a Ponto:** Utilizado quando há uma relação um-para-um entre o remetente e o destinatário. Cada mensagem na fila é enviada para um único destinatário e consumida uma vez.
- **Publicação/Subscrição:** Neste modelo, um remetente publica mensagens em um tópico, e múltiplos destinatários podem se inscrever nesse tópico para consumir as mensagens. Isso cria uma relação um-para-muitos.

Comunicação baseada em Eventos

Outro padrão arquitetural para comunicação entre aplicações é a comunicação baseada em eventos, onde as aplicações atuam como produtoras ou consumidoras de eventos, ou até mesmo desempenham ambos os papéis [IBM, 2020].

Um evento é o registro de algo que aconteceu ou de uma mudança de estado. Eventos são imutáveis, não podem ser modificados ou eliminados, e são ordenados por ordem de criação [Jansen e Saladas, 2020]. Quando uma aplicação executa uma ação ou ocorre uma mudança de estado, um evento é gerado e comunicado para outras aplicações interessadas, que estão inscritas para consumir esses eventos e processá-los posteriormente.

Esse padrão promove maior desacoplamento entre as aplicações, já que elas podem se comunicar de forma assíncrona, sem precisar ter conhecimento umas das outras.

Existem duas formas principais de transmissão de eventos:

- **Event Messaging (Publicação/Subscrição):** Semelhante ao modelo de publicação/subscrição mencionado anteriormente, onde os eventos são publicados em tópicos e consumidos por destinatários que se inscrevem nesses tópicos.
- **Event Streaming:** Neste modelo, os eventos são colocados em uma stream, onde permanecem mesmo após serem consumidos. Os consumidores podem se inscrever em streams a qualquer momento e consumir eventos, incluindo aqueles que ocorreram antes da inscrição [IBM (2020)]. A vantagem deste

modelo é que os eventos não são eliminados após o consumo, permitindo maior flexibilidade e histórico de eventos para os consumidores.

Ambos os modelos oferecem benefícios significativos em termos de flexibilidade e desacoplamento, contribuindo para sistemas mais escaláveis e resilientes.

2.2.5. Vantagens sobre uma arquitetura monolítica

A arquitetura de microsserviços oferece diversas vantagens em comparação com a arquitetura monolítica, muitas das quais são ampliadas devido à natureza dos microsserviços e suas fronteiras bem definidas. Essas vantagens são principalmente derivadas de conceitos de sistemas distribuídos, combinados com ocultação de informação e domain-driven design, tornando os microsserviços mais eficazes do que sistemas distribuídos tradicionais [Newman, 2021].

- **Resiliência**

Em uma aplicação monolítica, a falha de um único componente pode causar a falha de toda a aplicação. Para mitigar esse problema, uma solução comum é escalar a aplicação horizontalmente, adicionando mais máquinas para que, em caso de falha da máquina primária, outra possa assumir. No entanto, se a falha for causada por um erro em um componente específico, a aplicação continuará falhando, mesmo com múltiplas máquinas disponíveis.

Por outro lado, em uma arquitetura de microsserviços, cada serviço é uma unidade independente. A falha de um microsserviço não compromete toda a aplicação; apenas o serviço específico que falhou ficará temporariamente indisponível. Isso facilita a correção de erros, pois o impacto é limitado a um único serviço, que pode ser recompilado e reinstalado sem afetar o restante da aplicação [Newman, 2015].

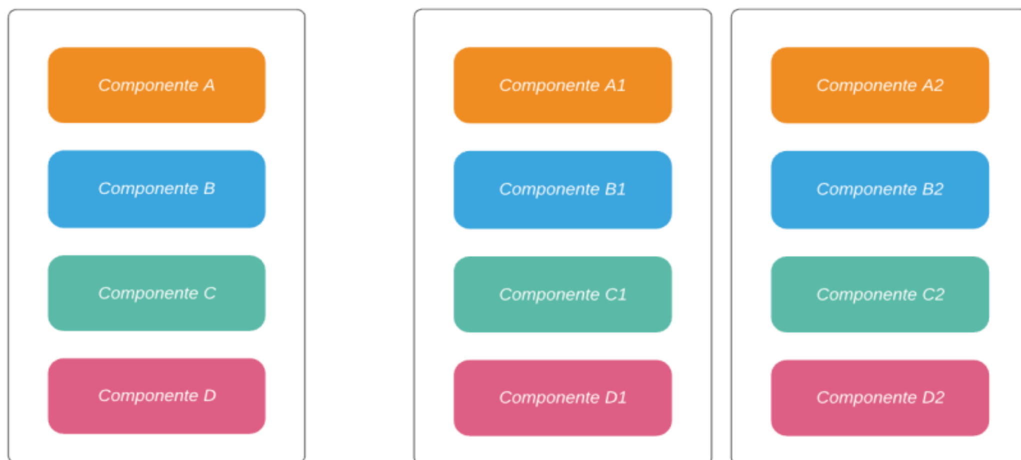
No entanto, a adoção de microsserviços por si só não garante maior robustez. Para alcançar uma aplicação mais resiliente, é necessário implementar mecanismos que minimizem a falha dos microsserviços ou a perda de mensagens. Isso pode incluir o uso de balanceadores de carga e a adoção de protocolos de comunicação que

assegurem que mensagens não sejam perdidas caso algum serviço falhe [Newman, 2021].

- **Escalonamento**

Em uma arquitetura monolítica, o escalonamento é realizado na aplicação como um todo. Por exemplo, se uma aplicação monolítica for composta por quatro componentes, todos esses componentes precisam ser escalados juntos, mesmo que apenas um deles esteja enfrentando alta demanda (como ilustrado na Figura 6).

Figura 6: Escalonamento de uma arquitetura monolítica

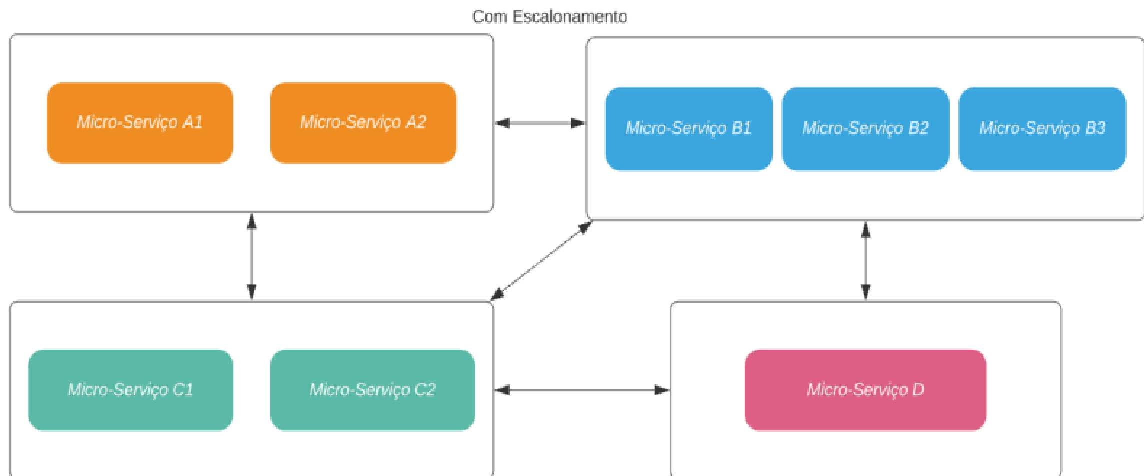


Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

(a) Sem escalonamento (b) Com escalonamento

Em contrapartida, em uma arquitetura de microsserviços, é possível escalar cada serviço de forma independente. Por exemplo, se uma aplicação composta por quatro microsserviços tem um serviço que está sob alta carga (como o microsserviço B na Figura 7), apenas esse serviço pode ser escalado para atender à demanda, sem a necessidade de escalonar os outros serviços que podem estar operando normalmente. Isso permite um uso mais eficiente dos recursos, ajustando o escalonamento conforme as necessidades específicas de cada serviço [Bruce e Pereira, 2018].

Figura 7: Arquitetura de microsserviços sem escalonamento.



Fonte: Construção de uma plataforma de e-commerce: Uma abordagem baseada numa arquitetura de microsserviços (2021)

Em uma arquitetura monolítica, qualquer mudança, mesmo que seja apenas uma linha de código, requer que toda a aplicação seja recompilada e reinstalada. Isso resulta em um processo de instalação que pode ser demorado e de alto risco, pois qualquer erro que passe despercebido pode exigir um retrabalho considerável para correção ou reversão da mudança, levando a maior tempo de inatividade [Richardson, 2018].

Por outro lado, na arquitetura de microsserviços, uma alteração em um serviço específico não exige a reinstalação de toda a aplicação. É possível atualizar apenas o microsserviço relevante, minimizando o impacto nas demais partes do sistema. Isso resulta em uma instalação mais rápida e permite que novas funcionalidades sejam entregues ao cliente com maior agilidade [Newman, 2015]. Se ocorrer um problema na nova instalação, ele estará limitado ao microsserviço afetado, facilitando a reversão e a mitigação do problema.

- **Substituível**

Em uma arquitetura monolítica, toda a funcionalidade da aplicação está concentrada em um único código base, o que torna a modificação de partes específicas do sistema um processo complexo e arriscado [Richardson, 2018].

Na arquitetura de microsserviços, cada serviço é uma unidade independente, o que facilita a substituição ou atualização de um microsserviço por uma nova

implementação. O impacto dessas mudanças é reduzido, uma vez que os outros serviços não são afetados diretamente [Newman, 2015].

- **Adoção de Novas Tecnologias**

Uma das principais vantagens dos microsserviços é a flexibilidade na escolha de tecnologias. Cada serviço pode ser desenvolvido com a tecnologia que melhor se adapta às suas necessidades específicas, permitindo que o sistema como um todo tire proveito do melhor desempenho possível para cada funcionalidade [Nadareishvili, 2016].

Em contraste, uma aplicação monolítica exige que todos os componentes sejam desenvolvidos na mesma tecnologia, limitando as possibilidades de otimização [Richardson, 2018].

- **Estrutura da Equipe**

A independência dos microsserviços permite que diferentes equipes trabalhem simultaneamente no desenvolvimento de uma aplicação, sem a necessidade de coordenação intensa entre elas. Cada equipe pode se concentrar em um único serviço, o que aumenta a autonomia e agilidade, além de facilitar a comunicação interna devido ao tamanho reduzido das equipes [Richardson, 2018].

Equipes menores, focadas em propósitos específicos, são mais eficientes e podem entregar resultados com maior rapidez e menos obstáculos, em comparação com equipes que trabalham em um código monolítico e altamente acoplado [Newman, 2021].

Entregas mais rápidas

A arquitetura de microsserviços facilita a implementação e entrega contínua de novas funcionalidades. Como os serviços são independentes, é mais fácil realizar modificações e reimplantar apenas os serviços afetados, sem a necessidade de reiniciar toda a aplicação [Newman, 2015]. Isso resulta em um ciclo de desenvolvimento mais ágil, permitindo que os clientes recebam novas funcionalidades mais rapidamente [Newman, 2021].

2.2.6. Desvantagens

Apesar das vantagens, a arquitetura de microsserviços também apresenta desafios significativos, que devem ser cuidadosamente gerenciados para garantir o sucesso da aplicação.

- **Complexidade**

A arquitetura de microsserviços é, por natureza, mais complexa do que uma monolítica. Como um sistema distribuído, composto por múltiplos serviços que se comunicam via rede, há desafios adicionais relacionados a latência, confiabilidade de mensagens, e gestão de múltiplas bases de dados. Esse nível de complexidade exige uma abordagem mais rigorosa para garantir a robustez e o desempenho do sistema [Richardson, 2018].

- **Dificuldade de Testar**

Testar uma aplicação composta por vários microsserviços pode ser complicado, uma vez que a funcionalidade pode depender da interação entre vários serviços. Isso torna o processo de testes mais difícil e pode exigir um ambiente de teste robusto e bem integrado para garantir a cobertura completa [Newman, 2021].

- **Experiência de Desenvolvimento**

À medida que o número de microsserviços aumenta, a experiência de desenvolvimento pode se deteriorar. Ambientes de desenvolvimento como a JVM têm limites sobre quantos processos podem ser executados simultaneamente em uma única máquina. Com um número elevado de microsserviços, o desenvolvimento local pode se tornar inviável, forçando os desenvolvedores a recorrerem à nuvem para testar suas aplicações, o que pode reduzir a agilidade [Newman, 2021].

2.2.7. Desafios na Adoção de Microsserviços

Além das desvantagens, a adoção de uma arquitetura de microsserviços traz desafios únicos que não são encontrados em uma arquitetura monolítica.

- **Definição de Microsserviços**

Um dos desafios iniciais é determinar quais microsserviços devem ser criados, suas responsabilidades, e como eles devem interagir. A definição correta das fronteiras de um microsserviço é crucial para evitar complicações futuras, e isso pode

ser uma tarefa complexa mesmo com a utilização de conceitos como Bounded Context [Bruce e Pereira, 2018].

- **Comunicação entre Microsserviços**

A comunicação entre microsserviços, que ocorre via rede, não é instantânea e pode ser sujeita a falhas ou latências variáveis. Isso exige um planejamento cuidadoso para garantir que os serviços se comportem conforme esperado, mesmo diante de falhas de comunicação [Newman, 2021].

- **Falhas de Microsserviços**

Um microsserviço pode falhar, levando à indisponibilidade de parte das funcionalidades da aplicação. Para mitigar esses riscos, é necessário implementar mecanismos de tolerância a falhas, como escalonamento horizontal e balanceamento de carga, garantindo que o serviço continue disponível [Newman, 2021].

- **Escalonamento Dinâmico**

O escalonamento de microsserviços em resposta a flutuações na demanda pode ser complexo. Garantir que novas instâncias sejam efetivamente utilizadas e que a carga seja distribuída adequadamente pelo balanceador de carga requer uma configuração meticulosa [Newman, 2021].

- **Visibilidade e Monitoramento**

Com múltiplos microsserviços distribuídos, identificar a origem de problemas na aplicação pode ser mais difícil. A necessidade de monitoramento centralizado e logs unificados é essencial para a detecção rápida de falhas e para garantir o desempenho contínuo da aplicação [Bruce e Pereira, 2018].

- **Microsserviço Muito Utilizado**

Quando um microsserviço é amplamente utilizado por outros, sua indisponibilidade pode comprometer a aplicação como um todo. É crucial implementar mecanismos de tolerância a falhas e escalonamento adequado para evitar que um serviço crítico se torne um ponto único de falha [Bruce e Pereira, 2018].

Concluindo, a arquitetura de microsserviços oferece várias vantagens em termos de escalabilidade, resiliência e agilidade no desenvolvimento e entrega de

software. No entanto, essas vantagens vêm acompanhadas de desafios significativos que exigem um planejamento cuidadoso e a adoção de práticas automatizadas para instalação, teste e monitoramento, a fim de garantir que o sistema funcione conforme o esperado e que as vantagens superem as complexidades introduzidas.

3. DESENVOLVIMENTO DA APLICAÇÃO COM MICROSSERVIÇOS

Durante o desenvolvimento de uma aplicação, é comum que ela não permaneça imutável. Mesmo após a entrega ao cliente, são necessárias mudanças e adaptações contínuas [Newman, 2015]. É praticamente impossível antecipar todas as funcionalidades e requisitos de uma aplicação desde o início. Por isso, é essencial projetá-la de forma que possa ser modificada com o mínimo de impacto possível.

Conforme uma aplicação monolítica cresce, ela incorpora novas funcionalidades e cria cada vez mais dependências entre seus componentes, resultando em um código mais acoplado. Em uma arquitetura monolítica, qualquer alteração, mesmo que pequena, pode ter um impacto significativo, exigindo a reinstalação de toda a aplicação [Fowler e Lewis, 2014]. Por outro lado, a adoção de uma arquitetura de microsserviços pode facilitar a implementação de mudanças e adição de novas funcionalidades, além de permitir entregas mais ágeis ao cliente. No entanto, a transição de uma arquitetura monolítica para microsserviços apresenta desafios, como a definição clara dos limites de cada microsserviço e a adaptação da comunicação entre eles [Newman, 2015].

Optar por uma arquitetura monolítica deve ser uma escolha deliberada, baseada na necessidade de alcançar algo que o sistema arquitetural atual não permite [Newman, 2021].

3.1. Migração Incremental

"If you do a big-bang rewrite, the only thing you’re guaranteed of is a big bang."

(Martin Fowler)

A migração para microsserviços não deve ser feita de maneira abrupta. É mais prudente realizar essa transição de forma incremental, extraindo funcionalidades uma de cada vez. Esse processo gradual permite que a equipe aprenda mais sobre microsserviços à medida que avança, além de minimizar os riscos de erros significativos [Newman, 2021].

Transformar uma aplicação monolítica em uma arquitetura de microsserviços pode ser um processo caro, especialmente se o sistema estiver altamente acoplado. Tentar fazer todas as mudanças de uma vez torna difícil identificar se a migração está sendo executada corretamente. Em vez disso, é muito mais eficaz dividir essa transformação em etapas menores, cada uma focada em uma única tarefa. Erros são inevitáveis, mas uma abordagem incremental permite reduzir a frequência e o impacto desses erros [Newman, 2021].

É recomendável começar a migração isolando um ou dois componentes da aplicação em microsserviços, implementá-los e, em seguida, avaliar o sucesso dessa mudança e seu impacto na aplicação como um todo [Newman, 2021].

Enquanto modificar e mover código é relativamente simples, graças às diversas ferramentas e editores que facilitam esse trabalho, alterar uma base de dados é muito mais complicado, e reverter essas mudanças é ainda mais difícil. O mesmo vale para a reescrita de uma API que é consumida por vários outros microsserviços, pois todos os serviços dependentes precisarão ser atualizados para acompanhar as mudanças [Newman, 2021].

Uma boa estratégia para iniciar uma migração é desenhar a arquitetura desejada para a aplicação. Isso ajuda a visualizar os microsserviços que serão necessários e as interações entre eles. É importante testar alguns exemplos de funcionalidades, como o registro de usuários, para identificar possíveis dependências cíclicas. Se essas dependências forem encontradas, é um sinal de que algo precisa ser ajustado para eliminar esses ciclos. Além disso, se dois microsserviços estiverem se comunicando intensamente, pode ser um indício de que eles deveriam ser combinados em um único serviço [Newman, 2021].

3.2. Definição da Plataforma

O objetivo deste projeto de dissertação é o desenvolvimento de uma aplicação para gestão de consultórios odontológicos, que seja escalável, segura e responsiva. A plataforma será destinada a dentistas e equipes administrativas, fornecendo uma interface centralizada para gerenciar agendamentos, prontuários, e informações dos pacientes. A solução permitirá uma gestão eficiente e integrada de consultórios odontológicos, abrangendo desde o cadastro e manutenção de registros de pacientes até a gestão de procedimentos odontológicos e do estoque.

3.3. Usuários

Nesta seção descrevem-se os utilizadores do sistema, que possuem interesse ou desempenham algum papel na plataforma

3.3.1. Usuários Diretos

- **Recepcionistas/Funcionários:** Usuários responsáveis pelo atendimento ao cliente, agendamento de consultas, e gestão de informações administrativas no consultório. Eles têm acesso ao sistema para realizar tarefas como registrar pacientes, organizar a agenda dos dentistas e gerenciar o fluxo de informações do consultório.
- **Paciente:** Usuário registrado com a finalidade de agendar consultas e acessar informações sobre seu histórico odontológico.
- **Dentista:** Profissional que se registra na plataforma para gerenciar seus pacientes, consultas e tratamentos.
- **Administrador:** Responsável pela gestão da aplicação, incluindo a adição ou remoção de usuários, aprovação de relatórios, e manutenção da base de dados.

3.3.2. Requisitos

Os requisitos foram organizados de acordo com os diferentes tipos de usuários da aplicação. Esses requisitos englobam tanto funcionalidades essenciais quanto aspectos não funcionais, como segurança e desempenho.

3.3.3. Tecnologia

A aplicação foi desenvolvida em Java, utilizando o Spring Boot para o backend, o que possibilita um desenvolvimento ágil e a integração eficiente com a base de dados, além da criação de APIs RESTful.

Para o banco de dados, optou-se pelo MySQL, devido à sua robustez e capacidade de lidar com grandes volumes de dados de forma eficiente. A integração com o banco de dados é feita através do Spring Data JPA, simplificando as operações de persistência.

Para o processo de compilação e deploy, utilizou-se Gradle, Docker e Kubernetes. Docker é utilizado para a criação de contêineres, permitindo a implantação isolada de componentes da aplicação. Kubernetes, por sua vez, gerencia

a orquestração dos contêineres, facilitando a escalabilidade e a gestão do ciclo de vida da aplicação.

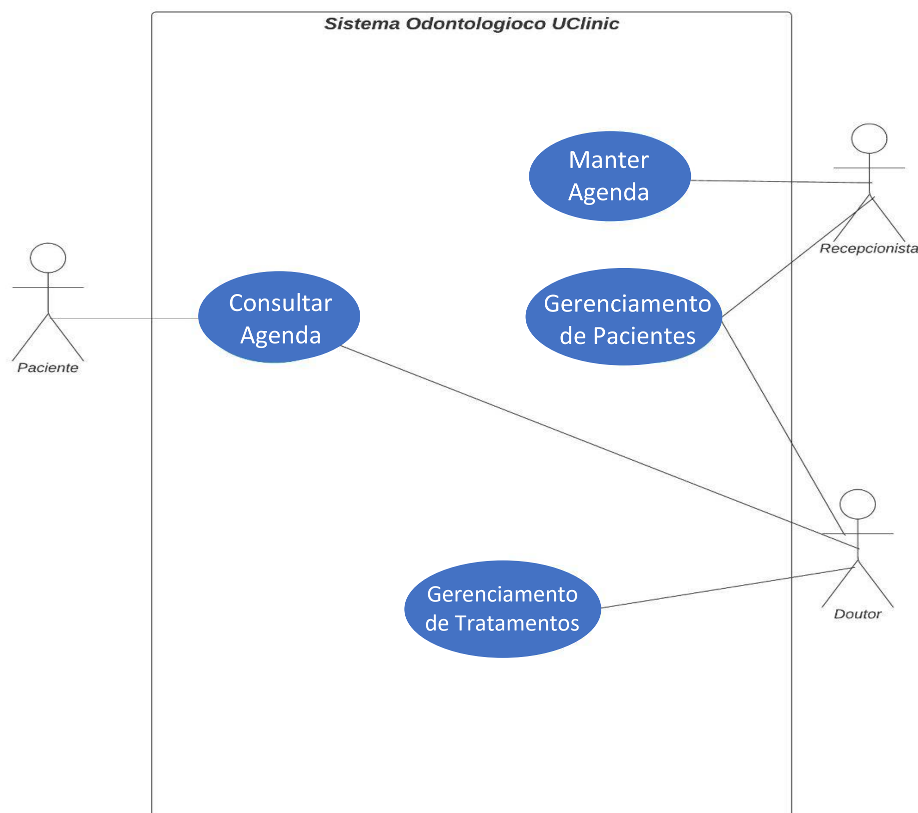
3.3.4. Modelo de Domínio

O modelo de domínio da aplicação foi projetado para refletir as necessidades específicas do gerenciamento odontológico, como o cadastro de pacientes, agendamento de consultas, registro de tratamentos e emissão de relatórios financeiros. A transição para uma arquitetura de microsserviços será planejada de modo a segmentar o sistema em componentes autônomos, cada um responsável por uma funcionalidade específica, como gestão de pacientes, agendamento e estoque.

3.3.5. Diagramação

Com os requisitos levantados e especificados, desenvolveu-se um Diagrama de Caso de Uso apresentado na Figura 8. Neste, representam-se os utilizadores da aplicação e as funcionalidades de cada um.

Figura 8: Diagrama de Caso de Uso.

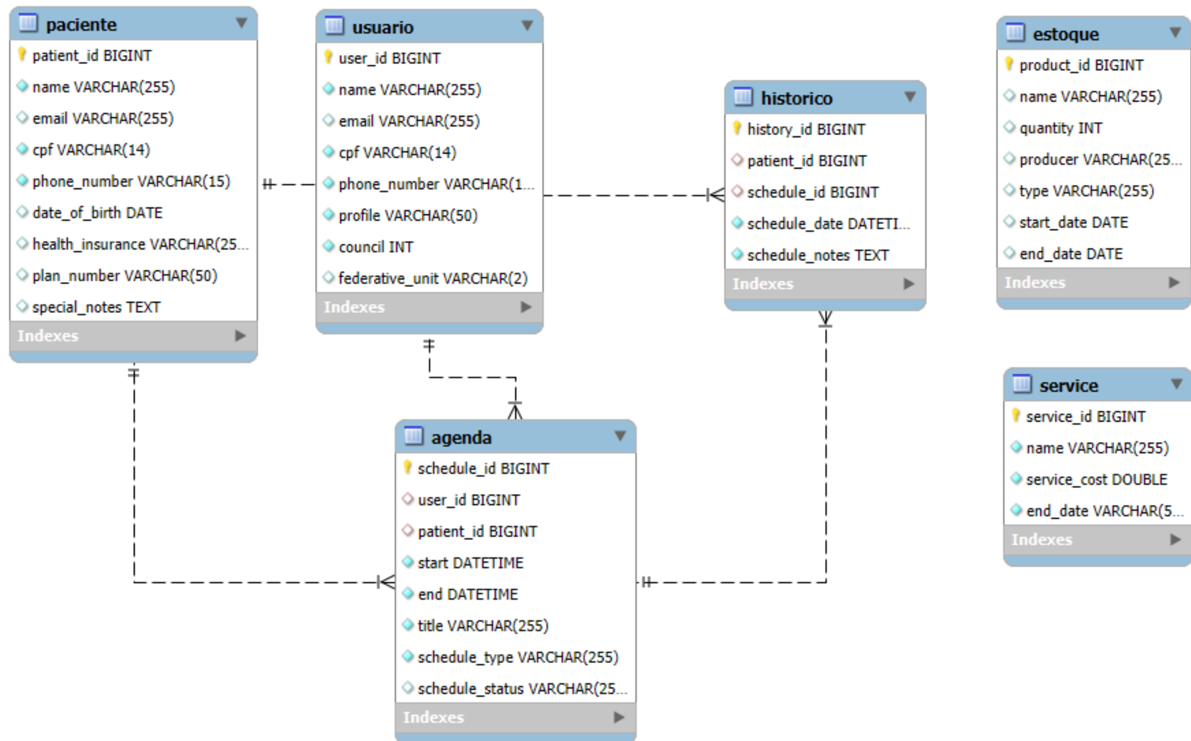


Fonte: Autor.

3.3.6. Modelo de Dados

O modelo de dados referente à aplicação encontra-se apresentado na Figura 9.

Figura 9: Modelo de dados.



Fonte: Autor.

3.3.7. Desenvolvimento do Backend

O objetivo deste projeto é desenvolver uma aplicação para gestão de consultórios odontológicos, iniciando com uma arquitetura baseada em microsserviços. Isso inclui a identificação de componentes que podem ser facilmente isolados e transformados em serviços independentes, bem como a adoção de práticas que promovam a modularidade e a escalabilidade do sistema.

3.3.8. Método de desenvolvimento

O desenvolvimento do backend foi conduzido com base nos princípios da metodologia Agile. Esta abordagem iterativa para a gestão de projetos e desenvolvimento de software permite a entrega rápida de valor aos clientes. Em vez de realizar uma implementação única e definitiva, a aplicação é desenvolvida e entregue em pequenas parcelas, de maneira incremental. Isso possibilita uma avaliação contínua dos requisitos do sistema, dos planos e dos resultados, permitindo uma adaptação ágil às mudanças. [Stellman e Greene, 2024]

Especificamente, foi utilizado o framework Scrum, uma implementação que define um conjunto de reuniões, ferramentas e diretrizes que auxiliam as equipes a organizarem e estruturar seu trabalho. Este framework estabelece um processo para

identificar as tarefas necessárias, designar responsabilidades, definir como as atividades serão realizadas e estabelecer prazos para sua conclusão. [Stellman e Greene, 2014].

3.3.9. Conexão à base de dados

O Spring Boot utiliza o arquivo application.properties para configurar a aplicação, permitindo a adição de parâmetros que podem ser acessados diretamente no código. Além disso, este arquivo é utilizado para configurar o acesso ao banco de dados, como mostrado no exemplo da Figura 10. Nele, é possível definir o nome de usuário, a senha, e o URL de conexão com o banco de dados. Embora a senha esteja visível no exemplo, é recomendável utilizar ferramentas que permitam criptografar essas informações sensíveis.

O exemplo também especifica que o banco de dados utilizado é o MySQL, define o driver correspondente e indica que a estratégia de geração das tabelas é update. Isso significa que, caso haja alguma alteração no modelo, ela será refletida automaticamente no esquema do banco de dados. Quando a aplicação está sendo executada em um ambiente local, e o banco de dados está configurado para rodar localmente, o acesso é realizado através do localhost.

Figura 10: Definição da base de dados.

```
spring.datasource.url=jdbc:mysql://localhost:3306/Uclinic?useSSL=false&createDatabaseIfNotExist=true&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.generate-ddl=true
```

Fonte: Autor.

3.3.10. Acesso à base de dados

Como mencionado na seção 3.3.4, foi utilizado o Spring Data JPA para acessar o banco de dados MySQL. Esta ferramenta serve como uma abstração que simplifica significativamente o código necessário para operações de banco de dados. Ao invés de implementar DAOs (Data Access Objects) manualmente, é possível utilizar interfaces que, combinadas com as anotações do Spring Boot, facilitam muito a escrita de código. Isso resulta em várias vantagens, como a redução da quantidade de código a ser gerido e mantido, maior consistência nos acessos aos dados, e configurações simplificadas para o acesso ao banco de dados.

Na Figura 11, vemos um exemplo de repositório, uma interface que funciona como um DAO. Essa interface estende a JpaRepository, o que permite ao Spring Data JPA localizar a interface e gerar automaticamente sua implementação. Esta implementação automática fornece métodos CRUD, e é possível adicionar métodos personalizados, como mostrado na Figura 12, onde duas funções específicas estão definidas.

Figura 11: Repositório de agendamento.

```
@Repository
public interface AgendaRepository extends JpaRepository<Agenda, Long> {
    1 usage   João Mendonça
    @Query("SELECT a FROM Agenda a JOIN a.userId u WHERE u.userId = :userId AND u.profile = 'Profissional da Saúde'")
    List<Agenda> findSchedule(@Param("userId") Long userId);

    1 usage   new *
    @Query("SELECT COUNT(a) FROM Agenda a WHERE a.scheduleStatus = 'Atendido' AND MONTH(a.start) = " +
            "MONTH(CURRENT_DATE) AND YEAR(a.start) = YEAR(CURRENT_DATE)")
    Long countAttendedAppointmentsInCurrentMonth();

    1 usage   new *
    @Query("SELECT COUNT(DISTINCT a.patientId) FROM Agenda a WHERE a.scheduleStatus = 'Atendido' AND MONTH(a.start) " +
            "= MONTH(CURRENT_DATE) AND YEAR(a.start) = YEAR(CURRENT_DATE)")
    Long countDistinctPatientsAttendedInCurrentMonth();

    no usages   João Mendonça
    List<Agenda> findByStartBetween(LocalDateTime start, LocalDateTime end);
    2 usages   new *
    List<Agenda> findByStartBetweenAndScheduleStatus(LocalDateTime start, LocalDateTime end, String scheduleStatus);
    no usages   João Mendonça
    List<Agenda> findAllByStartBetween(LocalDateTime start, LocalDateTime end);
}
```

Fonte: Autor.

3.3.11. Funcionamento de um agendamento

Nesta seção, demonstra-se como funciona um pedido à REST API da plataforma utilizando a framework Spring Boot, com foco na funcionalidade de criação de um agendamento. Para isso, é necessário criar um endpoint que permita à interface chamar essa funcionalidade e realizar a criação de um novo agendamento.

Na Figura 12, é possível observar a definição da classe AgendaController, que utiliza as seguintes anotações:

- **@RestController:** Esta anotação inclui a anotação @Controller, indicando que a classe é um controlador, especificamente um controlador REST.

- **@RequestMapping:** Anotação que define que todos os endpoints dentro deste controlador têm como prefixo a string especificada, neste caso, "/agendamento".
- **@AllArgsConstructor:** Cria um construtor que recebe um argumento para cada campo da classe. Isso é útil para inicializar todos os campos da classe de uma só vez quando um objeto é instanciado.

Com essas anotações, o AgendaController está preparado para gerenciar os pedidos REST relacionados aos agendamentos, facilitando a criação, atualização, consulta e exclusão de agendamentos dentro do sistema.

Figura 12: Repositório de agendamento.

```
@RestController
@AllArgsConstructor
@RequestMapping("/agenda")
public class AgendaController {

    @Autowired
    private AgendaRepository repo;
```

Fonte: Autor.

Com isso, é necessário implementar a função que trata da criação de um novo agendamento na plataforma. O método pode ser observado na Figura 13. Inicialmente, a função obtém o método da solicitação e usa o ObjectMapper para processar o corpo da solicitação JSON. O método então extrai e converte os dados relevantes, como o ID do dentista, o ID do paciente e a data e hora do agendamento.

Primeiramente, o método valida se o dentista está disponível para o horário solicitado. Em seguida, verifica se o paciente está registrado no sistema. Se todas as validações forem bem-sucedidas, a função cria um objeto de agendamento com os dados fornecidos.

O método, então, armazena o novo agendamento na base de dados e atualiza a agenda do dentista. Por fim, retorna uma resposta que confirma a criação do agendamento e inclui uma representação do novo agendamento.

Figura 13: Função para criação de um novo agendamento.

```
@CrossOrigin(origins = "http://localhost:3000")
@PostMapping("/inserir")
public ResponseEntity<String> enviarMensagem(@RequestBody String mensagem, HttpServletRequest request)
        throws JsonProcessingException {

    String method = request.getMethod();
    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode jsonNode = objectMapper.readTree(mensagem);

    JsonNode userId = jsonNode.get("userId");
    JsonNode patientId = jsonNode.get("patientId");
    JsonNode endNode = jsonNode.get("end");

    long user = Long.parseLong(userId.asText());
    long patient = Long.parseLong(patientId.asText());

    String endString = endNode.asText();

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy, HH:mm:ss");
    LocalDateTime localDateTime = LocalDateTime.parse(endString, formatter);
    String formattedDateTime = localDateTime.format(formatter);

    ((ObjectNode) jsonNode).put( fieldName: "userId", user);
    ((ObjectNode) jsonNode).put( fieldName: "patientId", patient);
    ((ObjectNode) jsonNode).put( fieldName: "end", formattedDateTime);

    String mensagemJson = jsonNode.toPrettyString();
    Message mensagemPOST = new Message();
    mensagemPOST.setMethod(method);
    mensagemPOST.setMessage(mensagemJson);
    String mensagemFinal = convertMensagemToJson(mensagemPOST);
    mensagemService.sendMessage(mensagemFinal);
    return ResponseEntity.status(HttpStatus.CREATED).body("Mensagem enviada com sucesso: " + mensagemJson);
}
```

Fonte: Autor.

Neste método, o processo é iniciado pela extração e conversão dos dados do JSON da solicitação. O método válida a disponibilidade do dentista e a existência do paciente antes de criar o agendamento e atualizar a agenda do dentista. Finalmente, retorna uma resposta confirmando a criação do agendamento

4. DESENVOLVIMENTO BACKEND

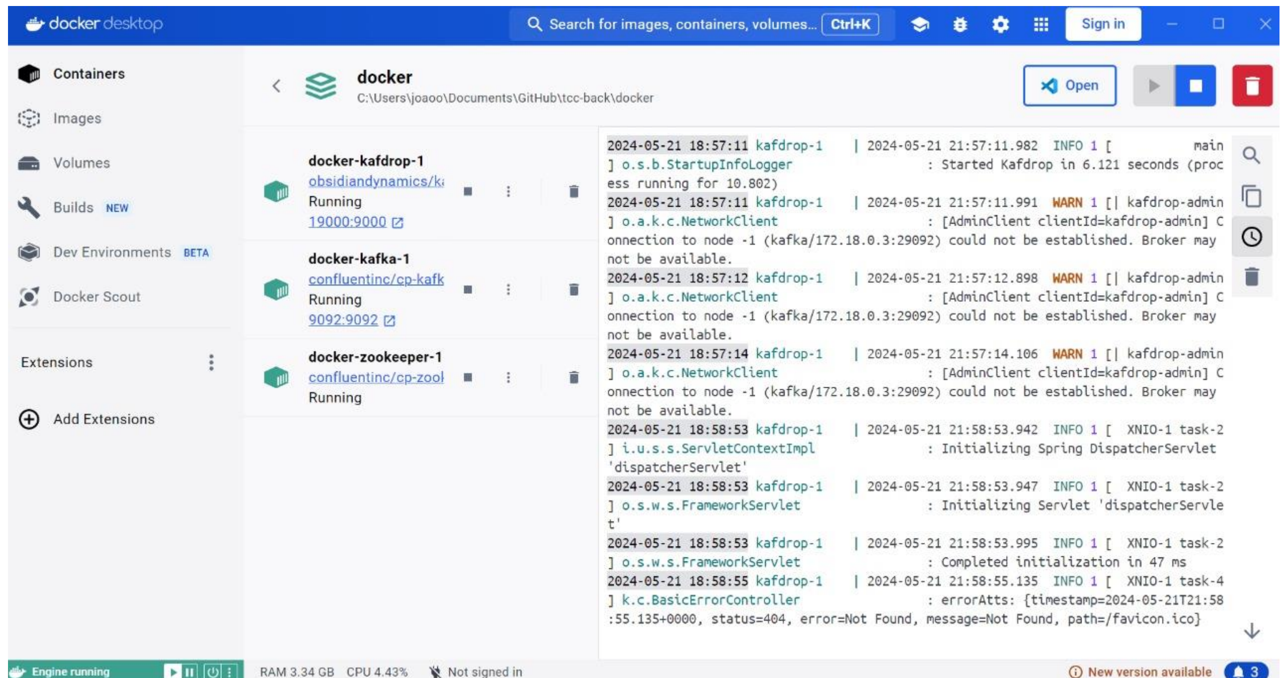
Nesta seção, são discutidas as tecnologias e ferramentas empregadas no desenvolvimento do projeto.

4.1. Docker

A Docker oferece uma abordagem inovadora e eficaz para o empacotamento de software, transformando a forma como os desenvolvedores criam, enviam e executam aplicativos. A tecnologia de contêineres permite que os aplicativos permaneçam fora da infraestrutura subjacente, garantindo portabilidade e consistência em vários ambientes. A criação de imagens específicas para projetos Spring oferece uma maneira fácil e confiável de empacotar e distribuir aplicativos Java. Ao encapsular todas as configurações e dependências do aplicativo em uma imagem Docker, os desenvolvedores podem garantir que os aplicativos funcionem de forma consistente em qualquer ambiente que suporte o Docker, desde o ambiente de desenvolvimento até o ambiente de produção.

Na Figura 14, podemos observar o uso do Docker Desktop para gerenciar contêineres em execução. Entre os contêineres mostrados, temos o docker-kafdrop-1, docker-kafka-1, e docker-zookeeper-1, que são essenciais para o funcionamento de uma arquitetura baseada em Kafka. O uso do Docker neste cenário é fundamental para garantir que todos os componentes do sistema sejam facilmente configurados e executados de maneira consistente, independente do ambiente. Além disso, o Docker Compose ajuda o Docker a funcionar melhor, simplificando a execução e definição de aplicativos compostos por vários contêineres. Os desenvolvedores podem usar o Docker Compose para definir a configuração de seus aplicativos em um arquivo YAML simples e usar um único comando para iniciar todos os contêineres necessários. Isso facilita a gestão de aplicativos complexos que incluem vários serviços interconectados, como bancos de dados, servidores web e microserviços. [Docker Docks, 2024]

Figura 14: Containers no Docker



Fonte: Autor.

4.2. GitHub e GitLab: Ferramentas Fundamentais para Desenvolvimento Colaborativo

Os recursos fundamentais para o desenvolvimento colaborativo de software incluem GitHub e GitLab, que proporcionam um ambiente solido e intuitivo, permitindo às equipes de desenvolvimento trabalharem de maneira conjunta e eficiente. Desenvolvedores podem colaborar transparentemente e acessível em software projects por meio de GitHub, uma plataforma lider baseada em nuvem. Os desenvolvedores garantem a segurança & integridade do trabalho que realizam, além de simplificar a colaboração com colegas de equipe e contribudores externos, ao hospedar o código-fonte em repositórios do GitHub. Com ferramentas abrangentes para simplificar o processo de revisão e integração de código, GitHub promove a qualidade e an eficácia do desenvolvimento de software colaborativo. Esses recursos incluem solicitações de pull, problemas e revisões de código.

De forma semelhante, uma plataforma alternativa chamada GitLab oferece uma variedade de ferramentas e capacidades que apoiam o desenvolvimento colaborativo. O GitLab ajuda as equipes a gerenciarem todo o ciclo de vida do desenvolvimento de software em um ambiente integrado, com foco em DevOps e integração contínua/entrega contínua (CI/CD). A automação de pipelines de implantação e o

controle de versão do código-fonte estão incluídos neste conjunto, fornecendo uma experiência abrangente e eficaz para o desenvolvimento e entrega de software.

As equipes de desenvolvimento podem aproveitar os benefícios do desenvolvimento colaborativo, como maior transparência, produtividade aprimorada e qualidade de código melhorada ao usar o GitHub ou o GitLab. Ao permitir que as equipes trabalhem de forma mais eficaz e fornecer valor aos usuários finais de forma rápida e consistente, essas plataformas são essenciais para modernizar os processos de desenvolvimento de software. Por fim, o GitHub e o GitLab são ferramentas vitais para equipes que desejam aumentar a eficiência e a colaboração em seus projetos de desenvolvimento de software. [GitHub Docs, 2024]

Na Figura 15, a branch principal utilizada é a branch desenvolvimento, que desempenha um papel fundamental na estratégia de desenvolvimento colaborativo. Essa branch serve como o ponto central para a implementação das funcionalidades e da arquitetura do sistema, permitindo uma evolução contínua do software. Através dela, as equipes de desenvolvimento podem integrar novos recursos e ajustes com base nos requisitos específicos e no feedback dos usuários, mantendo a coesão e a qualidade do projeto.

Figura 15: Página do GitHub.

Fonte: Autor.

4.3. Apache Kafka

A computação distribuída e o processamento de eventos em tempo real dependem do Apache Kafka. As aplicações podem lidar com grandes quantidades de dados e fluxos de eventos com eficiência e confiabilidade graças à sua arquitetura robusta e escalável. O processamento de requisições síncronas e assíncronas, como as transações financeiras e pagamentos em tempo real, é uma das principais aplicações. O Kafka também ajuda a integrar sistemas diversos, permitindo a troca de mensagens eficiente entre aplicativos e serviços distribuídos. A escalabilidade e a interoperabilidade são essenciais em ambientes como arquiteturas de microserviços e computação em nuvem. [Creating a Kafka Java Project using Maven (pom.xml), 2024]

4.4. MySQL

Desenvolvido pela Oracle Corporation, o MySQL é o mais popular sistema de gerenciamento de banco de dados SQL. Por conta de suas licenças para uso gratuito, rapidez e capacidade de realizar comandos SQL, a linguagem padrão para manipulação de registros em bancos de dados relacionais, o sistema ganhou bastante popularidade entre acadêmicos e desenvolvedores para gerenciar seus bancos de dados [MILANI, 2006].

4.5. Modelagem do Software

Nesta seção, será apresentada a análise dos requisitos funcionais e não funcionais, juntamente com a criação de diagramas para uma compreensão mais clara das funcionalidades e regras de negócio do software como um todo. Essas atividades proporcionam maior organização e orientação precisa para o desenvolvimento do projeto, garantindo que todas as etapas sejam executadas de forma alinhada aos objetivos e necessidades definidos.

4.5.1. Requisitos Funcionais

Os requisitos funcionais dizem respeito às funcionalidades do software, ou seja, o que ele é capaz de fazer, quais as necessidades que precisam ser atendidas e problemas que precisam ser solucionados. Dentre eles estão:

- **Agendamento de consultas:** o usuário poderá agendar a consulta do paciente, escolhendo o paciente cadastrado, o profissional da saúde que realizará a consulta, a data e hora e tipo de consulta.

- **Cadastro de pacientes:** o usuário antes de realizar o agendamento de consulta terá que cadastrar o paciente no sistema, inserindo informações pessoais importantes e convênio.
- **Cadastro de usuários:** o cadastro de usuários do sistema, os funcionários do consultório, será feito por um usuário administrador. No cadastro será atribuído o perfil de recepcionista ou de profissional de saúde ao usuário.
- **Serviços:** na página de serviços, será possível cadastrar os serviços que o consultório oferece, dando um preço e tempo de duração à ele.
- **Indicadores:** na página de indicadores, será possível ver o desempenho mensal do consultório com informações importantes como valor de entrada, total de atendimento e os serviços mais realizados.
- **Controle de estoque:** será possível fazer um controle de estoque do consultório, cadastrando todos os produtos, sua quantidade e data de validade, se houver.

4.5.2. Requisitos Não Funcionais

Os requisitos não funcionais dizem respeito ao que não é funcionalidade do software, focando na parte de desempenho, usabilidade, atendimento a padrões e afins. Dentre eles estão:

- **Atendimento a padrões:** atender seu objetivo de auxiliar os consultórios odontológicos na sua gestão
- **Confiabilidade:** ter recuperabilidade dos dados em caso de falhas e manter uma baixa ocorrência de erros.
- **Usabilidade:** ser intuitivo e de fácil compreensão, possuindo uma interface agradável e dando uma ótima experiência aos usuários.
- **Eficiência:** manter um alto desempenho usando os recursos disponíveis.
- **Manutenibilidade:** possuir fácil manutenção, para alcançar isso deve ser escrito um código limpo, organizado e padronizado para melhor compreensão futura do código.

4.6. Codificação

Em busca de garantir eficiência, escalabilidade e facilidade de manutenção ao longo do tempo, durante o desenvolvimento do software, foram escolhidas as tecnologias mais utilizadas no mercado atualmente. Para a linguagem de

programação principal, foi optado por Java, uma escolha robusta e amplamente utilizada em grandes aplicações [Eckel, 2003]. Java é conhecido por sua estabilidade, segurança e suporte extensivo, o que é essencial para aplicações que lidam com dados sensíveis, como informações de pacientes.

Para a criação das APIs REST que permitem a comunicação entre os diferentes módulos do sistema, foi utilizado o framework SpringBoot que facilita a configuração e a criação de serviços web, permitindo o desenvolvimento rápido e eficiente de APIs que são cruciais para integrar funcionalidades como agendamento de consultas, gerenciamento de prontuários e controle de estoque de materiais odontológicos.

A persistência de dados foi realizada com o banco de dados MySQL, escolhido por sua confiabilidade e performance. O MySQL oferece uma integração eficiente com Java, possibilitando o armazenamento seguro e rápido de informações críticas, como histórico de consultas, tratamentos realizados e dados financeiros dos pacientes [DuBois, 2008].

Para garantir a portabilidade e a consistência do ambiente de desenvolvimento do software, foi utilizado Docker. Docker permite que a aplicação e suas dependências sejam empacotadas em contêineres, garantindo que o software funcione de maneira consistente em qualquer ambiente, seja no computador do desenvolvedor ou em servidores de produção [Merkel, 2014]. Este quesito é essencial em cenários de implantação contínua, onde a aplicação precisa ser atualizada frequentemente sem interrupções no serviço.

O Maven foi escolhido como a ferramenta de gerenciamento de dependências e build, facilitando a automação do processo de build e gerenciamento de dependências, garantindo que todas as bibliotecas necessárias estejam atualizadas e que o software possa ser compilado e distribuído de forma eficiente. Com essa funcionalidade, é possível desenvolver novas funcionalidades, como sistemas de notificações para pacientes e ferramentas de análise de dados, sem se preocupar com problemas de dependências.

Para a arquitetura do software, foi utilizada uma abordagem baseada em microserviços. Esse estilo de arquitetura permite que diferentes componentes do sistema, como agendamento de consultas, gerenciamento de pacientes e serviços, sejam desenvolvidos e escalados de maneira independente [Newman, 2015]. Isso

facilita a manutenção e otimiza o tempo de desenvolvimento, pois os desenvolvedores podem trabalhar simultaneamente em diferentes microserviços.

O Kafka foi integrado ao projeto para gerenciar a comunicação assíncrona entre os microserviços, garantindo que as diferentes partes do sistema possam se comunicar de maneira eficiente e resiliente, processando grandes volumes de dados em tempo real. Isso é essencial para funcionalidades que exigem resposta rápida, como alertas em tempo real para mudanças de status de consultas ou atualizações de prontuários.

Para o controle de versão e colaboração, foi utilizado o GitHub, que facilita o versionamento do código e a colaboração entre os desenvolvedores da equipe, permitindo a integração contínua e a revisão de código [Loeliger e McCullough, 2012]. No desenvolvimento do software odontológico, isso garante que a equipe possa trabalhar de forma organizada e colaborativa, implementando e revisando novas funcionalidades de forma ágil e segura.

5. RESULTADOS

A API foi desenvolvida para fornecer funcionalidades robustas de gerenciamento de usuários, pacientes, serviços e login em um ambiente de clínica. Nesta sessão, será apresentado cada endpoint do software corresponde a um recurso específico que facilita operações essenciais para o sistema.

5.1. Agenda

A página '/agenda', como mostrado na Figura 16, é projetada para facilitar uma gestão eficiente e organizada dos horários de consulta, garantindo que os profissionais possam planejar suas agendas de forma otimizada e os pacientes possam ser atendidos de maneira adequada e pontual.

Figura 16: Endpoints da página Agenda.

agenda-resource

Método	Endpoint
PUT	/agenda/editar/{scheduleId}
POST	/agenda/inserir
GET	/agenda
GET	/agenda/{scheduleId}
DELETE	/agenda/deletar/{scheduleId}

Fonte: Autor

5.2. Dashboards

Na figura 17, a página '/dashboards' fornece recursos para visualização de dados e métricas importantes da clínica, oferecendo insights detalhados através de gráficos, tabelas e outros elementos visuais que ajudam na tomada de decisões estratégicas e na monitorização eficiente das operações.

Figura 17: Endpoints da página Dashboards

dashboard-resource

Método	Endpoint
GET	/usuarioAtivo
GET	/topServicos
GET	/recepcionistas
GET	/receita
GET	/profissionaisDaSaudeTotal
GET	/pacientesTratados
GET	/atendimentos
GET	/administrador

Fonte: Autor

5.3. Estoque

O serviço de '/estoque' oferece funcionalidades para gerenciar o inventário da clínica, incluindo operações para adicionar novos itens ao estoque, atualizar informações existentes, remover itens e consultar detalhes específicos por ID. Essa página é essencial para o controle e a manutenção eficaz dos recursos materiais da clínica, conforme a Figura 18.

Figura 18: Endpoints da página Estoque.

estoque-resource

PUT /estoque/editar/{productId}

POST /estoque/inserir

GET /estoque

GET /estoque/{productId}

DELETE /estoque/deletar/{productId}

Fonte: Autor

5.4. Login

No contexto de '/login', como mostrado na Figura 19, a API proporciona um mecanismo seguro de autenticação, validando as credenciais de usuário (email e senha) para garantir o acesso seguro aos recursos da aplicação.

Figura 19: Endpoints da página Login.

login-resource

POST /login

Fonte: Autor

5.5. Paciente

Para o recurso de ‘/paciente’, a API permite a inserção de novos registros de pacientes, atualização de informações, remoção de pacientes existentes e consulta detalhada de pacientes por ID. Há também uma funcionalidade especial para buscar as últimas consultas registradas, fornecendo um histórico detalhado que inclui data, notas de consulta e nome do profissional responsável, conforme na Figura 20.

Figura 20: Endpoints da página Paciente.

paciente-resource

PUT /paciente/editar/{patientId}

POST /paciente/inserir

POST /paciente/criarHistorico

GET /paciente

GET /paciente/{patientId}

GET /paciente/ultimasConsultas

DELETE /paciente/deletar/{patientId}

Fonte: Autor

5.6. Serviços

Na Figura 21 é apresentado a página ‘/servicos’, que é dedicada ao gerenciamento de serviços oferecidos pela clínica. Oferece operações para criar serviços, atualizar detalhes existentes, remover serviços e consultar informações detalhadas de um serviço específico por ID.

Figura 21: Endpoints da Página Serviços.

service-resource

PUT /servicos/editar/{serviceId}

POST /servicos/inserir

GET /servicos

GET /servicos/{serviceId}

DELETE /servicos/deletar/{serviceId}

Fonte: Autor

5.7. Usuários

Por fim, conforme ilustrado na Figura 22, a página '/usuario' oferece operações para inserir novos usuários, atualizar informações existentes, deletar registros e buscar detalhes de usuários por ID. Além disso, é possível listar todos os usuários cadastrados e filtrar por categorias específicas, como profissionais de saúde ou recepcionistas.

Figura 22- Endpoints da Página Usuários.

user-resource

PUT /usuario/editar/{userId}

POST /usuario/inserir

GET /usuario

GET /usuario/{userId}

GET /usuario/recepcionistas

GET /usuario/profissionaisDaSaude

DELETE /usuario/deletar/{userId}

Fonte: Autor

6. CONCLUSÃO E TRABALHO FUTURO

6.1. Conclusão

Ao final deste trabalho, pode-se afirmar que os objetivos inicialmente propostos foram plenamente atingidos, consolidando o desenvolvimento de um sistema backend robusto e eficiente para a gestão de clínicas e consultórios odontológicos, utilizando uma arquitetura baseada em microsserviços. Este projeto proporcionou uma solução tecnológica moderna e escalável, adaptada às necessidades específicas do setor de saúde, e pronta para evoluir conforme as demandas futuras.

A escolha pela arquitetura de microsserviços foi determinante para o sucesso do projeto. A modularidade inerente a essa abordagem permitiu que cada funcionalidade do software, desde a gestão de pacientes, até a administração de consultas e tratamentos fosse implementada de maneira independente. Isso trouxe não apenas facilidade de manutenção e atualização, mas também permitiu uma evolução contínua do sistema, com a possibilidade de incorporar novas funcionalidades sem impactar negativamente os módulos existentes.

Além disso, a arquitetura de microsserviços favoreceu a escalabilidade do sistema. Cada serviço pôde ser dimensionado de forma independente, respondendo adequadamente ao aumento de demanda em áreas críticas, como o agendamento de consultas ou o gerenciamento de estoques de materiais odontológicos. No ambiente de homologação, a escalabilidade flexível garantiu que o sistema continuasse a operar de maneira eficiente mesmo em momentos de alta carga, assegurando um desempenho otimizado e uma experiência de usuário satisfatória.

A interoperabilidade foi outro ponto forte da solução adotada. Essa flexibilidade é fundamental em um cenário de saúde em constante evolução, onde a capacidade de se adaptar e integrar novas tecnologias pode fazer a diferença na eficiência operacional e na qualidade do atendimento ao paciente.

No entanto, como em toda implementação de arquitetura distribuída, o projeto enfrentou desafios significativos. A complexidade na gestão de configurações e na orquestração dos serviços exigiu uma abordagem cuidadosa e o uso de ferramentas avançadas para monitoramento e logging distribuído. A gestão da comunicação entre os microsserviços, especialmente em um ambiente de saúde onde a confiabilidade e

a segurança dos dados são cruciais, também foi um aspecto desafiador, mas que foi superado com a adoção de boas práticas e padrões de comunicação robustos.

Por fim, é importante destacar que a arquitetura de microsserviços não apenas atendeu às necessidades atuais do projeto, mas também preparou o sistema para o futuro. A flexibilidade e escalabilidade garantem que o sistema possa continuar a evoluir, incorporando novas tecnologias e funcionalidades à medida que o setor de saúde e as demandas dos usuários mudam. Isso assegura a longevidade e relevância do software, permitindo que ele continue a oferecer valor a longo prazo.

Em suma, o desenvolvimento deste sistema de gestão para clínicas odontológicas, utilizando microsserviços, demonstrou ser uma solução não apenas tecnicamente sólida, mas também alinhada com as melhores práticas do mercado, proporcionando um sistema adaptável, escalável e pronto para atender às demandas de um ambiente de saúde dinâmico e em constante transformação.

6.2. Trabalho Futuro

Embora o trabalho realizado nesta dissertação tenha cumprido seus objetivos, sempre há espaço para melhorias e expansões. Novas funcionalidades, abordagens alternativas e requisitos emergentes podem ser incorporados ao sistema, especialmente em um campo tão dinâmico quanto o da arquitetura de microsserviços. A seguir, são apresentadas algumas sugestões para trabalhos futuros, que podem enriquecer e expandir o escopo deste estudo.

- **Aprofundamento do conhecimento em microsserviços:**

Apesar de todo o estudo e implementação realizados, a arquitetura de microsserviços é vasta e está em constante evolução. Continuar a explorar novos conceitos, técnicas e ferramentas relacionados a microsserviços é essencial para manter o sistema atualizado e preparado para os desafios futuros. Isso inclui o acompanhamento de novas práticas de design, padrões emergentes e tecnologias que possam melhorar a eficiência, segurança e escalabilidade da arquitetura.

- **Aplicação de novos padrões de migração:**

Durante o estado da arte, diversos padrões de migração foram apresentados, mas nem todos foram implementados. Uma linha de investigação futura interessante seria aplicar esses padrões não explorados e estudar a eficácia deles em diferentes

contextos. Além disso, buscar novos padrões de migração que possam ter surgido recentemente pode abrir novas oportunidades para otimizar o processo de transformação de arquiteturas.

- **Exploração de diferentes protocolos de comunicação:**

Atualmente, a comunicação entre os microsserviços foi realizada utilizando REST e um message broker. No entanto, existem outras formas de comunicação, como a comunicação por eventos, que podem oferecer benefícios adicionais em termos de desempenho, escalabilidade e desacoplamento dos serviços. Explorar e implementar essas alternativas poderia enriquecer a arquitetura do sistema e permitir novas abordagens para a integração dos microsserviços.

- **Aprimoramento da segurança dos microsserviços:**

A segurança é um aspecto crucial em qualquer sistema distribuído. Atualmente, os endpoints dos microsserviços estão acessíveis a todos os outros microsserviços sem restrições. Introduzir controles de acesso mais rigorosos e autenticação adequada pode aumentar significativamente a segurança do sistema. Investigar e implementar medidas como OAuth, JWT ou outras técnicas de autenticação e autorização são caminhos promissores para melhorar a proteção dos dados e a segurança da aplicação.

- **Simulação de migração em produção:**

O processo de migração foi realizado em um ambiente local, sem a preocupação com falhas críticas ou perda de dados. Um desafio futuro seria simular a migração em um ambiente de produção, onde erros podem ter consequências sérias. Isso exigiria a implementação de processos de rollback para garantir a segurança em caso de falhas, além de estratégias de migração de dados para minimizar o impacto no sistema durante o processo.

- **Implementação de um API Gateway:**

A arquitetura distribuída utilizada neste trabalho não contou com uma entidade centralizada para gerenciar a comunicação entre os microsserviços. No entanto, experimentar a implementação de um API Gateway poderia oferecer vários benefícios, como segurança centralizada, mapeamento eficiente de solicitações e monitoramento do sistema como um todo. Isso transformaria a

arquitetura de microsserviços em uma estrutura mais centralizada, com um ponto único de entrada que facilita a gestão e o controle da aplicação.

- **Desenvolvimento de um log centralizado:**

Em uma arquitetura de microsserviços, onde os serviços são executados separadamente, a depuração pode se tornar complexa devido à necessidade de analisar logs dispersos. A criação de um sistema de log centralizado permitiria uma depuração mais eficiente, consolidando os logs de todos os microsserviços em um único local. Isso tornaria o processo de identificar e resolver problemas mais rápido e eficaz, contribuindo para uma operação mais suave do sistema.

Essas direções futuras oferecem um vasto campo de exploração e aprimoramento para o trabalho desenvolvido. À medida que novas tecnologias e técnicas continuam a emergir, a aplicação desses conceitos poderá evoluir significativamente, garantindo que a plataforma de e-commerce permaneça competitiva e capaz de atender às demandas de um mercado em constante transformação.

REFERÊNCIAS BIBLIOGRÁFICAS

Artigos, Livros e Manuais

BRUCE, J.; PEREIRA, A. **Designing Scalable Systems: Patterns and Best Practices for Building Large-Scale Systems**. O'Reilly Media, 2018.

CONWAY, M. E. **How do Committees Invent? Datamation,** v. 14, n. 4, p. 28-31, 1968.

DOYLE, J.; et al. **Microservices in Practice: A Guide to Architectural Design and Implementation.** Springer, 2021.

DRAGONI, N.; et al. **Microservices: Yesterday, Today, and Tomorrow. In: Present and Ulterior Software Engineering**. Springer, 2017.

DU BOIS, P. **MySQL**. Addison-Wesley Professional, 2008.

ECKEL, B. **Thinking in Java**. Prentice Hall, 2003.

EVANS, E. **Domain-Driven Design: Tackling Complexity in the Heart of Software.** Addison-Wesley, 2003.

FIELDING, R. T.; TAYLOR, R. N. **Principled Design of the Modern Web Architecture.** ACM Transactions on Internet Technology (TOIT), v. 2, n. 2, p. 115-150, 2000.

FOWLER, M. **Patterns of Enterprise Application Architecture.** Addison-Wesley Professional, 2019.

FOWLER, M. **Refactoring: Improving the Design of Existing Code.** Addison-Wesley, 2013.

JANSEN, R.; SALADAS, P. **Architecting for Scale: Designing Resilient Systems with Microservices.** O'Reilly Media, 2020.

JAVED, M. **Architecting Modern Software Applications.** Packt Publishing, 2019.

KREPS, J.; NARKHEDE, N.; RAO, J. **Kafka: A Distributed Messaging System for Log Processing.** LinkedIn, 2014.

LEWIS, J.; FOWLER, M. **Microservices: A Definition of This New Architectural Term**. Martin Fowler Blog, 2014. Disponível em: https://martinfowler.com/articles/microservices.html.

LOELIGER, J.; MCCULLOUGH, M. **Version Control with Git.** O'Reilly Media, 2012.

MERKEL, D. **Docker: Lightweight Linux Containers for Consistent Development and Deployment.** Linux Journal, n. 239, Article 2, 2014.

MILANI, A. **Banco de Dados Relacional: Modelagem e Implementação.** Elsevier, 2006.

NADAREISHVILI, I.; MITRA, R.; MCHAUGHAN, M.; AMUNDSON, M. **Microservice Architecture: Aligning Principles, Practices, and Culture**. O'Reilly Media, 2016.

NEWMAN, S. **Building Microservices.** O'Reilly Media, 2015.

NEWMAN, S. **Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith.** O'Reilly Media, 2021.

PINHEIRO, J. A. **Técnicas de Programação Orientada a Objetos.** São Paulo: Editora Ciência Moderna, 2010.

RICHARDSON, C. **Microservices Patterns: With Examples in Java.** Manning Publications, 2018.

STELLMAN, A.; GREENE, J. **Learning Agile: Understanding Scrum, XP, Lean, and Kanban.** O'Reilly Media, 2014.

THÖNES, J. **Microservices.** IEEE Software, v. 32, n. 1, p. 116-116, 2015.

WALLS, C. **Spring in Action.** Manning Publications, 2021.

Documentação e Fontes Online

Apache Kafka Documentation. Disponível em: https://kafka.apache.org/documentation/. Acesso em: 30 jul. 2024.

Docker Docs. Disponível em: https://docs.docker.com/. Acesso em: 7 abr. 2024.

Docker Hub. Disponível em: https://hub.docker.com/. Acesso em: 15 maio 2024.

GitHub Docs. Disponível em: https://docs.github.com/pt. Acesso em: 7 abr. 2024.

GitLab Documentation. Disponível em: https://docs.gitlab.com/. Acesso em: 28 abr. 2024.

IBM (2020). **Modernizing Enterprise Systems with Cloud-Based Solutions.** IBM White Paper. Disponível em: https://www.ibm.com/cloud.

Kafka Project (2024). **Creating a Kafka Java Project using Maven (pom.xml).** Disponível em: https://kafka.apache.org/documentation/.

MicroProfile Documentation. Disponível em: https://microprofile.io/. Acesso em: 3 maio 2024.

Mozilla Developer Network (MDN). **Tecnologia Web para Desenvolvedores.** Disponível em: https://developer.mozilla.org/pt-BR/docs/Web. Acesso em: 24 mar. 2024.

Spring Boot Documentation. Disponível em: https://spring.io/projects/spring-boot. Acesso em: 26 abr. 2024.

Teses e Trabalhos Acadêmicos

SOMI, M. **Desenvolvimento de Interface de Usuário de um Moderno Aplicativo Web.** Politecnico di Torino, 2021. Disponível em: https://webthesis.biblio.polito.it/secure/30076/1/tesi.pdf.