UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

PEDRO HENRIQUE GRIGORIO LIMA

AVALIAÇÃO PRÁTICA DA ABORDAGEM *MONOLITH FIRST* NO DESENVOLVIMENTO DE APLICAÇÕES COM ARQUITETURA DE MICROSSERVIÇOS

QUIXADÁ

2025

PEDRO HENRIQUE GRIGORIO LIMA

AVALIAÇÃO PRÁTICA DA ABORDAGEM *MONOLITH FIRST* NO DESENVOLVIMENTO DE APLICAÇÕES COM ARQUITETURA DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientador: Prof. Dr. Jefferson de Carvalho Silva.

QUIXADÁ

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

L71a Lima, Pedro Henrique Grigorio.
Avaliação prática da abordagem Monolith First no desenvolvimento de aplicações com arquitetura de microsserviços / Pedro Henrique Grigorio Lima. – 2025.
72 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2025.
Orientação: Prof. Dr. Jefferson de Carvalho Silva .

1. Monolith First. 2. Microsserviços. 3. Domain-Driven Design. 4. Clean Architecture. 5. Apache Kafka. I. Título.

CDD 005.1

PEDRO HENRIQUE GRIGORIO LIMA

AVALIAÇÃO PRÁTICA DA ABORDAGEM *MONOLITH FIRST* NO DESENVOLVIMENTO DE APLICAÇÕES COM ARQUITETURA DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: 07/03/2025.

BANCA EXAMINADORA

Prof. Dr. Jefferson de Carvalho Silva (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Bruno Góis Mateus
Universidade Federal do Ceará (UFC)

Prof. Dr. Wladimir Araújo Tavares
Universidade Federal do Ceará (UFC)

A Deus, que me permitiu estar onde estou hoje e ter guiado meus passos. Aos meus pais, por todo o amor, confiança e por ter investido em mim. Aos meus amigos, que me deram apoio nos momentos difíceis e foram fundamentais para que eu não desistisse.

AGRADECIMENTOS

Primeiramente a Deus, pois sem ele eu jamais teria sequer iniciado essa caminhada.

Aos meus pais, pela confiança, amor e dedicação, que apesar das dificuldades sempre confiaram em mim e me manteram firme e confiante.

Ao Prof. Dr. Jefferson de Carvalho Silva, pela orientação, paciência e valiosas contribuições durante todo o desenvolvimento deste trabalho.

Aos meus amigos, pelo apoio incondicional ao longo desta jornada. Nos momentos mais difíceis, vocês estiveram ao meu lado, oferecendo palavras de incentivo e tornando essa caminhada mais leve e significativa.

Aos professores da faculdade, que compartilharam conhecimento e experiência, contribuindo não apenas para a realização deste trabalho, mas também para minha formação acadêmica e profissional.

A todos que, de alguma forma, fizeram parte dessa trajetória, meu sincero agradecimento.

"No que diz respeito ao empenho, ao compromisso, ao esforço, à dedicação, não existe meio-termo. Ou você faz uma coisa bem-feita ou não faz." (Ayrton Senna)

RESUMO

Este trabalho tem como objetivo avaliar a abordagem *Monolith First* no desenvolvimento de sistemas em microsserviços, por meio de um estudo de caso que envolve a construção de um sistema inicialmente como um monólito modular, seguido de sua migração para microsserviços. A metodologia adotada consiste em cinco etapas: levantamento de requisitos, projeto do sistema, desenvolvimento do monólito modular, migração para microsserviços e avaliação da abordagem utilizada. A análise foca na eficácia da transição, destacando os benefícios da utilização de *Domain-Driven Design* (DDD) e *Clean Architecture* para a estruturação do sistema. O estudo também explora os desafios enfrentados, como a implementação de comunicação entre serviços utilizando Apache Kafka e *Google Remote Procedure Call* (gRPC), e como a replicação de dados foi fundamental para garantir a independência entre os microsserviços. Os resultados demonstram que a abordagem *Monolith First* facilita a definição de contextos delimitados, reduz a complexidade inicial do desenvolvimento e possibilita uma migração mais ágil e eficiente para microsserviços. A pesquisa conclui que essa estratégia é altamente recomendada para projetos que buscam uma arquitetura distribuída bem estruturada.

Palavras-chave: *monolith first*; microsserviços; *domain-driven design*; *clean architecture*; apache kafka; gRPC.

ABSTRACT

This paper aims to evaluate the Monolith First approach in the development of microservices systems, through a case study involving the development of a system initially as a modular monolith, followed by its migration to microservices. The methodology adopted consists of five stages: requirements gathering, system design, modular monolith development, migration to microservices, and evaluation of the approach used. The analysis focuses on the effectiveness of the transition, highlighting the benefits of using Domain-Driven Design (DDD) and Clean Architecture in structuring the system. The study also explores the challenges faced, such as the implementation of communication between services using Apache Kafka and gRPC, and how data replication was essential to ensure service independence. The results demonstrate that the Monolith First approach facilitates the definition of bounded contexts, reduces the initial complexity of development, and allows for a faster and more efficient migration to microservices. The research concludes that this strategy is highly recommended for projects aiming to build a well-structured distributed architecture.

Keywords: monolith first; microservices; domain-driven design; clean architecture; apache kafka; gRPC.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicações
DDD	*Domain-Driven Design*
gRPC	*Google Remote Procedure Call*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
JWT	*JSON Web Token*
RPC	Chamada de Procedimento Remoto
UI	Interface de Usuário
UML	Linguagem de Modelagem Unificada

SUMÁRIO

1 INTRODUÇÃO

Uma aplicação com arquitetura monolítica é aquela que, embora possa ser formada por vários módulos, possui um único executável, enquanto que uma aplicação com arquitetura de microsserviços é uma aplicação distribuída onde todos os seus módulos ou elementos são microsserviços e podem ser executados de forma independente (Velepucha; Flores, 2023). A arquitetura de microsserviços ganhou força significativa, em parte devido ao seu potencial para fornecer produtos de software escaláveis, robustos, ágeis e resistentes a falhas (Abgaz *et al.*, 2023). No entanto, a transição para uma arquitetura de microsserviços não está isenta de desafios, visto que o processo de decomposição de monólitos existentes em microsserviços coesos se mostra uma etapa complexa, devido à dificuldade de se definir fronteiras claras entre cada serviço (Oumoussa; Saidi, 2024).

A abordagem *Monolith First* surge como uma estratégia para mitigar esses desafios (Fowler, 2015). De acordo com Fowler (2015), não se deve começar um novo projeto com microsserviços, mesmo se tiver certeza de que seu aplicativo será grande o suficiente para valer a pena. Laigner *et al.* (2020) recomendam a criação de um monólito antes da migração para os microsserviços, pois enfrentaram desafios na criação de uma arquitetura orientada a serviços do zero. A recomendação de se iniciar com um monólito é pautada na facilitação do desenvolvimento inicial e oferecimento de uma visão clara do domínio do problema, permitindo que a equipe refine os requisitos antes de adicionar a camada de complexidade extra que acompanha a arquitetura de microsserviços (Fowler, 2015).

O problema em começar com microsserviços é que eles só funcionam bem se forem definidos limites claros e estáveis entre os serviços, ou seja, elaborar o conjunto certo de *Bounded Contexts* (contexto delimitado), que é um conceito central no *Domain-Driven Design* (DDD) (Fowler, 2015). O contexto delimitado estabelece limites conceituais que ajudam a manter a consistência do domínio (Özkan *et al.*, 2023). Nesse sentido, Vural e Koyuncu (2021) buscaram saber se o DDD leva a encontrar a modularidade ideal de um microsserviço e concluíram que a abordagem é um bom ponto de partida e que se mostra uma ótima metodologia para identificar a granularidade ideal de microsserviços.

Ao aplicar um DDD em uma arquitetura monolítica tradicional, poderíamos ter diferentes domínios que, por sua vez, poderiam se tornar módulos (Tsechelidis *et al.*, 2023). Nesse sentido, o monólito modular é uma variação da arquitetura monolítica tradicional que consiste em módulos separados que podem trabalhar independentemente, mas que ainda pre-

cisam ser combinados para implantação (Newman, 2019). De acordo com Newman (2019), o monólito modular traz muitos benefícios da arquitetura de microsserviços, como a separação de responsabilidades e a possibilidade de separação de times para cada módulo, e da arquitetura monolítica, como a simplicidade operacional de uma única base de código, facilitando assim a integração. Além disso, a criação de um monólito modular facilita a transição para a arquitetura de microsserviços e é uma das melhores formas de migrar um sistema legado (Vernon; Tomasz, 2021).

Apesar da popularidade da abordagem *Monolith First*, há poucos estudos que detalham seu processo de implementação e as melhores práticas associadas. Assim, este trabalho busca explorar essa estratégia de forma prática, aplicando-a na criação de um sistema desde sua concepção, utilizando um estudo de caso para avaliar os benefícios e os desafios dessa abordagem. Durante o processo, serão analisadas as melhores práticas, as abordagens recomendadas e os resultados obtidos na transição para uma arquitetura de microsserviços. O principal público-alvo deste trabalho são desenvolvedores interessados em compreender as etapas envolvidas no uso da abordagem *Monolith First*, bem como em aprender como projetar sistemas escaláveis desde o início, contribuindo para a adoção consciente e eficiente da arquitetura de microsserviços. A aplicação prática e as avaliações realizadas pretendem preencher lacunas na literatura existente e servir como referência para futuros estudos e implementações.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é avaliar a abordagem *Monolith First* para o desenvolvimento de microsserviços durante o desenvolvimento de uma aplicação, investigando como a construção inicial de um sistema monolítico pode servir como uma base sólida para a posterior migração para uma arquitetura de microsserviços.

1.1.2 Objetivos Específicos

- Implementar um monólito modular como etapa intermediária no processo de migração para microsserviços.
- Implementar o uso de mensageria assíncrona na comunicação entre microsserviços, considerando como a integração de sistemas de mensageria pode impactar a arquitetura e a

performance do sistema.

- Avaliar a aplicação de padrões e práticas de arquitetura na construção de um sistema baseado na abordagem *Monolith First* e sua transição para microsserviços.
- Avaliar o processo de desenvolvimento e documentar principais aprendizados e dificuldades.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo, serão apresentados alguns dos conceitos mais importantes para o entendimento deste trabalho. Na Seção 2.1 serão explicados os conceitos sobre microsserviços. Na Seção 2.2 será apresentado o conceito de *Monolith First* e os tipos de monólito. Por fim, na Seção 2.3 serão apresentados conceitos importantes sobre arquitetura e *design* de *software*.

2.1 Microsserviços

De acordo com Velepucha e Flores (2023), uma aplicação com arquitetura monolítica é caracterizada por sua natureza centralizada, onde todos os componentes do sistema são interdependentes. Por outro lado, a arquitetura de microsserviços é uma aplicação distribuída composta por vários serviços autônomos e independentes, cada um executando sua própria função específica e se comunicando por meio de Interface de Programação de Aplicações (API)s ou outros mecanismos de comunicação (Velepucha; Flores, 2023).

Os microsserviços representam uma abordagem moderna na engenharia de *software* que tem ganhado popularidade nos últimos anos (Oumoussa; Saidi, 2024). De acordo com Oumoussa e Saidi (2024), essa arquitetura quebra as aplicações tradicionais monolíticas em serviços independentes, permitindo que sejam projetados, testados e implantados de forma individualizada. Uma das principais características dos microsserviços é sua capacidade de serem desenvolvidos de forma independente e serem operados da mesma maneira que em uma aplicação monolítica (Velepucha; Flores, 2023). A decomposição funcional das aplicações é fundamental nesse contexto, permitindo a construção de aplicações ou serviços em um nível superior, combinando diversos microsserviços (Oumoussa; Saidi, 2024).

Os microsserviços oferecem vantagens significativas em relação às arquiteturas monolíticas tradicionais. Uma das principais vantagens é a capacidade de escalabilidade, permitindo que cada serviço seja dimensionado independentemente com base na demanda específica (Hasselbring; Steinacker, 2017). Além disso, os microsserviços facilitam a manutenção e evolução contínua do *software*, pois cada serviço pode ser atualizado, testado e implantado de forma independente, promovendo maior agilidade no desenvolvimento, enquanto que na arquitetura monolítica uma única alteração pode afetar o comportamento de todo o sistema (Velepucha; Flores, 2023). Outra vantagem é a resiliência, uma vez que falhas em um serviço não afetam o sistema global devido ao isolamento dos microsserviços (Velepucha; Flores, 2023).

2.1.1 Comunicação entre serviços

A comunicação é um elemento fundamental de uma aplicação de microsserviços. Microsserviços comunicam entre si para realizar uma tarefa útil (Bruce; Pereira, 2018). Nesse sentido, uma das principais decisões a serem tomadas diz respeito a como os serviços devem colaborar, ou seja, devemos decidir se a comunicação será síncrona ou assíncrona (Newman, 2014).

De acordo com Kleppmann (2017), quando um serviço precisa se comunicar com outro através de uma rede, existem diferentes formas de organizar essa comunicação. A forma mais comum envolve dois principais agentes: o cliente e o servidor. Neste caso, o servidor expõe uma API e os clientes podem se conectar e realizar chamadas a essa API exposta (Kleppmann, 2017). Durante a chamada, o cliente aguarda a resposta do servidor antes de prosseguir com a execução de suas próximas ações. Esse comportamento caracteriza a comunicação síncrona. Geralmente, essa é a primeira abordagem que vem à mente, sendo utilizada quando o resultado de uma ação é necessário para a execução da próxima (Bruce; Pereira, 2018).

Por outro lado, a comunicação assíncrona pode ser útil para tarefas de longa duração, onde uma conexão aberta por um longo período é inviável (Newman, 2014). Além disso, ela também é eficaz quando existe a necessidade de baixa latência, pois esperar pela resposta durante uma chamada pode atrasar o processo (Newman, 2014). No entanto, isso tem um custo, visto que interações assíncronas são mais difíceis de gerenciar, já que o comportamento geral do sistema não segue mais uma sequência linear (Bruce; Pereira, 2018).

2.1.1.1 Google Remote Procedure Call (gRPC)

O gRPC é uma tecnologia de comunicação baseada em Chamada de Procedimento Remoto (RPC), projetada para facilitar a interação entre serviços escritos em diferentes linguagens de programação (Richardson, 2018). Ele utiliza o HTTP/2 como protocolo de transporte e *Protocol Buffers* como formato de serialização de mensagens, garantindo um mecanismo binário compacto e eficiente (Richardson, 2018). Diferentemente do REST, que se baseia em padrões de comunicação sobre *Hypertext Transfer Protocol* (HTTP) e utiliza *JavaScript Object Notation* (JSON) para troca de dados, o gRPC define contratos de serviço fortemente tipados e permite a geração automática de código para clientes e servidores (Richardson, 2018). Além disso, o gRPC suporta diferentes tipos de chamadas, incluindo *streaming* bidirecional, permitindo uma

troca contínua de mensagens entre cliente e servidor (Kleppmann, 2017).

O gRPC suporta tanto chamadas de procedimento remoto simples, onde o cliente faz uma solicitação e recebe uma resposta, quanto chamadas de streaming, permitindo a troca de múltiplas mensagens entre o cliente e o servidor ao longo do tempo (Kleppmann, 2017). Essa flexibilidade facilita a implementação de sistemas que exigem comunicação contínua, como em casos de transmissão de dados ou processamento em tempo real. Além disso, o formato binário do Protocol Buffers permite que as APIs evoluam de forma eficiente, mantendo a compatibilidade retroativa, já que os campos de uma mensagem podem ser ignorados se não forem reconhecidos pelo receptor (Richardson, 2018).

Embora o gRPC ofereça vantagens em termos de performance e flexibilidade, ele possui alguns desafios. Por exemplo, a configuração e o consumo de APIs gRPC podem ser mais complexos para clientes em JavaScript quando comparados ao uso de APIs REST com JSON (Richardson, 2018). Além disso, o uso de HTTP/2 pode ser um obstáculo para redes que não o suportam, como algumas versões mais antigas de *firewalls* (Richardson, 2018). Apesar desses desafios, o gRPC é uma alternativa atraente ao REST, principalmente quando se busca uma comunicação eficiente e a interoperabilidade entre serviços escritos em diferentes linguagens (Bruce; Pereira, 2018).

Em comparação com outras tecnologias de RPC, o gRPC se destaca por sua simplicidade e integração com o ecossistema de *Protocol Buffers*, permitindo um *design* de serviço claro e um desempenho superior devido ao uso de um formato binário (Kleppmann, 2017). Essa abordagem é particularmente vantajosa em sistemas distribuídos, onde a comunicação eficiente entre serviços pode impactar diretamente a escalabilidade e a performance geral do sistema (Bruce; Pereira, 2018).

2.1.1.2 Apache Kafka

A mensageria assíncrona é um mecanismo fundamental para a comunicação entre serviços em arquiteturas distribuídas, permitindo a emissão e o consumo de eventos de forma desacoplada (Bruce; Pereira, 2018). Nesse modelo, os serviços não se comunicam diretamente por chamadas síncronas, mas emitem eventos que são processados por consumidores interessados, sem que o emissor precise conhecer ou coordenar suas ações (Newman, 2014). Um exemplo prático desse modelo pode ser visto em um sistema de e-commerce, onde o serviço de pedidos publica um evento informando a criação de um pedido. Outros serviços, como faturamento e

envio, podem consumir essa mensagem e processá-la em seus próprios tempos, garantindo que cada etapa ocorra de maneira independente e tolerante a falhas (Bruce; Pereira, 2018).

Mensagens assíncronas normalmente requerem um agente de comunicação, um componente de sistema independente que recebe eventos e os distribui aos consumidores de eventos (Bruce; Pereira, 2018). Um exemplo amplamente conhecido e utilizado desses agentes é o Apache Kafka. Em geral, os agentes de mensagens funcionam enviando uma mensagem para uma fila ou tópico específico, onde o agente se encarrega de entregar essa mensagem a um ou mais consumidores ou assinantes vinculados (Kleppmann, 2017). Inclusive, um mesmo tópico pode ter vários produtores e consumidores simultaneamente (Kleppmann, 2017). Esses agentes são projetados para serem escaláveis, mas adicionam uma camada extra de complexidade, pois é necessário executar um novo sistema para desenvolver os serviços (Newman, 2014).

2.1.2 Decomposição de microsserviços

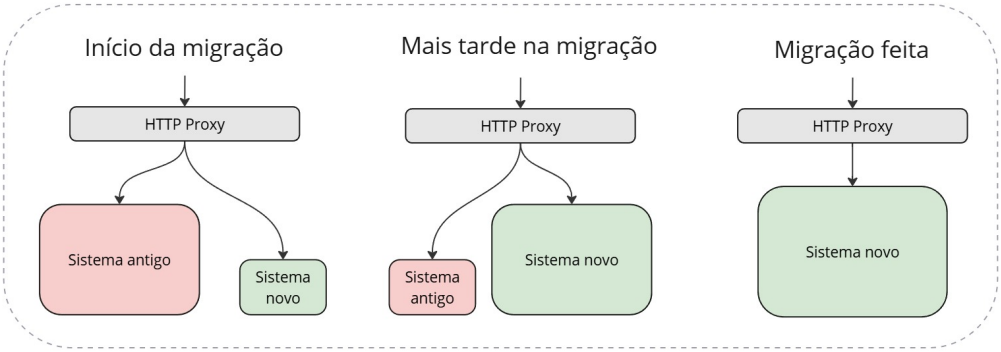
A decomposição de um monólito em microsserviços é uma tarefa complicada e cara (Abgaz *et al.*, 2023). Uma decomposição inadequada pode resultar em serviços muito pequenos, aumentando a complexidade da comunicação entre os serviços (Jamshidi *et al.*, 2018). Por outro lado, serviços muito grandes podem comprometer a independência e a autonomia dos microsserviços, dificultando a manutenção e a evolução individual de cada componente (Abgaz *et al.*, 2023). Portanto, identificar os limites corretos e estabelecer uma estratégia eficaz de decomposição são desafios cruciais que os desenvolvedores enfrentam ao adotar arquiteturas baseadas em microsserviços.

Entre os métodos amplamente utilizados para a decomposição de microsserviços, destacam-se o *Strangler Fig Pattern* (ou Padrão Figueira Estranguladora) e o *Branch by Abstraction* (ou Ramificação por Abstração) (Newman, 2019).

O *Strangler Fig Pattern* é uma abordagem gradual para a migração de sistemas monolíticos para arquiteturas baseadas em microsserviços (Fowler, 2004). Inspirado pela forma como a figueira estranguladora cresce e substitui a árvore hospedeira, esse padrão envolve a criação de novos microsserviços que substituem funcionalidades específicas do sistema monolítico (Fowler, 2004). A transição é feita de forma incremental, com o novo sistema gradualmente assumindo o tráfego e as responsabilidades do antigo. Isso permite uma adaptação menos arriscada e mais controlada, reduzindo a possibilidade de interrupções no serviço durante o processo de migração (Newman, 2019). A Figura 1 exemplifica o processo de migração utilizando este método para um

sistema que utiliza HTTP, onde é implementado um *proxy* reverso que redireciona as requisições para o sistema antigo ou para o novo. Nesta figura, são apresentados três momentos no processo. O início da migração, onde temos a maior parte do sistema composto pela implementação antiga e um novo sistema começando a ser construído. Na sequência, temos o sistema novo comportando a maior parte das funcionalidades e, por fim, temos o sistema novo substituindo por completo o sistema antigo.

Figura 1 – Processo de migração utilizando o método *Stangler Fig*

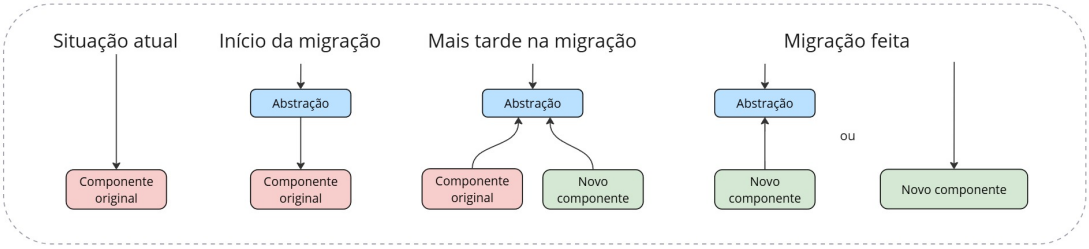


Fonte: Adaptado de (Milanović, 2023)

Por outro lado, o *Branch by Abstraction* oferece uma abordagem diferente para a decomposição de sistemas. Essa técnica envolve a introdução de uma camada de abstração que permite que o sistema monolítico e os novos microsserviços coexistam e interajam de forma simultânea (Milanović, 2023). A camada de abstração atua como um intermediário, direcionando as chamadas para os novos microsserviços ou para o sistema monolítico, conforme necessário (Milanović, 2023). A vantagem do *Branch by Abstraction* é que ele permite uma transição mais fluida e uma integração gradual, permitindo que a mudança seja realizada em fases e com menor impacto no funcionamento geral do sistema (Newman, 2019). A Figura 2 ilustra o processo de migração utilizando o método *Branch by Abstraction*, onde é implementada uma abstração que utiliza o componente original do sistema e, em seguida, cria-se uma nova implementação para a abstração. Após a migração, o novo componente toma o lugar do componente original.

Ambos os métodos oferecem vantagens distintas e podem ser escolhidos com base nas necessidades específicas do projeto, na complexidade do sistema monolítico e nos objetivos de migração (Newman, 2019). A escolha do método adequado é essencial para garantir uma transição bem-sucedida para uma arquitetura de microsserviços, minimizando riscos e maximizando a eficiência da implementação.

Figura 2 – Processo de migração utilizando o método *Branch by Abstraction*



Fonte: Adaptado de (Milanović, 2023)

2.2 *Monolith First*

De acordo com Fowler (2015), a abordagem *Monolith First* sugere que, ao iniciar o desenvolvimento de um sistema, pode ser vantajoso começar com uma arquitetura monolítica antes de considerar uma migração para microsserviços. Essa estratégia é baseada na ideia de que um sistema monolítico pode proporcionar uma base sólida para o desenvolvimento inicial e a compreensão do domínio antes de enfrentar a complexidade adicional de uma arquitetura distribuída.

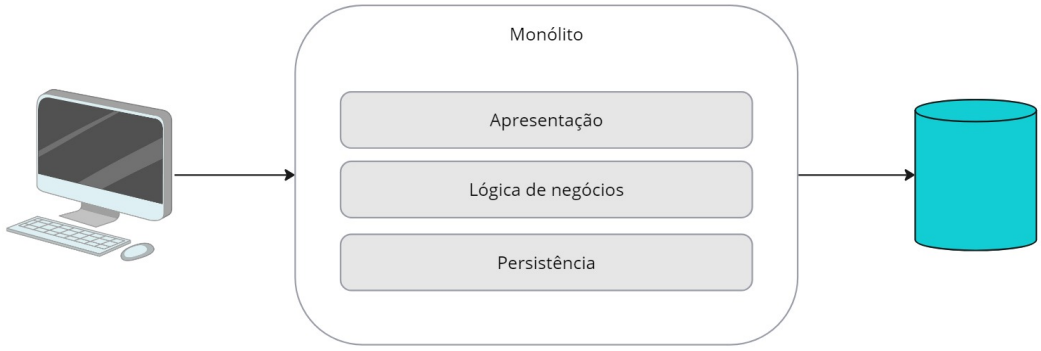
Começar com um monólito permite que a equipe de desenvolvimento concentre seus esforços na construção de um sistema funcional e coeso sem a sobrecarga de gerenciar múltiplos serviços e suas interações desde o início (Fowler, 2015). Em um sistema monolítico, todos os componentes estão integrados em uma única base de código, o que facilita a implementação de funcionalidades e a iteração inicial (Newman, 2019). Esse enfoque pode acelerar o desenvolvimento e permitir que os desenvolvedores aprendam mais sobre as necessidades e os requisitos do sistema antes de fazer a transição para microsserviços (Fowler, 2015).

A vantagem de iniciar com um monólito é que ele oferece uma visão clara do domínio do problema e permite que a equipe refine suas decisões arquitetônicas com base em uma compreensão mais completa do sistema (Fowler, 2015). Uma vez que o sistema monolítico atinge um nível significativo de complexidade, escalabilidade ou necessidade de autonomia dos componentes, a transição para microsserviços pode ser considerada como uma opção mais adequada. A migração gradual, em que se aplicam técnicas como o *Strangler Fig Pattern* ou *Branch by Abstraction*, pode facilitar essa transição, permitindo uma mudança mais controlada (Newman, 2019).

2.2.1 Monólito tradicional

Um monólito é uma abordagem arquitetural onde uma aplicação é desenvolvida como um único e indivisível bloco de código e que representa uma única unidade de integração (Newman, 2019). Tradicionalmente, os monólitos são construídos com um alto nível de acoplamento, utilizando padrões de arquitetura mais simples, apenas com uma divisão de camadas em apresentação, lógica de negócios e persistência, mas sem uma separação por domínios (Richards; Ford, 2020). Este modelo de monólito pode ser visualizado na Figura 3.

Figura 3 – Monólito Tradicional



Fonte: Adaptado de (Milanović, 2023)

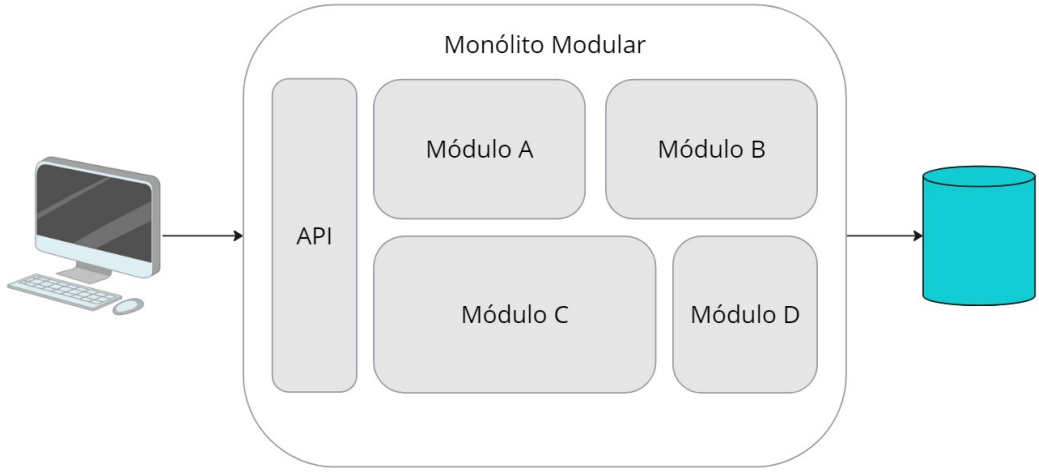
Os monólitos geralmente são mais vulneráveis aos perigos do acoplamento, especificamente, acoplamento de implementação e de implantação (Newman, 2019). Nesse sentido, é comum se deparar com sistemas que se encaixam na "arquitetura"*Big Ball of Mud* (Grande Bola de Lama), que se refere a um sistema bagunçado e sem limites claros definidos de cada componente (Vernon; Tomasz, 2021). Dessa forma, o termo monólito geralmente é visto como algo legado e ruim, quando o problema na verdade não ocorre por conta de o sistema ser monolítico, mas sim por ter sido construído de uma forma ruim (Vernon; Tomasz, 2021).

De acordo com Newman (2019), os sistemas monolíticos tem diversas vantagens, como a implantação mais simples, sem toda a complexidade de sistemas distribuídos, o que resulta em fluxos de trabalho mais simples e possivelmente manutenções mais rápidas. Além disso, um sistema monolítico facilita o reuso de código, visto que todo o sistema esta na mesma base de código, enquanto que em um sistema distribuído teria que decidir se o código será copiado ou se será movido para um serviço (Newman, 2019).

2.2.2 Monólito modular

A modularidade é a solução utilizada pelos humanos para lidar com problemas complexos (Vernon; Tomasz, 2021). O Monólito Modular, representado na Figura 4, é uma variação da arquitetura monolítica tradicional que consiste em módulos separados que podem trabalhar independentemente, mas que ainda precisam ser combinados para implantação (Newman, 2019). Diferentemente de um monólito convencional, onde todos os componentes e funcionalidades do sistema estão interligados de forma coesa e, muitas vezes, acoplada, o monólito modular é projetado para ter uma estrutura interna mais definida e segmentada, permitindo que diferentes partes do sistema sejam desenvolvidas, testadas e mantidas de forma mais independente. Essa abordagem busca introduzir um maior nível de organização e separação de responsabilidades dentro de um único código base (Richards; Ford, 2020).

Figura 4 – Monólito Modular



Fonte: Adaptado de (Milanović, 2023)

A modularidade é uma característica crucial que facilita a transição futura para uma arquitetura de microsserviços. De acordo com Vernon e Tomasz (2021) começar com um monólito modular é a melhor maneira de passar do legado para os microsserviços. Nesta perspectiva, um monólito modular é tipicamente dividido em módulos que representam cada domínio do sistema, cada um encapsulando sua lógica e dados relacionados (Richards; Ford, 2020). Esses módulos comunicam-se entre si através de interfaces bem definidas, reduzindo o acoplamento e promovendo a reutilização de código.

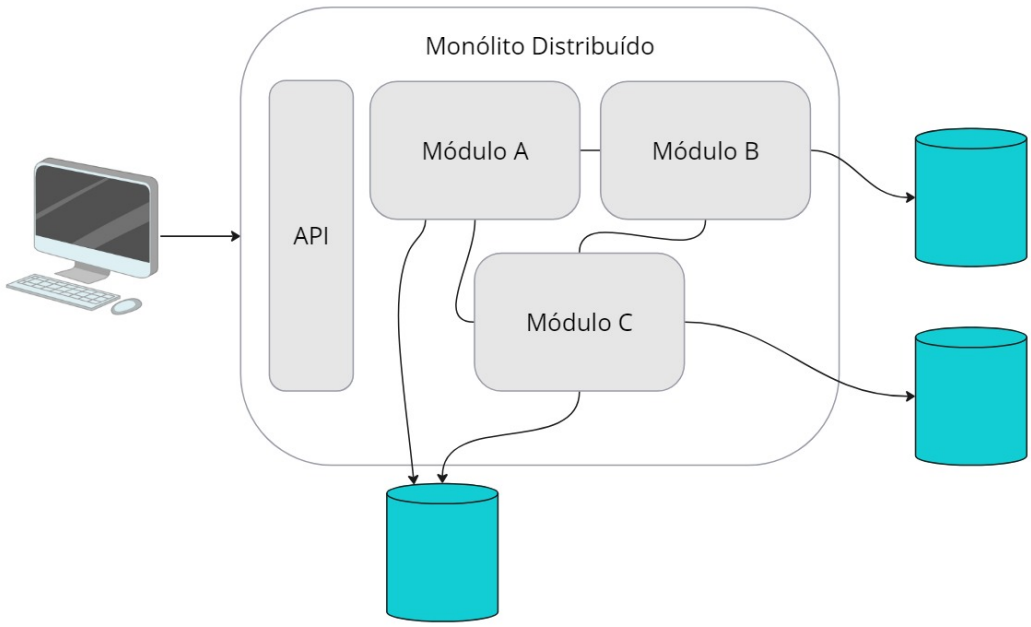
A adoção de um monólito modular permite que a equipe de desenvolvimento obtenha muitos dos benefícios da arquitetura de microsserviços, como a separação de responsabilidades

e a capacidade de escalar partes específicas do sistema, enquanto ainda mantém a simplicidade operacional de uma única base de código, assim, possuindo tanto os benefícios do monólito, como os benefícios dos microsserviços (Newman, 2019). Isso significa que, ao precisar de uma divisão em serviços menores, a organização interna do monólito modular já está preparada para essa evolução. A modularização, portanto, atua como uma ponte natural entre o monólito e os microsserviços, permitindo que a transição seja mais suave e menos disruptiva (Vernon; Tomasz, 2021).

2.2.3 Monólito distribuído

O Monólito Distribuído, representado na Figura 5, refere-se a uma situação em que, apesar de um sistema ser tecnicamente composto de vários serviços ou componentes distribuídos, ele ainda opera de maneira tão fortemente acoplada que se assemelha a um monólito em termos de complexidade e interdependência (Newman, 2019). Essa arquitetura é frequentemente vista como um *antipattern*, pois combina o pior dos dois mundos: a complexidade operacional de um sistema distribuído com as limitações de um monólito acoplado (Newman, 2019).

Figura 5 – Monólito Distribuído



Fonte: Adaptado de (Milanović, 2023)

Na transição da abordagem *Monolith First* para uma arquitetura de microsserviços, é fundamental evitar cair na armadilha do monólito distribuído. Quando a decomposição do monólito inicial não é cuidadosamente planejada, ou quando os serviços resultantes não são

suficientemente independentes, o sistema pode acabar com múltiplos serviços que dependem fortemente uns dos outros, exigindo *deploys* sincronizados e dificultando a escalabilidade e a resiliência (Newman, 2019). Esse tipo de acoplamento pode ocorrer devido ao tipo de comunicação, causando dependências temporais (Vernon; Tomasz, 2021) ou pelo compartilhamento de bibliotecas, onde mudanças podem impactar vários serviços (Ford *et al.*, 2021).

Evitar a criação de um monólito distribuído envolve garantir que cada serviço resultante da decomposição seja verdadeiramente autônomo, com dados e lógica de negócios isolados, e que a comunicação entre eles seja minimizada e gerida por meio de contratos claros e estáveis. É crucial assegurar que cada serviço tenha bem definidas suas fronteiras e responsabilidades, pois o custo de desatar o nó de um monólito distribuído é alto (Newman, 2019).

2.3 Arquitetura e *design* de *software*

Ao falar de desenvolvimento de *software*, cada programador pode escrever código de formas diferentes. Entretanto, quando falamos da construção de um sistema que pode escalar, é extremamente importante que a utilização de padrões amplamente usados (Velepucha; Flores, 2023).

2.3.1 Domain-Driven Design (DDD)

O DDD é uma abordagem de desenvolvimento de *software* que se concentra em alinhar a estrutura do código-fonte com o domínio de negócios do sistema. Introduzido por Eric Evans em 2003, DDD busca conectar a implementação a um modelo de domínio em constante evolução (Vural; Koyuncu, 2021). De acordo com Fowler, DDD centra o desenvolvimento na programação de um modelo de domínio que possui uma compreensão aprofundada dos processos e regras do domínio (Fowler, 2020).

De acordo com Özkan *et al.* (2023), o DDD se baseia em três princípios fundamentais que transformam a abordagem tradicional de *design* de *software*. Primeiro, enfatiza a importância de focar no núcleo do domínio e na lógica do domínio, em vez de se concentrar apenas na arquitetura ou nos detalhes de implementação. Em segundo lugar, promove o uso de um modelo de domínio para orientar *designs* complexos. Finalmente, DDD recomenda a colaboração contínua com especialistas do domínio para aprimorar o modelo da aplicação e resolver questões

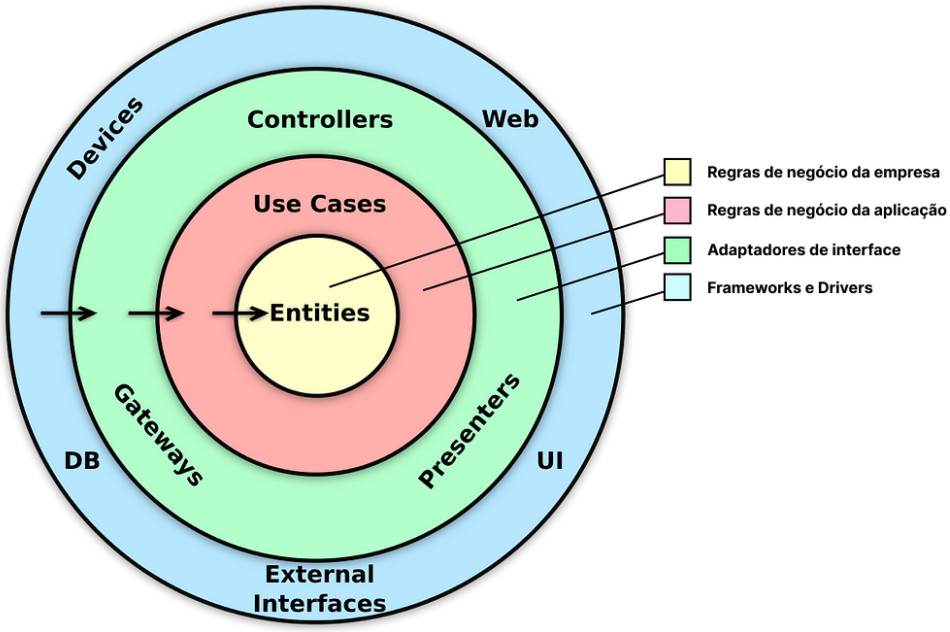
relacionadas (Özkan *et al.*, 2023). Esses princípios promovem uma compreensão mais profunda do problema e uma integração mais coesa entre os membros da equipe.

Além disso, o DDD introduz conceitos cruciais como Contexto, Modelo, Linguagem Ubíqua e Contexto Delimitado. O Contexto define o ambiente necessário para compreender um modelo, enquanto o Modelo é um sistema de abstrações que descreve aspectos do domínio. A Linguagem Ubíqua, por sua vez, é uma linguagem compartilhada usada por toda a equipe para garantir uma comunicação eficaz e coesa. O Contexto Delimitado estabelece limites conceituais que ajudam a manter a consistência do modelo e a evitar confusões (Özkan *et al.*, 2023). Com essas definições, o DDD permite um desenvolvimento mais alinhado ao modelo do negócio, promovendo uma colaboração eficaz e uma melhor compreensão entre todos os envolvidos.

2.3.2 Clean Architecture

Clean Architecture, ou Arquitetura Limpa, foi proposta por Robert C. Martin, também conhecido como "Uncle Bob", com o objetivo de integrar os melhores conceitos de arquiteturas já existentes, como a Arquitetura Hexagonal (*Ports and Adapters*), Arquitetura em Camadas e *Onion Architecture* (Martin, 2017). De acordo com Martin (2017), essa abordagem organiza sistemas em camadas concêntricas, cada uma com responsabilidades bem definidas, de forma que as dependências fluam sempre das camadas externas para as internas. O diagrama que representa a *Clean Architecture* pode ser visualizado na Figura 6.

Figura 6 – Arquitetura Limpa



Fonte: Adaptado de (Martin, 2017)

Os principais conceitos da *Clean Architecture* estão fundamentados nos princípios *SOLID*, que promovem sistemas modulares, flexíveis e de fácil manutenção (Martin, 2017). Esses princípios garantem que os sistemas desenvolvidos sejam independentes de *frameworks*, o que possibilita a troca de tecnologias externas sem impacto significativo no núcleo do sistema. Além disso, permitem a independência da Interface de Usuário (UI), possibilitando alterações na camada de apresentação sem afetar as regras de negócio. Também promovem a independência em relação ao banco de dados e tornam os sistemas mais fáceis de testar, pois cada camada possui responsabilidades bem definidas e isoladas (Martin, 2017).

De acordo com Martin (2017), no centro da *Clean Architecture* estão as Entidades, que representam as regras de negócio e abstrações fundamentais do sistema. Essas entidades são cercadas por Casos de Uso, que descrevem a lógica da aplicação. Esta camada é cercada por uma camada de adaptadores de interface, que tratam a comunicação entre a camada de casos de uso e a de serviços externos. Por fim, na camada mais externa estão os *frameworks* e *drivers*, que servem como mecanismos de entrada e saída de dados.

A implementação prática da *Clean Architecture* geralmente envolve estruturar o aplicativo em camadas distintas, cada uma com suas responsabilidades e dependências específicas. De acordo com Fadhlurrohman (2024), um *design* típico dessa arquitetura pode incluir as seguintes camadas:

- **Camada de Domínio**: Contém a lógica de negócios principal e os modelos de domínio do aplicativo, independentemente de quaisquer preocupações ou estruturas externas.
- **Camada de Infraestrutura**: Gerencia dependências externas, como bancos de dados, APIs de terceiros ou serviços. Inclui implementações de repositórios, objetos de acesso a dados e outros componentes relacionados à infraestrutura.
- **Camada de Aplicação**: Coordena as interações entre as camadas de domínio e infraestrutura, implementando casos de uso e regras de negócios sem estar vinculada a mecanismos de entrega específicos.
- **Camada de Apresentação**: Lida com preocupações de interface do usuário, como renderização de visualizações e manipulação de entrada do usuário. Ela se comunica com a camada de aplicação para recuperar dados e executar lógica de negócios.

Ao estruturar o aplicativo dessa maneira, a *Clean Architecture* promove uma separação clara de preocupações e impõe a inversão de dependência, onde módulos de alto nível não dependem de módulos de baixo nível, mas sim de abstrações (Fadhlurrohman, 2024).

De acordo com Martin (2017), a separação clara entre as camadas facilita a evolução do *software*, melhora a legibilidade do código e promove a reutilização de componentes. Além disso, a independência de agentes externos torna o sistema mais resiliente a mudanças tecnológicas. Essas características tornam a *Clean Architecture* uma escolha particularmente adequada para projetos de longa duração, onde mudanças de requisitos são frequentes.

2.3.3 Princípios SOLID

SOLID são cinco princípios da programação orientada a objetos que facilitam no desenvolvimento de softwares, tornando-os fáceis de manter e estender (Paixão, 2019). Os princípios *SOLID* orientam sobre a organização das funções e estruturas de dados em classes, bem como sobre a maneira como essas classes devem se relacionar entre si (Martin, 2017). De acordo com Martin (2017), embora os princípios tenham sido inicialmente associados à programação orientada a objetos, eles se aplicam a qualquer tipo de agrupamento de funções e dados. A seguir, são apresentados os cinco princípios *SOLID*:

- **Princípio da Responsabilidade Única (SRP)**: Um módulo ou classe deve ter uma única razão para mudar.
- **Princípio Aberto-Fechado (OCP)**: O *software* deve ser aberto para extensão, mas fechado para modificação.
- **Princípio da Substituição de Liskov (LSP)**: Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar a correção do programa.
- **Princípio da Segregação de Interfaces (ISP)**: Uma classe não deve ser forçada a implementar interfaces que não usa.
- **Princípio da Inversão de Dependência (DIP)**: Dependências de alto nível não devem depender de dependências de baixo nível; ambas devem depender de abstrações.

Esses princípios são aplicados no nível intermediário de *design*, ajudando a definir a estrutura e a interação entre classes e módulos (Martin, 2017). A aplicação correta dos princípios *SOLID* ajuda o desenvolvedor a escrever códigos mais limpos, diminuindo o acoplamento e facilitando refatorações (Paixão, 2019).

3 TRABALHOS RELACIONADOS

Foram encontrados na literatura alguns estudos que se relacionam com este trabalho. Nesta seção, esses estudos serão apresentados e comparados com o trabalho proposto.

3.1 *Sliceable Monolith: Monolith First, Microservices Later*

Montesi *et al.* (2021) propuseram uma metodologia de desenvolvimento para uma aplicação com arquitetura de microsserviços chamada *Sliceable Monolith*, que se assemelha ao monólito modular. Nesse sentido, o objetivo era manter a simplicidade do desenvolvimento de um monólito e automatizar o processo de decomposição, que era possível graças à linguagem Jolie e à ferramenta Jolie Slicer.

Os autores aplicaram a metodologia em uma aplicação de gerenciamento de estacionamentos privados que possuem estações de carregamento de veículos elétricos, cuja ideia é que os proprietários compartilhem suas estações por meio de incentivo financeiro. Assim sendo, o monólito foi desenvolvido e testado localmente e, em seguida, foi utilizada a ferramenta Jolie Slicer, que, com apenas uma linha de comando, separou todos os serviços em subdiretórios e criou o arquivo *docker compose* para a integração na nuvem.

3.2 *From a Monolithic Big Data System to a Microservices Event-Driven Architecture*

O trabalho proposto por Laigner *et al.* (2020) tinha como objetivo transformar um sistema monolítico de *Big Data* para gerenciamento de frota de caminhões em uma arquitetura de microsserviços orientada a eventos. O sistema original, desenvolvido para processar grandes volumes de dados, apresentava desafios relacionados à escalabilidade e manutenção devido à sua estrutura monolítica. O projeto visou melhorar a eficiência e a flexibilidade do sistema, abordando questões como escalabilidade horizontal e integração contínua, por meio da decomposição em microsserviços independentes. Esta abordagem também buscou aprimorar a capacidade de resposta do sistema e permitir uma melhor gestão dos dados em tempo real.

Para alcançar esse objetivo, os autores realizaram uma *Action Research* (pesquisa-ação), para investigar as razões que motivavam a adoção da arquitetura de microsserviços, definiram a nova arquitetura e documentaram as dificuldades e lições aprendidas. A pesquisa conduzida foi extremamente importante para os desenvolvedores entenderem o sistema existente e quais características deviam ser mantidas. Apesar de o sistema já existir, havia sido construído

por outra equipe, então os desenvolvedores que realizaram a migração não haviam completo domínio sobre o sistema. Ainda assim, o desenvolvimento do sistema foi iniciado diretamente na arquitetura de microsserviços ao invés de construir uma versão monolítica para adequação da equipe, o que gerou diversas dificuldades e aprendizados durante o trabalho. Embora não tenha sido usado, no fim do trabalho os autores recomendam a utilização da abordagem *Monolith First*, como aconselhado por Fowler, como forma de entender melhor os requisitos do sistema.

3.3 *Design and Implementation of Microservices System Based on Domain-Driven Design*

Fajar *et al.* (2020) propuseram a transformação de uma aplicação monolítica existente em microsserviços, utilizando os conceitos de DDD. O domínio da aplicação abordado é o *e-procurement*, que se refere a um sistema de contato, negociação e compra de suprimentos ou serviços, usado para gerenciar inventário, pedidos e fornecedores. A abordagem DDD foi utilizada para identificar e isolar os diferentes domínios e subdomínios dentro do sistema, facilitando a decomposição em microsserviços. O objetivo foi melhorar a modularidade, escalabilidade e flexibilidade da aplicação, alinhando a estrutura do sistema com as necessidades de negócios.

Do ponto de vista técnico, o trabalho envolveu a aplicação de práticas de DDD para definir limites de contexto e criar microsserviços que operam de maneira autônoma, mas coordenada. Foi implementado um *API Gateway* para gerenciar o tráfego de solicitações e coordenar a comunicação entre os microsserviços, além de utilizar uma solução de mensageria para assegurar uma comunicação assíncrona e eficiente entre os serviços.

3.4 Análise Comparativa

Assim como este trabalho, o trabalho proposto por Montesi *et al.* (2021) aborda o conceito de *Monolith First* como facilitador no desenvolvimento de microsserviços. Entretanto, o trabalho de Montesi *et al.* (2021) tem como objetivo apresentar uma nova metodologia de desenvolvimento de microsserviços, baseada na ferramenta Jolie Slicer e na linguagem de programação Jolie, que, de certa forma, se assemelha a construção de um monólito modular.

O trabalho de Laigner *et al.* (2020) tem como principal objetivo reportar a experiência na substituição de um sistema monolítico legado por um sistema que utiliza microsserviços. Nesse sentido, este trabalho é o que mais se aproxima do trabalho proposto, pois o trabalho proposto pretende avaliar o processo de desenvolvimento do sistema a partir da abordagem

Monolith First. Além disso, o sistema desenvolvido por Laigner *et al.* (2020) possui semelhanças na arquitetura que será desenvolvida no trabalho proposto, como a comunicação movida a eventos, utilizando um sistema de mensageria.

Por fim, o trabalho de (Fajar *et al.*, 2020) se assemelha ao trabalho proposto pela utilização do DDD para construção de microsserviços. Entretanto, os autores utilizaram essa abordagem para definir os domínios do sistema e delimitar cada serviço, enquanto o trabalho proposto utiliza ainda na etapa do sistema monolítico, com o intuito de desenvolver o monólito modular. Assim como o trabalho anterior e o trabalho proposto, o trabalho de (Fajar *et al.*, 2020) também utiliza comunicação assíncrona com sistema de mensageria, o que permite o desacoplamento dos serviços e aumenta a escalabilidade do sistema.

Dessa forma, o trabalho proposto combina as ideias apresentadas nos trabalhos analisados para estruturar sua abordagem. A adoção do monólito modular como etapa inicial do desenvolvimento é inspirada na proposta de Montesi *et al.* (2021), permitindo uma transição mais fluida para a arquitetura de microsserviços. Além disso, seguindo a abordagem de Laigner *et al.* (2020), este trabalho busca não apenas implementar a migração, mas também relatar e avaliar o processo, documentando os desafios e benefícios da estratégia adotada. Por fim, a definição dos domínios e delimitação dos serviços é baseada na aplicação de DDD, conforme sugerido por (Fajar *et al.*, 2020), sendo utilizada desde a fase monolítica para garantir uma modularização coerente e facilitar a migração posterior.

A relação entre os trabalhos citados e o trabalho proposto pode ser visualizada no Quadro 1.

Quadro 1 – Quadro comparativo entre os trabalhos relacionados e este trabalho

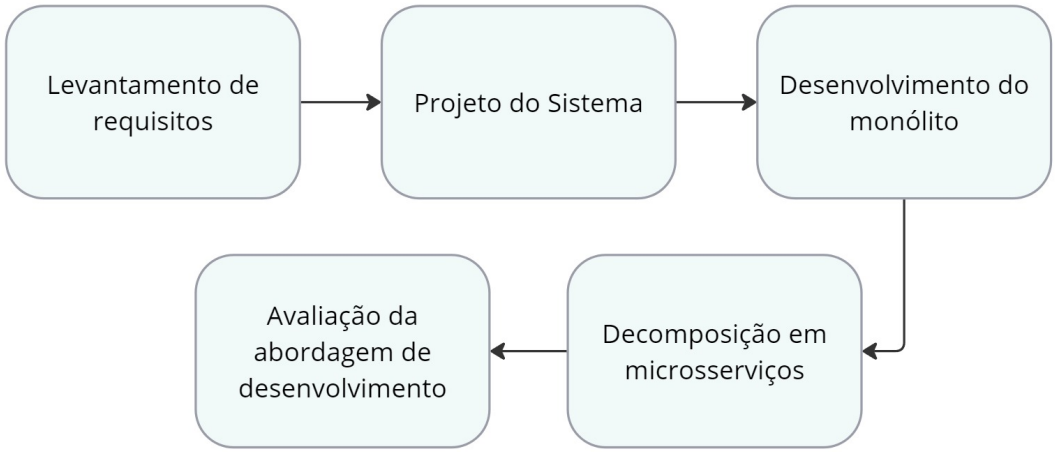
	Monolith first	Monólito modular	Utiliza DDD	Utiliza mensageria	Avalia o processo de desenvolvimento
Este trabalho	X	X	X	X	X
Montesi *et al.* (2021)	X	X			
Laigner *et al.* (2020)				X	X
Fajar *et al.* (2020)			X	X	

Fonte: Elaborado pelo autor

4 METODOLOGIA

Neste Capítulo, será apresentada a sequência de passos necessários para a realização do trabalho. A Figura 7 apresenta a sequência de passos que foi seguida.

Figura 7 – Passos para a realização do trabalho



Fonte: Elaborado pelo autor

A metodologia adotada neste trabalho segue a abordagem de estudo de caso, na qual um sistema foi desenvolvido e analisado com o objetivo de avaliar a transição de um monólito modular para uma arquitetura de microsserviços. O presente trabalho se caracteriza como uma pesquisa de caráter exploratório, pois busca compreender os desafios e benefícios da abordagem *Monolith First* para a migração de um sistema para microsserviços, documentando as experiências e dificuldades encontradas durante o desenvolvimento. Além disso, possui um caráter descritivo, uma vez que detalha as etapas do desenvolvimento e as decisões técnicas tomadas ao longo do processo.

Na Seção 5.1 será apresentado o primeiro passo da metodologia, que consiste no levantamento de requisitos, identificando as necessidades e definições iniciais do sistema. Em seguida, na Seção 5.2, será realizado o projeto do sistema, que envolve a criação de diagramas técnicos e a prototipação de interface. Após essa etapa, na Seção 5.3, inicia-se o desenvolvimento da versão monolítica da aplicação, que serviu como base para a posterior decomposição. Na Seção 5.4, é realizada a decomposição do sistema em microsserviços, juntamente com a integração de sistemas de mensageria para comunicação assíncrona. Por fim, na Seção 5.5, será detalhado o processo de avaliação da abordagem de desenvolvimento, documentando os desafios enfrentados e recomendações para projetos similares.

4.1 Levantamento de requisitos

O sistema que foi desenvolvido tem como objetivo a divulgação e aplicação de vagas para repúblicas estudantis. Nesse sentido, uma análise de requisitos foi conduzida de forma sistemática para garantir uma compreensão aprofundada das necessidades e expectativas dos usuários em relação à busca de moradias para estudantes. O processo foi dividido em três etapas:

- **Pesquisa e Análise de Soluções Existentes**: Inicialmente, foram investigadas soluções existentes na literatura e no mercado sobre plataformas e aplicativos para busca de moradias estudantis. Essa pesquisa inclui a avaliação de aplicativos móveis e plataformas *web*, analisando suas funcionalidades, usabilidade e pontos fortes e fracos. Essa análise é importante para compreender as melhores práticas e as lacunas existentes nos sistemas atuais.
- **Identificação de Necessidades e Funcionalidades**: Com base na análise das soluções existentes, foram identificadas as principais necessidades dos usuários e as funcionalidades desejadas para a nova aplicação. Isso inclui a consideração das demandas dos usuários finais e dos aspectos que poderiam melhorar a experiência de busca e gestão de moradias.
- **Definição de Requisitos**: Com as necessidades identificadas, foi criada uma lista de requisitos para a aplicação. Esses requisitos foram organizados em formato de histórias de usuário, que descrevem as funcionalidades do sistema a partir da perspectiva dos usuários finais. A utilização de histórias de usuário facilita a compreensão das expectativas dos usuários e assegura que as funcionalidades propostas atendam a suas necessidades de forma clara e prática.

Esse levantamento detalhado de requisitos fornece uma base sólida para o desenvolvimento do sistema, assegurando que a aplicação atenda às expectativas dos usuários de maneira eficaz e eficiente. A abordagem sistemática adotada garante que todos os aspectos relevantes foram considerados e que o sistema resultante estará alinhado com as necessidades reais dos usuários.

4.2 Projeto do sistema

Na fase de projeto do sistema, foram desenvolvidos artefatos cruciais para a implementação do sistema, consolidando a etapa anterior de levantamento de requisitos e preparando o terreno para o desenvolvimento. Esta etapa compreende a criação de um protótipo de alta

fidelidade, a elaboração dos diagramas de casos de uso, de classes e a decisão sobre as tecnologias a serem utilizadas.

O protótipo de alta fidelidade foi desenvolvido utilizando a ferramenta Figma, que proporciona uma representação visual detalhada das telas e funcionalidades do sistema. Esse protótipo permite uma avaliação precisa da usabilidade e do fluxo de interação, possibilitando ajustes e refinamentos antes do início efetivo do desenvolvimento. A criação deste protótipo envolve a definição de todos os elementos visuais e interativos, garantindo que o *design* esteja alinhado com as expectativas dos usuários.

Além disso, foram elaborados diagramas essenciais para a compreensão e implementação do sistema. Estes diagramas foram criados utilizando o Creately e o Miro:

- **Diagrama de Casos de Uso**: Descreve as interações entre os usuários e o sistema, identificando os principais casos de uso e as funcionalidades que o sistema deve suportar. Este diagrama fornece uma visão geral das operações e dos requisitos funcionais, ajudando a garantir que todas as necessidades dos usuários sejam abordadas.
- **Diagrama de Classes**: Representa a estrutura do sistema, mostrando as classes, seus atributos, métodos e as relações entre elas. Este diagrama é fundamental para o planejamento da arquitetura do sistema e para a definição das interações entre os diferentes componentes.

Os diagramas foram construídos com o objetivo de descrever de forma detalhada a estrutura e o comportamento do sistema, oferecendo uma visão clara da arquitetura e das relações entre os componentes. Esses diagramas são ferramentas cruciais para orientar a implementação e assegurar que a construção do sistema siga um plano bem definido.

Além disso, o desenvolvimento do sistema seguiu o DDD. Por esse motivo, nesta etapa são definidos os *Bounded Contexts* do sistema, permitindo a divisão do sistema em subdomínios, o que facilitou a separação dos microsserviços.

Por fim, foi realizada uma análise detalhada para a tomada de decisão sobre as tecnologias a serem utilizadas na implementação. Foram avaliadas várias opções de ferramentas e *frameworks*, considerando fatores como adequação às necessidades do sistema, compatibilidade, escalabilidade e facilidade de uso.

Com o projeto do sistema concluído, a próxima etapa é a implementação do sistema. Este planejamento detalhado garante que o desenvolvimento seja orientado por uma base sólida e bem estruturada, facilitando a construção eficiente e eficaz do sistema.

4.3 Desenvolvimento do monolíto

No desenvolvimento da aplicação, adotamos a abordagem *Monolith First*, iniciando com a construção de um monólito modular antes da migração para microsserviços. Para estruturar o monólito, utilizamos os conceitos do DDD, aplicados à definição dos módulos da aplicação. A separação foi feita considerando os limites naturais do domínio, buscando garantir que cada módulo encapsulasse um conjunto coeso de regras de negócio. Essa abordagem permitiu que os módulos fossem projetados desde o início como unidades independentes dentro do monólito, facilitando sua futura transição para microsserviços.

A implementação seguiu os princípios da *Clean Architecture*, organizando o código em quatro camadas principais. A camada de domínio concentrou as regras de negócio, garantindo que ficassem isoladas de detalhes técnicos. A camada de aplicação foi responsável pelos casos de uso, orquestrando a interação entre as regras de negócio e os demais componentes do sistema. A camada de apresentação expôs os endpoints da API, servindo como interface entre o sistema e os consumidores externos. Por fim, a camada de infraestrutura lidou com integrações externas, como a implementação de repositórios, bibliotecas de terceiros e demais serviços externos necessários ao funcionamento do sistema.

Com o monólito modular totalmente desenvolvido, foi possível planejar sua decomposição de forma estruturada, garantindo uma transição controlada para a arquitetura de microsserviços.

4.4 Decomposição em microsserviços

Neste estágio, realizamos a decomposição do sistema em microsserviços, seguindo um processo estruturado para garantir uma transição eficiente e organizada. O primeiro passo foi a definição dos serviços, analisando os limites de cada subdomínio com base no DDD e identificando quais funcionalidades deveriam ser separadas. Esse mapeamento foi essencial para garantir coesão e minimizar dependências entre os serviços.

A migração dos módulos do monólito para microsserviços foi realizada de forma incremental, garantindo que cada serviço fosse extraído e validado antes de iniciar a migração do próximo módulo. Para garantir a comunicação assíncrona e evitar acoplamento excessivo, integramos o Apache Kafka, permitindo a troca de mensagens entre serviços por meio de eventos.

Além disso, para manter a independência dos serviços e evitar consultas constantes

entre microsserviços, aplicamos uma estratégia de replicação de dados, garantindo que cada serviço tivesse as informações necessárias para operar autonomamente. No entanto, essa abordagem pode introduzir desafios como inconsistências entre os dados replicados, aumento da complexidade na sincronização e maior consumo de armazenamento. Para mitigar esses problemas, utilizamos o Apache Kafka como mecanismo de mensageria, garantindo que as mudanças nos dados fossem propagadas de forma assíncrona e confiável entre os serviços. Dessa forma, sempre que um novo usuário era criado no serviço de usuários, uma mensagem era publicada no Kafka, permitindo que outros serviços, como o de aplicações, mantivessem uma cópia atualizada da informação necessária para seu funcionamento.

Essa abordagem garantiu que a decomposição ocorresse de forma estruturada, permitindo avaliar a transição de forma controlada antes da conclusão do processo.

4.5 Avaliação da abordagem de desenvolvimento

A etapa de avaliação do trabalho visa analisar a eficácia da abordagem *Monolith First* no contexto do desenvolvimento de microsserviços. Essa fase foi composta por três principais atividades: análise crítica da implementação do monólito, revisão da transição para microsserviços e documentação dos aprendizados.

Primeiramente, foi conduzida uma análise crítica do monólito modular desenvolvido, identificando como a estrutura e modularização do sistema contribuem para a clareza do domínio e a preparação para a migração. Em seguida, foi realizada a revisão da transição do monólito para a arquitetura de microsserviços, avaliando a eficácia da decomposição, a integração de sistemas de mensageria e a coesão dos serviços resultantes. Por fim, foram documentados os principais aprendizados e dificuldades encontradas ao longo do processo, permitindo uma reflexão sobre a viabilidade da abordagem *Monolith First* e fornecendo recomendações para futuros projetos que optem por essa estratégia.

5 RESULTADOS

Neste Capítulo, serão apresentados os resultados dos passos descritos na seção de procedimentos metodológicos. Na Seção 5.1 são apresentados os requisitos do sistema. Na Seção 5.2 serão apresentadas as escolhas tecnológicas para o desenvolvimento. Na Seção 5.3 é apresentada a modelagem Linguagem de Modelagem Unificada (UML) do sistema. Na Seção 5.4 são apresentadas as decisões tecnológicas do projeto. Na Seção 5.5 é apresentado o protótipo de alta fidelidade da plataforma *web*. Na Seção 5.6 são apresentados os resultados referentes ao desenvolvimento do monólito modular. Na Seção 5.7 é apresentada a arquitetura em microsserviços desenvolvida. Na Seção 5.8 é apresentado o resultado alcançado em relação à comunicação síncrona da aplicação. Na Seção 5.9 é apresentado o resultado alcançado em relação à comunicação assíncrona que ocorre no sistema. Na Seção 5.10 é apresentado como a replicação de dados é tratada na aplicação. Por fim, na Seção 5.11 é apresentada a experiência de desenvolvimento e avaliação final da abordagem.

5.1 Análise da Concorrência

O sistema desenvolvido, intitulado *myRepublic*, assim como todos os concorrentes elencados, tem como objetivo desenvolver um sistema *web* com o intuito de facilitar a busca e oferta por vagas em repúblicas. Cada um desses sistemas apresenta diferentes abordagens para atender às necessidades de anunciantes e buscadores de vagas.

O *Student Housing* se destaca pela presença de um sistema de aplicação para vagas, permitindo que interessados manifestem diretamente seu interesse nos anúncios. O *Republic Search*, por sua vez, possui um algoritmo de recomendação que sugere vagas com base em preferências do usuário. Já o República Fácil integra a API do *Google Maps* para facilitar a visualização da localização das repúblicas e conta com um sistema de favoritos.

Além dessas diferenças, todos os sistemas analisados oferecem funcionalidades básicas, como gerenciamento de conta, criação e gestão de anúncios e busca com filtros personalizados. A usabilidade também varia entre as soluções, com alguns sistemas priorizando uma interface mais intuitiva e responsiva, enquanto outros carecem de otimizações para diferentes dispositivos.

A análise comparativa pode ser visualizada no Quadro 2, destacando as funcionalidades presentes em cada sistema avaliado.

Quadro 2 – Análise de concorrência

	Student Housing	*Republic Search*	República Fácil
Sistema de recomendação		X	
Geolocalização			X
Sistema de Favoritos			X
Gerenciamento de conta	X	X	X
Gerenciamento básico de anúncios	X	X	X
Filtros de busca	X	X	X
Interface agradável	X		
Aplicação para vagas	X		
Design responsivo			X
Sistema de Notificações			
Gerenciamento de anúncios facilitado			

Fonte: Elaborado pelo autor

5.2 Requisitos

Durante a etapa de levantamento de requisitos, foram consideradas as necessidades do público-alvo e a análise da concorrência. A partir dessas informações, foram definidos os requisitos essenciais que a aplicação deveria atender. Todas as funcionalidades foram mapeadas e documentadas em forma de histórias de usuário.

O sistema proposto, intitulado *myRepublic*, busca oferecer funcionalidades essenciais para facilitar tanto a busca quanto a oferta de vagas em repúblicas. Ele inclui o gerenciamento de conta, criação e gestão de anúncios, busca com filtros personalizados e uma interface responsiva e intuitiva. Além disso, se diferencia pela implementação de um sistema de notificações dentro da plataforma, permitindo que usuários sejam informados sobre mudanças nos anúncios, interações e outras ações relevantes.

Outro diferencial é o gerenciamento de anúncios facilitado, em que os anúncios são automaticamente pausados quando a lotação da república atinge seu limite. Isso reduz o trabalho manual do anunciante e melhora a precisão das informações disponíveis na plataforma. Além disso, o sistema permite a aplicação direta para vagas, similar ao *Student Housing*, tornando o processo mais rápido e eficiente para os interessados.

De acordo com os requisitos elencados na Tabela 1, o usuário anunciante deve poder

gerenciar seus anúncios, filtrá-los por *status*, aceitar interessados e ter seus anúncios pausados automaticamente com base na lotação de vagas. O usuário buscador pode realizar buscas de acordo com a localidade e aplicar para vagas. Além disso, visitantes podem visualizar anúncios, mas sem interação. Todos os usuários autenticados devem ter acesso ao gerenciamento de conta e à aba de notificações.

Vale ressaltar que, apesar de os requisitos apontarem diferentes perfis de usuários, o sistema não os separa formalmente. Todos os usuários podem tanto publicar anúncios quanto aplicar para vagas, garantindo uma experiência mais fluida.

Tabela 1 – Requisitos funcionais do sistema

Requisito	História de Usuário	Descrição
RF01	Gerenciar anúncios	Como anunciante, quero poder criar, visualizar, editar, pausar e excluir anúncios para que eu possa gerir meus anúncios conforme necessário.
RF02	Filtrar anúncios	Como anunciante, quero poder filtrar meus anúncios entre pausados e ativos, para que eu possa facilmente encontrar meus anúncios com base no *status*.
RF03	Aceitar interessados	Como anunciante, quero ter a opção de aceitar um interessado na vaga, para que eu possa confirma a aplicação e gerenciar a ocupação da república.
RF04	Pausar anúncio automaticamente	Como anunciante, quero que meu anúncio seja pausado automaticamente quando atingir o número máximo de pessoas, para que eu não precise gerenciar o *status* manualmente.
RF05	Aplicar para vaga	Como buscador, quero aplicar para uma vaga e enviar uma mensagem, para que eu possa demonstrar meu interesse diretamente ao anunciante.
RF06	Acessar informações sem *login*	Como visitante, quero visualizar informações de repúblicas sem estar logado, para que eu possa explorar o sistema e avaliar as opções disponíveis.
RF07	Gerenciamento de conta	Como usuário logado, gostaria de uma seção de configurações de conta, para que eu possa gerenciar minhas informações pessoais.
RF08	Receber notificações	Como usuário logado, gostaria de receber notificações sobre interações no sistema, para que eu esteja sempre informado de atividades relevantes.

Fonte: Elaborado pelo autor

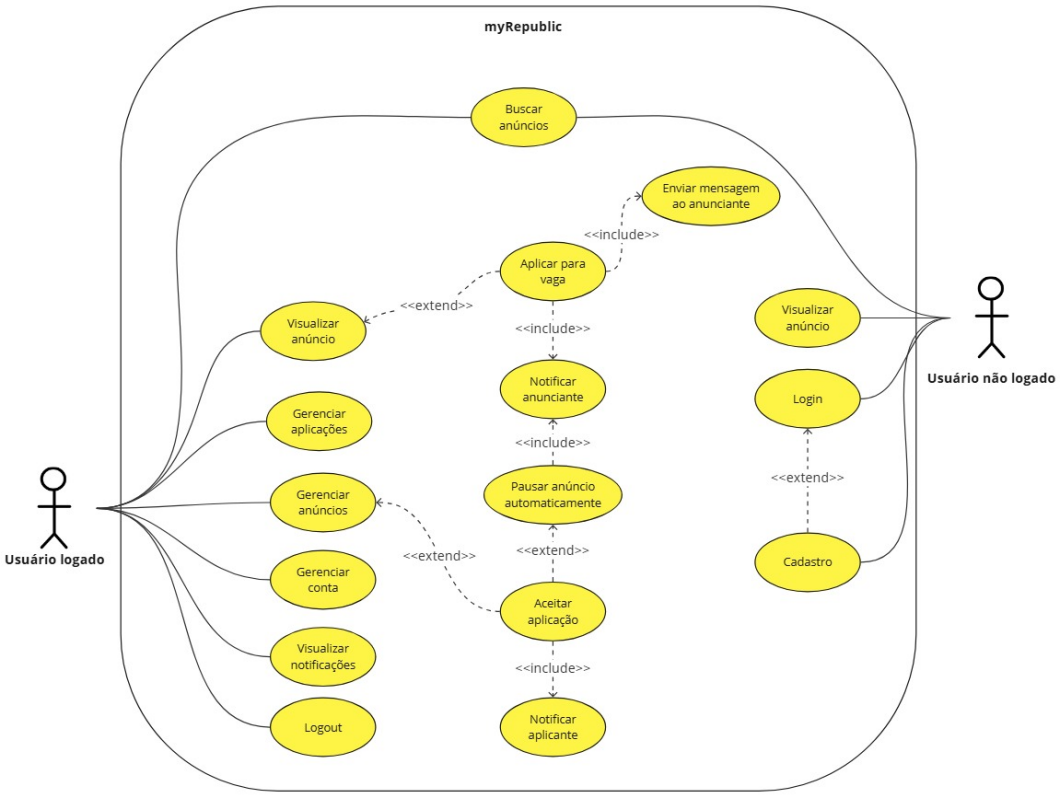
5.3 Modelagem

Posteriormente, foi realizada a etapa de projeto do sistema, que se inicia com a modelagem UML. Nesse sentido, foram criados os diagramas de casos de uso e de classes, para

representar o que o sistema deve ter e como cada componente do sistema deve se comportar para alcançar o objetivo da plataforma.

O diagrama de casos de uso pode ser visualizado na Figura 8 e representa o que cada tipo de usuário pode fazer no sistema. Nessa perspectiva, o sistema possui dois atores, o usuário autenticado e o usuário não autenticado. O usuário autenticado tem acesso a todas as funcionalidades do sistema, como notificações, configurações, gerenciamento de anúncios, busca de anúncios, aplicar para vagas e outras funcionalidades complementares. Por outro lado, o usuário não autenticado também pode usar o sistema, porém com acesso limitado, podendo apenas buscar anúncios e visualizar as informações.

Figura 8 – Diagrama de casos de uso do sistema

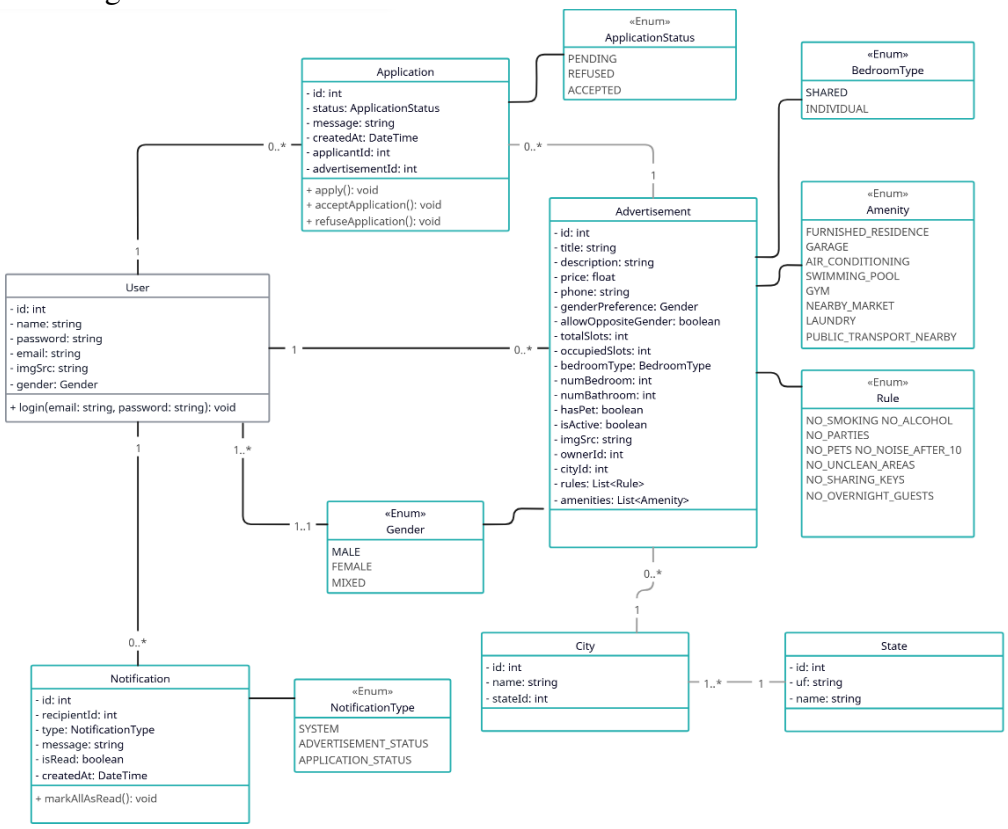


Fonte: Elaborado pelo autor

O diagrama de classes pode ser visualizado na Figura 9 e representa como cada parte do sistema deve se comportar. Neste aspecto, temos como principais entidades da nossa aplicação os usuários, os anúncios, as aplicações e as notificações. Ademais, temos classes complementares e enumeradores para representar certas informações na aplicação, como o tipo de notificação, o gênero do usuário e gênero preferencial no anúncio, tipo do quarto, *status* da aplicação, regras, comodidades, cidade e estado.

O diagrama de classes representa também como as entidades do sistema se relacio-

Figura 9 – Diagrama de classes sistema

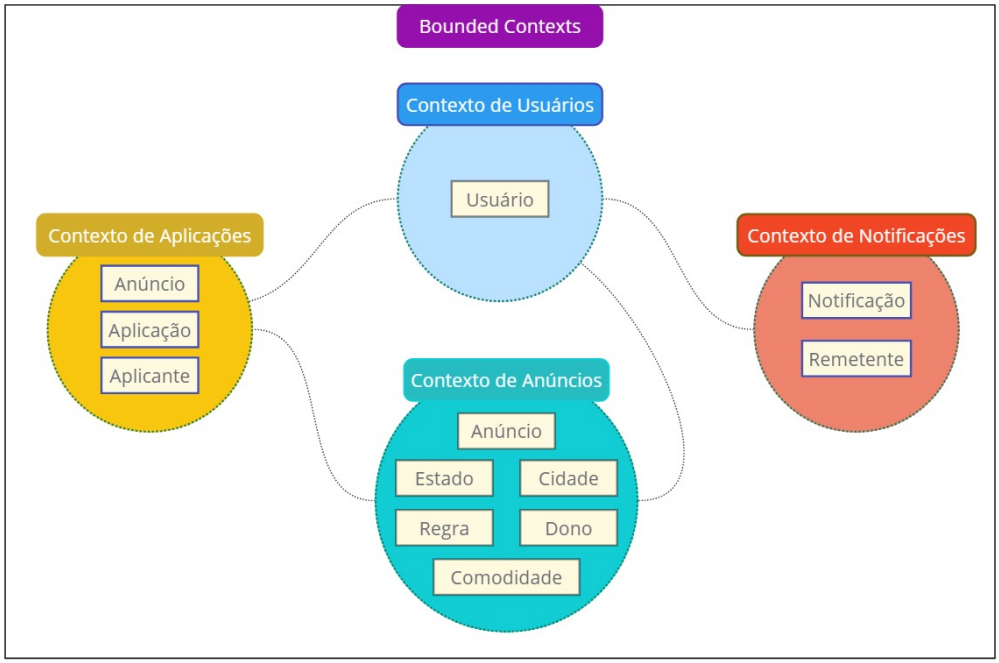


Fonte: Elaborado pelo autor

nam entre si. Nesse sistema, o usuário pode ser tanto um anunciante, como um aplicante, então ele possui um relacionamento de um para muitos com os anúncios, que representa os anúncios criados pelo usuário, e uma relação de um para muitos com as aplicações. Anúncios também possuem uma relação de um para muitos com aplicações, modelando o sistema de aplicações do sistema. Além disso, cada usuário pode ter muitas notificações e apenas um gênero. Cada anúncio pode ter muitas regras e comodidades, mas apenas um gênero preferencial, um tipo de quarto e uma cidade, que por sua vez, possui um estado associado. Por fim, as notificações devem ter apenas um tipo de notificação e aplicações devem ter um *status*.

Além disso, foram definidos os *Bounded Contexts* do sistema, que podem ser visualizados na Figura 10. A definição dos *Bounded Contexts* permite a definição de cada subdomínio, que posteriormente virarão microsserviços independentes. Temos quatro contextos mapeados: usuários, anúncios, aplicações e notificações. Cada subdomínio possui seu próprio conjunto de identidades e sua própria linguagem ubíqua, onde mesmas entidades podem ser chamadas por termos diferentes. O contexto de aplicações trabalha com dados de usuários, mas neste contexto são reconhecidos como aplicantes. Da mesma forma, no contexto de notificações, usuários são tratados como remetentes e, no contexto dos anúncios, usuários são tratados como donos.

Figura 10 – *Bounded Contexts* do sistema



Fonte: Elaborado pelo autor

5.4 Tecnologias

Durante a fase de projeto do sistema, foram definidas as tecnologias que serão utilizadas no projeto, cujas principais estão apresentadas na Figura 11. Este trabalho utilizará como principal tecnologia o Nest.js, um *framework* para Node.js, para desenvolvimento do sistema monolítico e nos futuros microsserviços. Esta escolha foi devido ao suporte da tecnologia ao desenvolvimento de microsserviços e pelo *framework* organizar o ambiente em módulos, se alinhando com a metodologia utilizada no desenvolvimento.

Além disso, o sistema utiliza Kafka para a implementação do sistema de mensageria, garantindo comunicação assíncrona eficiente e desacoplamento entre os serviços. Para comunicação síncrona entre o API Gateway e os serviços, será utilizado gRPC. O gRPC foi escolhido devido à sua alta performance e capacidade de lidar com chamadas rápidas e eficientes, especialmente em sistemas distribuídos como o adotado nesta arquitetura. O gRPC permite a comunicação entre os microsserviços de maneira mais eficiente que o tradicional HTTP/REST, já que utiliza o protocolo HTTP/2, o que traz benefícios como multiplexação de requisições, compressão de cabeçalhos e transmissão de dados binários, garantindo menor latência nas chamadas e maior escalabilidade.

O sistema também utilizará Docker para a orquestração de todos os serviços da

aplicação. O uso do Docker garante que todos os ambientes de desenvolvimento, teste e produção sejam consistentes, evitando problemas de configuração e simplificando o processo de implantação.

Além das tecnologias citadas, o projeto também utilizará:

- Next.js no *front-end*, escolhido por sua capacidade de renderização híbrida (SSR e SSG), otimização de desempenho e boa integração com backends em Node.js;
- TailwindCSS, um *framework* de estilização que permite a criação de interfaces modernas e responsivas de maneira produtiva, reduzindo a necessidade de arquivos CSS adicionais;
- Prisma para manipulação dos bancos de dados, proporcionando um ORM moderno, tipado e otimizado para uso com TypeScript e Nest.js;
- Git para controle de versão, garantindo rastreamento de mudanças no código.
- Amazon S3, um serviço de armazenamento em nuvem da AWS, utilizado para armazenar arquivos da aplicação de forma escalável, segura e com alta disponibilidade.

Figura 11 – Principais tecnologias utilizadas

Fonte: Elaborado pelo autor

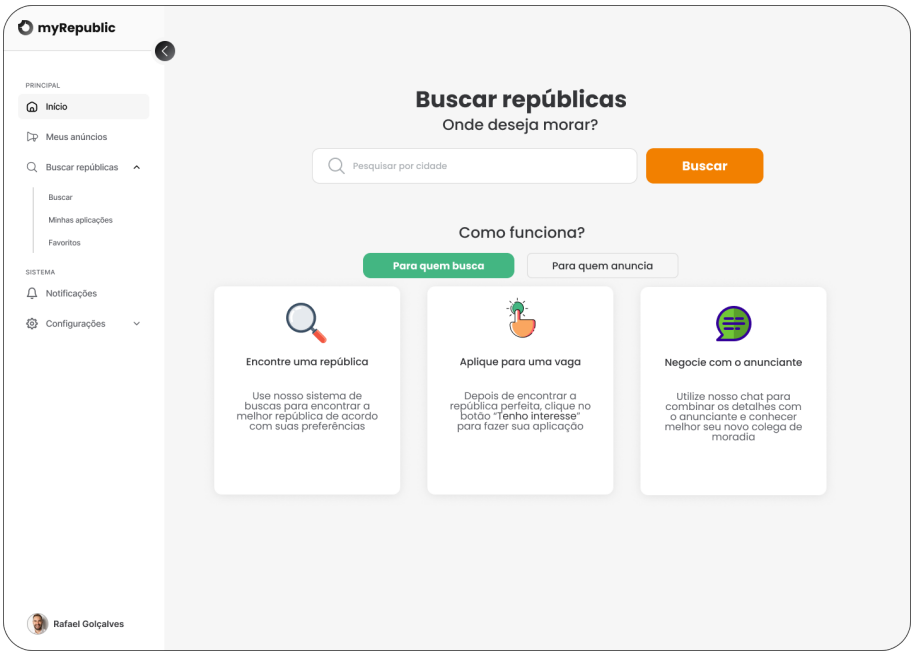
5.5 Protótipo de alta fidelidade

Para finalizar a etapa de projeto do sistema, foi criado um protótipo de alta fidelidade do sistema. Nesse sentido, o protótipo tem como objetivo definir as telas da aplicação, tanto para consolidar as funcionalidades elencadas, quanto para guiar o desenvolvimento do sistema.

A primeira tela da aplicação é a tela de início e está representada na Figura 12. Nessa tela, o usuário pode realizar uma busca de repúblicas por localidade e também apresenta um breve tutorial de como a plataforma funciona, tanto da visão de quem busca, quanto da visão de quem anuncia.

Começando pelas telas de quem anuncia, a Figura 13 apresenta a listagem dos anúncios do usuário na aba "Meus anúncios". Nessa tela, o usuário pode gerenciar seus anúncios,

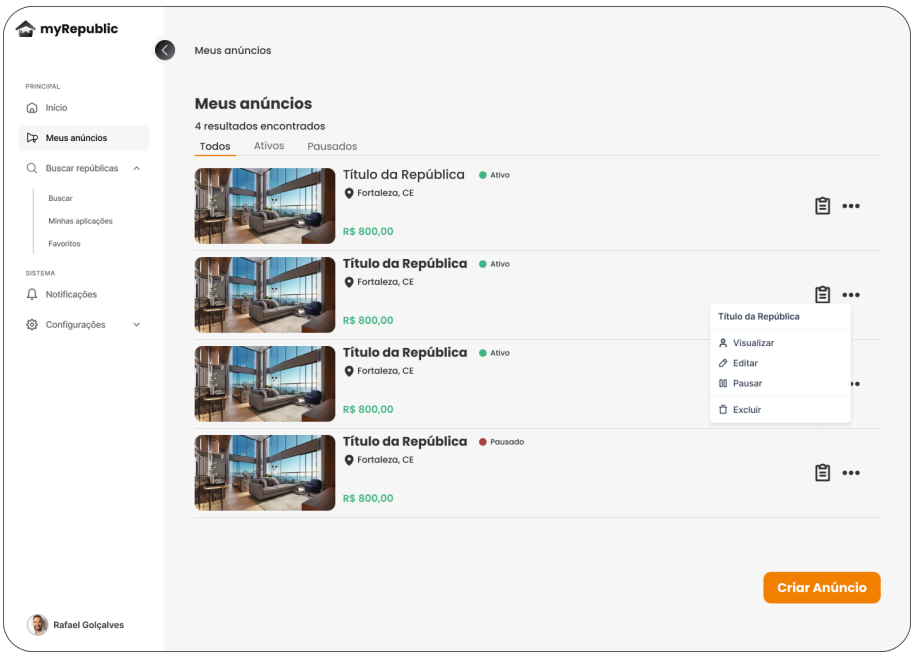
Figura 12 – Tela inicial



Fonte: Elaborado pelo autor

abrir a seção de aplicações e criar um novo anúncio. O usuário também tem a possibilidade de filtrar os anúncios por *status*.

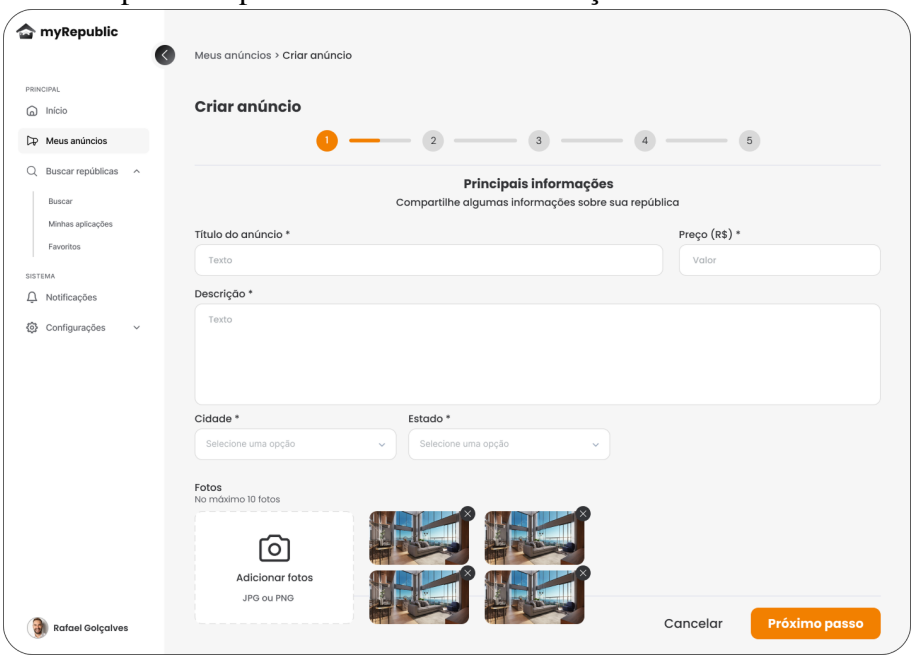
Figura 13 – Tela com anúncios do usuário



Fonte: Elaborado pelo autor

Durante a criação de um anúncio, é apresentado ao usuário um formulário de múltiplas etapas. A primeira etapa pode ser visualizada na Figura 14 e o usuário preenche as principais informações do anúncio, como título, descrição, preço, estado, cidade e fotos.

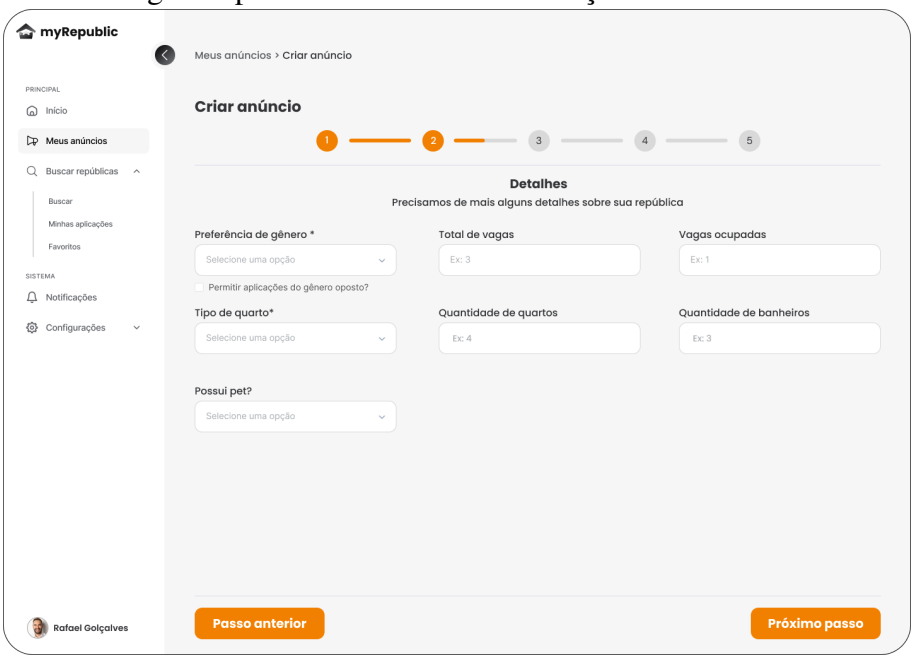
Figura 14 – Tela do primeiro passo do formulário de criação de anúncio



Fonte: Elaborado pelo autor

A Figura 15 representa o segundo passo do formulário de criação de anúncio. Neste passo, o usuário irá preencher os detalhes da república, como a preferência de gênero, o total de vagas, a quantidade de vagas ocupadas, tipo de quarto, quantidade de quartos, quantidade de banheiros e se a república possui um animal de estimação.

Figura 15 – Tela do segundo passo do formulário de criação de anúncio

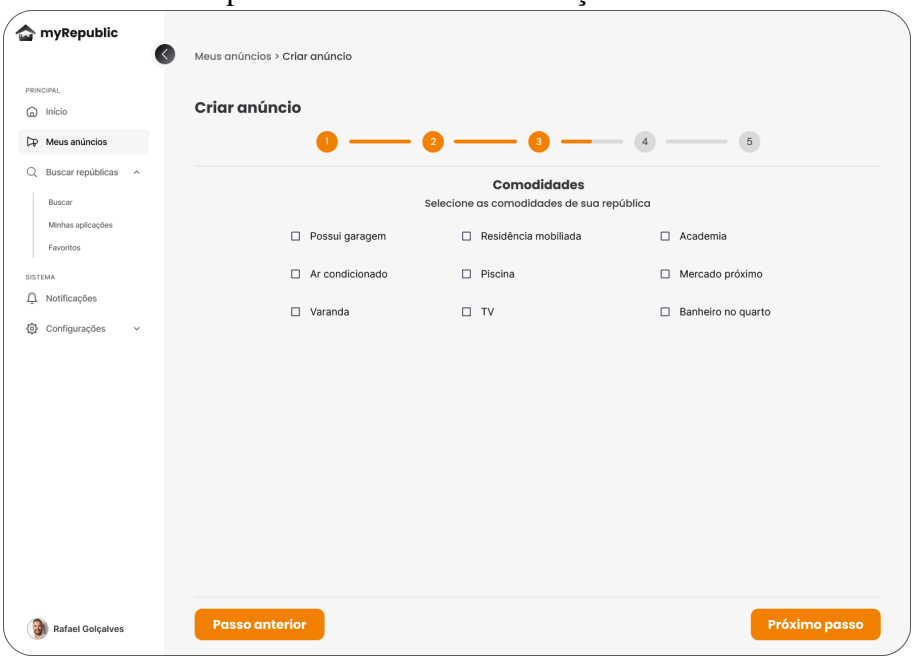


Fonte: Elaborado pelo autor

No terceiro passo, representado na Figura 16, o usuário preenche as comodidades

da república. Essa tela possui um conjunto de caixas e o usuário marca apenas aquelas que se encaixam com seu anúncio, podendo também não preencher nenhuma.

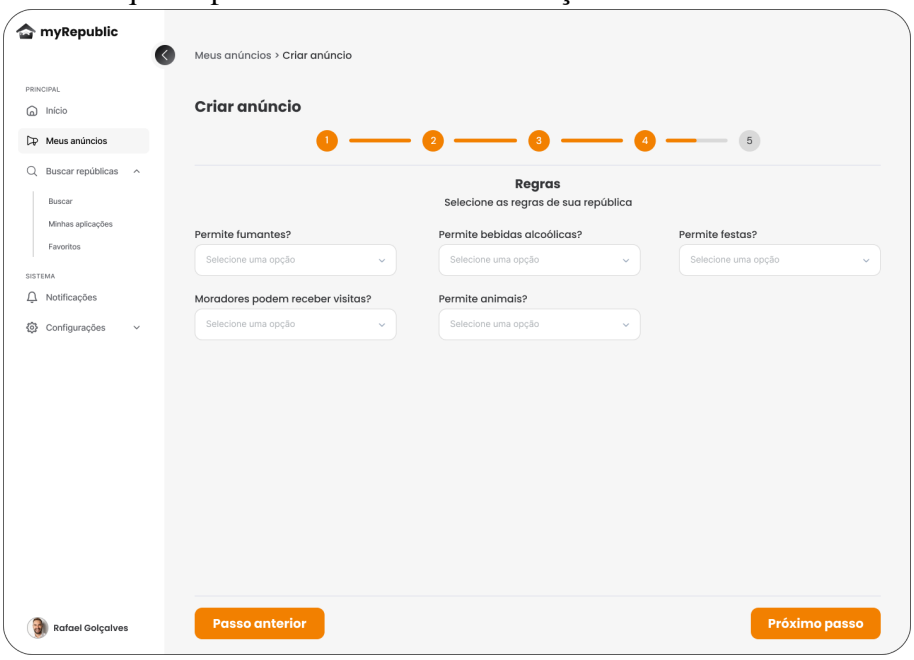
Figura 16 – Tela do terceiro passo do formulário de criação de anúncio



Fonte: Elaborado pelo autor

No quarto passo, ilustrado na Figura 17, o usuário irá definir as regras da república, como a permissão de fumantes, de bebidas alcoólicas, de visitas, festas e animais de estimação.

Figura 17 – Tela do quarto passo do formulário de criação de anúncio

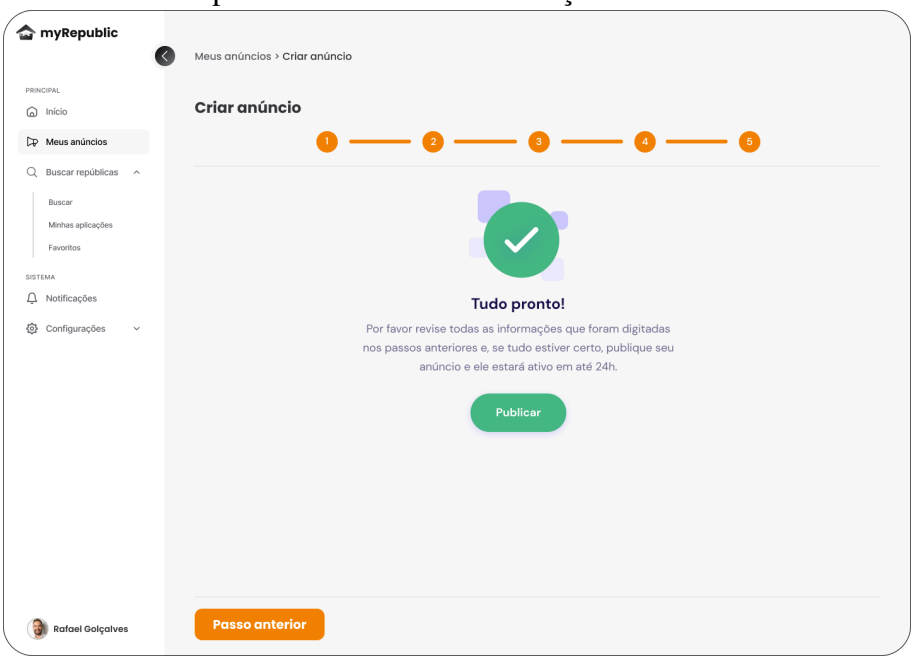


Fonte: Elaborado pelo autor

Por fim, o último passo do formulário, representado na Figura 18, apresenta apenas

uma confirmação para o usuário de que o formulário foi completamente preenchido, também o alertando para que verifique as informações preenchidas antes de publicar. Dessa forma, encerram-se as telas referentes ao usuário anunciante.

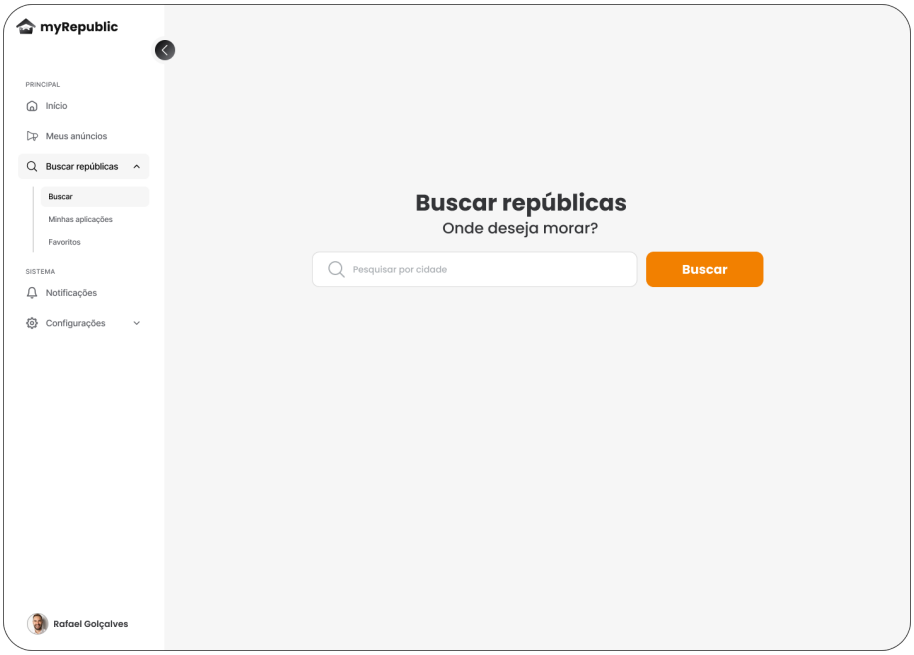
Figura 18 – Tela do último passo do formulário de criação de anúncio



Fonte: Elaborado pelo autor

Agora, serão apresentadas as telas referentes ao usuário que busca uma república, que estão contidas no menu "Buscar repúblicas".

Figura 19 – Tela de buscar repúblicas por localidade

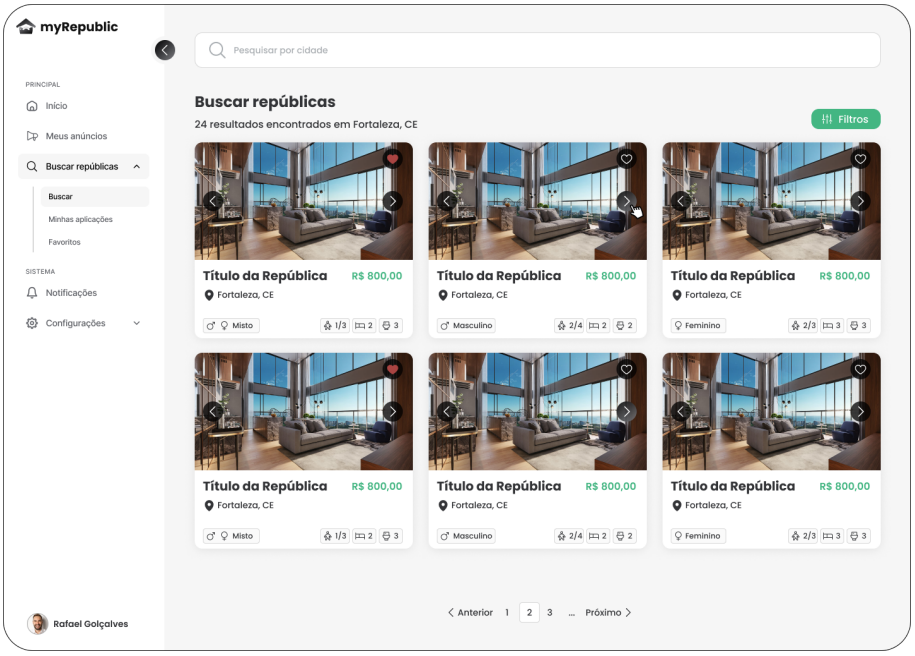


Fonte: Elaborado pelo autor

Primeiramente, ao acessar este menu, o usuário irá se deparar com a tela representada na Figura 19, que pede que o usuário digite a localidade em que ele deseja procurar uma república. Ao digitar, o campo irá exibir uma lista de possíveis localidades que possuem o texto que o usuário está digitando, com o objetivo de facilitar a busca.

Ao realizar a busca, o usuário será redirecionado para a tela apresentada na Figura 20, onde são exibidos os anúncios para aquela localidade em uma grade responsiva e com um sistema de paginação, onde o usuário pode navegar e visualizar os anúncios.

Figura 20 – Tela resultado de busca por localidade

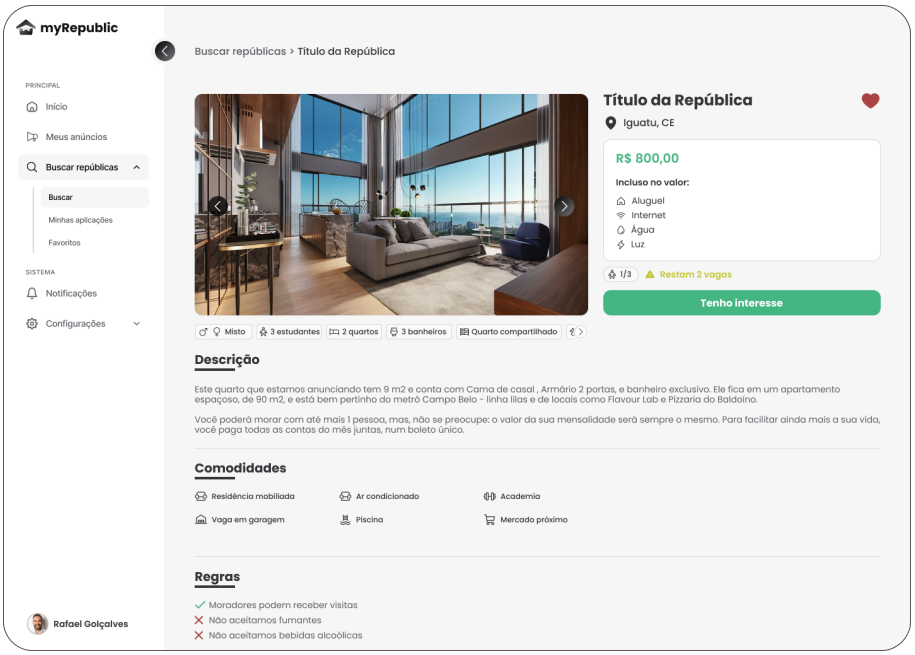


Fonte: Elaborado pelo autor

Ao clicar em um anúncio, o usuário será redirecionado para a tela representada na Figura 21, onde ele pode verificar as informações da república, como título, valores, regras, comodidades, imagens, entre outros detalhes. Além disso, o usuário pode realizar a aplicação para a vaga, enviando uma mensagem para o d0no do anúncio.

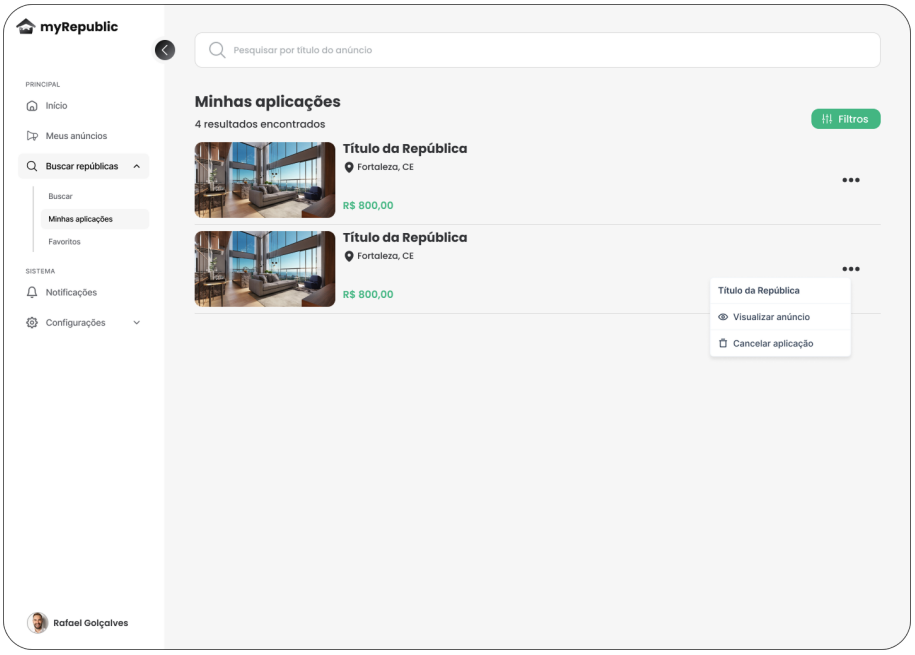
Por fim, ao realizar uma aplicação, o usuário pode visualizar suas aplicações no menu "Minhas aplicações"dentro de "Buscar repúblicas". Esta tela está representada na Figura 22 e é semelhante à tela de gerenciamento de anúncios. Entretanto, as opções dos anúncios são apenas de visualizar ou de cancelar a aplicação.

Figura 21 – Tela detalhes do anúncio



Fonte: Elaborado pelo autor

Figura 22 – Tela aplicações do usuário



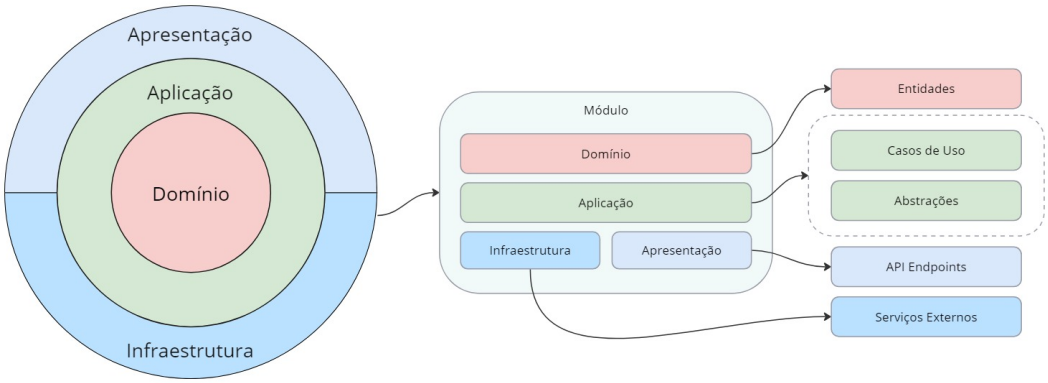
Fonte: Elaborado pelo autor

5.6 Monólito modular

O sistema foi construído utilizando os conceitos de DDD e dividido em módulos, os quais foram desenvolvidos utilizando o padrão de arquitetura *Clean Architecture*. A estrutura dos módulos pode ser visualizada na Figura 23.

O sistema foi composto por cinco módulos: Autenticação, usuários, anúncios,

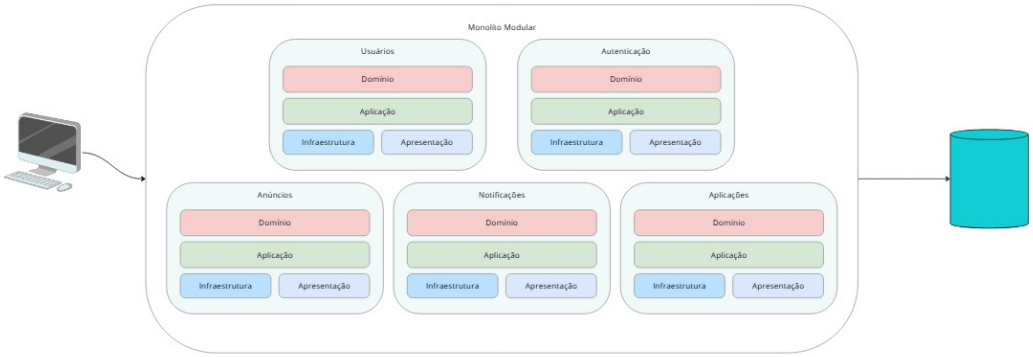
Figura 23 – Estrutura dos módulos da aplicação



Fonte: Elaborado pelo autor

aplicações e notificações. A divisão do sistema em módulos foi pensada para promover uma maior organização e facilitar a evolução da aplicação, já com a visão de uma futura transição para microsserviços. Cada módulo é responsável por uma funcionalidade específica e contém suas próprias lógicas e camadas de persistência, com a comunicação entre eles sendo feita de forma desacoplada através de interfaces bem definidas. O esquema do monólito modular do sistema está representado na Figura 24, sendo formado por cinco módulos e se comunicando com um único banco de dados.

Figura 24 – Monólito modular do sistema



Fonte: Elaborado pelo autor

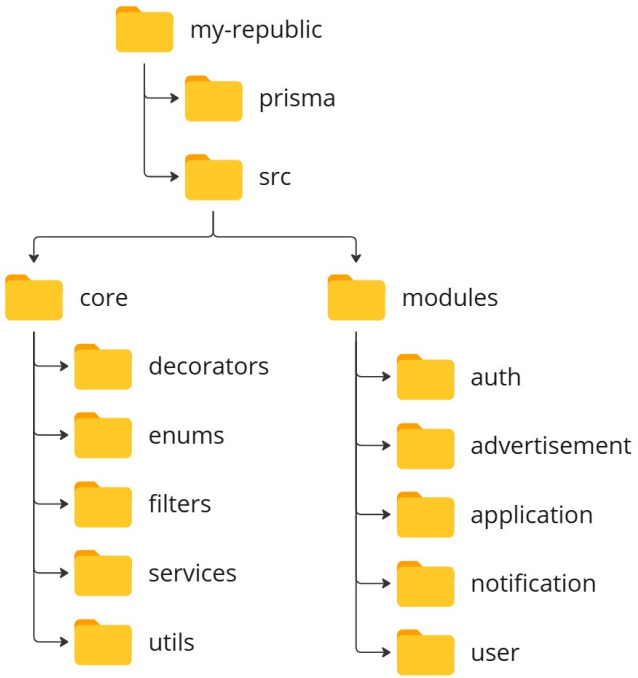
- **Módulo de Autenticação**: Responsável por gerenciar o processo de *login*, registro de usuários, autenticação de credenciais e emissão de *tokens* de acesso. Ele é crucial para garantir a segurança da aplicação e será um ponto central de interligação com os outros módulos, uma vez que a autenticação é necessária na maior parte das interações do sistema.
- **Módulo de Usuários**: Gerencia os dados dos usuários, incluindo seu cadastro e gerenciamento de perfil. Ele interage diretamente com os outros módulos para fornecer informações personalizadas aos usuários e garantir que as ações sejam associadas ao usuário correto

dentro do sistema.

- **Módulo de Anúncios**: Responsável por permitir a criação, exibição e gerenciamento de anúncios. Ele também contém toda a lógica de negócio referente a localização e busca.
- **Módulo de Aplicações**: Lida com o gerenciamento das aplicações dos anúncios. Este módulo envolve interações complexas com os módulos de usuários e anúncios e, por esse motivo, faz sentido que seja um módulo a parte.
- **Módulo de Notificações**: Responsável por todo o gerenciamento de notificações dentro da aplicação.

A estrutura de pastas do sistema reflete a estrutura descrita anteriormente. A Figura 25 representa a estrutura de pastas do sistema como um todo. No nível mais alto existe uma pasta *prisma*, com os arquivos relacionados ao banco de dados, e uma pasta *src*, com o restante da aplicação. Dentro da pasta *src* existe uma pasta *core*, com componentes que são utilizados em diferentes módulos, e uma pasta *modules*, com a implementação de cada um dos módulos do sistema.

Figura 25 – Estrutura de pastas do sistema

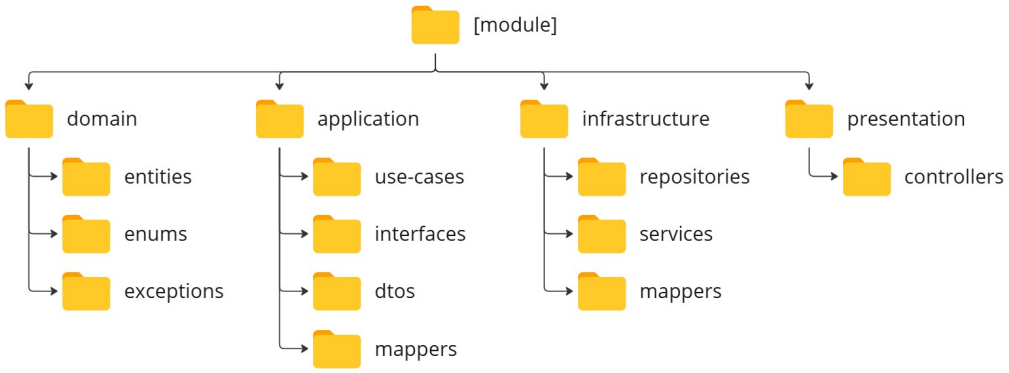


Fonte: Elaborado pelo autor

A estrutura de pastas dos módulos é ilustrada na Figura 26. Nela, a pasta *domain* contém as entidades, exceções e enumeradores; a pasta *application* agrupa os casos de uso,

abstrações, *DTOs* e *mappers*; a pasta *infrastructure* abriga os repositórios, serviços externos e *mappers*; e, por fim, a pasta *presentation* reúne os controladores.

Figura 26 – Estrutura de pastas dos módulos



Fonte: Elaborado pelo autor

A arquitetura modular escolhida permite que, à medida que o sistema cresce, os módulos possam ser facilmente desacoplados e convertidos em microsserviços independentes. A definição clara dos limites de cada módulo já contribui para uma transição mais tranquila para uma arquitetura distribuída no futuro.

Essa abordagem também facilita a manutenção e evolução do sistema, pois mudanças em um módulo podem ser feitas de forma isolada, sem causar impacto direto nas demais funcionalidades, o que é uma característica desejável tanto em sistemas monolíticos quanto em microsserviços.

Além disso, a escolha da *Clean Architecture* isola as regras de negócio do sistema de serviços externos, o que facilita a troca desses serviços, a manutenção e, consequentemente, a migração do sistema. Um exemplo disso é a funcionalidade de *login* do sistema, que utiliza uma biblioteca para geração de *tokens JSON Web Token* (JWT). Para isolar o caso de uso do serviço externo, são criadas interfaces, onde o serviço implementa a interface e o caso de uso depende apenas da interface, injetando a implementação.

Código-fonte 1 – Caso de Uso para *Login*

```
1 @Injectable()
2 export class LoginUseCase {
3   constructor(private tokenService: TokenService) {}
4
```

```
5   async execute(authUserDto: AuthUserDto): Promise<
        AuthResponseDto> {
6     const payload: PayloadDto = {
7       sub: authUserDto.id,
8       name: authUserDto.name,
9       email: authUserDto.email,
10      gender: authUserDto.gender,
11      imgSrc: authUserDto.imgSrc,
12    };
13
14    const token = this.tokenService.generateToken(payload);
15
16    return { user: authUserDto, access_token: token };
17  }
18 }
```

Código-fonte 2 – Interface para Serviço de *Token*

```
1 export abstract class TokenService {
2   abstract generateToken(payload: PayloadDto): string;
3 }
```

Código-fonte 3 – Implementação da interface

```
1 @Injectable()
2 export class JwtTokenService implements TokenService {
3   constructor(private jwtService: JwtService) {}
4
5   generateToken(payload: PayloadDto): string {
6     const token = this.jwtService.sign(payload);
7
8     return token;
```

```
    }
}
```

Os códigos do *front-end* [1] e do *back-end* [2] do sistema estão disponibilizados em repositórios públicos para qualquer um que tenha interesse em analisar o código na íntegra.

5.7 Arquitetura de Microsserviços

Durante a construção dos microsserviços, cada módulo da aplicação original foi migrado para um serviço independente, com exceção do módulo de autenticação, que permaneceu centralizado no *API Gateway*, responsável por atuar como ponto de entrada do sistema. Como resultado, a arquitetura final do sistema é composta por um *API Gateway*, quatro microsserviços independentes, cada um com seu próprio banco de dados, e um agente de mensagens (Kafka) para o disparo de eventos. A comunicação entre o cliente e o *API Gateway* ocorre por meio de requisições HTTP, enquanto a comunicação entre o *API Gateway* e os microsserviços é realizada via gRPC, garantindo maior eficiência e desempenho nas chamadas internas. Para evitar acoplamento entre os microsserviços, foi adotada uma abordagem que evita comunicações síncronas diretas entre eles. Em vez disso, optou-se por replicar os dados essenciais para cada serviço e utilizar o Kafka para sincronização assíncrona. A Figura 27 ilustra a arquitetura resultante.

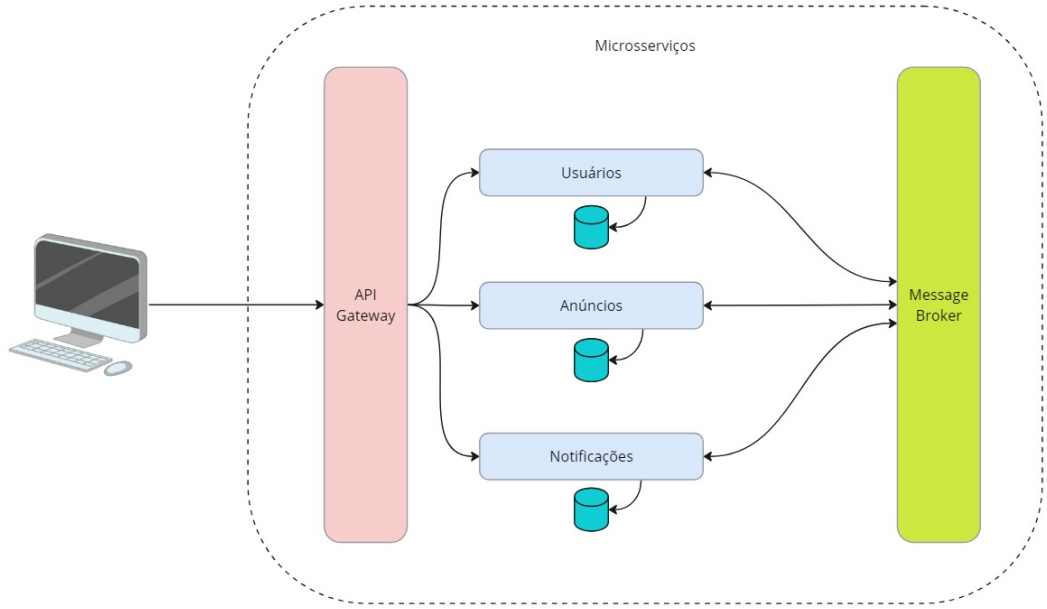
Para viabilizar a implantação dos microsserviços e facilitar a orquestração dos serviços, foi utilizado o Docker com um arquivo *docker-compose.yml*. Esse arquivo define todos os serviços necessários, incluindo o *API Gateway*, os microsserviços, o Kafka para comunicação assíncrona e o Zookeeper para gerenciar o Kafka.

5.8 Comunicação Síncrona

No sistema, grande parte das requisições acontece de forma síncrona. Para maximizar a performance do sistema, utilizamos gRPC para comunicação entre *API Gateway* e microsserviços. No contexto desta arquitetura, o *API Gateway* atua como intermediário entre o cliente e os microsserviços, utilizando gRPC para comunicar-se com os microsserviços internos de forma eficaz. Embora o modelo de microsserviços em si incentive a comunicação assíncrona

[1] https://github.com/pedrogrigorio/my-republic-frontend
[2] https://github.com/pedrogrigorio/my-republic-backend

Figura 27 – Arquitetura de Microsserviços



Fonte: Elaborado pelo autor

para reduzir o acoplamento entre os serviços, algumas operações exigem chamadas síncronas, como a recuperação de dados essenciais ou a execução de ações que precisam ser realizadas em tempo real.

Um exemplo de uso do gRPC neste sistema ocorre quando o *API Gateway* recebe uma requisição HTTP de um cliente para obter os detalhes de um usuário. O *API Gateway* então realiza uma chamada síncrona via gRPC para o microsserviço responsável pelos usuários, buscando as informações necessárias para devolver ao cliente. Para a comunicação ser possível, ambas as partes da comunicação devem conhecer as interfaces de comunicação, que no contexto do gRPC são definidas utilizando *Protocol Buffers* (Proto) por meio de um arquivo *.proto*, como o exemplo abaixo.

Código-fonte 4 – Exemplo de arquivo *.proto*

```
1 syntax = "proto3";
2
3 package user;
4
5 service UserService {
6   rpc getUserById (GetUserByIdRequest) returns (
      UserResponse);
7 }
```

```
8
9  message GetUserByIdRequest {
10   int32 id = 1;
11 }
12
13 message User {
14   int32 id = 1;
15   string name = 2;
16   string email = 3;
17   string imgSrc = 4;
18   string gender = 5;
19 }
20
21 message UserResponse {
22   User user = 1; // Dados do usu rio
23 }
```

Fonte: Elaborado pelo autor

Código-fonte 5: Endpoint no API-Gateway

```
1  @Get(':id')
2  async getUser(@Param('id') userId: string) {
3    const id = parseInt(userId);
4    const { user } = await firstValueFrom(
5      this.userService.getUserById({ id }),
6    ).catch((e) => {
7      throw new RpcException(e);
8    });
9
10   return user;
11 }
```

Fonte: Elaborado pelo autor

Código-fonte 6: Ponto de entrada do microsserviço de usuários

```
@GrpcMethod('UserService', 'getUserById')
async getUser(data: any) {
  try {
    const user = await this.getUserById.execute(data.id);
    return { user };
  } catch (error) {
    throw GrpcExceptionHandler.handleError(error);
  }
}
```

Fonte: Elaborado pelo autor

5.9 Comunicação Assíncrona

Além da comunicação síncrona via gRPC, os microsserviços também se comunicam de forma assíncrona através do *Apache Kafka*. Um exemplo prático dessa comunicação ocorre quando um anúncio atinge o limite de vagas disponíveis. O fluxo do evento funciona da seguinte maneira:

- O microsserviço de aplicações recebe a solicitação de aplicação de um usuário em um anúncio.
- Ao aceitar a aplicação, o serviço dispara um evento chamado *application.accepted* para o *Kafka*.
- O microsserviço de anúncios consome esse evento e incrementa a contagem de vagas preenchidas.
- Se todas as vagas forem preenchidas, o anúncio é automaticamente pausado e um novo evento *advertisement.paused* é emitido.
- O microsserviço de notificações consome esse evento e envia uma notificação ao dono do anúncio.

A implementação do fluxo pode ser visualizada a seguir. No caso de uso abaixo, o serviço de aplicações aceita uma inscrição de vaga e dispara um evento para notificar que uma aplicação foi aceita.

Código-fonte 7: Caso de Uso de Aceitar Aplicação

```
export class AcceptApplicationUseCase {
  constructor(...) {}

  async execute(applicationId: number) {
    const application = await this.applicationRepository.
       findById(applicationId);
    if (!application) {
      throw new ApplicationNotFoundException(`Application
         with id ${applicationId} not found`);
    }

    if (application.advertisement.isActive === false) {
      throw new AdvertisementPausedException(`Advertisement
          paused`);
    }

    application.status = ApplicationStatus.ACCEPTED;
    this.kafkaClient.emit('application.accepted', {
      id: application.id,
      advertisementId: application.advertisementId,
      applicantId: application.applicantId,
    });

    await this.applicationRepository.update(application);
  }
}
```

Fonte: Elaborado pelo autor

Quando um evento *application.accepted* é disparado, o serviço de anúncios escuta e processa esse evento para atualizar a quantidade de vagas preenchidas:

Código-fonte 8: Observador do Evento de Aplicação Aceita no Serviço de Anúncios

```
@MessagePattern('application.accepted')
async handleIncrementOccupiedSlots(@Payload() data: any) {
  try {
    await this.incrementOccupiedSlotsUseCase.execute(data.
       advertisementId);
  } catch (error) {
    console.log(error);
  }
}
```

Fonte: Elaborado pelo autor

No caso de uso abaixo, o serviço incrementa a quantidade de vagas preenchidas e, se atingir o limite, pausa o anúncio e dispara um novo evento advertisement.paused:

Código-fonte 9: Caso de Uso de Incrementar Vagas Ocupadas

```
@Injectable()
export class IncrementOccupiedSlotsUseCase {
  constructor(...) {}

  async execute(advertisementId: number): Promise<void> {
    const advertisement = await this.
       advertisementRepository.findById(advertisementId);

    if (!advertisement) {
      throw new AdvertisementNotFoundException(`
         Advertisement with id ${advertisementId} not found
         `);
    }

    advertisement.occupiedSlots += 1;

    if (advertisement.occupiedSlots === advertisement.
```

```
        totalSlots) {
      advertisement.isActive = false;
      this.kafkaClient.emit('advertisement.paused', {
        id: advertisement.id,
        title: advertisement.title,
        ownerId: advertisement.ownerId,
      });
    }

    await this.advertisementRepository.update(advertisement
        );
    this.kafkaClient.emit('advertisement.updated', {
      id: advertisement.id,
      title: advertisement.title,
      imgSrc: advertisement.imgSrc,
      price : advertisement.price,
      cityName : advertisement.city.name,
      stateUF: advertisement.state.uf,
      isActive: advertisement.isActive,
    });
  }
}
```

Fonte: Elaborado pelo autor

Por fim, o serviço de notificações escuta o evento *advertisement.paused* e cria uma notificação para o dono do anúncio:

Código-fonte 10: Observador do Evento de Anúncio Pausado no Serviço de Notificações

```

@MessagePattern('advertisement.paused')
async handleAdvertisementPaused(@Payload() data: any) {
  try {
    await this.createNotificationUseCase.execute({
```

```
6         recipientId: data.ownerId,
7         type: NotificationType.ADVERTISEMENT_PAUSED,
8         message: `Seu an ncio intitulado "${data.title}" foi
              pausado automaticamente devido ao  preenchimento
              das vagas.`,
9       });
10    } catch (error) {
11      console.log(error);
12    }
13  }
```

Fonte: Elaborado pelo autor

5.10 Replicação de Dados

No sistema desenvolvido é comum que um serviço precise retornar dados que residem em outros serviços. Esse cenário normalmente exige uma comunicação síncrona entre os serviços. Por exemplo, no caso do serviço de anúncios, ao retornar informações sobre um anúncio, seria necessário incluir dados sobre o proprietário do anúncio, que são armazenados no serviço de usuários. O caminho mais simples seria o *API Gateway* fazer uma requisição ao serviço de anúncios, que, por sua vez, faria outra requisição ao serviço de usuários para obter as informações necessárias, retornando, então, todos os dados.

Entretanto, essa abordagem apresenta um problema: ela resulta em um acoplamento forte entre os serviços, violando o princípio de independência dos microsserviços. Cada serviço deve ser responsável por seus próprios dados e funcionalidades, evitando dependências diretas entre eles. Para resolver essa questão, optou-se pela replicação de dados relevantes e pela sincronização assíncrona utilizando um sistema de mensageria. Essa abordagem permitiu que os serviços evitassem realizar buscas em outros serviços, visto que mantinham todos os registros necessários. Esta estratégia traz complexidades em relação à consistência dos dados, mas permite que os serviços atuem independentemente. Nesta lógica, caso um microsserviço caia ou esteja sobrecarregado, os outros continuam atuando normalmente.

A replicação de dados visa manter uma cópia das informações necessárias localmente no serviço que precisa delas. Por exemplo, o serviço de anúncios pode manter uma tabela de

usuários em seu banco de dados, que é atualizada de forma assíncrona toda vez que um novo usuário é criado no serviço de usuários. Essa atualização é feita por meio de eventos gerados pelo serviço de usuários e consumidos pelo serviço de anúncios, garantindo que os dados estejam sincronizados sem a necessidade de uma comunicação síncrona entre os serviços.

No exemplo abaixo, ao criar um novo usuário no serviço de usuários, o evento *user.created* é emitido, e o serviço de anúncios, que é responsável por acompanhar esses eventos, irá processá-los e atualizar sua própria base de dados com as informações do novo usuário.

Código-fonte 11: Caso de Uso de Cadastro de Usuários

```
@Injectable()
export class SignUpUseCase {
  constructor(
    private userRepository: UserRepository,
    private hashingService: HashingService,
    private readonly kafkaClient: ClientKafka,
  ) {}

  async execute(signUpDto: SignUpDto): Promise<void> {
    const { name, email, password, passwordConfirm, gender
       } = signUpDto;

    if (password !== passwordConfirm) {
      throw new PasswordNotMatchException('Passwords do not
          match');
    }

    const existingUser = await this.userRepository.
       findByEmail(email);

    if (existingUser) {
      throw new EmailAlreadyExistsException(
        `The email ${email} already exists.`,
```

```
21        );
22      }
23
24      const hashedPassword = await this.hashingService.hash(
            password, 10);
25
26      const user = new User({
27        name,
28        email,
29        password: hashedPassword,
30        gender,
31      });
32
33      const createdUser = await this.userRepository.create(
            user);
34
35      this.kafkaClient.emit('user.created', {
36        id: createdUser.id,
37        name: createdUser.name,
38        imgSrc: createdUser.imgSrc,
39      });
40    }
41  }
```

Fonte: Elaborado pelo autor

Ao ocorrer a criação de um usuário, o serviço de usuários emite o evento user.created, que é então consumido pelo serviço de anúncios. O serviço de anúncios escuta esse evento e, em seguida, executa uma ação específica para tratar as informações do usuário recém-criado. Nesse caso, o serviço de anúncios mantém uma cópia dos dados do usuário em seu próprio banco de dados, garantindo que as informações estejam disponíveis para consultas subsequentes, sem a necessidade de uma comunicação síncrona com o serviço de usuários.

Código-fonte 12: Observador do Evento de Usuário Criado no Serviço de Anúncios

```
@MessagePattern('user.created')
async handleUserCreated(@Payload() data: any) {
  try {
    await this.createOwnerUseCase.execute(data)
  } catch (error) {
    console.log(error)
  }
}
```

Fonte: Elaborado pelo autor

Esse processo de replicação assíncrona, em vez de depender de requisições síncronas, permite que os serviços permaneçam desacoplados e independentes, respeitando o princípio fundamental dos microsserviços.

A versão do sistema em microsserviços [3] foi desenvolvida em um repositório à parte e está pública para qualquer pessoa interessada.

5.11 Avaliação da abordagem de desenvolvimento

A abordagem *Monolith First* utilizada no desenvolvimento deste sistema demonstrou ser altamente eficaz para a construção de uma arquitetura de microsserviços bem estruturada. O desenvolvimento inicial do monólito modular utilizando DDD e *Clean Architecture* proporcionou um entendimento aprofundado da aplicação, permitindo a identificação precisa dos limites de cada subdomínio.

Um dos principais benefícios observados foi a clareza na definição dos serviços, especialmente no caso da funcionalidade de aplicações para vagas. Inicialmente, houve dúvidas sobre se essa funcionalidade pertencia ao módulo de usuários ou ao de anúncios. No entanto, com o uso de DDD e a modelagem dos contextos, ficou evidente que aplicações constituíam um subdomínio próprio, com sua própria linguagem ubíqua. Essa descoberta foi essencial para a posterior transição para microsserviços, pois permitiu que o módulo de aplicações gerenciasse apenas os dados relevantes de outros módulos, facilitando, posteriormente, a replicação de dados.

A modularidade promovida pela *Clean Architecture* também se mostrou um fator

[3] https://github.com/pedrogrigorio/my-republic-microservices

chave para a migração eficiente. A separação de regras de negócio e a independência dos módulos dentro do monólito permitiram que a migração fosse realizada sem a necessidade de reescrever código manualmente. A organização interna do sistema garantiu que os módulos pudessem ser extraídos e transformados em serviços independentes com mínima refatoração.

Outro ponto positivo foi a redução da complexidade inicial do desenvolvimento, uma vez que o monólito modular eliminou a necessidade de lidar com comunicação distribuída e sincronização de dados desde o começo. Isso possibilitou um foco maior na construção das regras de negócio e na estruturação dos módulos. Como resultado, a migração para microsserviços ocorreu de maneira rápida e eficiente, levando aproximadamente uma semana para ser concluída.

Durante a migração, os principais desafios encontrados foram relacionados à implementação da comunicação entre serviços. A comunicação assíncrona utilizando Apache Kafka foi relativamente simples de integrar, permitindo o desacoplamento eficaz dos serviços e garantindo a consistência dos dados através de eventos. Já a comunicação síncrona via gRPC exigiu mais esforço, devido à necessidade de definição de interfaces nos arquivos *proto* e ao tratamento de exceções. Apesar dessas dificuldades, o fato de o sistema já estar estruturado e funcional facilitou significativamente a adaptação para esse novo modelo de comunicação.

No geral, a experiência reforça que a abordagem *Monolith First* é altamente recomendada para projetos que visam uma arquitetura de microsserviços bem definida. A fase inicial de desenvolvimento monolítico permitiu um entendimento aprofundado do sistema e a correta delimitação dos serviços, mitigando problemas comuns de acoplamento excessivo e comunicação ineficiente. Além disso, a combinação do *Monolith First* com DDD e *Clean Architecture* mostrou-se vantajosa para a modularização, replicação de dados e facilidade de migração. Um resumo da avaliação da abordagem de desenvolvimento pode ser visualizado no Quadro 3.

Dessa forma, pode-se concluir que iniciar com um monólito antes da transição para microsserviços não apenas reduz a complexidade inicial, como também contribui para a construção de uma arquitetura distribuída mais sólida e bem planejada. Além disso, este estudo conclui também que a utilização de padrões arquiteturais que visam à modularização do sistema é extremamente importante para uma melhor etapa de migração, facilitando a identificação correta dos limites de cada serviço e promovendo a independência entre eles.

Quadro 3: Avaliação da abordagem

Critério	Avaliação
Complexidade inicial	O desenvolvimento inicial como monólito modular reduziu a complexidade ao evitar desafios de comunicação distribuída no início do projeto
Definição de serviços	Facilitada pelo uso de DDD, permitindo uma separação clara dos subdomínios.
Modularidade e organização	*Clean Architecture* proporcionou separação entre regras de negócio e infraestrutura, facilitando a migração.
Tempo de migração	Aproximadamente uma semana, beneficiado pela modularidade prévia do monólito.
Desafios na comunicação	A implementação de Apache Kafka foi simples, enquanto a comunicação via gRPC exigiu maior esforço devido à definição de interfaces e tratamento de exceções.
Replicação de dados	A modularização inicial facilitou a replicação de dados, garantindo independência entre os serviços após a migração.
Benefícios gerais	Melhor entendimento do sistema, redução de problemas de acoplamento e uma transição estruturada para microsserviços.

Fonte: Elaborado pelo autor

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 Conclusões

A análise da abordagem *Monolith First* no desenvolvimento de sistemas em microsserviços demonstrou ser uma estratégia eficiente e eficaz, principalmente quando combinada com práticas como DDD e *Clean Architecture*. O estudo de caso realizado ao longo deste trabalho evidenciou os benefícios dessa abordagem na construção de uma arquitetura de microsserviços bem estruturada, desde a fase inicial de desenvolvimento até a migração para um sistema distribuído.

A fase de desenvolvimento do monólito modular proporcionou uma compreensão profunda do sistema e uma definição clara dos contextos delimitados. A utilização de DDD foi fundamental para a identificação dos limites dos subdomínios, como no caso da gestão das aplicações para vagas, onde ficou evidente a independência do subdomínio, facilitando a posterior replicação de dados.

A migração para microsserviços, que ocorreu de forma rápida e eficiente, foi facilitada pela modularidade promovida pela *Clean Architecture*. A separação das responsabilidades e a independência dos módulos dentro do monólito possibilitaram a extração dos módulos para serviços independentes com mínima refatoração. Esse processo evidenciou a importância de iniciar com um monólito modular para reduzir a complexidade e os desafios iniciais no desenvolvimento de sistemas distribuídos.

Embora a comunicação síncrona via gRPC tenha apresentado desafios, especialmente na definição das interfaces e no tratamento de exceções, a comunicação assíncrona utilizando Apache Kafka foi bem-sucedida e eficaz, permitindo o desacoplamento entre os serviços e a manutenção da consistência dos dados através de eventos.

Por fim, a abordagem provou ser uma estratégia altamente recomendada para projetos que buscam uma transição bem-sucedida para microsserviços. A experiência deste trabalho reforça que a fase inicial de desenvolvimento monolítica não só reduz a complexidade do projeto, como também contribui para a criação de uma arquitetura distribuída mais sólida, bem planejada e capaz de escalar de forma eficiente.

A combinação de DDD, *Clean Architecture* e a abordagem permite não apenas uma migração mais tranquila, mas também a construção de sistemas mais resilientes, modulares e de fácil manutenção a longo prazo, proporcionando uma base sólida para a evolução do sistema conforme as necessidades do negócio se expandem.

6.2 Trabalhos Futuros

Este trabalho abre espaço para diversas possibilidades de pesquisa e desenvolvimento futuros relacionados à abordagem *Monolith First*. Um dos caminhos promissores seria a aplicação dessa abordagem utilizando diferentes padrões arquiteturais, distintos de DDD e *Clean Architecture*, para avaliar como outras estratégias impactam a modularidade e a migração para microsserviços.

Outra possibilidade de estudo futuro seria explorar o processo de implantação do sistema desenvolvido. A transição de um ambiente local para um ambiente de produção, considerando aspectos como escalabilidade, balanceamento de carga e estratégias de *deploy*, traria desafios complexos que merecem uma análise aprofundada. A implementação de mecanismos como orquestração de *containers* e infraestrutura como código poderia ser avaliada para tornar essa transição mais eficiente.

Além disso, uma análise mais detalhada do desempenho do sistema monolítico *versus* a versão em microsserviços poderia ser conduzida. Testes de carga, latência e consumo de recursos poderiam fornecer dados concretos sobre os impactos da migração, permitindo uma comparação mais precisa entre os modelos arquiteturais. Essa avaliação auxiliaria na tomada de decisões em projetos futuros que considerem a adoção da abordagem *Monolith First*.

Dessa forma, este estudo não apenas valida a eficácia da abordagem, mas também abre caminho para novas investigações que podem aprimorar ainda mais sua aplicação no desenvolvimento de sistemas distribuídos.

REFERÊNCIAS

ABGAZ, Y.; MCCARREN, A.; ELGER, P.; SOLAN, D.; LAPUZ, N.; BIVOL, M.; JACKSON, G.; YILMAZ, M.; BUCKLEY, J.; ; CLARKE, P. Decomposition of monolith applications into microservices architectures: A systematic review. **IEEE Transactions on Software Engineering**, IEEE, v. 11, p. 4213–4242, 2023.

BRUCE, M.; PEREIRA, P. A. **Microservices in Action**. [*S. l.*]: Manning Publications, 2018. ISBN 978-1-617-29445-7.

FADHLURROHMAN, F. **The Importance of Clean Architecture in Software Development**. 2024. Disponível em: https://zikazama.medium.com/the-importance-of-clean-architecture-in-software-development-4c2da43c7fcf. Acesso em: 26 dez. 2024.

FAJAR, A. N.; NOVIANTI, E.; FIRMANSYAH. Design and implementation of microservices system based on domain-driven design. **International Journal of Emerging Trends in Engineering Research**, WARSE, v. 8, n. 7, p. 213–220, 2020.

FORD, N.; RICHARDS, M.; SADALAGE, P.; DEHGHANI, Z. **Software Architecture: The Hard Parts**: Modern trade-off analyses for distributed architectures. [*S. l.*]: O'Reilly Media, 2021. ISBN 978-1-492-08689-5.

FOWLER, M. **Strangler Fig Application**. 2004. Disponível em: https://martinfowler.com/bliki/StranglerFigApplication.html. Acesso em: 2 ago. 2024.

FOWLER, M. **Monolith First**. 2015. Disponível em: https://martinfowler.com/bliki/MonolithFirst.html. Acesso em: 2 ago. 2024.

FOWLER, M. **Domain Driven Design**. 2020. Disponível em: https://martinfowler.com/bliki/DomainDrivenDesign.html. Acesso em: 4 ago. 2024.

HASSELBRING, W.; STEINACKER, G. Microservice architectures for scalability, agility and reliability in e-commerce. **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**, IEEE, v. 11, 2017.

JAMSHIDI, P.; PAHL, C.; MENDONçA, N. C.; LEWIS, J.; TILKOV, S. Microservices: The journey so far and challenges ahead. **IEEE Software**, IEEE, v. 11, p. 24–35, 2018.

KLEPPMANN, M. **Designing Data-Intensive Applications**: The big ideas behind reliable, scalable, and maintainable systems. [*S. l.*]: O'Reilly Media, 2017. ISBN 978-1-449-37332-0.

LAIGNER, R.; KALINOWSKI, M.; DINIZ, P.; BARROS, L.; CASSINO, C.; LEMOS, M.; ARRUDA, D.; LIFSCHITZ, S.; ZHOU, Y. From a monolithic big data system to a microservices event-driven architecture. **46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**, IEEE, p. 213–220, 2020.

MARTIN, R. C. **Clean Architecture**: A craftsman's guide to software structure and design. [*S. l.*]: Pearson, 2017. ISBN 978-0-13-449416-6.

MILANOVIć, D. M. **Why should you build a (modular) monolith first?** 2023. Disponível em: https://newsletter.techworld-with-milan.com/p/why-you-should-build-a-modular-monolith. Acesso em: 23 ago. 2024.

MONTESI, F.; PERESSOTTI, M.; PICOTTI, V. Sliceable monolith: Monolith first, microservices later. **2021 IEEE International Conference on Services Computing (SCC)**, IEEE, p. 364–366, 2021.

NEWMAN, S. **Building Microservices**: Designing fine-grained systems. [*S. l.*]: O'Reilly Media, 2014. ISBN 978-1-491-95035-7.

NEWMAN, S. **Monolith to Microservices**: Evolutionary patterns to transform your monolith. [*S. l.*]: O'Reilly Media, 2019. ISBN 978-1-492-07554-7.

OUMOUSSA, I.; SAIDI, D. R. Evolution of microservices identification in monolith decomposition: A systematic review. **IEEE Access**, IEEE, v. 12, p. 23389–23405, 2024.

PAIXãO, J. R. da. **O que é SOLID**: O guia completo para você entender os 5 princípios da poo. 2019. Disponível em: https://medium.com/desenvolvendo-com-paixao/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530. Acesso em: 26 dez. 2024.

RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture**: An engineering approach. [*S. l.*]: O'Reilly Media, 2020. ISBN 978-1-492-04340-9.

RICHARDSON, C. **Microservices Patterns**: With examples in java. [*S. l.*]: Manning Publications, 2018. ISBN 978-1-617-29454-9.

TSECHELIDIS, M.; NIKOLAIDIS, N.; MAIKANTIS, T.; AMPATZOGLOU, A. Modular monoliths the way to standardization. **ESAAM '23: Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum**, Association for Computing Machinery, p. 49–52, 2023.

VELEPUCHA, V.; FLORES, P. A survey on microservices architecture: Principles, patterns and migration challenges. **IEEE Access**, IEEE, v. 11, p. 88339–88358, 2023.

VERNON, V.; TOMASZ, J. **Strategic Monoliths and Microservices**: Driving innovation using purposeful architecture. [*S. l.*]: Addison-Wesley Publishing, 2021. ISBN 978-0-137-35546-4.

VURAL, H.; KOYUNCU, M. Does domain-driven design lead to finding the optimal modularity of a microservice? **IEEE Access**, IEEE, v. 9, p. 32721–32733, 2021.

ÖZKAN, O.; BABUR Önder; BRAND, M. van den. Refactoring with domain-driven design in an industrial context. **Empirical Software Engineering**, Springer, v. 28, 2023.