Pontifícia Universidade Católica de São Paulo
Especialização em Engenharia de Software

LUIZ CARLOS DOS SANTOS JUNIOR

MIGRAÇÃO DE MONOLITO PARA MICROSSERVIÇOS

Relatório da experiência do projeto integrador

SÃO PAULO – SP

2024

LUIZ CARLOS DOS SANTOS JUNIOR

MIGRAÇÃO DE MONOLITO PARA MICROSSERVIÇOS
Relatório da experiência do projeto integrador

Trabalho de Conclusão de Curso apresentado à banca examinadora da Pontifícia Universidade Católica de São Paulo, como exigência parcial para obtenção do título de Especialista em Engenharia de Software, sob a orientação da Prof. Msc. José Teodoro da Silva.

São Paulo

2024

RESUMO

Este trabalho aborda o desafio técnico e estratégico de migrar uma aplicação monolítica para uma arquitetura de microserviços, explorando os benefícios em termos de escalabilidade, resiliência e manutenibilidade. O estudo apresenta tanto a fundamentação teórica quanto a experiência prática de um projeto integrador acadêmico, no qual foi realizada a modernização da arquitetura de um e-commerce de pet shop, destacando o planejamento, a execução e os desafios enfrentados. A transformação envolve a adoção de novas tecnologias, como Kubernetes, RabbitMQ, NGINX e ferramentas de monitoramento (OpenTelemetry e Grafana), com o objetivo de construir uma solução robusta, escalável e tolerante a falhas.

Todos os relatos aqui apresentados refletem a vivência de quem participou ativamente do desenvolvimento e das discussões deste projeto. Os códigos-fonte utilizados no projeto estão disponíveis no GitHub: https://github.com/puppyplace.

Palavras-chave: arquitetura de microsserviços, migração de monolito, e-commerce, arquitetura de software, evolução de sistema**s.**

ABSTRACT

This work addresses the technical and strategic challenges of a monolithic to a microservices architecture's migration. Challenges like exploring trade-offs in terms of scalability, resilience, and maintainability. The study presents both the theoretical foundation and the practical experience during the development of the 'Projeto Integrador'. Following its development. Here, we discuss and present the modernization of a pet-shop e-commerce, emphasizing the planning, execution, and challenges faced by the team. The transformation from monolith to microservices involved the adoption of new technologies such as Kubernetes, RabbitMQ, NGINX, and monitoring tools (OpenTelemetry and Grafana), aiming to build a robust, scalable, and fault-tolerant solution.

The source codes used in the project are available on GitHub: https://github.com/puppyplace.

Keywords: microservices architecture, monolith migration, e-commerce, software architecture, system evolution.

DOS SANTOS JUNIOR, LUIZ CARLOS

MIGRAÇÃO DE MONOLITO PARA MICROSSERVIÇOS Relatório da experiência do projeto integrador. LUIZ CARLOS DOS SANTOS JUNIOR. -- São Paulo: [s.n.], 2024. 21p.

Orientador: José Teodoro da Silva.

Trabalho de Conclusão de Curso (Especialização) - Pontifícia Universidade Católica de São Paulo. Especialização em Engenharia de Software, 2024.

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

Na atual crescente demanda por soluções desacopladas e altamente escaláveis, as empresas estão optando por modernizar seus sistemas legados para uma arquitetura de micro serviços buscando aumentar a disponibilidade e resiliência dessas aplicações. A arquitetura monolítica, embora funcional, apresenta limitações significativas em termos de escalabilidade horizontal, distribuição de carga e resiliência. Por outro lado, a arquitetura de micro serviços tem se mostrado uma alternativa viável e eficaz para superar essas limitações, permitindo que sistemas sejam estruturados em componentes menores, independentes e facilmente gerenciáveis (FOWLER, 2014).

Em um cenário de e-commerce, por exemplo, é possível observar que o volume de buscas geralmente supera o número de modificações nos cadastros de clientes ou no fechamento de pedidos (Hellerstein, J. M., Stonebraker, M., & Hamilton, J. ,2007). Dessa forma, a arquitetura de micro serviços permite a alocação de um maior número de instâncias para o serviço de busca, como, por exemplo, dez instâncias, enquanto os serviços de cadastro e fechamento de pedidos podem ser dimensionados com um número reduzido de instâncias, como duas ou três. Em contraste, na arquitetura monolítica, essa flexibilidade de escalabilidade não é possível. Nesse modelo, é necessário iniciar múltiplas instâncias do monólito, que engloba todos os serviços, como busca, fechamento de pedidos e cadastro, resultando em um consumo excessivo de recursos, como memória e capacidade de processamento, devido à incapacidade de escalar e isolar as funcionalidades de maneira independente, uma vez que todas as operações são executadas em um único processo.

A separação de serviços em uma arquitetura de micro serviços não é trivial, pois envolve mudanças significativas na estrutura do sistema e no modo como suas partes interagem. Em uma arquitetura monolítica, os diferentes módulos podem ser interconectados diretamente por meio da memória compartilhada e das pilhas internas de um único processo. No entanto, essa capacidade é perdida quando se adota a abordagem de micro serviços, uma vez que múltiplas instâncias independentes de processos isolados são criadas. Isso gera o desafio da comunicação entre processos, um problema fundamental que surge na interação entre os micro serviços (Newman, 2015).

A comunicação interprocesso (IPC, do inglês *Inter-Process Communication*) refere-se aos mecanismos que possibilitam a troca de informações entre processos distintos, podendo ocorrer de forma independente ou cooperativa. Em sua forma independente, a execução de um processo não é afetada por outros, enquanto na comunicação cooperativa, a execução de

um processo pode depender da interação com outro. Esses mecanismos tornam-se essenciais para garantir a sincronização e a troca de dados entre os micro serviços, o que representa um desafio adicional em termos de desempenho e escalabilidade (Tanenbaum & Van Steen, 2007).

No contexto da arquitetura de micro serviços, é fundamental reavaliar as estratégias de transferência de dados entre as diferentes partes do sistema. As abordagens comumente adotadas na indústria para facilitar essa comunicação incluem: comunicação via rede (como REST, RPC, entre outros), compartilhamento de arquivos, compartilhamento de banco de dados e comunicação através de filas de mensagens (como RabbitMQ, ActiveMQ, etc.) (Fowler, 2015).

Neste contexto, o processo de reengenharia e migração visam garantir que o sistema seja capaz de lidar com o aumento de carga e falhas em componentes de forma eficiente, mantendo a integridade e a performance da aplicação ao longo do tempo (Newman, 2015).

Este trabalho relata o estudo de caso do projeto desenvolvido no primeiro semestre para analisar as particularidades e os desafios da migração de uma arquitetura monolítica para uma abordagem baseada em micro serviços. Em vez de uma única aplicação com milhões de linhas de código, teremos diversas pequenas aplicações independentes, que podem variar de dezenas a centenas ou até milhares. A lógica de negócios será distribuída entre essas aplicações, possibilitando um crescimento mais eficaz das instâncias e uma maior flexibilidade na escalabilidade do sistema.

O desafio proposto neste trabalho é revisar a aplicação desenvolvida durante o projeto integrador no primeiro ano deste curso, com o objetivo de alcançar uma solução que seja não apenas robusta e escalável, mas também tolerante a falhas e de fácil manutenção.

2. FUNDAMENTAÇÃO TEÓRICA

A migração de uma aplicação monolítica para uma arquitetura de microsserviços é um processo técnico e estratégico que visa melhorar aspectos como escalabilidade, resiliência e manutenção de sistemas. Esse paradigma se tornou popular devido às suas vantagens em ambientes que exigem alta disponibilidade e flexibilidade, como em plataformas de e-commerce (GERVINO, 2020; NEWMAN, 2015)

2.1. ARQUITETURA MONOLÍTICA

Uma arquitetura monolítica caracteriza-se por sua estrutura única e integrada, onde todos os componentes do sistema são executados em um único processo. Isso implica que as operações, desde a interação com o usuário até o acesso ao banco de dados, estão intimamente acopladas. Embora a abordagem monolítica seja eficiente em sistemas pequenos ou iniciais, ela apresenta grandes desafios quando a demanda do sistema cresce, como apontado por Fowler (2014). A escalabilidade é limitada, pois a aplicação inteira precisa ser escalada, o que leva a um consumo excessivo de recursos. Além disso, qualquer modificação exige a reconstrução e implantação de toda a aplicação, o que aumenta a complexidade e o risco de falhas.

2.2. ARQUITETURA MICROSSERVIÇOS

Os microsserviços são definidos como uma abordagem arquitetural onde uma aplicação é dividida em serviços menores, independentes e distribuídos. Cada micro serviço é responsável por uma funcionalidade específica e pode ser desenvolvido, implantado e escalado de forma autônoma (FOWLER, 2014; NEWMAN, 2015).
Essa arquitetura facilita a adoção de práticas de escalabilidade horizontal e promove a resiliência ao isolar falhas em componentes individuais.

2.3. COMPARAÇÃO ENTRE MONOLITO E MICROSSERVIÇOS

A principal diferença entre uma arquitetura monolítica e uma baseada em microsserviços reside na flexibilidade e no isolamento dos serviços. Enquanto no monólito toda a aplicação é gerenciada como um único bloco, nos microsserviços cada serviço tem sua

infraestrutura própria, podendo ser escalado e mantido de maneira independente. A tabela abaixo resume as principais comparações:

Aspecto	**Monolito**	**Microsserviços**
Escalabilidade	Escala o sistema inteiro, com alto consumo de recursos	Escala serviços individualmente (FOWLER, 2015).
Manutenção	Difícil devido à dependência entre os componentes	Modular, com serviços independentes (NEWMAN, 2015).
Complexidade	Simples no início, mas cresce com o sistema	Mais complexo inicialmente, mas flexível no longo prazo
Tolerância a Falhas	Impacta o sistema inteiro.	Isolamento de falhas em serviços (TANENBAUM e VAN STEEN, 2007).

2.4. DESAFIOS: MIGRAÇÃO DE MONOLITO PARA MICROSSERVIÇOS

A migração de um sistema monolítico para microsserviços não é trivial e apresenta desafios significativos. Segundo Gervino (2020), a principal dificuldade está na decomposição adequada do sistema, o que exige uma revisão completa das interações entre os componentes. Além disso, a comunicação entre microsserviços é mais complexa, pois envolve protocolos de comunicação distribuída, como REST, gRPC ou mensageria (RabbitMQ, Kafka), o que pode impactar o desempenho e a latência do sistema, conforme discutido por Newman (2015).

A introdução de microsserviços exige também a adoção de novas práticas e ferramentas de desenvolvimento, como orquestração com Kubernetes, monitoramento distribuído com OpenTelemetry, Prometheus e Grafana, e estratégias de integração contínua (CI/CD) para garantir que os serviços sejam implantados de forma ágil e segura.

A observabilidade é um pilar fundamental em arquiteturas de microsserviços, permitindo monitorar o desempenho e detectar gargalos em tempo real. Ferramentas como OpenTelemetry, Prometheus e Grafana desempenham papéis importantes nesse contexto,

garantindo que métricas críticas sejam coletadas e visualizadas de maneira eficiente (TURNER et al., 2020).

2.5. PADRÕES DE MIGRAÇÃO

- **Strangler Fig Pattern**: Substituição incremental de partes do sistema monolítico por microsserviços (FOWLER, 2015).
- **Decompose by Business Capabilities**: Divisão de serviços com base nas capacidades de negócio (NEWMAN, 2015).

2.6. O CASO E-COMMERCE: CONSULTAS VS ESCRITAS

No contexto de e-commerce, o comportamento dos usuários é marcado por um volume expressivamente maior de operações de consulta (como buscas por produtos) em relação às operações de escrita (como cadastro de clientes e fechamento de pedidos). Conforme destacado por **Hellerstein, Stonebraker e Hamilton (2007)**, sistemas transacionais em domínios de alto tráfego geralmente enfrentam cenários onde consultas superam significativamente inserções e atualizações, tanto em frequência quanto em complexidade.

Esse padrão exige soluções arquiteturais que permitam dimensionar adequadamente os serviços responsáveis pelas consultas. A arquitetura de microsserviços oferece uma abordagem eficaz para lidar com essa demanda, permitindo que os serviços sejam escalados de forma independente com base em suas necessidades específicas. Por exemplo, um serviço de busca pode ser replicado em várias instâncias para lidar com altos volumes de requisições de consultas simultâneas, enquanto os serviços de cadastro de clientes e processamento de pedidos podem ser mantidos com um número reduzido de instâncias, economizando recursos computacionais.

Portanto, adotar a escalabilidade seletiva e aproveitar tecnologias modernas de busca e processamento de dados são práticas essenciais para o sucesso de um sistema de e-commerce moderno. Essa abordagem, viabilizada por microsserviços, não apenas melhora a eficiência operacional, mas também reduz custos e permite uma adaptação mais rápida às mudanças nas demandas do mercado (RICHARDSON, 2018).

3. O PROJETO INTEGRADOR

O projeto integrador tem por objetivo apresentar aos grupos de alunos um problema para que eles participem de todas as fases do ciclo de desenvolvimento de um produto. A disciplina foi iniciada no primeiro semestre de 2021, e o grupo ao qual pertencia apresentou uma proposta de plataforma para um marketplace voltado ao e-commerce de pet shops, buscando atender às necessidades específicas desse segmento de mercado. O grupo era formado por quatro alunos, dividindo entre 3 backends e 1 frontend. E este projeto visa não apenas o desenvolvimento de uma solução tecnológica, mas também a aplicação de metodologias ágeis e boas práticas de engenharia de software ao longo de todo o processo de criação e implementação do sistema.

3.1. ARQUITETURA PARA ALCANÇAR AS REGRAS DE NEGÓCIO

A decisão sobre as funcionalidades a serem adotadas foi baseada na elaboração de um PM Canvas, uma ferramenta estratégica utilizada para mapear e visualizar as necessidades do projeto, a visão de negócio e as estratégias a serem aplicadas. O PM Canvas é um modelo que auxilia na definição e organização dos objetivos do projeto, identificando elementos chave como público-alvo, proposta de valor, métricas de sucesso, canais de comunicação, entre outros. Esse processo permitiu uma compreensão mais ampla das estratégias e das regras a serem implementadas ao longo do desenvolvimento. O PM Canvas utilizado e o resultado da discussão com o grupo pode ser visualizado na imagem 1.

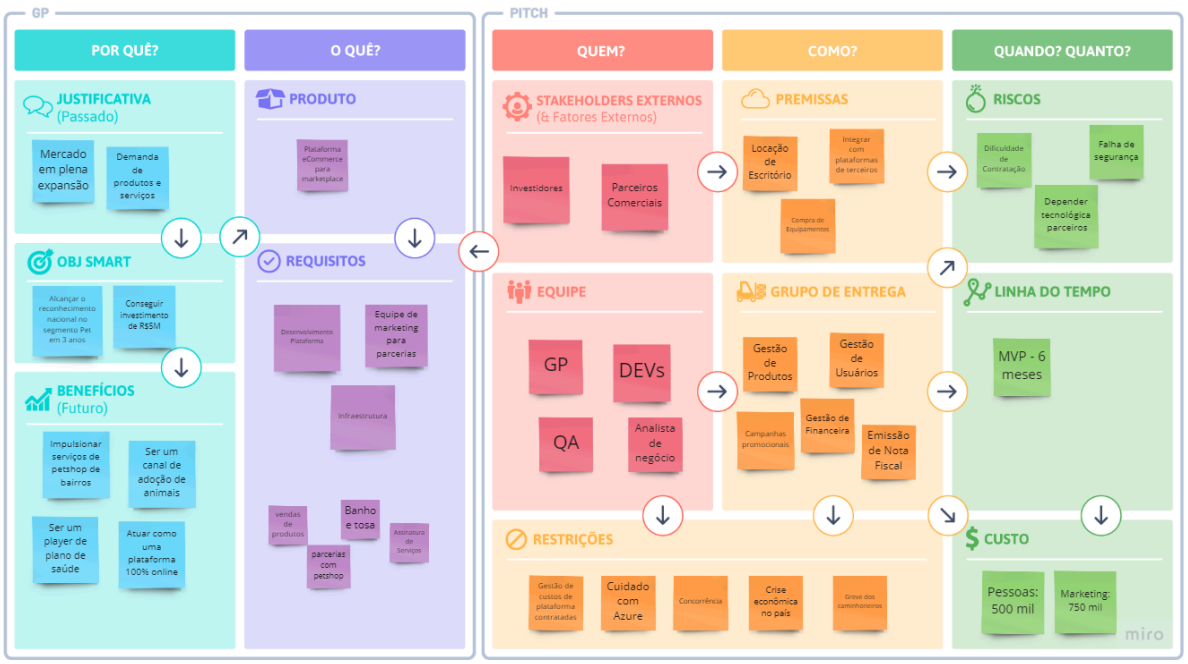


Figura 1 - Project Management Canvas

Para desenvolver uma aplicação que atendesse ao modelo de negócio proposto e permanecesse dentro do prazo estipulado pela disciplina do projeto integrador, adotamos uma arquitetura monolítica. Essa arquitetura inicial foi concebida com base na abordagem de MVP (Minimum Viable Product), que se caracteriza pela criação de um conjunto de funcionalidades essenciais, com o objetivo de validar a viabilidade do negócio em curto espaço de tempo.

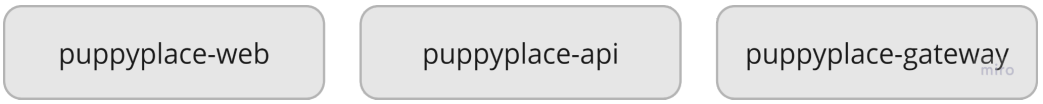
A arquitetura monolítica é um modelo tradicional de desenvolvimento de software no qual todos os componentes de uma aplicação são integrados e executados em um único processo. Nesse modelo, os diferentes módulos do sistema, como interface do usuário, lógica de negócios e acesso a dados, são interligados e compartilham a mesma base de código. Isso resulta em uma aplicação coesa, onde a comunicação entre os módulos ocorre de forma direta e eficiente, já que todos estão dentro de um único espaço de memória e executando no mesmo processo (Bass, Clements, & Kazman, 2013).

Uma das principais vantagens da arquitetura monolítica é a simplicidade no desenvolvimento inicial, pois a gestão de dependências e a comunicação entre os componentes são mais diretas e não exigem mecanismos complexos de comunicação interprocessos. Além disso, a implantação é facilitada, pois o sistema pode ser distribuído e executado como uma única unidade.

O MVP não deve ser confundido com a versão final do produto, pois seu objetivo é

garantir que as características mínimas necessárias para o lançamento do negócio estejam presentes, mesmo que com capacidade operacional limitada ou restrita a um público específico, de modo a gerar valor a partir dos pontos identificados durante a elaboração do PM Canvas (Ries, 2011; Blank, 2013).

Dessa forma, a arquitetura monolítica foi dividida em apenas três camadas principais, visando atender aos requisitos fundamentais para o lançamento inicial:



- **puppyplace-web**: frontend
- **puppyplace-api**: backend com as regras do negócio;
- **puppyplace-gateway-api**: interface com a API de pagamentos da Juno (*https://www.juno.com.br*)

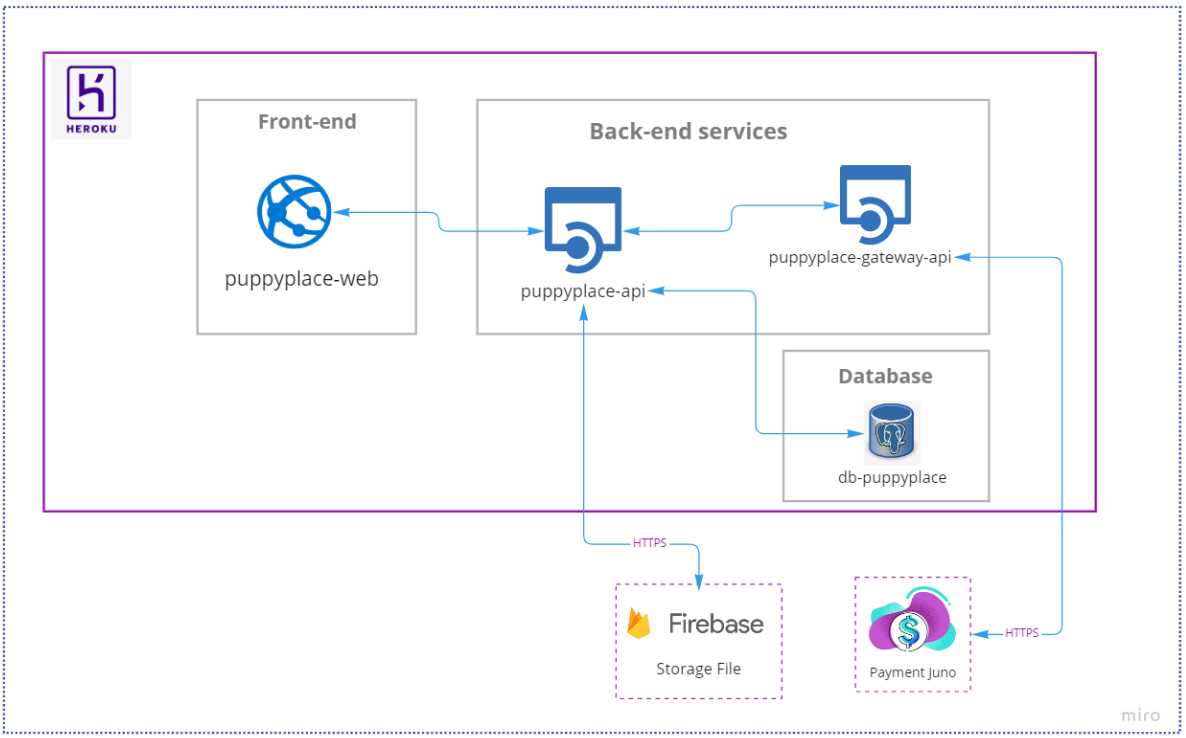


Figura 2 - Arquitetura Monolítica

Embora haja uma camada separada para o serviço de integração com o gateway de pagamento, as regras de negócio permanecem concentradas no serviço monolítico de backend

puppyplace-api e no banco de dados *db-puppyplace*. Isso caracteriza a aplicação como monolítica, uma vez que todas as funcionalidades e componentes do sistema estão presentes em uma única aplicação, sem a separação ou distribuição entre diferentes serviços independentes, como ocorre em arquiteturas mais modulares, como em arquitetura de micro serviços.

3.2. TECNOLOGIAS E FERRAMENTAS

Concluída a definição arquitetural, iniciou-se a análise sobre as tecnologias mais adequadas para a construção da aplicação. Na escolha das tecnologias, a principal preocupação não foi com tendências ou modismos, mas com a robustez do suporte da comunidade open source, a disponibilidade de documentação abrangente e a capacidade de resolver problemas de forma eficiente. Além disso, foi levado em consideração o nível de familiaridade e as competências individuais dos membros do grupo, a fim de evitar a adoção de ferramentas desconhecidas pela maioria, o que poderia comprometer o tempo de desenvolvimento.

O diagrama a seguir ilustra as tecnologias e componentes selecionados, bem como as suas respectivas interconexões, para a implementação proposta no MVP.

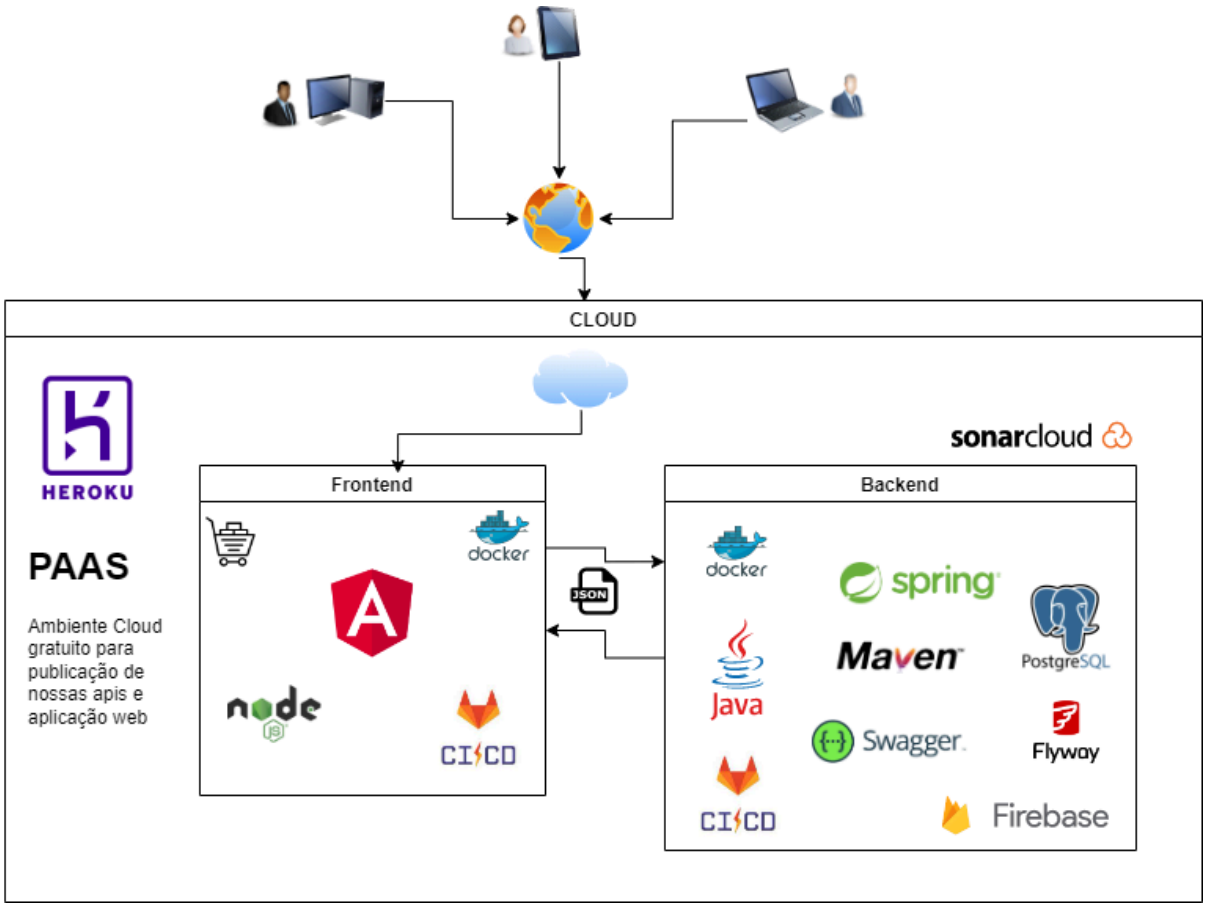


Imagem 3 - Desenho das Tecnologias

A escolha do sistema de versionamento foi orientada pela busca por um ambiente simples e automatizado, priorizando ferramentas gratuitas e de fácil integração. Nesse contexto, optou-se pelo GitLab como repositório de código-fonte, aproveitando seus recursos nativos para configurar a esteira de Integração e Entrega Contínua (CI/CD). O CI/CD é um conjunto de práticas que visa automatizar as etapas de integração e entrega de software, permitindo que o código seja testado e implantado de forma contínua e sem interrupções, melhorando a eficiência e reduzindo erros humanos (Fowler, 2018).

A integração contínua (CI) envolve a automação do processo de integração do código, onde mudanças feitas por diferentes desenvolvedores são incorporadas frequentemente ao repositório principal. Isso facilita a identificação e correção de erros de maneira precoce. Já a entrega contínua (CD) refere-se à prática de automatizar a entrega das mudanças de código para produção, permitindo uma atualização constante e rápida da aplicação. Essas práticas têm se mostrado essenciais para melhorar a qualidade do software e aumentar a velocidade de desenvolvimento (Humble & Farley, 2010).

Dentro do contexto do projeto, absorvi a tarefa de configurar a esteira de CI/CD paralelamente ao desenvolvimento do backend, motivado pelo meu interesse em aprender como automatizar os processos de integração e entrega contínua.

A plataforma de nuvem Heroku foi selecionada devido à sua gratuidade e à facilidade de integração com o GitLab, permitindo a automação do processo de deploy contínuo. Além disso, o Heroku possibilitou a criação de instâncias gratuitas de banco de dados PostgreSQL.

Outras ferramentas, como o Google Firebase Cloud, foram escolhidas por oferecerem um limite de uso gratuito antes da cobrança, permitindo que custos sejam gerados apenas quando a demanda da aplicação se torne significativa. O Firebase Cloud foi utilizado como armazenamento de arquivos (Storage) para as imagens dos produtos e do site.

Para facilitar a experiência de desenvolvimento, adotou-se o Swagger para a documentação automática da API. Para as migrações de banco de dados, utilizou-se o Flyway, um componente que facilita mudanças graduais e incrementais na estrutura de tabelas e entidades, garantindo controle e versionamento da evolução do banco de dados.

Na verificação da qualidade do código, foi integrado o SonarCloud à esteira de desenvolvimento, permitindo a análise estática do código e a identificação de 'bad smells' logo após o envio dos commits no repositório, além de monitorar a cobertura de testes.

Por fim, a escolha de componentes e linguagens como Java, Spring e Angular foi fundamentada na experiência prévia da equipe com essas tecnologias, bem como na forte comunidade open source, que proporciona fácil acesso a documentação e resolução de problemas.

3.3. DESENVOLVIMENTO DA APLICAÇÃO

A organização das atividades necessárias para a construção da aplicação foi estruturada em módulos e suas respectivas funcionalidades, por meio de refinamento contínuo com a equipe, e planejada com foco na entrega de valor ao produto, seguindo os princípios da metodologia ágil, especificamente o modelo Scrum. O Scrum é uma metodologia ágil que organiza o trabalho em ciclos curtos e iterativos, conhecidos como sprints, com o objetivo de fornecer entregas incrementais de valor ao cliente e ao produto. Para o gerenciamento das histórias de usuário, foi utilizada a ferramenta Jira, que facilita o

acompanhamento das atividades e a organização do fluxo de trabalho, conforme os princípios do Scrum (Schwaber, 2017).

Embora a equipe tenha tentado adotar o modelo de trabalho baseado em sprints, a dificuldade de conciliar as agendas dos membros, considerando que a maior parte do tempo disponível para o desenvolvimento do projeto estava restrita aos finais de semana, e o não cumprimento das cerimônias exigidas pelo Scrum, levou à decisão de transitar para o modelo Kanban. O Kanban, também uma abordagem ágil, permite um fluxo contínuo de trabalho sem a necessidade de ciclos fixos, proporcionando maior flexibilidade e fluidez no gerenciamento das atividades. Isso foi considerado mais adequado à dinâmica da equipe, permitindo a absorção das tarefas conforme a evolução das entregas e a disponibilidade dos membros da equipe (Anderson, 2010).

A imagem 4 ilustra as páginas 'Home' e 'Checkout' do MVP, respectivamente.

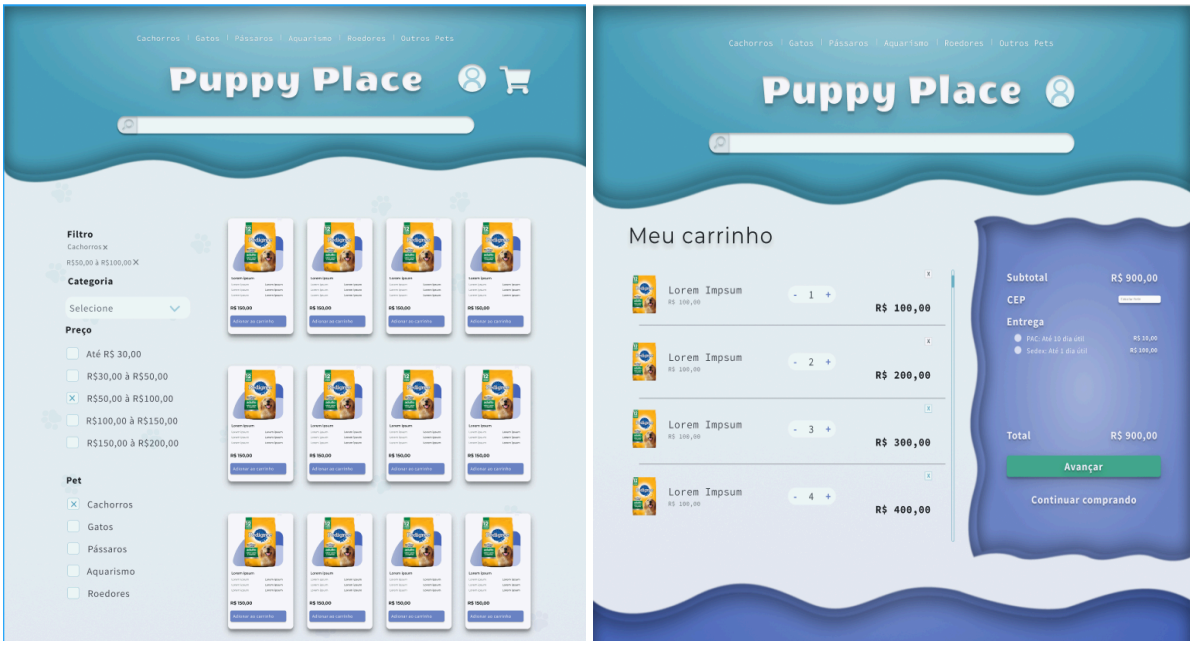


Figura 4 - Interface Home e Checkout, respectivamente.

Neste estágio do projeto, a interface visual do frontend estava funcional e a integração com a API do backend foi realizada para validar as regras de negócio da aplicação web. No entanto, a integração com plataformas de pagamento ainda não havia sido implementada. Com a conclusão das funcionalidades iniciais, iniciamos o desenvolvimento de um micro serviço para viabilizar a integração com uma plataforma de pagamento externa. A escolha

recaiu sobre a Juno Pagamentos (www.juno.com.br), que oferece um plano gratuito para transações de baixo volume, permitindo uma integração com sua API de forma síncrona.

Esse micro serviço foi projetado para interagir com a API da Juno, realizando a comunicação entre a aplicação e a plataforma de pagamento. Com a implementação desta funcionalidade, o sistema passou a ser capaz de processar pagamentos, integrando-se de maneira eficiente com a solução de pagamentos externa. Ao final dessa etapa, diversos módulos, pacotes e endpoints foram desenvolvidos, o que contribuiu para a definição da arquitetura de software e dos componentes que compõem o projeto, consolidando sua estrutura de serviços e garantindo o funcionamento adequado da aplicação.

3.4. DETALHES DA ARQUITETURA DE SOFTWARE E SEUS MÓDULOS

A arquitetura da monolítica representada na imagem 5, corresponde ao MVP que entregamos no projeto integrador ao final do primeiro semestre de 2021 deste curso.

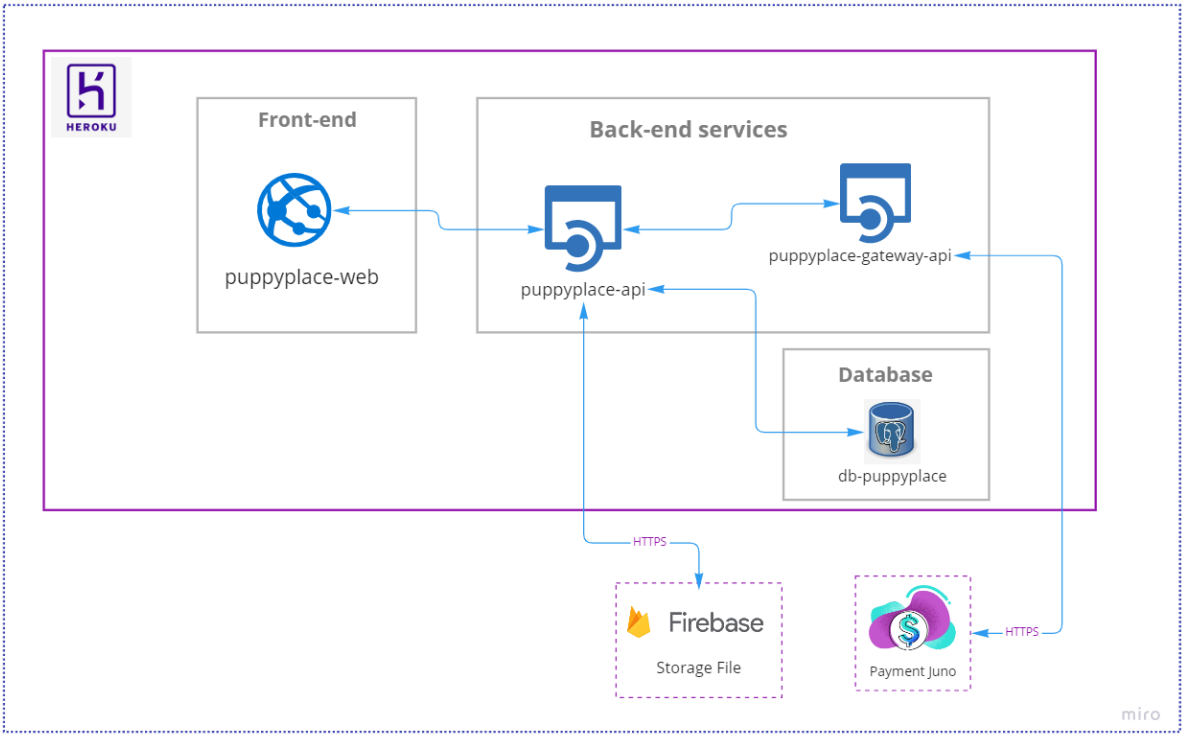


Figura 5 - Arquitetura da Solução

- **PuppyPlace-Web:** Trata-se do serviço de frontend da plataforma, voltado para o cliente final. A aplicação é responsável por fornecer a interface visual e interagir com o backend para validar e exibir as informações pertinentes ao usuário. O código-fonte

do projeto pode ser acessado através do repositório no GitHub: https://github.com/puppyplace/puppyplace-web.

- **PuppyPlace-API:** Este é o serviço de backend da plataforma, implementado como um monolito, responsável por processar as lógicas de negócio e fornecer os dados necessários para o frontend. O código-fonte deste serviço pode ser consultado no repositório no GitHub: https://github.com/puppyplace/puppyplace-api.
- **PuppyPlace-Gateway-API**: Este serviço atua como um gateway de integração entre a plataforma do e-commerce e a API da Juno Pagamentos, responsável por processar as transações financeiras. Foi desenvolvido para facilitar a comunicação com a plataforma de pagamentos, garantindo que as requisições de pagamento sejam processadas de maneira eficiente e segura. O código-fonte do projeto pode ser acessado através do repositório no GitHub: https://github.com/puppyplace/puppyplace-pagamentos-api.
- **Diagrama Entidade-relacionamento (ER):** Mostra o relacionamento entre as principais entidades do banco de dados relacional da puppy-place

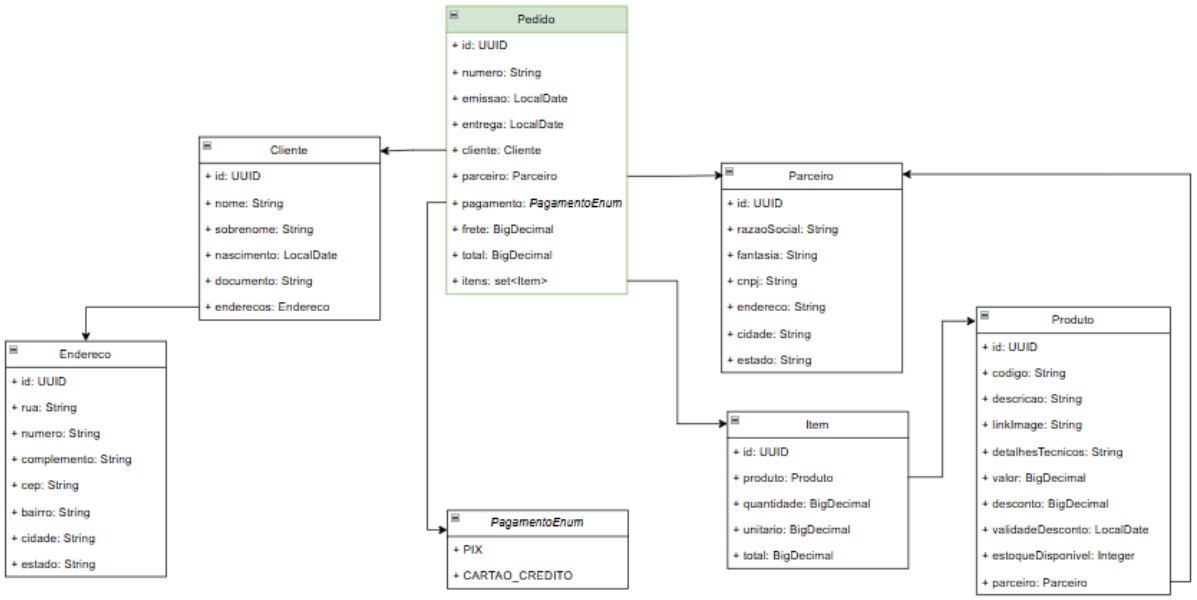


Figura 6 - Diagrama das entidades

4. A EVOLUÇÃO DO MVP

No início do segundo semestre do curso, a proposta para o projeto integrador teve como premissas a implementação de novas funcionalidades, a migração do provedor de cloud

de Heroku para Azure, e a transição da arquitetura monolítica para uma abordagem baseada em **microsserviços**. Os **microsserviços** são uma arquitetura de desenvolvimento de software na qual a aplicação é dividida em pequenos serviços independentes, cada um responsável por uma funcionalidade específica. Esses serviços são projetados para serem altamente escaláveis, fáceis de manter e implantáveis de maneira independente, o que facilita a evolução do sistema de forma mais ágil e eficiente (Newman, 2015). Essa abordagem permite que cada serviço seja desenvolvido, escalado e implementado de forma isolada, o que melhora a flexibilidade do sistema como um todo (Fowler, 2014).

A decisão de adotar uma arquitetura de microsserviços surgiu pela necessidade de maior escalabilidade e flexibilidade para evoluir o sistema conforme as demandas de negócio, além de permitir uma distribuição de cargas de trabalho mais eficiente. Outra razão para essa mudança foi a busca por um maior desacoplamento entre os módulos, o que possibilita atualizações e implementações contínuas sem a necessidade de interferir em outros componentes do sistema, característica fundamental de microsserviços (Richardson, 2018).

Para atender a essas novas diretrizes, a arquitetura da aplicação foi totalmente reformulada, com o objetivo de transformar cada módulo em um micro serviço, mantendo o sistema legado como uma camada de **BFF** (Backend for Frontend). O **BFF** é uma abordagem arquitetural onde uma camada de backend é responsável por orquestrar a comunicação entre o frontend e os microsserviços. Essa camada foi projetada para abstrair a complexidade dos microsserviços e fornecer ao frontend um contrato de dados simples e consolidado, de forma a reduzir a complexidade nas interações entre a interface de usuário e o backend (Adelsberger et al., 2020).

A escolha de manter o monolito como BFF visou minimizar o impacto no frontend, uma vez que os contratos de integração, tanto de entrada quanto de saída, já estavam previamente desenvolvidos, testados e validados. Com isso, o foco da reformulação estava em desacoplar os módulos do sistema e garantir a consistência na transferência de dados entre as APIs. Para facilitar esse processo, adotou-se a estratégia de criar uma **Library** contendo os **DTOs** (Data Transfer Objects), que serviriam como contratos de dados reutilizáveis dentro do ecossistema de microsserviços do backend.

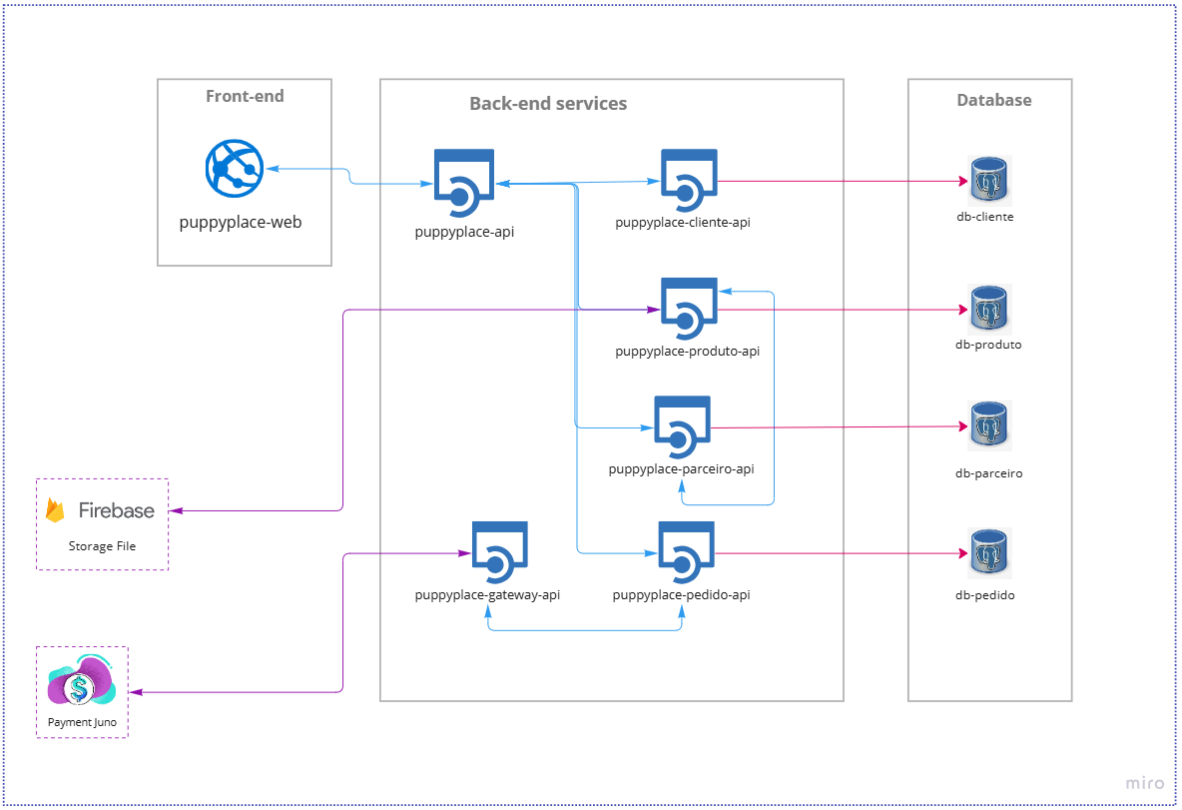


Figura 7 - Arquitetura microservice proposta

A partir da nova arquitetura para micros serviços e o planejamento das atividades, iniciei o processo de provisionamento da infraestrutura na plataforma de cloud Azure, considerando as necessidades de escalabilidade e gerenciamento eficientes dos microsserviços. Para viabilizar a migração, foi essencial adotar práticas de dockerização dos microsserviços, garantindo a criação de containers para cada um deles, além de implementar a orquestração desses containers com Kubernetes. Dessa forma, buscamos garantir a agilidade e a escalabilidade necessárias para suportar a arquitetura de microsserviços.

A infraestrutura foi configurada utilizando diversos componentes da Azure, cada um com uma função específica para atender aos requisitos do projeto. Os principais componentes utilizados foram:

Azure Container Registry (ACR): Serviço responsável por armazenar e gerenciar imagens de containers Docker de forma segura e eficiente. O ACR foi utilizado para armazenar as imagens dos microsserviços que foram desenvolvidas e dockerizadas (Microsoft, 2024).

 **Azure SQL Database**: Serviço de banco de dados relacional baseado em nuvem, altamente escalável e seguro. A escolha do Azure SQL Database foi motivada pela necessidade de manter uma solução de banco de dados relacional com alta disponibilidade e integração fácil com os microsserviços do sistema (Microsoft, 2024).

 **Azure DevOps**: Plataforma de colaboração e integração contínua, utilizada para automatizar o ciclo de vida do desenvolvimento de software, incluindo a construção, testes e deploy dos microsserviços. O Azure DevOps foi configurado para suporta toda a esteira de **CI/CD (Integração Contínua e Entrega Contínua)**, facilitando o controle de versão e as atualizações do sistema (Microsoft, 2024).

 **Azure Kubernetes Service (AKS)**: Serviço de orquestração de containers baseado em Kubernetes, utilizado para gerenciar, escalonar e monitorar os microsserviços em containers. O AKS foi adotado para garantir que todos os containers do backend fossem gerenciados de forma eficiente, com escalabilidade automática conforme a demanda (Microsoft, 2024).

 **Azure Static Web Apps**: Serviço destinado ao deploy de aplicações frontend baseadas em conteúdo estático. Utilizamos o Azure Static Web Apps para realizar o deploy da aplicação frontend, que é otimizada para escalabilidade, com integração direta ao GitHub para automação do processo de publicação (Microsoft, 2024).

Com intuito de experimentação tecnológica em ambiente acadêmico, apoiei a equipe na decisão estratégica de não limitar a implementação dos novos microsserviços à utilização exclusiva da linguagem Java. Incluiu-se o Python como uma segunda linguagem de programação, motivada pelo interesse e curiosidade de aprendizado de cada membro da equipe. Dessa forma, os serviços relacionados a **Produto** e **Parceiro** seriam desenvolvidos em Python, e além da infra eu também desenvolvi o produto, e tínhamos apenas um único integrante da equipe com conhecimento em Python, o qual se incumbiu de fornecer suporte.

O provisionamento da infraestrutura na **Azure** foi realizado de forma relativamente simples, uma vez que as configurações padrão foram utilizadas para a criação e o

gerenciamento dos recursos necessários para a execução dos microsserviços.

O planejamento das atividades foi registrado no **Azure DevOps Board** seguindo com a metodologia **Kanban** para gerenciar o fluxo de trabalho, uma vez que não havia dedicação total ao projeto, o que inviabilizaria a implementação do modelo de **Sprint**.

Durante o desenvolvimento, surgiram diversos desafios, sendo o principal deles a ausência de documentação adequada dos micros serviços e da modelagem de dados. Isso levou a equipe a recorrer a um processo de **engenharia reversa** para entender o código existente, o que dificultou a segmentação e o isolamento de funcionalidades essenciais para a migração e adaptação à nova arquitetura de microsserviços.

No que tange ao modelo de banco de dados, a transição para microsserviços apresentou desafios adicionais. Como cada serviço passa a ser independente, algumas chaves de tabelas, que anteriormente estavam associadas ao banco de dados monolítico, precisaram ser reestruturadas para suportar o novo modelo. Por exemplo, a tabela de **Pedido** foi vinculada ao **Cliente** no banco de dados monolítico, mas, no modelo de microsserviços, essa associação precisou ser redefinida, já que as entidades passaram a residir em bancos de dados separados, implicando na necessidade de novas estratégias de integração entre os dados distribuídos.

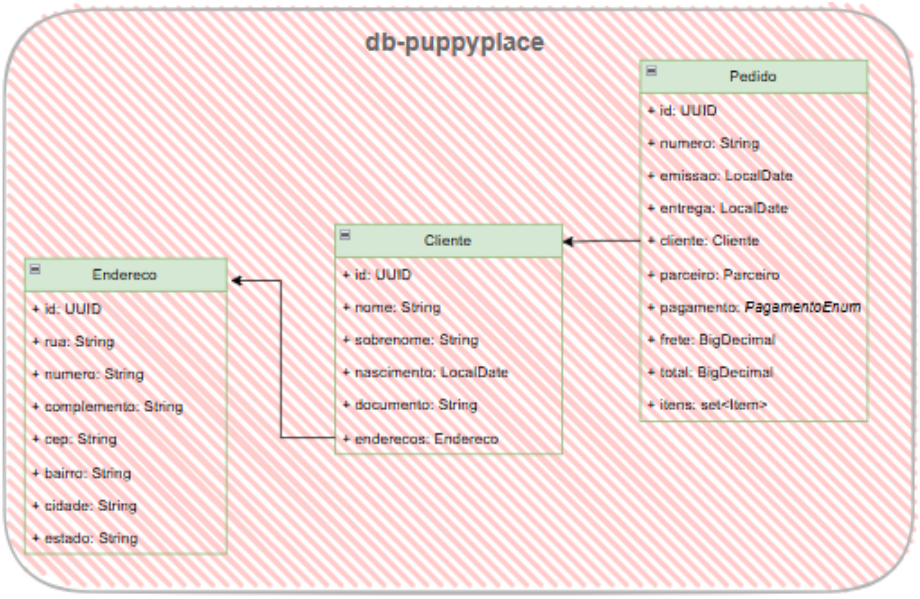


Figura 8 - Integridade relacional monolito Pedido x Cliente

No contexto da arquitetura de microsserviços, um dos desafios significativos envolve

a estruturação e a gestão de dados, especialmente quando se lida com a segregação das bases de dados. Ao migrar de um modelo monolítico para microsserviços, algumas chaves de tabelas, que antes estavam interligadas em um único banco de dados, deixam de existir como uma única entidade. Isso ocorre porque, em um sistema distribuído, os dados são segmentados entre os diversos microsserviços, sendo armazenados em bancos de dados independentes. Essa abordagem implica na necessidade de uma nova forma de organizar e acessar os dados que eram anteriormente referenciados de forma centralizada.

Uma técnica que se mostra eficaz para lidar com esse desafio é o **Domain-Driven Design (DDD)**. O DDD é uma abordagem que visa a segmentação dos dados e a definição clara de limites entre os diferentes domínios de um sistema. No contexto de microsserviços, essa técnica permite a segregação dos dados em diferentes contextos, permitindo que cada micro serviço tenha seu próprio banco de dados e controle sobre suas próprias entidades. Como exemplo, no caso de uma tabela **Pedido x Cliente** em um sistema monolítico, ao adotar DDD, cada micro serviço relacionado poderia manter a tabela **Pedido** e a tabela **Cliente** em seus respectivos bancos de dados. A chave de relacionamento entre as tabelas, anteriormente comum, seria substituída por outra forma de comunicação entre os microsserviços, como eventos ou APIs, garantindo a consistência eventual dos dados e a integridade do sistema (Evans, 2004).

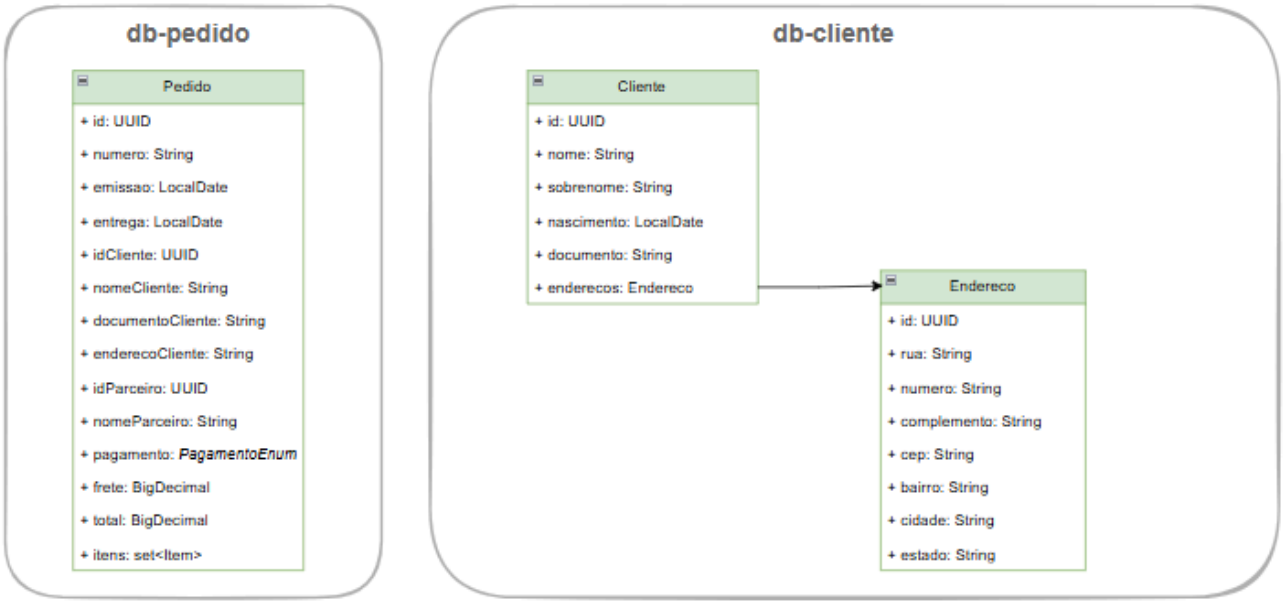


Figura 9 - Representação Pedido x Cliente em db separados

Essa segmentação através do DDD permite a criação de contextos limitados, onde

cada serviço pode ser tratado de maneira independente e com maior flexibilidade na evolução de suas entidades e relacionamentos. Além disso, a técnica facilita a manutenção de uma arquitetura de dados mais escalável e resiliente, fundamental para sistemas distribuídos como os baseados em microsserviços.

No processo de remodelagem do banco de dados de pedidos, foram incluídas informações básicas sobre o cliente e o parceiro. Para quaisquer dados adicionais necessários, o serviço de pedidos precisaria consultar os serviços dedicados ao cliente ou ao parceiro, a fim de enriquecer as informações e garantir a consistência dos dados. No entanto, a execução dessa remodelagem de dados, aliada à implementação de novos microsserviços em linguagens distintas, juntamente com a falta de documentação adequada, resultou em dificuldades significativas. Como consequência, não conseguimos completar a migração para a arquitetura de microsserviços de maneira bem-sucedida e dentro do prazo estabelecido.

A migração para microsserviços não foi concluída com sucesso, mas, por meio dessa experiência, adquirimos valiosas lições aprendidas. Em processos de migração, é imprescindível seguir as boas práticas descritas na literatura técnica e realizar um planejamento cuidadoso, pois um único erro pode comprometer o sucesso de todo o projeto. Em nosso caso, o erro foi iniciar o desenvolvimento sem antes validar um módulo de forma isolada. Uma migração bem-sucedida exige uma prova de conceito (PoC) para avaliar os riscos e identificar possíveis obstáculos antes de implementar as modificações em larga escala. A PoC permite medir de forma analítica os desafios a serem enfrentados, o que possibilita um planejamento mais eficiente e a mitigação dos problemas que possam surgir.A PoC é uma abordagem fundamental para testar a viabilidade técnica de uma solução, como no caso de uma migração para microsserviços, antes de sua aplicação em larga escala (RAHMAN et al., 2019).

A partir dessa experiência, discutimos o tema com o nosso orientador, Prof. José Teodoro, que compartilhou uma experiência de sucesso na migração gradual de sistemas monolíticos para microsserviços. Com base nas melhores práticas observadas, sugerimos uma nova abordagem arquitetural que permita a transição de um sistema monolítico para microsserviços de forma incremental, minimizando riscos e otimizando o processo de migração.

5. AVALIANDO ESTRATÉGIAS PARA REALIZAR A MIGRAÇÃO

A migração de uma arquitetura monolítica para microsserviços é um processo complexo que exige planejamento cuidadoso e a escolha de estratégias adequadas para minimizar riscos e garantir uma transição bem-sucedida. As principais estratégias ou patterns (padrões) para realizar essa migração incluem:

5.1. Strangler Fig Pattern (Padrão da Figueira Estranguladora)

O *Strangler Fig Pattern* é uma das abordagens mais populares e eficazes para migrar de um sistema monolítico para microsserviços. Nessa estratégia, um novo sistema baseado em microsserviços é gradualmente introduzido ao lado do sistema monolítico. A cada iteração, partes do sistema monolítico são substituídas pelos novos microsserviços, até que o monolito seja completamente substituído. Esse padrão reduz o risco ao permitir que o sistema existente continue funcionando enquanto a migração é realizada de forma incremental. (Fowler, M., 2015)

5.2. Incremental Approach (Abordagem Incremental)

A abordagem incremental envolve a migração de funcionalidades específicas ou módulos do sistema monolítico para microsserviços, um de cada vez. Essa estratégia permite que as equipes se concentrem em uma parte do sistema por vez, realizando migrações em fases. A cada fase, novos microsserviços são introduzidos enquanto a base monolítica permanece em operação. Isso reduz os riscos ao permitir testes constantes e manutenção do sistema durante a migração. (Bass, L., Clements, P., & Kazman, R., 2012)

5.3. Decompose by Business Capabilities (Decomposição por Capacidades de Negócio)

Neste padrão, a migração é feita de acordo com as capacidades de negócio do sistema, em vez de focar em funções técnicas. Isso significa que os serviços são quebrados e organizados conforme as necessidades do negócio, como pedidos, clientes ou pagamentos. A decomposição é baseada em uma análise cuidadosa de como a empresa opera, e cada novo micro serviço reflete uma função de negócio independente. Essa abordagem facilita a

integração de microsserviços com equipes de negócios e proporciona uma arquitetura mais alinhada com a estratégia empresarial. (Newman, S., 2015)

5.4. Decompose by Subdomains (Decomposição por Subdomínios)

A decomposição por subdomínios é um conceito derivado do *Domain-Driven Design (DDD)*. Aqui, a migração é feita com base nos subdomínios da aplicação, ou seja, cada micro serviço é responsável por um subdomínio específico da lógica de negócios. Essa abordagem é ideal quando o sistema é complexo e há uma clara distinção entre os diferentes componentes do sistema. Cada micro serviço é projetado para ser independente, com seu próprio banco de dados e regras de negócio. O DDD é uma abordagem útil para mapear os limites contextuais e ajudar na decomposição do monolito. (Evans, E., 2004)

5.5. API Gateway Pattern

O padrão de API Gateway ajuda a gerenciar a comunicação entre os microsserviços e o front-end de forma centralizada. Ao migrar de monolito para microsserviços, a implementação de um *API Gateway* facilita o controle das rotas e das integrações, fornecendo uma interface unificada para os consumidores da aplicação. O API Gateway também pode oferecer funcionalidades como balanceamento de carga, autenticação, controle de tráfego e monitoramento. (Fowler, M., 2014)

5.6. Database Per Service (Banco de Dados por Serviço)

Ao migrar para microsserviços, uma das principais mudanças é a separação do banco de dados. No monolito, geralmente existe um único banco de dados compartilhado por todas as partes da aplicação. Na arquitetura de microsserviços, cada serviço tem seu próprio banco de dados, o que permite que ele evolua independentemente dos demais. Isso melhora a escalabilidade e a resiliência do sistema, mas requer um planejamento cuidadoso para garantir consistência entre os dados distribuídos. (Newman, S., 2015)

5.7. Proxy Pattern

Este padrão envolve o uso de proxies para redirecionar as solicitações do sistema monolítico para os novos microsserviços à medida que eles são migrados. O proxy pode ser usado para gerenciar as comunicações de forma eficiente durante a migração gradual, sem

impactar a experiência do usuário final. A ideia é que as solicitações do front-end ainda sejam processadas pelo sistema monolítico, mas são redirecionadas para os microsserviços onde necessário.(Newman, S., 2015)

6. PROPOSTA DE MIGRAÇÃO DE ARQUITETURA MONOLÍTICA PARA MICROSSERVIÇOS: ESTRATÉGIAS E BENEFÍCIOS

Em cenários complexos, como a migração de uma arquitetura monolítica de um e-commerce com funcionalidades de MarketPlace, é essencial adotar práticas que garantam a alta disponibilidade do sistema (7x24h) e minimizem impactos em produção. Dado o desafio de realizar essa transformação em um ambiente crítico, a adoção de estratégias recomendadas pela literatura técnica é fundamental para alcançar uma transição bem-sucedida e gradativa para uma arquitetura de microservices.

A abordagem proposta baseia-se na aplicação do padrão **Strangler Fig**, combinado com o uso de um **API Gateway** como elemento central de orquestração. O padrão Strangler Fig permite a substituição incremental de funcionalidades monolíticas, isolando novas implementações como microsserviços e mantendo o sistema legado ativo até que todos os módulos sejam migrados. Essa estratégia reduz significativamente o risco de falhas críticas, pois opera em um ambiente controlado, permitindo testes e validações contínuas antes da transição completa para produção.

O uso de um API Gateway complementa essa abordagem ao centralizar as chamadas dos clientes, mascarando a coexistência do monólito e dos microsserviços. Isso assegura que os usuários finais não percebam mudanças estruturais durante o processo de migração, proporcionando uma experiência contínua e sem interrupções.

Essa combinação de estratégias apresenta características favoráveis à viabilização da migração de forma gradual e sem impactos significativos. Além disso, a abordagem permite lidar com eventuais falhas de maneira controlada, assegurando que o sistema mantenha sua operação em condições críticas enquanto a arquitetura é progressivamente transformada.

6.1. MONITORAMENTO E OBSERVABILIDADE

O conceito de monitoramento e observabilidade é essencial para garantir o desempenho adequado de sistemas complexos, como aqueles baseados em microsserviços. Neste contexto, ao migrarmos para uma arquitetura de microsserviços, a camada de monitoramento se torna um componente crítico para acompanhar métricas como tempo de resposta, taxa de erros, e identificar gargalos de desempenho em tempo real. A observabilidade vai além do simples monitoramento ao proporcionar insights detalhados sobre o comportamento do sistema, facilitando a detecção de anomalias e melhorando a resiliência e a disponibilidade do sistema (Turner et al., 2020).

Em um ambiente de microsserviços, a telemetria distribuída e o uso de métricas em tempo real são fundamentais para a adaptação dinâmica do sistema, garantindo que eventuais falhas possam ser rapidamente mitigadas sem afetar a experiência do usuário (Granowski et al., 2019).

é uma coleção de APIs, bibliotecas, agentes e instrumentações que permite a coleta de dados de monitoramento de aplicações, como métricas, logs e rastreamento de transações. Ele fornece uma maneira padronizada de instrumentar o código para coletar esses dados, ajudando a garantir que as métricas e logs sejam consistentes entre diferentes componentes e serviços. O Open Telemetry, por ser open source, oferece flexibilidade e integração com diversos sistemas de backend para análise de desempenho.

é uma ferramenta de monitoramento e alertas altamente escalável e com forte suporte na comunidade open source. Ele coleta e armazena métricas em tempo real em um formato de série temporal, o que o torna ideal para ambientes de microsserviços. O Prometheus se destaca pela sua eficiência na coleta de métricas, a flexibilidade na configuração de consultas e a capacidade de fornecer alertas com base em condições específicas, como falhas de serviço ou picos de latência .

é uma plataforma de visualização de métricas que pode ser integrada ao Prometheus para exibir os dados coletados de uma forma compreensível e personalizada. Com o Grafana, é possível criar dashboards interativos e intuitivos, que ajudam a monitorar o desempenho dos serviços em tempo real e a identificar rapidamente problemas potenciais,

como picos de latência ou aumento de erros. A visualização proporciona insights valiosos sobre a saúde do sistema e facilita a tomada de decisões para ajustes e melhorias .

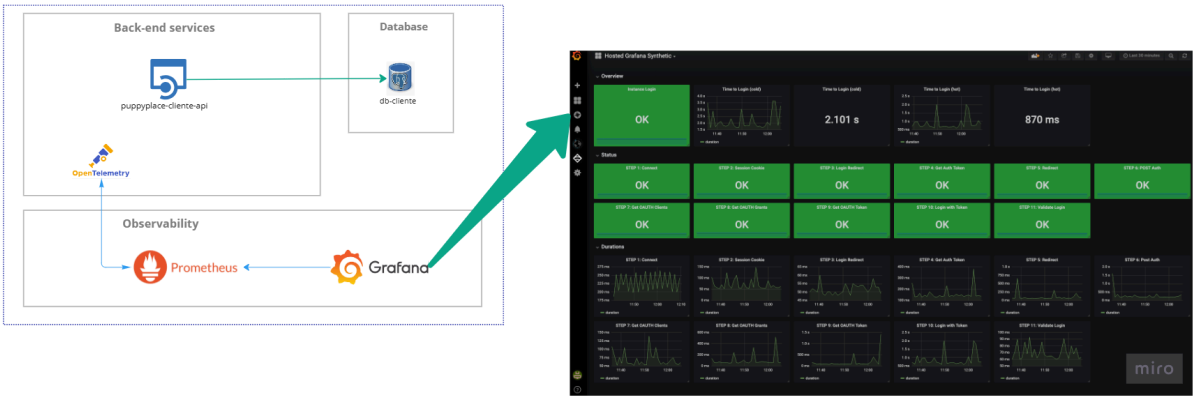


Figura 10 - Monitoramento & Observabilidade

A integração dos três componentes — Open Telemetry, Prometheus e Grafana — possibilita uma implementação robusta de monitoramento e observabilidade, fundamental em arquiteturas de microsserviços. O Open Telemetry é responsável por coletar dados de desempenho e métricas de todos os microsserviços, os quais são enviados para o Prometheus, que os armazena de forma eficiente e em formato de séries temporais. Em seguida, os dados são visualizados por meio do Grafana, que permite a criação de dashboards personalizados, oferecendo insights detalhados sobre a performance do sistema, taxas de erro e possíveis falhas.

Essa arquitetura proporciona uma visão abrangente e em tempo real da aplicação, essencial para garantir a alta disponibilidade e o bom funcionamento do sistema, permitindo a rápida identificação de problemas e a aplicação de correções antes que impactem os usuários finais. Além disso, o monitoramento contínuo e a visualização dinâmica dos dados contribuem para decisões informadas, assegurando que o sistema opere de maneira otimizada e resiliente.

6.2. GATEWAY

Na arquitetura proposta, uma camada de gateway será implementada utilizando o servidor NGINX como Load Balancer, com a finalidade de distribuir as requisições de forma equilibrada entre os diversos serviços da camada de backend. Essa abordagem visa garantir escalabilidade horizontal, redundância e tolerância à falha, características essenciais para a

manutenção de sistemas resilientes e de alta disponibilidade. O NGINX é amplamente utilizado como balanceador de carga devido à sua alta performance e flexibilidade, permitindo a distribuição eficiente de tráfego entre várias instâncias de serviço (Nguyen et al., 2017).

Além disso, será incorporada à camada de gateway uma implementação de segurança para autenticação das requisições. Nesse contexto, uma instância do NGINX atuará como autenticador de token JWT (JSON Web Token), oferecendo proteção ao backend contra acessos diretos ou não autorizados. O uso de JWT tem se consolidado como uma solução eficiente para autenticação e autorização em arquiteturas distribuídas, uma vez que permite que a identidade do usuário seja validada de forma segura e sem a necessidade de armazenamento de sessões no servidor (Santos et al., 2020).

Outra prática recomendada que será aplicada na camada de gateway é a implementação de Rate Limiting, que visa a limitação de requisições por unidade de tempo. Essa técnica é essencial para prevenir que serviços internos sejam sobrecarregados por um volume excessivo de tráfego, além de proteger o sistema contra ataques de negação de serviço (DoS) (Wang et al., 2019).

Por fim, o NGINX será utilizado como Proxy Reverso com regras de roteamento. O objetivo é encaminhar cerca de 30% das requisições para o novo micro serviço. Isso garante flexibilidade e controle sobre o tráfego, possibilitando o desacoplamento gradual dos módulos do sistema à medida que a migração para microsserviços avança, o percentual de redirecionamento aumenta até chegar aos 100%. Isto permite um controle centralizado sobre o roteamento das requisições e melhorando a experiência do usuário (Choi et al., 2021).

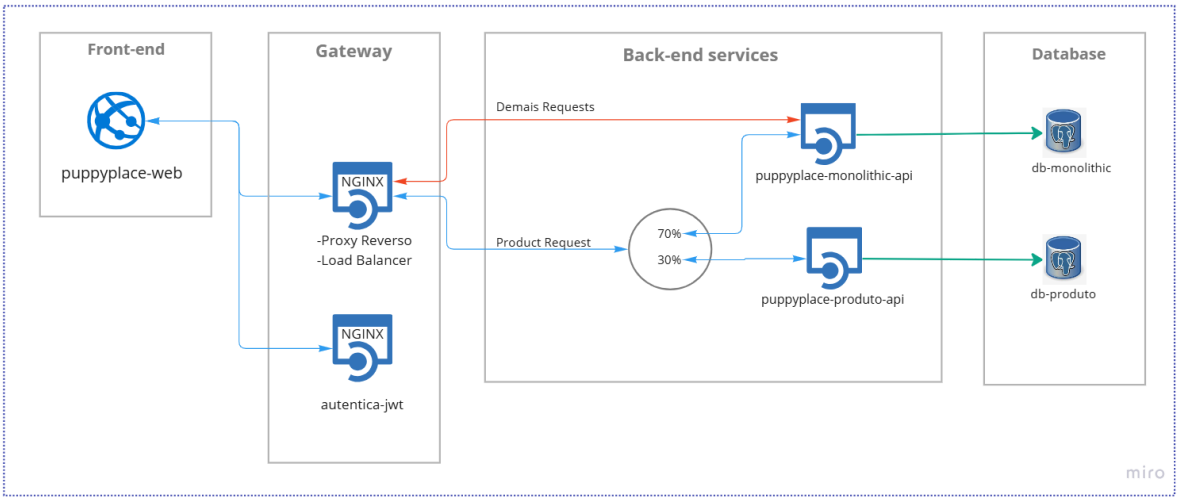


Figura 11 - Camada Gateway com NGNIX

6.3. ARQUITETURA EVOLUTIVA

A arquitetura evolutiva é uma abordagem de design de sistemas que prioriza a flexibilidade e a capacidade de adaptação contínua ao longo do tempo, em vez de buscar uma solução otimizada e imutável desde o início. Esse conceito é especialmente útil em ambientes ágeis e de alta incerteza, como no desenvolvimento de software ou na migração de arquiteturas monolíticas para microsserviços. Em sistemas complexos e dinâmicos, a arquitetura evolutiva oferece uma estratégia mais robusta ao permitir que o sistema seja ajustado conforme novas necessidades ou feedback surgem durante o desenvolvimento.

O princípio central dessa abordagem é que os sistemas devem ser projetados para serem continuamente melhorados, com base em experiências reais e nas mudanças de contexto. Em vez de construir uma arquitetura completamente ideal desde o início, foca-se na criação de uma base funcional e escalável, que será progressivamente refinada. Assim, a equipe pode lidar com a incerteza e com a evolução das demandas sem comprometer a capacidade de adaptação futura.(Lewis, J., & Fowler, M.,2014)

Principais Características:

6.3.1. **Iteração e Adaptação**: O sistema evolui por meio de ciclos contínuos, com ajustes baseados no feedback dos usuários e stakeholders.

6.3.2. **Flexibilidade**: A arquitetura é projetada para acomodar mudanças sem grandes refatorações, permitindo uma adaptação ágil.

6.3.3. **Débitos Técnicos Planejados**: Algumas práticas podem ser temporariamente adiadas para acelerar a entrega inicial, com a promessa de serem corrigidas posteriormente.

6.3.4. **Foco na Entrega de Valor**: A arquitetura evolutiva prioriza a entrega contínua de valor ao negócio, o que permite testar o sistema em um ambiente real e ajustá-lo conforme necessário.

A adoção dessa abordagem implica um entendimento claro dos desafios e das oportunidades que surgem à medida que o sistema é desenvolvido. O conceito de Domain-Driven Design (DDD) também está intimamente relacionado à arquitetura evolutiva, pois fornece uma metodologia para lidar com sistemas complexos e de rápido desenvolvimento, permitindo que a arquitetura evolua conforme o domínio de negócios se desenvolve e se refina. (Evans, E. ,2003)

6.4. MICROSSERVIÇOS

Para adotar uma estratégia eficiente de separação em microsserviços, a chave está em definir limites claros de responsabilidade e de comunicação entre os serviços, sempre mantendo em mente o princípio de independência e escala horizontal.

Algumas das estratégias práticas para a separação de microsserviços, são:

6.4.1. **Baseada no Domínio de Negócio:** Uma das abordagens mais eficazes é adotar Domain-Driven Design (DDD), que propõe a modelagem do sistema de acordo com o domínio de negócios. O DDD divide o sistema em diferentes Bounded Contexts (Contextos Delimitados), que representam áreas distintas de um sistema que podem ser isoladas em microsserviços. Dentro de cada contexto, as entidades, as regras de negócios e as operações devem estar fortemente acopladas, mas com pouca ou nenhuma dependência com outros contextos.

6.4.2. **Baseada na Funcionalidade:** A separação por funcionalidade envolve a criação de microsserviços de acordo com funcionalidades ou áreas específicas dentro do

sistema. Esse padrão é mais direto e pode ser adequado quando o domínio do sistema é simples ou já está bem definido.

6.4.3. **Estratégia de Serviços Independentes:** Uma estratégia que funciona bem em sistemas grandes e complexos é a separação de microsserviços autônomos, onde cada micro serviço possui seu próprio banco de dados e a comunicação entre eles é feita via APIs, tipicamente RESTful ou gRPC. Essa estratégia evita problemas de dependências e transações distribuídas, favorecendo a consistência eventual (eventual consistency), que é uma característica importante em arquiteturas distribuídas.

Boas Práticas Adicionais:

- ➔ **Manutenção de Coesão e Baixo Acoplamento**: Para garantir que os microsserviços permaneçam eficientes e fáceis de manter, deve-se manter alta **coesão** (todas as funcionalidades relacionadas a um contexto de negócios devem estar dentro do mesmo micro serviço) e baixo acoplamento (os microsserviços devem depender minimamente uns dos outros).
- ➔ **Gerenciamento de Dados**: No contexto de microsserviços, cada serviço deve ter seu próprio banco de dados (princípio do Database per Service). Isso garante que um micro serviço possa evoluir independentemente dos outros sem causar dependências rígidas.
- ➔ **Testes e Automação:** A separação em micro serviços requer uma **a**utomação robusta de testes para garantir que cada micro serviço funciona corretamente isoladamente e também quando integrado aos outros. A integração contínua e a entrega contínua (CI/CD) são essenciais para manter a qualidade do sistema à medida que ele evolui.

Uma das abordagens mais eficazes, como mencionado, é adotar o Domain Driven Design (DDD), especialmente porque buscamos delimitar os contextos específicos para cada micro serviço. Essa estratégia permite uma separação clara das entidades de banco de dados e das funcionalidades de cada serviço, promovendo um baixo acoplamento entre os serviços e garantindo sua independência. Ao isolar cada serviço dentro de seu contexto de domínio, conseguimos reduzir a complexidade e facilitar a evolução de cada micro serviço de forma autônoma. Isso facilita a manutenção e a escalabilidade, mantendo a arquitetura flexível e resiliente a mudanças.

O nosso foco com isso é garantir que o código que funciona no monólito seja implantado em um micro serviço e que funcione exatamente igual. Isso nos dará uma segurança argumentativa em defesa dessa proposta, pois caso um bug de negócio ocorra, existe alta probabilidade que o bug também ocorra no monólito.

Antes de iniciar o desacoplamento, é necessário alterar a arquitetura de comunicação entre a camada gateway/Load Balance implementando o NGINX, como visto anteriormente, para que o componente de roteamento seja o responsável por orquestrar as requests até o endpoint de destino, que pode estar no monólito (legado) ou nos novos microsserviços, como abordamos no item de proxy reverso. Lembrando, que neste momento inicial, a arquitetura está sendo adaptada com a implementação da camada do NGINX como intermediário e todas as requisições estão direcionadas para o monolito. Conforme o desacoplamento acontece, as rotas serão modificadas no NGINX redirecionando as requisições para os novos microsserviços.

Portanto, seguindo esta estratégia teremos uma arquitetura conforme ilustrado no desenho a seguir.

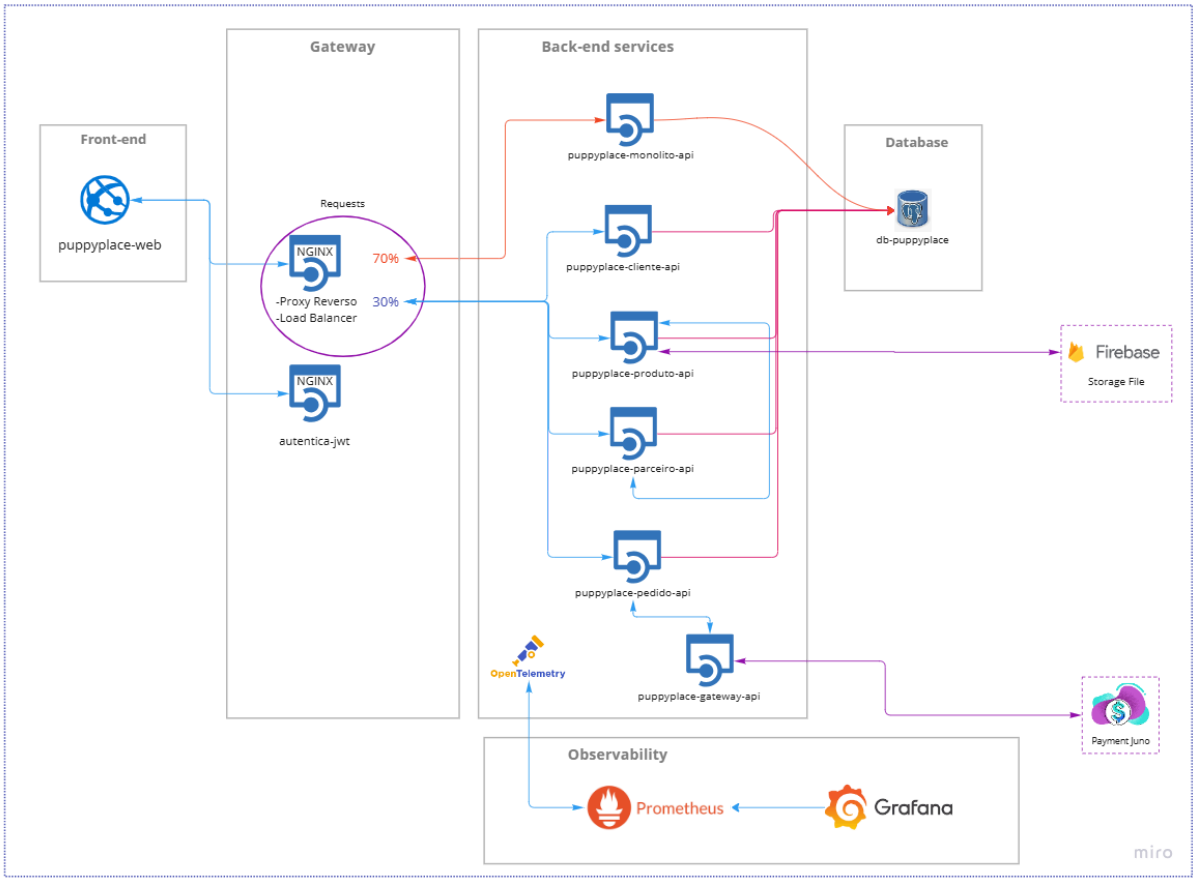


Figura 12 - Arquitetura inicial para favorecer migração

.

6.5. BANCO DE DADOS

Ao adotar uma arquitetura de microsserviços, um dos desafios mais complexos e sensíveis refere-se à gestão dos dados. Em um ambiente de microsserviços, a independência de dados é crucial para garantir a escalabilidade, resiliência e baixa dependência entre os serviços. No entanto, enquanto os microsserviços são desacoplados, o problema de dados compartilhados entre serviços continua a exigir uma solução eficaz.

Inicialmente, propomos a manutenção do monolito como a golden source (fonte primária de dados) até que os dados sejam replicados e estejam devidamente integrados nos bancos de dados específicos de cada micro serviço. Esse processo envolve a replicação dos dados, com cada micro serviço controlando e operando apenas sobre suas próprias tabelas, mantendo a independência e isolamento dos dados.

Para ilustrar, consideremos o caso do catálogo de produtos. Inicialmente, o catálogo de produtos está armazenado no banco de dados monolítico, e ao desacoplar o micro serviço de produto (no caso, o serviço puppyplace-produto-api), o banco de dados principal ainda continua sendo o monolito. A partir dessa situação, a estratégia adotada é a replicação de dados para o novo banco de dados de produtos, mantendo a consistência entre as fontes de dados.

A solução para a replicação de dados entre o monolito e os microsserviços envolve a implementação de uma camada de mensageria, utilizando o RabbitMQ. Após a execução de uma operação de cadastro (POST) realizada pelo serviço no monolito, uma mensagem contendo o objeto de dados, como no caso do produto, é publicada em uma fila do RabbitMQ. Esse mecanismo garante que o microserviço puppyplace-replica-produto-api consuma a mensagem, atualizando o banco de dados exclusivo para o domínio de produto, pertencente ao micro serviço.

Esse processo de replicação assíncrona, por meio de filas e mensagens, assegura a consistência eventual dos dados entre o monolito e os microsserviços. A consistência eventual é um conceito fundamental em arquiteturas de microsserviços, pois permite que os dados se tornem consistentes ao longo do tempo, sem a necessidade de uma sincronização

imediata entre os sistemas (Newman, 2015; Fowler, 2015). Essa abordagem, amplamente adotada, facilita a transição de sistemas monolíticos para uma arquitetura de microsserviços sem impactar a operação do sistema.

Além disso, a utilização de mensageria não apenas facilita a separação de dados, mas também contribui para a observabilidade do sistema. Utilizando ferramentas como o Grafana, é possível monitorar em tempo real as métricas relacionadas à replicação de dados, como a comparação entre a quantidade de registros gravados no banco de dados do monolito e no banco de dados de produtos dos microsserviços. Este monitoramento proporciona uma visão clara e contínua da integridade e performance do sistema, sendo essencial para a manutenção da qualidade e disponibilidade.

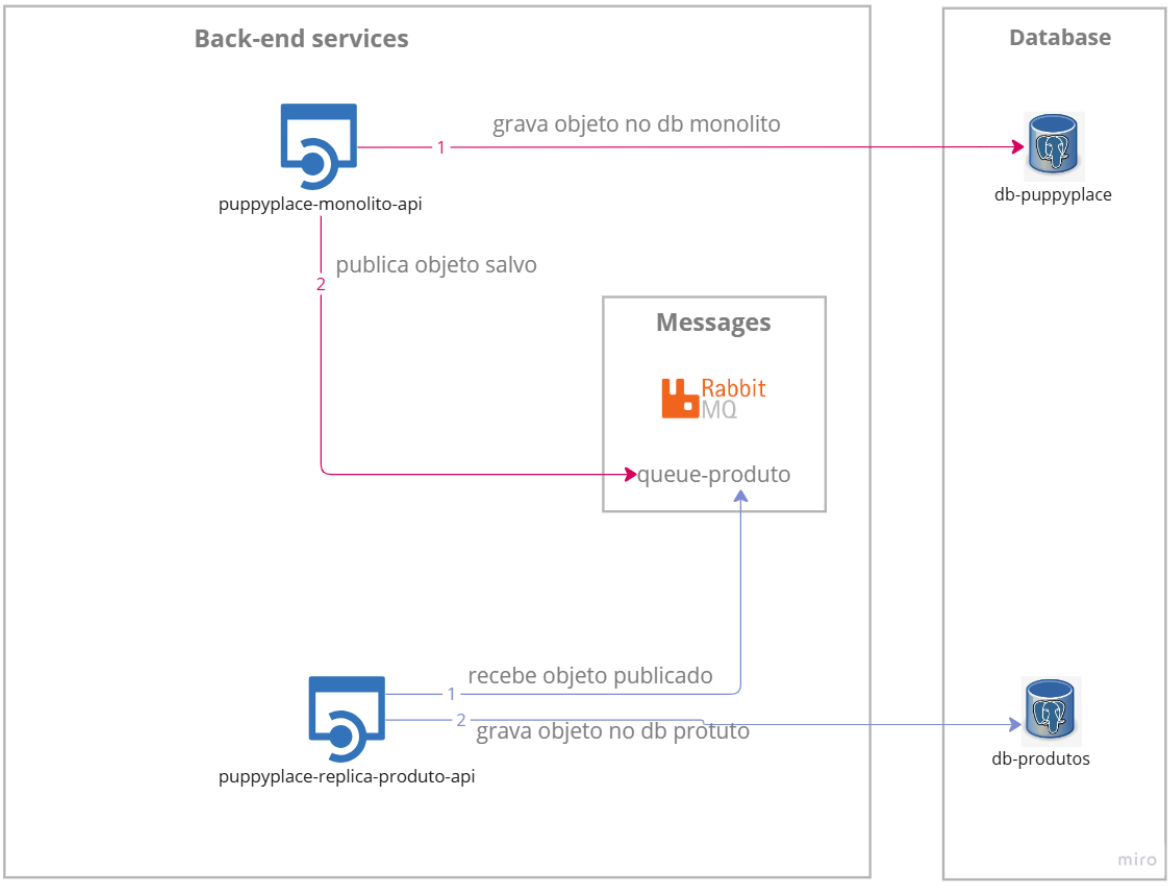


Figura 13 - Fluxo de réplica de dados

Este fluxo garante que, durante um intervalo de tempo determinado, o monolito possa ser mantido em operação enquanto ocorre a migração dos dados. A migração será realizada por meio de scripts SQL, que extraem os dados do banco de dados do monolito e os replicam no novo banco de dados, enquanto um mecanismo de atualização online assegura que os dados permaneçam sincronizados. Esse processo fornece uma visão contínua da evolução do banco de dados, permitindo o acompanhamento e a validação do enriquecimento das bases de dados.

Esse mecanismo possibilita que, gradualmente, o micro serviço, como no caso do serviço de produtos, possa se conectar diretamente ao seu banco de dados específico. Após essa mudança, a necessidade de replicação de dados será descontinuada, pois os dados estarão completamente isolados e consistentes nos novos bancos de dados de domínio, alinhando-se com as práticas de consistência eventual (Newman, 2015).

Essa abordagem oferece uma transição mais fluida e segura para a arquitetura de microsserviços, permitindo que a migração ocorra sem interromper os serviços existentes e sem risco de perda de dados.

Adotando essa estratégia, ao longo do tempo, os dados serão distribuídos para seus respectivos bancos de dados, proporcionando que os microserviços adquiram autonomia gradualmente. Esse processo permitirá que cada microserviço opere de forma independente, seguindo seu fluxo de negócios específico. Eventualmente, com a conclusão da migração, os microserviços estarão 100% desacoplados, com suas próprias bases de dados e responsabilidades, como ilustrado no diagrama final a seguir. Esse processo de transição é fundamental para garantir a escalabilidade e resiliência do sistema, conforme recomendado por diversas práticas de arquitetura de microsserviços (Newman, 2015; Fowler, 2015).

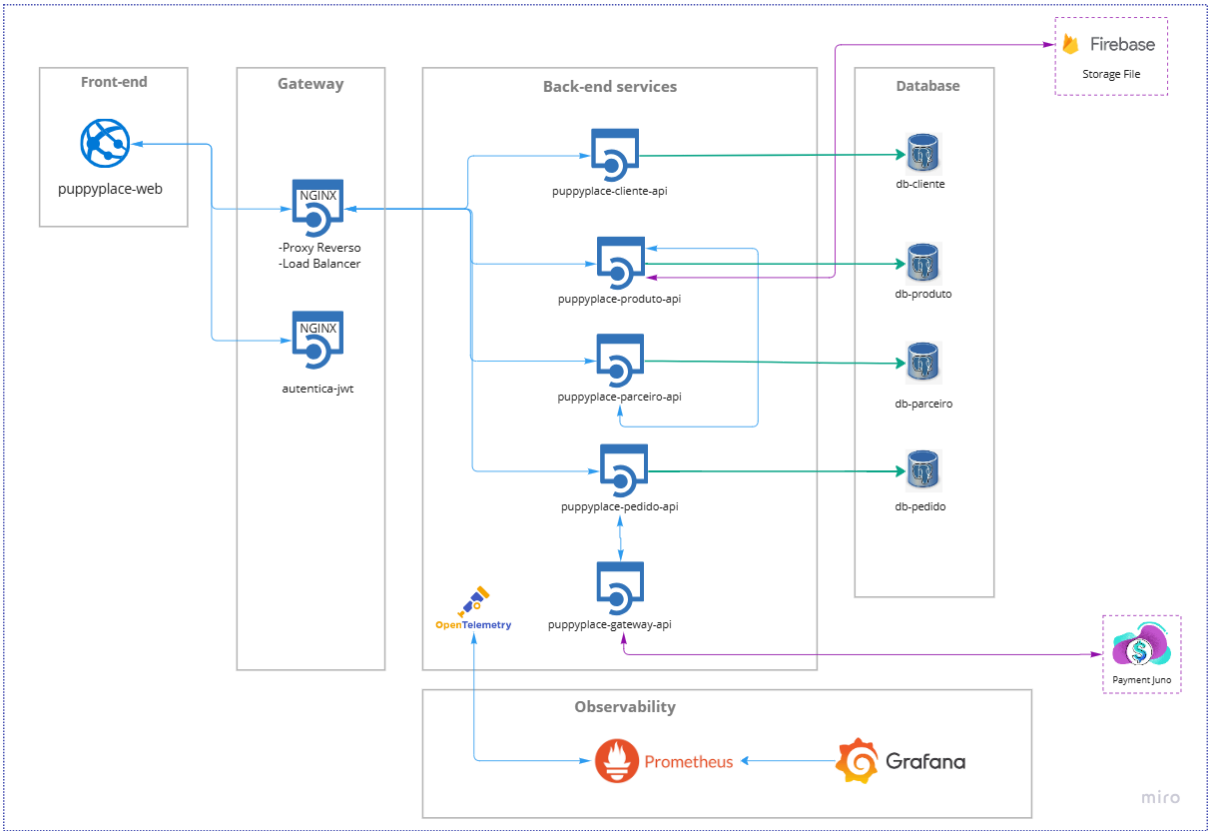


Figura 14 - Arquitetura Final para Microsserviços

7. DISCUSSÃO E CONCLUSÃO

A migração de sistemas monolíticos para uma arquitetura de microsserviços representa um desafio técnico significativo, mas ao mesmo tempo oferece grandes benefícios em termos de escalabilidade, resiliência e manutenção. Neste trabalho, exploramos a complexidade dessa transformação através de uma análise prática realizada no projeto integrador do curso. O objetivo foi desenvolver uma solução robusta que atendesse a demandas de disponibilidade e modularidade, essenciais para o crescimento e a competitividade no mercado atual.

Ao longo da implementação, encontramos dificuldades que destacaram a importância de um planejamento cuidadoso e progressivo. A transição para microsserviços mostrou-se mais desafiadora do que o esperado, principalmente devido à necessidade de reformulação da infraestrutura, orquestração de containers e integração de novas tecnologias, como o Kubernetes e o NGINX como proxy reverso e balanceador de carga. Essas ferramentas foram essenciais para garantir uma distribuição eficiente das requisições e promover um sistema tolerante a falhas.

Uma lição importante foi a necessidade de testes e validações incrementais, em vez de uma abordagem de migração direta. Portanto, aproveitando da experiência e conhecimento do Professor José Teodoro que me apoiou nesta construção e sugeriu a adoção de uma arquitetura evolutiva, focamos no desacoplamento gradual dos módulos mais acessados, permitindo uma transição controlada e com menor risco de indisponibilidade. A introdução de uma camada de monitoramento e observabilidade, com ferramentas como Open Telemetry e Grafana, demonstrou ser fundamental para identificar gargalos e avaliar a performance do sistema de forma contínua. Mesmo que no início sigamos com os indicadores default gerados pelo Open Telemetry.

Em resumo, o trabalho sugere que, embora a migração para microsserviços exija um esforço inicial elevado e um domínio técnico avançado, os benefícios compensam o investimento. Os resultados obtidos apontam para uma arquitetura mais ágil, escalável e preparada para atender novas demandas, provando ser uma escolha estratégica para sistemas que buscam flexibilidade e adaptação rápida às mudanças do mercado. A experiência adquirida reforça a importância de adotar uma abordagem estruturada, guiada por boas práticas e com foco em uma migração gradual e minimamente disruptiva.

REFERÊNCIAS BIBLIOGRÁFICAS

- **ADLESBERGER, Stefan; et al.** Building microservices with Spring Boot. 1. ed. 2020.
- **ANDERSON, David J.** Kanban: successful evolutionary change for your technology business. 2010. Blue Hole Press.
- **BASS, Len; CLEMENTS, Paul; KAZMAN, Rick.** Software architecture in practice. 3. ed. Addison-Wesley, 2013.
- **BASS, Len; CLEMENTS, Paul; KAZMAN, Rick.** Software architecture in practice. 3. ed. Addison-Wesley, 2012.
- **BLANK, Steve.** The four steps to the epiphany: successful strategies for products that win. 2013. K&S Ranch.
- **CHOI, S.; LEE, J.; KIM, D.** Proxy reverses and load balancing in microservice architectures. *Journal of Cloud Computing*, 2021.
- **EVANS, Eric.** Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, 2003.
- **EVANS, Eric.** Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, 2004.
- **FOWLER, Martin.** Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2018.

- **FOWLER, Martin.** Microservices: a definition of this new architecture. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 07 dez. 2024.
- **FOWLER, Martin.** Microservices: a definition of this new architectural style. 2015. Disponível em: http://martinfowler.com/articles/microservices.html. Acesso em: 07 dez. 2024.
- **FOWLER, Martin.** Patterns of Enterprise Application Architecture. Addison-Wesley, 2014.
- **FOWLER, Martin.** Strangler fig pattern. 2015. Disponível em: https://martinfowler.com/bliki/StranglerFigApplication.html. Acesso em: 07 dez. 2024.
- **GERVINO, Mariana Trevisoli.** "Migração de software monolítico para micro serviços: uma revisão sistemática da literatura." *Revista Interface Tecnológica*, v. 17, n. 1, p. 17-28, 2020.
- **GRANOWSKI, T.; PETERS, J.; HENDERSON, S.** Real-time monitoring with Prometheus and Grafana. *Journal of Distributed Systems*, 2019.
- **HELLERSTEIN, J. M.; STONEBRAKER, M.; HAMILTON, J.** Architecture of a database system. *Foundations and Trends in Databases*, v. 1, n. 2, p. 141–259, 2007.
- **LEWIS, James; FOWLER, Martin.** Microservices: a definition of this new architectural term. 2014. Recuperado de: https://martinfowler.com/articles/microservices.html.
- **MICROSOFT.** Azure container registry. Disponível em: https://learn.microsoft.com/en-us/azure/container-registry/. Acesso em: 07 dez. 2024.
- **MICROSOFT.** Azure DevOps. Disponível em: https://azure.microsoft.com/en-us/services/devops/. Acesso em: 07 dez. 2024.
- **MICROSOFT.** Azure Kubernetes service. Disponível em: https://learn.microsoft.com/en-us/azure/aks/. Acesso em: 07 dez. 2024.
- **MICROSOFT.** Azure SQL database. Disponível em: https://learn.microsoft.com/en-us/azure/sql-database/. Acesso em: 07 dez. 2024.
- **MICROSOFT.** Azure static web apps. Disponível em: https://learn.microsoft.com/en-us/azure/static-web-apps/. Acesso em: 07 dez. 2024.
- **NEWMAN, Sam.** Building microservices. 1. ed. O'Reilly, 2015.
- **NEWMAN, Sam.** Building microservices: designing fine-grained systems. 2015. O'Reilly Media.
- **RAHMAN, M. M.; et al.** Proof of concept for migrating monolithic architecture to microservices: a case study. *International Journal of Computer Applications*, v. 178, n. 4, p. 14-21, 2019.
- **RICHARDSON, Chris.** Microservices patterns: with examples in Java. 1. ed. Manning Publications, 2018.
- **RIES, Eric.** The lean startup: how today's entrepreneurs use continuous innovation to create radically successful businesses. Crown Business, 2011.
- **SCHWABER, Ken.** Scrum: the art of doing twice the work in half the time. Crown Business, 2017.

- **TURNER, L.; CARMEL, B.; GRANT, M.** Implementing observability in microservice architectures. *Software Engineering Review*, 2020.
- **WANG, X.; ZHANG, Y.; LIU, T.** Rate limiting and security in modern web systems. *Security and Privacy Journal*, 2019.
- **NGUYEN, H.; SAI, T.; CHOI, B.** Load balancing with NGINX for scalable systems. *IEEE Transactions on Cloud Computing*, 2017.
- **SANTOS, L.; PEREIRA, R.; ARAÚJO, V.** JWT for secure authentication in distributed systems. *International Journal of Computer Science*, 2020.