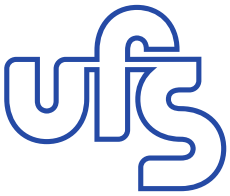


UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Desenvolvimento de aplicações com arquitetura de microsserviços

Trabalho de Conclusão de Curso

João Paulo Feitosa Secundo

São Cristóvão – Sergipe

2025

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

João Paulo Feitosa Secundo

Desenvolvimento de aplicações com arquitetura de microsserviços

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Rafael Oliveira Vasconcelos e Admilson De Ribamar Lima Ribeiro

São Cristóvão – Sergipe

2025

Resumo

O presente trabalho analisa o desenvolvimento de aplicações com arquitetura de microsserviços, expondo as características desta abordagem arquitetural e reunindo e discutindo práticas usadas no desenvolvimento de aplicações que a usa, por meio de pesquisa e revisão bibliográfica. Também serão discutidas e usadas algumas ferramentas para o desenvolvimento de uma aplicação de exemplo. O resultado é um conjunto de características comuns, práticas bem consolidadas e ferramentas úteis no desenvolvimento de tais aplicações. Ademais, foi identificado que certas práticas têm circunstâncias subjetivas e devem ser ponderadas antes de aplicadas, pois nem sempre são consideradas favoráveis, por vezes sendo julgadas positivas por alguns autores e negativas por outros.

Palavras-chave: arquitetura de *software*. microsserviços. desenvolvimento. práticas. ferramentas.

Abstract

This paper analyzes the development of applications with microservice architecture, exposing the characteristics of this architectural approach and gathering and discussing practices used in the development of applications that use it, through literature research and review. Some tools will also be discussed and used for the development of an example application. The result is a set of common characteristics, well established practices and useful tools in the development of such applications. Furthermore, it has been identified that certain practices have subjetive circumstances and must be pondered before applied, for they are not always considered favorable, sometimes being judged positive by some authors and negative by others.

Keywords: software architecture. microservices. development. practices. tools.

Lista de ilustrações

Lista de abreviaturas e siglas

API	*Application Programing Interface* - Interface para programação de aplicação
HTTP	*HyperText Transfer Protocol* - Protocolo de transferência de hipertexto
DoS	*Denial of Service* - Negação de serviço
RFC	*Request For Comments* - Pedido de comentários
JSON	*JavaScript Object Notation* - Notação de objeto JavaScript
DDD	*Domain-Driven Design* - Projeto orientado a domínio
SSL	*Secure Socket Layer* - Camada de soquete seguro
RPC	*Remote Procedure Call* - Chamada de procedimento remoto
AWS	*Amazon Web Services* - Serviços web da Amazon
IDE	*Integrated Development Environment* - Ambiente integrado de desenvolvimento
CI	*Continuous Integration* - Integração contínua
CD	*Continuous Delivery* e/ou *Continuous Deployment* - Entrega contínua e/ou Implantação contínua
CDC	*Consumer-Driven contract* - Contratos orientados ao consumidor
DVCS	*Distributed Version Control System* - Sistema de controle de versão distribuido
CVCS	*Centralized Version Control System* - Sistema de controle de versão centralizado
ORM	*Object-Relational Mapping* - Mapeamento objeto-relacional
AMQP	*Advanced Message Queuing Protocol* - Protocolo avançado de enfileiramento de mensagens
CLI	*Command Line Interface* - Interface de Linha de Comando

Sumário

1 Introdução

O crescimento da Internet e a onipresença da computação móvel tem mudado o jeito como *software* é desenvolvido nos últimos tempos. Todos que têm contato com a área do desenvolvimento de *software* provavelmente conhecem o termo *SaaS (Software as a Service)*, ou *software* como um serviço. Entretanto, essa expressão significa mais do que apenas um modelo de negócio. A tendência que tem-se observado na indústria do *software* é a de oferecer *software* não mais como um pacote completo e fechado, mas sim como um pacote flexível e em constante melhoria, o que implica na mudança do foco dos desenvolvedores para a criação de aplicações modulares, e que permitam que mudanças sejam desenvolvidas e implantadas rápida, fácil e independetemente (XU et al., 2016; Oracle Corporation, 2021).

Essa mudança de foco implicou no surgimento de novas abordagens de arquitetura e organização de *software*, e uma dessas tem ganho grande popularidade na indústria do *software* por facilitar a criação de aplicações que são multilíngues, facilmente mantidas e implantadas, escaláveis, e altamente disponíveis. Inspirada na arquitetura orientada a serviços, ela se chama arquitetura de microsserviços, e é considerada por muitos profissionais da engenharia de software como a melhor maneira de arquitetar uma aplicação de *software* como um serviço atualmente. Entretanto, como tudo na computação, há um *trade-off* (uma troca), pois assim como há benefícios, também há desvantagens e desafios no emprego de uma arquitetura de microsserviços, os quais também são discutidos neste trabalho (Middleware Lab, 2021; WASEEM et al., 2021).

1.1 Objetivos

1.1.1 Objetivo geral

Analisar o desenvolvimento de aplicações com arquitetura de microsserviços.

1.1.2 Objetivos específicos

- Caracterizar a arquitetura de microsserviços;

- Apresentar e discutir práticas comumente usadas no desenvolvimento de aplicações com arquitetura de microsserviços;

- Apresentar ferramentas que são frequentemente usadas e que cumprem propósitos importantes em aplicações com arquitetura de microsserviços;

- Contextualizar essas ferramentas, apontando os problemas que resolvem e necessidades que suprem, assim como seus pontos positivos e negativos;

- Desenvolver uma aplicação exemplar com arquitetura de microsserviços, usando uma combinação das ferramentas e práticas apresentadas;

1.2 Metodologia

Para o desenvolvimento deste trabalho, inicialmente foi feita uma pesquisa exploratória sobre a arquitetura de microsserviços, com o objetivo de ganhar maior familiaridade com o tema. Depois de definidos os objetivos iniciou-se uma pesquisa bibliográfica e os trabalhos mais relevantes foram filtrados e revisados.

Como foi constatado que não existe uma definição formal para a arquitetura de microsserviços, para caracterizá-la e para reunir práticas foram extraídas dos trabalhos as características ou práticas que apareceram com frequência ou que foram mencionadas como imprescindíveis pelos autores.

Para apresentar e contextualizar as ferramentas frequentemente usadas em aplicações com arquitetura de microsserviços, uma nova pesquisa bibliográfica foi feita para conhecer as mais comumente usadas, entender como funcionam e os problemas que resolvem, assim como suas vantagens e limites. Após isso, e depois de decidido o domínio da aplicação exemplar, as ferramentas e práticas a serem usadas no desenvolvimento da aplicação foram escolhidas de acordo com as necessidades resultantes, tendo em mente os limites do contexto do desenvolvimento deste trabalho.

2

Fundamentação teórica

Este capítulo aprensenta uma introdução sobre as arquiteturas monolítica e de microsserviços e analisa trabalhos relacionados.

Assim como este capítulo, a maioria dos engenheiros de *software* fala da arquitetura de microsserviços contrastando-a com a arquitetura monolítica porque essa última é a abordagem mais comumente usada de arquitetura de *software*, porém é importante ter em mente que existem outros tipos de arquitetura que não se encaixam nem como de microsserviços nem como monolíticas (FOWLER, 2015a).

2.1 As aplicações monolíticas

Aplicações monolíticas, também chamadas de monólitos, são aplicações que possuem as camadas de acesso aos dados, de regras de negócios e de interface de usuário em um único programa em uma única plataforma. As aplicações monolíticas são autocontidas, totalmente independentes de outras aplicações e são feitas não para uma tarefa em particular, mas sim para serem responsáveis por todo o processo para completar determinada função, assim tendo pouca ou nenhuma modularidade. Elas podem ser organizadas das mais variadas formas e fazer uso de padrões arquiteturais, mas são limitadas em muitos outros aspectos, apresentados na subseção 2.1.2 (MONOLITHIC. . . , 2022).

2.1.1 Benefícios

O maior e melhor benefício da arquitetura monolítica é sua simplicidade. Os monólitos são simples para desenvolver, para implantar, e para escalar, ademais uma aplicação simples é uma aplicação mais fácilmente entendida pelos seus desenvolvedores, fato que por sí só já melhora sua manutenibilidade. Porém, vale ressaltar que simplicidade não necessariamente implica em facilidade.

Outra vantagem dos monólitos é a facilidade de desenvolvimento da sua infraestrutura. Por não possuirem dependências com outras aplicações e nem precisarem se dedicar a comunicação externa, os monólitos têm uma infraestrutura fácil de elaborar.

Entretanto, esses benefícios só são válidos até certo ponto. Depois que uma aplicação monolítica ou o time que a desenvolve cresce muito, ela pode se tornar um emaranhado complexo de funcionalidades que são difíceis de diferenciar, de separar e de manter, e os problemas dos monólitos começam a ficar evidentes (RICHARDSON, 2018).

2.1.2 Limitações

2.1.2.1 Crescimento, velocidade de desenvolvimento, e manutenção

Quando aplicações monolíticas chegam a certo tamanho, pode se tornar muito difícil desenvolver funcionalidades novas, ou mesmo prover manutenção às já existentes, devido a diversos problemas que começam a surgir, tais como: lentidão da IDE por conta do tamanho do código, prejudicando a produtividade dos desenvolvedores; sobrecarregamento do contêiner ou máquina que hospeda a aplicação, aumentando o tempo de início; e dificuldade de entendimento da aplicação e de realizar alterações nela, diminuindo a velocidade de desenvolvimento e a qualidade do código. Padrões de organização podem amenizar a situação, mas não eliminam o problema (RICHARDSON, 2018).

2.1.2.2 Escalabilidade

O escalamento de aplicações monolíticas também pode se tornar um problema, pois ele só pode ser feito em uma dimensão. É possível escalar o volume de operações executando mais cópias de um monólito, mas não é possível fazer isso com o volume dos dados, pois cada cópia precisará acessar todos os dados, o que aumenta o consumo de memória e o tráfego de entradas e saídas (*I/O*). Também não é possível escalar os componentes independentemente, não permitindo ajustar poder de processamento ou memória quando ou onde adequado (RICHARDSON, 2018).

2.1.2.3 Reutilização

O alto acoplamento entre as partes de aplicações monolíticas dificulta a reutilização delas, o que pode causar esforço e código repetidos.

2.1.2.4 Implantação

Realizar alterações em qualquer componente de um monólito implica na necessidade de reimplantar toda a aplicação, mesmo as partes que não têm ligação com as alterações, o que aumenta riscos associados a falhas na implantação e consequentemente desencoraja a prática de implantação contínua (RICHARDSON, 2018).

2.1.2.5 Confiabilidade e resiliência

O alto acoplamento existente entre as partes da aplicação monolítica permite que falhas relativamente pequenas possam prejudicar toda a aplicação, inclusive as partes que não tiveram relação com a falha.

2.1.2.6 Flexibilidade de tecnologias

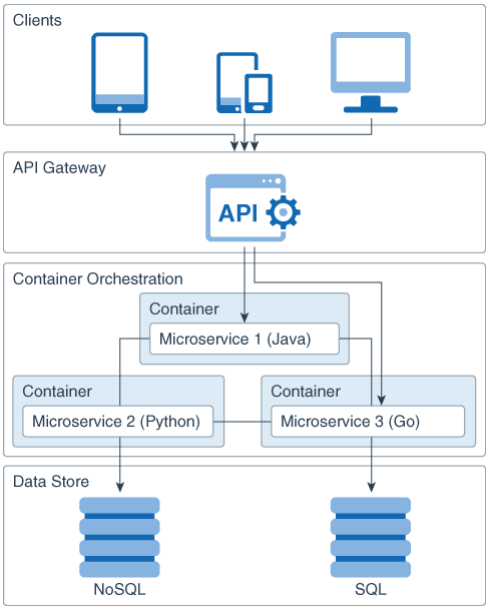
As escolhas de tecnologias para novas funcionalidades são mais limitadas - um projeto tende a usar apenas um certo grupo de tecnologias porque realizar *upgrades* ou mudanças de tecnologias é uma tarefa complexa e pode causar problemas de compatibilidade (RICHARDSON, 2018).

2.1.2.7 Divisão de times

Quando um monólito alcança determinado tamanho, é desejável dividir os desenvolvedores em times que têm foco em partes funcionais ou do domínio específicas da aplicação. Entretanto, ter times separados no desenvolvimento de uma mesma aplicação monolítica é mais difícil e menos proveitoso, porque nunca serão totalmente independentes, visto que precisam coordenar o desenvolvimento e as implantações da aplicação (RICHARDSON, 2018).

2.2 Os microsserviços

Figura 1 – Aplicação com arquitetura de microsserviços



Fonte: Oracle Corporation (2021)

Microsserviços é uma abordagem de arquitetura de *software*. Aplicações com uma arquitetura de microsserviços são separadas em partes, chamadas de microsserviços, que são classificadas em tipos (apresentados na subseção 2.2.1) e se comunicam por meio de uma rede. Microsserviços oferecem capacidades de negócio (funcionalidades relacionadas às regras de negócio da aplicação) ou capacidades de plataforma (funcionalidades relacionadas ao ambiente de execução da aplicação), tratando um aspecto em particular da aplicação. Eles se comunicam por meio de APIs bem definidas, contratos de dados e configurações. O "micro" em microsserviços faz referência não ao tamanho do serviço, mas sim ao seu escopo de funcionalidade, pois trata-se de oferecer apenas uma determinada funcionalidade, tornando-se especialistas nela. Assim sendo, microsserviços não necessariamente devem ser pequenos em tamanho, mas fazem apenas uma tarefa e a fazem bem (FAMILIAR, 2015; NEWMAN, 2021).

Sendo especialistas em apenas uma tarefa, microsserviços têm características e comportamentos que os diferenciam de outras arquiteturas orientadas a serviços, os quais serão discutidos no Capítulo 3.

A Figura 1 exemplifica uma aplicação com arquitetura de microserviços. Inicialmente os usuários da aplicação (camada *Clients*) fazem requisições à API para obter as informações desejadas. O *API Gateway* é reponsável por gerenciar as chamadas aos microsserviços e fará as devidas requisições para os devidos microsserviços (localizados na camada *Container Orchestration*). Esses microsserviços, então, executarão a lógica apropriada de acordo com a requisição recebida, possivelmente usando informações registradas no banco de dados apropriado (camada *Data Store*).

2.2.1 Tipos de microsserviços

2.2.1.1 Serviço de dados (*data service*)

Tipo de serviço de mais baixo-nível. É responsável por receber e tratar dados, assim fornecendo acesso a determinado domínio e suas regras. Quando um serviço de dados realiza apenas operações relacionadas a um determinado domínio da aplicação, ele também é chamado de serviço de domínio.

2.2.1.2 Serviço de negócio (*business service*)

Em determinados momentos as operações precisam de mais de um modelo do domínio para serem representadas em um serviço. Assim, os serviços de negócio agregam dados e oferecem operações mais complexas. Eles englobam vários serviços de domínio e proveem uma funcionalidade do negócio de nível mais alto, podendo também encapsular domínios relacionados. Por exemplo, em um site de cursos *online*, um serviço de negócio poderia prover uma funcionalidade chamada "Matricular Aluno", que envolveria as operações de inserir aluno no serviço de cursos, inserir aluno no serviço de pagamento, e inserir aluno no serviço de

gamificação.

2.2.1.3 Serviço de tradução (*translation service*)

Um serviço de tradução é um intermediário entre a aplicação e um recurso externo, provendo uma forma de acessar esse recurso. No caso desse serviço externo sofrer mudanças, pode-se realizar as alterações consequentemente necessárias em apenas um lugar, nesse serviço de tradução. Por exemplo, a aplicação pode consumir uma API externa por meio do serviço de tradução, pedindo para que ele faça uma requisição para essa API, e então recebendo a resposta.

2.2.1.4 Serviço de ponta (*edge service*)

É um serviço que serve diretamente ao cliente, sendo customizado para atender necessidades específicas desse cliente. Por exemplo, pode existir um serviço de ponta para clientes móveis e outro serviço de ponta para clientes web.

2.3 Trabalhos relacionados

2.3.1 Microservices, IoT and Azure - capítulo 2: What is a microservice, por Familiar (2015)

O Capítulo 2 do livro de Bob Familiar descreve o que é um microsserviço, suas características e implicações, benefícios e desafios.

Como explicado por Familiar (2015), microsserviços fazem uma coisa e fazem bem. Eles representam capacidades de negócio definidas usando o projeto orientado a domínio (DDD), são testados a cada passo do *pipeline* de implantação, e lançados por meio de automação como serviços independentes, isolados, altamente escaláveis e resilientes em uma infraestrutura em nuvem distribuída. Pertecem a um time único de desenvolvedores, que trata o desenvolvimento do microsserviço como um produto, entregando *software* de alta qualidade em um processo rápido e iterativo com envolvimento do cliente e satisfação como métrica de sucesso.

Em contraste com o presente trabalho, Familiar (2015) não aborda práticas e ferramentas usadas no desenvolvimento de microsserviços.

2.3.2 A Systematic Mapping Study on Microservices Architecture in DevOps, por Waseem, Liang e Shahin (2020)

Esse trabalho tem o objetivo de sistematicamente identificar, analisar, e classificar a literatura sobre microsserviços em DevOps. Inicialmente o leitor é contextualizado no mundo dos microsserviços e a cultura DevOps. Os autores usam a metodologia de pesquisa de um mapeamento sistemático da literatura publicada entre Janeiro de 2009 e Julho de 2018. Após

selecionados 47 estudos, é feita a classificação deles de acordo com os critérios definidos pelos autores, e então é feita a discussão sobre os resultados obtidos - são expostos a quantidade de estudos sobre determinados tópicos em microsserviços, problemas e soluções, desafios, métodos de descrição, padrões de projeto, benefícios, suporte a ferramentas, domínios, e implicações para pesquisadores e praticantes.

Em contraste com o presente trabalho, Waseem, Liang e Shahin (2020) não abordam as características dos microsserviços, entretanto mapeiam desafios enfrentados e soluções empregadas.

2.3.3 Design, monitoring, and testing of microservices systems: The practitioners' perspective, por Waseem et al. (2021)

Esse trabalho tem o objetivo de entender como sistemas de microsserviços são projetados, monitorados e testados na indústria. Foi conduzida uma pesquisa relativamente grande que obteve 106 respostas e 6 entrevistas com praticantes de microsserviços. Os resultados obtidos identificam os desafios que os praticantes enfrentam e as soluções empregadas no projeto, monitoramento e teste de microsserviços. Também é feita uma discussão profunda sobre os resultados, da perspectiva dos praticantes, e sobre as implicações para pesquisadores e praticantes.

Em contraste com o presente trabalho, Waseem et al. (2021) não abordam as características dos microsserviços.

2.3.4 Building Microservices: Designing Fine-Grained Systems, por Newman (2021)

Newman (2021) aborda as caracterísicas e o desenvolvimento de microsserviços, desde a modelagem até a implantação, entretanto foca em ideias em vez de tecnologias, pois reconhece que detalhes de implementação e ferramentas estão sempre em mudança.

Diferente do presente trabalho, Newman (2021) não discute ferramentas para a implementação dos conceitos e práticas discutidas.

3

Características, vantagens e desafios da arquitetura de microsserviços

Este capítulo aprensenta as características e as vantagens da arquitetura de microsserviços, assim como os riscos, desafios e desvantagens que as acompanham.

Em geral não existe uma definição formal do que a arquitetura de microsserviços tem ou não tem, entretanto as características aqui apresentadas são características comuns observadas em arquiteturas que se encaixam como microsserviços. Assim sendo, nem todas as arquiteturas de microsserviços terão essas características, apesar de ser esperado que a maioria delas estejam presentes (FOWLER; LEWIS, 2014).

3.1 Sistema distribuído

A arquitetura de microsserviços forma naturalmente um sistema altamente distribuido, o que implica em alguns comportamentos e características. Uma delas é que qualquer chamada a um serviço está sujeita a falhas, portanto microsserviços devem ser projetados para serem resilientes, isso é, tolerantes a falhas e ter um tempo de recuperação razoável quando algum problema inevitavelmente acontecer (FAMILIAR, 2015).

Um benefício proveniente dessa distribuição é que microsserviços podem ser usados em soluções e cenários de uso diferentes, contudo, para tanto devem ser escaláveis, responsivos e configuráveis, para assim alcançar um bom desempenho independente do cenário de uso. Outro benefício dessa distribuição é a autonomia e o isolamento, o que significa que microsserviços são unidades auto-contidas de funcionalidade, com dependências de outros serviços fracamente acopladas, então podem ser projetados, desenvolvidos, testados e implantados independentemente de outros serviços (FOWLER; LEWIS, 2014; FAMILIAR, 2015).

Um desafio advindo dessa distribuição é que a comunicação é complexa e falível, geralmente dependendo de APIs e contratos de dados para definir como os microsserviços interagem, portanto eles são orientados-a-mensagens. Essa comunicação é melhor discutida na seção 4.12 e subseção 4.13.1 (FAMILIAR, 2015).

3.2 Flexibilidade na escolha de ferramentas

Como mencionado anteriormente, cada microsserviço disponibiliza suas funcionalidades por meio de APIs e contratos de dados em uma rede. Usando esse meio, a comunicação independe da arquitetura que o microsserviço faz uso, o que possibilita que cada microsserviço escolha seu sistema operacional, linguagem, serviços de apoio e demais ferramentas necessárias. Além disso, os microsserviços podem ser desenvolvidos usando uma linguagem de programação e estrutura que melhor se adapte ao problema que ele é projetado para resolver, possuindo mais flexibilidade, entretanto é importante notar que usar muitas ferramentas diferentes aumenta a complexidade do sistema (Oracle Corporation, 2021; FAMILIAR, 2015).

3.3 Alta velocidade de desenvolvimento

Ter um time independente responsável por cuidar do ciclo de desenvolvimento e sua automação permite uma alta velocidade de desenvolvimento para os microsserviços, muito maior do que fazendo o equivalente para uma solução monolítica (FAMILIAR, 2015).

3.4 Componentização

Há muito tempo na indústria do *software* deseja-se construir sistemas apenas juntando componentes, assim como se faz no mundo físico. Na computação, um componente é definido como uma unidade de *software* que é atualizável e substituível independentemente. Apesar de ser muito comum o uso de pacotes e bibliotecas (padrão de projeto conhecido como *sidecar*), que podem ser considerados componentes, há maneiras diferentes de se componentizar *software* que são características dos microsserviços. Os microsserviços também podem utilizar pacotes e bibliotecas como componentes, contudo nessa arquitetura a maneira principal e mais eficiente para componentizar o *software* é justamente dividí-lo em microsserviços. Entretanto, essa divisão não é uma tarefa simples - pelo contrário, definir adequadamente os limites dos microsserviços é um dos desafios mais complexos e importantes desta arquitetura, mas pode ser facilitado pelo uso de abordagens bem consolidadadas de design de *software*, como o *Domain-Driven Design* (DDD), discutido na seção 4.2 (FOWLER; LEWIS, 2014).

Como mencionado acima, uma maneira de se alcançar certo nível de componentização em uma aplicação é pelo uso de múltiplas bibliotecas como componentes em um único processo, porém nesse caso uma mudança em qualquer desses componentes resulta na necessidade de reimplantar toda a aplicação. Por outro lado, se essa mesma aplicação é decomposta em múltiplos serviços, é provável que uma mudança em um serviço só obrigaria a reimplantação do mesmo serviço, assim tendo-se uma implantação mais simples e rápida. Ademais, por ter um alto nível de componentização e independência, os microsserviços são reusáveis por natureza e portanto

torna-se mais fácil criar soluções e produtos por meio da combinação de multiplos microsserviços (FOWLER; LEWIS, 2014; FAMILIAR, 2015).

Contudo, usar serviços dessa forma também traz algumas desvantagens. Uma delas é que será necessário considerar como mudanças em um serviço podem afetar seus consumidores. A abordagem tradicional para resolver esse problema é o uso de versionamento no serviço, entretanto essa prática não é bem-vista no mundo dos microsserviços. A melhor solução é projetar serviços para serem o mais tolerante possíveis a mudanças nos serviços a que consomem. Outra desvantagem é que a comunicação remota é muito mais complexa e custosa, portanto o método de comunicação escolhido deve ser implementado de modo flexível. Ademais, realocar responsabilidades entre componentes é mais difícil quando se trata de processos diferentes (FOWLER; LEWIS, 2014).

3.5 Portabilidade

A portabilidade de uma aplicação diz respeito a quão facilmente ela pode ser executada em diferentes ambientes de execução com variadas configurações de plataforma e infraestrutura. Ela não é inerente aos microsserviços, mas é facilitada por terem escopo e tamanho limitados. No entanto, faz-se necessário o uso de técnicas que aumentem essa portabilidade, como por meio de automação e externalização de configurações, para que assim os microsserviços possam ser implantados de forma fácil e eficiente em ambientes de execução variados. (FAMILIAR, 2015).

3.6 Versionável e substituível

Apesar do versionamento de microsserviços não ser recomendado por dificultar a operação e o entendimento deles, quando necessário é possível manter versões diferentes de um mesmo serviço executando ao mesmo tempo, assim proporcionando retrocompatibilidade e um processo de migração mais suave. Além disso, serviços podem ser atualizados ou mesmo substituidos sem ocasionar indisponibilidade do serviço, pelo uso de ferramentas de provisionamento e técnicas de implantação apropriadas. (FAMILIAR, 2015).

3.7 Menor erosão de *software*

A erosão de *software* é inevitável em qualquer aplicação - conforme um sistema cresce e envelhece, é natural que torne-se mais difícil dar-lhe manutenção. Porém, com componentes modularizados e organizados adequadamente, uma aplicação com arquitetura de microsserviços tende a erodir muito mais lentamente e crescer mais horizontalmente (em vez de verticalmente) do que uma não modularizada, favorecendo uma vida útil mais longa da aplicação.

3.8 Complexidade e desafios

O uso da arquitetura de microsserviços implica num grande aumento de complexidade não apenas na infraestrutura, mas também em muitas etapas do ciclo de desenvolvimento do *software*, como no *debug*, nos testes, e no monitoramento por exemplo. Além disso, o uso de diversas ferramentas diferentes pode ocasionar problemas por inexperiência dos desenvolvedores. Dessa forma, existem muitos desafios no desenvolvimento de aplicações com arquitetura de microsserviços, que normalmente só conseguem ser compensados em aplicações mais complexas. E como em qualquer tipo de arquitetura, existem benefícios, riscos, desvantagens e desafios, portanto para determinar se adotá-la é uma escolha sábia é necessário entendê-los e aplicá-los ao contexto específico da aplicação e dos desenvolvedores (RICHARDSON, 2021; FOWLER, 2015a).

Um dos principais desafios no desenvolvimento de uma aplicação com arquitetura de microsserviços é a definição adequada dos limites de cada microsserviço, que pode ser facilitado pelo uso de abordagens bem consolidadadas de *design* de *software*, como o DDD. Outros desafios são: complexidade de projeto, complexidade operacional, consistência de dados, comunicação e manutenção. De acordo com Xu et al. (2016), os **três grandes desafios** do desenvolvimento de aplicações com arquitetura de microsserviços são (1) Como programar sistemas que consistem de um grande número de serviços executando em paralelo e distribuidos em um conjunto de máquinas, (2) Como reduzir a sobrecarga de comunicação causada pela execução de grandes números de pequenos serviços e (3) Como sustentar a implantação flexível de serviços em uma rede para conseguir realizar o balanceamento de carga (FOWLER, 2015b).

3.8.1 Teorema CAP

CAP é um acrônimo para *Consistency, Availability and Partition tolerance* (Consistencia, Disponibilidade e Tolerância à partição), e o teorema CAP prova que é impossível, em um sistema distribuido, se ter consistência estrita de dados, disponibilidade contínua de todos os nós e tolerância à falhas de comunicação entre os nós ao mesmo tempo.

Um sistema distribuido que se mantém completamente disponível e tolera partições (AP) precisará lidar com a consistência eventual (em vez de estrita) dos dados durante uma falha. Um sistema distribuido que mantém consistência estrita dos dados e tolera partições (CP) precisará ficar indisponível enquanto se recupera de uma falha. Já um sistema que mantém consistência estrita dos dados e continua completamente disponível durante uma falha, na verdade não pode ser um sistema distribuido, pois sendo intolerante à partições, não poderia estar se comunicando por meio de uma rede. Assim, não se pode abandonar a tolerância à partições em sistemas distribuidos, reduzindo a escolha a consistência estrita ou disponibilidade contínua. Como toda aplicação com arquitetura de microsserviços naturalmente forma um sistema distribuido, é importante ter isso em mente ao escolher um modelo de gerenciamento de dados. (IBM, 2022;

NEWMAN, 2021)

4

Práticas no desenvolvimento de aplicações com arquitetura de microsserviços

Este capítulo apresenta e discute práticas comumente seguidas no desenvolvimento de aplicações com arquitetura de microsserviços.

4.1 Começar pela arquitetura monolítica

Fowler (2015b) defende o uso de arquiteturas monolíticas para desenvolver novas aplicações. Mesmo os defensores dos microsserviços dizem que há custos e riscos no uso desta arquitetura, os quais desaceleram o time de desenvolvimento, assim favorecendo monólitos para aplicações mais simples. Esse fato leva a um argumento forte para a escolha de uma arquitetura monolítica mesmo se for acreditado que haverá benefícios mais tarde com o uso da arquitetura de microsserviços, por duas razões. A primeira é conhecida como *Yagni - You're not gonna need it*, ou "Você não precisará disso", um preceito do método ágil *ExtremeProgramming* que diz que uma capacidade que acredita-se ser necessária no futuro não deve ser implementada agora por quê "você não precisará disso". A segunda razão é que microsserviços só funcionarão bem se os limites forem muito bem estabelecidos, e para tanto, constrói-se um monólito primeiro para que se possa descobrir os limites antes de serem impostos grandes obstáculos neles pela divisão dos microsserviços (FOWLER, 2015b).

Além disso, Fowler (2014) também afirma que existem 3 pré-requisitos para se adotar uma arquitetura de microsserviços, e que é mais fácil lidar com as operações de um monólito bem definido do que de um ecossistema de pequenos serviços. Assim sendo, pode-se considerar uma boa prática começar pela arquitetura monolítica até que o sistema já esteja bem definido e estes pré-requisitos sejam atendidos - provisionamento rápido, monitoramento básico, e implantação rápida de aplicação - explicados na subseção 4.1.1, subseção 4.1.2 e subseção 4.1.3 respectivamente (FOWLER, 2014).

Já Tilkov (2015) contesta essa prática e afirma que não se deve começar pela arquitetura monolítica se o objetivo for uma arquitetura de microserviços. Ele afirma que o melhor momento para se pensar em dividir um sistema é justamente quando ele está sendo construido, e que

é extremamente difícil dividir um sistema *brownfield* (sistema desenvolvido a partir de outro pré-existente). Entretanto, ele reconhece que para dividir um sistema, deve-se conhecer muito bem o domínio, e que o cenário ideal para o desenvolvimento de microsserviços é quando se está desenvolvendo uma segunda versão de um sistema existente. Fowler (2015b) reconhece esses argumentos como válidos e reforça que existem, sim, benefícios de se começar por uma arquitetura de microsserviços, mas ainda existem poucas histórias de aplicações com arquiteturas de microsserviços e mais estudos de casos são necessários para saber como determinar a melhor escolha inicial de arquitetura (TILKOV, 2015; FOWLER, 2015b).

Lumetta (2018) afirma que para decidir a abordagem arquitetural inicial de uma aplicação é necessário considerar o contexto do negócio, da própria aplicação, e do time que a irá desenvolver, e que existem condições que configuram a melhor escolha. Ele descreve 3 condições que tornam a adoção de uma arquitetura de microserviços para uma nova aplicação uma boa escolha: (1) há necessidade de entrega de serviços rapida e independentemente; (2) parte da plataforma precisa ser extremamente eficiente; e (3) planeja-se aumentar o time. Ele também descreve 3 condições que tornam a adoção de uma arquitetura monolítica uma boa escolha: (1) o time ainda está em crescimento; (2) o produto sendo construído é não comprovado ou é uma prova de conceito; e (3) o time não tem experiência com microsserviços (LUMETTA, 2018).

Percebe-se então que existem tanto razões para se começar pelos microsserviços como razões para se começar com uma arquitetura mais simples. Porém, não foi observado um consenso sobre quais seriam exatamente as razões para adotar ou não uma arquitetura de microsserviços para uma nova aplicação desde o início de seu desenvolvimento. Há nesse ponto, portanto, espaço para mais discussões e pesquisas.

4.1.1 Requisito dos microsserviços: Provisionamento rápido

No contexto da computação, provisionamento significa disponibilizar um recurso necessário para o funcionamento de uma aplicação. Para produzir *software*, é necessário provisionar diversos recursos, tanto para os desenvolvedores quanto para o cliente. Naturalmente, o provisionamento torna-se mais fácil em plataformas de serviços de computação em nuvem - na AWS, por exemplo, para conseguir uma nova máquina, basta iniciar uma nova instância e acessá-la - um processo muito rápido quando comparado ao *on-premises*, onde seria necessário comprar uma nova máquina, esperar chegar, configurá-la e só então ela estaria pronta. Além disso, o provisionamento rápido requer automação de tarefas relacionadas (FOWLER, 2014).

4.1.2 Requisito dos microsserviços: Monitoramento básico

Toda aplicação precisa lidar com erros e problemas, porém em uma arquitetura distribuida, existem naturalmente mais lugares suscetíveis a problemas, por existirem mais componentes que são fracamente acoplados, estando sujeitos não só a falhas no código, mas também na

comunicação, na conexão, ou até falhas físicas. Portanto, o monitoramento é crucial nesse tipo de arquitetura, favorecendo uma rápida detecção dos problemas. Ademais, o monitoramento também pode ser usado para detectar problemas de negócio, como por exemplo uma redução nos pedidos de um *site* de vendas (FOWLER, 2014).

4.1.3 Requisito dos microsserviços: Implantação rápida

Na arquitetura de microserviços a implantação é feita separadamente para cada microsserviço. Com muitos serviços para gerenciar, ela pode se tornar uma tarefa árdua, portanto será novamente necessário um grande nível de automação nessa etapa, geralmente envolvendo um *pipeline* de implantação, que deve ser automatizado o máximo possível (FOWLER, 2014).

4.2 Microsserviços com *Domain-Driven Design* (DDD)

Como mencionado anteriormente, definir adequadamente os limites dos microsserviços é um dos desafios mais complexos e importantes no desenvolvimento de aplicações com tal arquitetura. De fato, isso também é verdade mesmo para arquiteturas mais simples, pois o conhecimento e modelagem do domínio são problemas intrínsecos do desenvolvimento de *software*, e não são habilidades facilmente ensinadas ou aprendidas. (EVANS, 2003)

Domain-Driven Design (DDD) é uma abordagem para o desenvolvimento de *software* que foca na modelagem do domínio do negócio de forma alinhada com suas regras e conceitos. Essa abordagem enfatiza a colaboração entre desenvolvedores e especialistas do domínio para criar um modelo preciso, reduzindo a complexidade e garantindo que o *software* reflita fielmente os processos do mundo real tratados. (EVANS, 2003)

Um dos pilares do DDD é a linguagem ubíqua, um vocabulário compartilhado que unifica a comunicação entre técnicos e especialistas do negócio. Além disso, a abordagem sugere dividir o sistema em contextos delimitados (*bounded contexts*), que favorecem modularidade e independência entre diferentes partes (e consequentemente equipes) da aplicação, o que se aplica especialmente bem aos microsserviços, por serem naturalmente componentizados. (EVANS, 2003)

Embora o uso do DDD possa trazer diversos benefícios para aplicações, espcialmente as com arquiteturas complexas como a de microsserviços, sua implementação exige um profundo entendimento do domínio e competência na modelagem dele. Entretanto, quando aplicado corretamente, essa abordagem proporciona maior alinhamento com o negócio, melhor manutenção e evolução do *software* e favorece a independência de diferentes equipes. (EVANS, 2003)

4.3 A metodologia 12-fatores

A metodologia 12-fatores é um conjunto de diretivas para o desenvolvimento de aplicações com modelo de *software* como um serviço. Ela resume a experiência de diversos desenvolvedores experientes com o desenvolvimento desse tipo de aplicação, e tem foco no crescimento da aplicação, na dinâmica entres os times e na erosão do *software*. Cumprir esses fatores proporciona à aplicação: Um formato declarativo para automação de configuração de ambientes de execução, favorecendo a portabilidade; Adequação à implantação em plataformas na nuvem, evitando a necessidade de infraestrutura *on-premises*; Redução da divergência entre o ambiente de desenvolvimento e de lançamento, facilitando a entrega contínua; e Simplicidade no escalamento. Qualidades, essas todas, que são altamente desejáveis em microsserviços (WIGGINS, 2017; RODRIGUES, 2016)

É importante salientar que em um sistema distribuido como em uma aplicação com arquitetura de microsserviços, uma "aplicação" no contexto da metodologia 12-fatores constitui um único microsserviço, portanto o cumprimento e os benefícios da metodologia são separados para cada um deles. Posto isso, para alcançar os benefícios citados, os microsserviços devem seguir as seguintes diretivas:

I. Base de código única - Cada microsserviço deve ter uma base de código única e particular, com rastreamento utilizando controle de versão, e devem existir diferentes versões implantáveis, como por exemplo uma versão de homologação e uma de produção. Ter mais de um microsserviço compartilhando uma mesma base de código, como na abordagem *monorepo*, constitui uma violação dessa diretiva.

II. Dependências portáteis e isoladas - Cada microsserviço deve declarar suas dependências e isolá-las das de outros projetos. Isso é facilmente alcançado pelo uso de um gerenciador de pacotes, que mantém um arquivo declarando todas as dependências e gerencia o isolamento dos pacotes instalados na máquina. Um exemplo em projetos Python é o *pip*, que declara as dependências e o *virtualenv*, que lida com o isolamento dos pacotes instalados para cada projeto na máquina;

III. Externalizar configurações - Qualquer informação que varia de acordo com o ambiente de execução, como credenciais de acesso a um banco de dados por exemplo, devem ser armazenadas separadas do código, como em arquivos que não são rastreados pelo sistema de controle de versão ou em variáveis de ambiente;

IV. Serviços de apoio são anexos - Os microsserviços e sua lógica não devem depender de um serviço de apoio específico. Um serviço de apoio é qualquer serviço consumido por meio da rede como parte de sua operação usual, como um banco de dados. Caso o banco de dados usado seja o MySQL, por exemplo, deve ser possível trocar para o PostgreSQL, ou mesmo para um outro banco de dados na nuvem sem nenhuma mudança no código, apenas na configuração;

V. Separação entre construção, lançamento, e execução - Deve-se separar estritamente

as etapas de construção, lançamento e execução. Na etapa de construção, usa-se os arquivos no repositório para criar um programa iniciável. Na etapa de lançamento, o programa iniciável combina-se com a configuração do ambiente de execução atual para criar um lançamento, que deve ser imutável e ter um identificador único (como, por exemplo, versão 1.6.2) para propósitos de controle de versão. Na etapa de execução, o lançamento e seus serviços de apoio são executados no ambiente apropriado.

VI. Processos *stateless* (sem estado) - O microsserviço deve ser executado como um ou mais processos que não armazenam estado e não dividem espaço de memória. Qualquer informação que precise persistir deve ser salva em serviços de apoio, geralmente um banco de dados, assim diminuindo o acoplamento e facilitando o escalamento;

VII. Vínculo de porta - Um microsserviço deve ser auto-contido, sendo ele mesmo responsável por expor e escutar em uma porta, em vez de depender de uma injeção no ambiente de execução, que é normalmente feita por um servidor web;

VIII. Simultaneidade - Além de não guardarem estado, os processos em um microsserviço devem ser independentes e categorizados em tipos diferentes. Dessa forma, é possível usar tipos diferentes de processos para atender cargas de trabalho diferentes e todos podem ser escalados independente e horizontalmente com facilidade.

IX. Descartabilidade - Os processos de um microsserviço devem ser descartáveis, podendo ser iniciados ou interrompidos rapidamente sem perda de informação importante;

X. Paridade de ambientes de execução - Deve-se manter todos os ambientes de execução o mais semelhantes possível. Essa é uma diretiva simples mas que envolve vários aspectos, em especial o intervalo entre o término do desenvolvimento de uma função e sua implantação; a diferença entre quem desenvolve e quem implanta; e a diferença entre as ferramentas e serviços de apoio usados para cada ambiente de execução. Esse intervalo e essas diferenças devem ser minimizados para aumentar a paridade entre os ambientes de execução, assim favorecendo a entrega contínua;

XI. Registros (*Logs*) - Registros devem ser tratatos como um fluxo de eventos e enviados para a saída padrão do processo, para que uma outra ferramenta lide com o tratamento e armazenamento;

XII. Processos administrativos - Processos que são executados pontualmente, como migrações de bancos de dados, devem ser versionados e constituem parte do lançamento. Eles devem ser executados facilmente num ambiente idêntico ao ambiente onde os processos do microsserviços estão sendo executados, aderindo também ao isolamento de dependências mencionado no fator II.

4.4 Produtos, não projetos

A maioria dos times de desenvolvedores trabalham sob o seguinte modelo de projeto: O objetivo é entregar uma peça de *software*, que quando entregue é considerada como completa. Após isso, o *software* é passado para um time de manutenção e o time que o desenvolveu é desfeito. Os praticantes de microsserviços tendem a evitar esse modelo, em vez disso adotando a ideia de que um time deve ser o dono de um produto - não projeto - durante todo seu ciclo de vida. Um exemplo de empresa que adota esse modelo é a Amazon, exercendo a ideia de "você constroi, você executa", na qual um time de desenvolvimento é totalmente responsável por um *software* em produção (um produto). Dessa forma, o time adquire pleno conhecimento de como seu produto se comporta e como seus usuários o utilizam, o que é importante pois também terá que realizar o suporte aos usuários do produto (FOWLER; LEWIS, 2014).

Essa prática também está ligada a separação da aplicação por capacidades de negócio - em vez de enxergar o *software* como um conjunto de funcionalidades a serem implementadas, cria-se uma relação entre os desenvolvedores e os usuários, na qual a questão é como o *software* pode auxiliar o usuário a aumentar a capacidade de negócio. Tal prática também pode ser aplicada em aplicações monolíticas, embora a divisão em microsserviços facilita a crição de relações entre os desenvolvedores de serviços e seus usuários (FOWLER; LEWIS, 2014).

4.5 Desenvolver e compartilhar ferramentas

Em vez de apenas usar um conjunto de padrões definidos para desenvolver microsserviços, é preferível produzir ferramentas úteis que outros desenvolvedores possam usar para resolver problemas similares aos que eles enfrentam. Essas ferramentas geralmente são extraídas de implementações maiores e compartilhadas com um grupo mais amplo, geralmente por meio de um modelo de código aberto. Com o Git e o GitHub se tornando ferramentas tão populares, práticas de código aberto estão cada vez mais comuns (FOWLER; LEWIS, 2014).

A Netflix é um exemplo de organização que segue essa filosofia. Compartilhar código útil e muito bem testado como bibliotecas incentiva outros desenvolvedores a resolver problemas semelhantes de maneiras semelhantes, ao mesmo tempo que mantém a possibilidade de escolher uma abordagem diferente, se necessário. As bibliotecas compartilhadas tendem a se concentrar em problemas comuns, como armazenamento de dados, comunicação entre processos e automação de infraestrutura (FOWLER; LEWIS, 2014).

4.6 Descentralização dos dados

Para o gerenciamento de dados, há a possibilidade de compartilhar um banco de dados entre diferentes microsserviços, porém isso é visto como um anti-padrão, pois microsserviços diferentes possuem necessidades distintas de armazenamento e acesso a dados. Dessa forma, uma

aplicação com arquitetura de microsserviços tem melhor isolamento, segurança e disponibilidade quando os microsserviços possuem um modelo de dados independente e gerenciam seu próprio banco de dados particular, inclusive tendo a possibilidade de usar sistemas de banco de dados diferentes. Além disso, os dados persistidos por esses bancos de dados particulares só devem ser acessados diretamente pelo serviço que o contém, e outros serviços que necessitem desses dados precisarão enviar uma requisição. Com cada serviço tendo seu próprio banco de dados, a escalabilidade dele e do seu banco de dados pode ser feita independentemente dos outros serviços. Assim, serviços que recebem poucos acessos podem ter bancos menos potentes e mais baratos, e vice-versa (Oracle Corporation, 2021; FOWLER; LEWIS, 2014).

Entretanto, essa descentralização tem implicações para o gerenciamento de fluxos de negócios que envolvem escritas em múltiplos bancos de dados. Geralmente a abordagem para se garantir consistência nas escritas é pelo uso de transações quando atualizando múltiplos recursos, porém o uso de transações resulta em um acoplamento temporal, o que pode resultar em problemas quando existem muitos serviços envolvidos em um único fluxo. Ademais, transações distribuidas são notoriamente difíceis de implementar, portanto arquiteturas de microsserviços normalmente realizam coordenação sem transações entre serviços, com reconhecimento claro de que consistência pode ser apenas consistência eventual (em vez de estrita) e que problemas serão lidados pela compensação de operações (FOWLER; LEWIS, 2014).

4.7 CI/CD

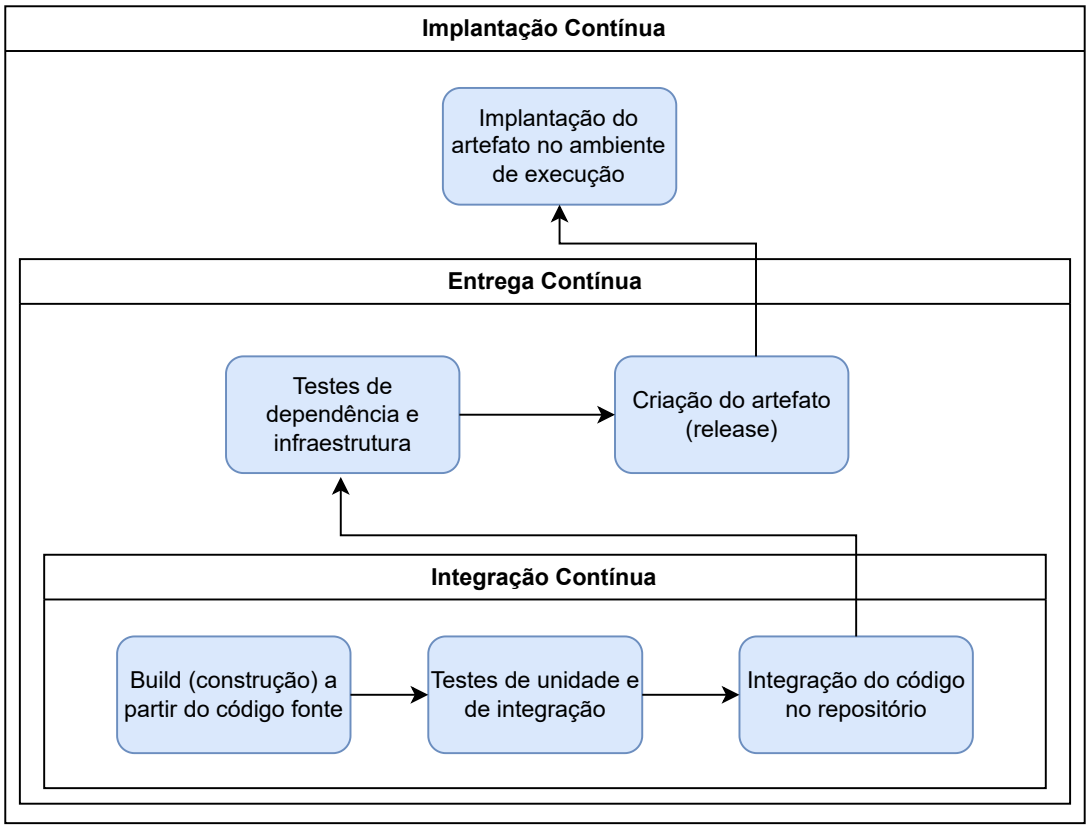
CI/CD é um método para entregar aplicações e mudanças nelas aos clientes com frequência, envolvendo CI e CD. CI é um acrônimo de *Continuous Integration* (integração contínua), e diz respeito à automação de como o novo código feito pelo desenvolvedor integra-se ao repositório principal. CD é um acrônimo de *Continuous Delivery* e/ou *Continuous Deployment* (entrega contínua e/ou implantação contínua). Entrega contínua diz respeito à automação de como o código no repositório se transforma em artefatos, que definem uma versão da aplicação e que são implantáveis num ambiente de execução. Implantação contínua diz respeito a automação de como esses artefatos são de fato implantados no ambiente de execução. Apesar de determinar essas definições para o presente trabalho, foram observadas algumas inconsistencias nas referências utilizadas tanto nas definições para os termos CI e CD quanto nos limites do que essas práticas englobam. Em particular para o termo CD, que pode significar *Continuous Delivery*, *Continuous Deployment*, ou mesmo os dois. (Harness Incorporated, 2021; Red Hat Incorporated, 2022)

Dito isso, não é importante se ater à uma definição específica; apenas deve-se entender que CI/CD é um processo, muitas vezes visualizado como um *pipeline*, que envolve a adição de um alto nível de automação e monitoramento no ciclo de vida do desenvolvimento de *software* como um serviço e auxilia profissionais de desenvolvimento e de operações a trabalhar mais eficiente e colaborativamente. Ademais, CI/CD faz com que processos sejam previsíveis e repetíveis,

diminuindo o espaço para erros humanos. Considerando que a automação e monitoramento são essenciais para diversas práticas recomendadas no desenvolvimento de microsserviços, CI/CD é uma prática de extrema importância (Red Hat Incorporated, 2022; GitLab Incorporated, 2022a).

Por fim, não existe uma estrutura ou um conjunto específico de passos que todo *pipeline* de CI/CD deve ter; a estrutura e os passos contidos dependem dos requisitos dos times envolvidos e da aplicação. No caso dos microsserviços, CI/CD é implementado para cada microsserviço separadamente, podendo ter configurações variadas para cada um. A Figura 2 ilustra uma possível estrutura de um *pipeline* de CI/CD.

Figura 2 – Exemplo de *pipeline* de CI/CD



Fonte: Autor

4.7.1 Integração contínua (CI)

CI (*Continuous Integration* - Integração contínua) é uma prática no desenvolvimento de *software* onde os desenvolvedores integram o código em um repositório compartilhado frequentemente. Muitos times afirmam que essa prática leva a uma grande redução nos problemas de integração e os permite desenvolver *software* coeso mais rapidamente. Além disso, melhora a identificação do progresso do desenvolvimento, facilita a identificação e remoção de falhas e *bugs*, e aumenta a experiência e a confiança do time nos testes e *build* do código desenvolvido.

A integração contínua por sí só requer apenas uma ferramenta de controle de versão para ser feita, mas existem práticas bem consolidadas na indústria do desenvolvimento de *software* que incrementam esse processo, apresentadas a seguir. (FOWLER, 2006).

4.7.1.1 Uso de apenas um repositório fonte

Deve haver apenas um repositório por projeto ou microsserviço, compartilhado por toda a equipe desenvolvedora e contendo tudo aquilo que é necessário para uma instalação rápida e funcional do ambiente de desenvolvimento. Isso inclui mas não é limitado a - código, *scripts*, migrações e esquemas de bancos de dados, arquivos de propriedades e configurações de IDE. Além disso, o conteúdo do repositório deve estar disponível para todos, assim aumentando a visibilidade e facilitando o monitoramento do progresso do conteúdo (GitLab Incorporated, 2022a; FOWLER, 2006).

4.7.1.2 *Build* rápido

Um *build* (construção) da aplicação trata-se da transformação dos arquivos no repositório em um programa iniciável. *Builds* lentos afetam negativamente a integração contínua, atrasando integrações e diminuindo a frequência do *feedback*. Além disso, todas as etapas do *build* devem ser simples de executar, idealmente por meio de um único comando (FOWLER, 2006).

4.7.1.3 Automação de testes em novas integrações

As integrações de cada novo *commit* no repositório devem ser testadas, o que pode ser uma tarefa árdua e demorada dependendo de como o *build* da aplicação é feito e da quantidade de testes, especialmente se feitos manualmente. Desse modo, deve-se elaborar uma bateria de testes, e executá-la de forma automática a cada novo *build*, para assim detectar falhas rapidamente e aumentar a qualidade do *software* (GitLab Incorporated, 2022a; FOWLER, 2006).

4.7.1.4 Servidor de integração

Muitas vezes não é possível (nem recomendado) executar todos os passos do *pipeline* de integração na máquina do desenvolvedor, seja por questões de tempo, de capacidade computacional ou pela falta de garantia de que o desenvolvedor realmente executará os passos. Nesses casos, recomenda-se providenciar um local que centralizará a integração do novo código desenvolvido com o repositório principal. Tal local é chamado de *CI Daemon*, ou servidor de integração, e ele é responsável por executar todos os passos do *pipeline* de integração, assim como disponibilizar informações e relatórios sobre os passos executados. Também é recomendado estabelecer condições para que a integração do novo código seja efetuada, tal como a execução bem-sucedida de todos os testes, assim garantindo que apenas o código que cumpra tais condições seja integrado e consequentemente aumentando a qualidade do código e do produto (FOWLER, 2006; HUMBLE; FARLEY, 2010).

4.7.1.5 Estabilidade do ramo principal

Um dos principais motivos para se usar a integração contínua é a garantia de que a equipe sempre estará trabalhando a partir de uma base de código estável. Se o ramo principal está instável, é tarefa de toda a equipe resolver o problema o mais rápido possível, pois nenhum código poderá ser desenvolvido a partir daquele ramo até que esteja estável e confiável novamente. Geralmente o melhor meio de resolver esse problema é reverter os *commits* problemáticos, mas se a solução for simples, integrar um novo *commit* pode ser suficiente. (FOWLER, 2006).

4.7.2 Entrega/Implantação contínua (CD)

CD significa entrega contínua e/ou implantação contínua, conceitos relacionados e às vezes usados alternadamente. Em ambos os casos, trata-se da automação de fases avançadas do *pipeline* de implantação. A entrega contínua é uma evolução da integração contínua e envolve todo o ciclo do projeto, até a criação da nova versão da aplicação, mas a implantação dessa nova versão no ambiente de execução ainda é feita manualmente. A implantação contínua engloba a entrega contínua e adiciona o passo de automatizar a implantação da nova versão no ambiente de execução. A finalidade da entrega contínua é garantir o mínimo de esforço na implantação de novas alterações, enquanto a da implantação contínua é sempre manter o ambiente de execução atualizado com as últimas alterações. A adoção da implantação contínua deve ser ponderada, pois para alguns negócios é preferível uma taxa de implantações mais baixa. A arquitetura de microsserviços tem uma ótima afinidade com a implantação contínua, por ser naturalmente modularizada e mais facilmente testável (GitLab Incorporated, 2022a; Red Hat Incorporated, 2022).

Os benefícios da entrega contínua incluem: risco reduzido na implantação, pois como as mudanças são menores, há menos possibilidades de problemas, e caso haja, o conserto é mais simples; visualização do progresso, que não será simplesmente por trabalho "completo", mas sim por trabalho entregue; e *feedback* do usuário mais rápida e frequentemente (FOWLER, 2013).

Anti-padrão em CD: gerenciamento manual de ambientes

Diferenças entre ambientes que deveriam ser iguais, ou o mais similares possível, por exemplo homologação e produção. Diferenças entre réplicas. Resulta em implantações não confiáveis. Deve-se tratar a configuração de ambiente como código, com versionamento e automatizado. (HUMBLE; FARLEY, 2010)

Anti-padrão em CD: implantação manual

Realizar os passos da implantação manualmente resulta em uma implantação lenta e propícia a erros. Recomenda-se automatizar a implantação o suficiente para que possa ser feita

com apenas o clique de um botão, ou, caso seu negócio permita, ser completamente automática. (HUMBLE; FARLEY, 2010)

Anti-padrão em CD: baixa frequência de implantação

Implantar o *software* com baixa frequência resulta em pouca colaboração e entendimento entre a equipe de desenvolvimento e a de operações. Quanto mais frequente é a implantação, menor é a dificuldade dela. (HUMBLE; FARLEY, 2010; FOWLER, 2011)

4.7.2.1 Estratégias de lançamento

Estratégias de lançamento dizem respeito a como atualizar uma aplicação em produção, e a escolha da estratégia depende de fatores como risco, tempo de inatividade e a necessidade de controle sobre a liberação de funcionalidades. Algumas estratégias fazem uso de *dark launching* (lançamento escuro), que significa que algumas funcionalidades estão disponíveis para alguns usuários sem que eles saibam, permitindo testes e monitoramento com usuários reais antes do lançamento completo. É importante notar que implantação (*deploy*) e lançamento (*release*) são conceitos distintos, embora frequentemente usados de forma intercambiável. Implantação refere-se à ação técnica de implantar a nova versão do *software* em um ambiente de execução, enquanto lançamento é o momento em que a versão é disponibilizada aos usuários, uma decisão que envolve considerações de negócio.

Ao separar implantação e lançamento, as equipes técnicas podem realizar implantações mais ágeis, por não depender de decisões de negócios. Isso permite que a área técnica trabalhe com mais autonomia e frequência nas implantações, enquanto a área de negócios ainda pode decidir o melhor momento para liberar a nova versão para o público, alinhando o lançamento com objetivos estratégicos. A seguir são apresentadas algumas estratégias comuns de lançamento.

Blue-Green Deployment

No *Blue-Green Deployment*, quando deseja-se lançar uma nova versão, são mantidos um ambiente de produção com a versão antiga (*Blue*) e um com a versão nova (*Green*). Para permitir uma transição suave entre versões para os usuários, o tráfego é redirecionado para o ambiente novo após a validação da nova versão, favorecendo uma atualização segura e diminuindo riscos. Além disso, caso ocorra algum problema, é possível rapidamente reverter para o ambiente antigo, sendo necessário apenas mudar o redirecionamento. (FOWLER, 2010)

Canary Release

No *Canary Release*, a nova versão é disponibilizada para um número gradual dos usuários. Inicialmente, a nova versão é implantada em uma parte restrita do ambiente de execução, sem tráfego significativo, permitindo testes preliminares. Em seguida, um subconjunto de usuários

é direcionado para essa versão, possibilitando a detecção de falhas ou problemas antes que a atualização afete todos os usuários. Se nenhum erro crítico for identificado, a distribuição da nova versão ocorre de forma progressiva, garantindo maior controle e diminuindo riscos. Essa abordagem permite um monitoramento mais eficiente do desempenho e da estabilidade da nova versão, também possibilitando uma fácil reversão (*rollback*) da implantação caso necessário, assim protegendo a experiência dos usuários. (SATO, 2014)

Feature Toggles

Com os *Feature Toggles*, também conhecidos como *Feature Flags*, funcionalidades podem ser ativadas ou desativadas sem a necessidade de modificar o código-fonte ou realizar novas implantações. Essa abordagem permite maior flexibilidade no lançamento de novas funcionalidades, facilitando testes A/B, lançamentos graduais e controle operacional de determinadas partes do sistema, inclusive podendo ser feitas de forma que o usuário decida se quer ou não usar a nova funcionalidade. Existem diferentes categorias de Feature Toggles, incluindo *Release Toggles*, que gerenciam a ativação progressiva de novas features; *Experiment Toggles*, que suportam testes controlados; *Ops Toggles*, que ajustam o comportamento do sistema em tempo de execução; e *Permissioning Toggles*, que regulam o acesso de usuários a funcionalidades específicas. No entanto, o uso excessivo desses *toggles* pode aumentar a complexidade do código, tornando essencial a adoção de boas práticas para seu gerenciamento, como a documentação clara e a remoção de *toggles* obsoletos. (HODGSON, 2017)

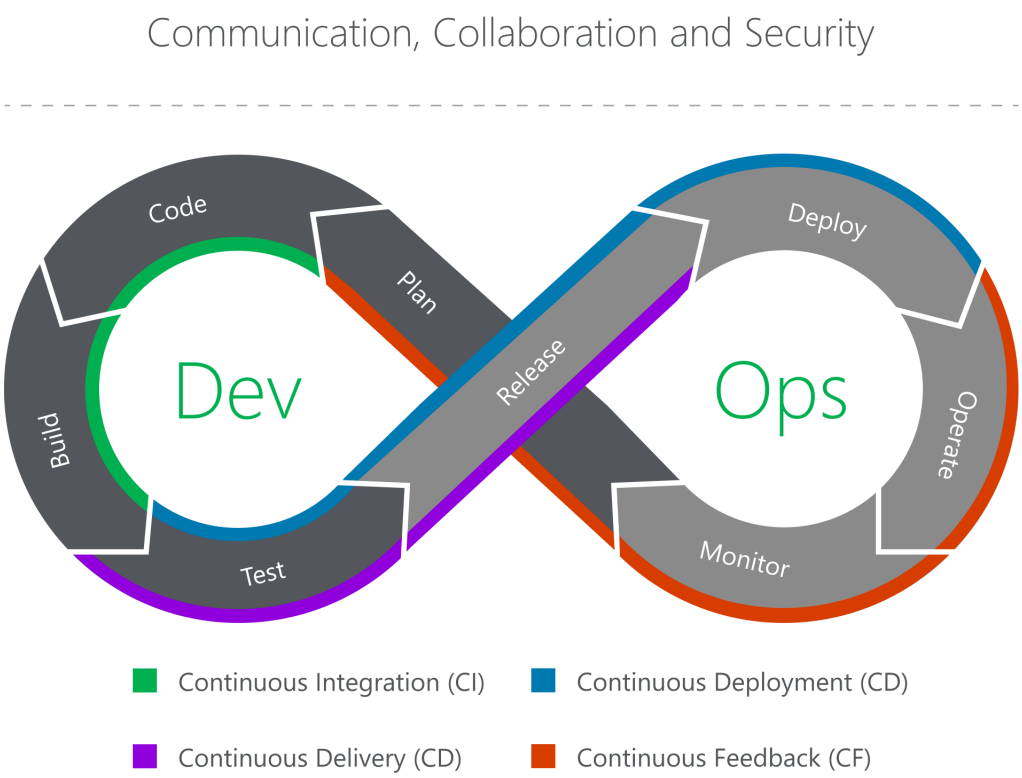
4.8 DevOps

Historicamente, existe uma diferença no foco do trabalho entre desenvolvedores e operadores que já está arraigada em muitas empresas que lidam com o desenvolvimento de *software*. Enquanto por um lado as equipes de desenvolvimento tentam ser o mais eficientes possível, entregando tarefas completas sempre que possível, por outro, a equipe de operações valoriza a estabilidade, considerando cada alteração como uma possível causa de problemas. Assim, uma equipe preza pela velocidade em novas funcionalidades, enquanto a outra, pela estabilidade. Com isso em mente, integrantes da indústria do desenvolvimento de *software* começaram a conceber um movimento, chamado DevOps, com o objetivo de alinhar o foco das equipes e favorecer o trabalho em conjunto para conseguir alcançar uma entrega contínua e ao mesmo tempo manter o *software* funcional no ambiente de execução.

DevOps, ilustrado na Figura 3, é um movimento cultural que visa a integração e otimização do processo de aprendizagem e colaboração entre os integrantes de equipes relacionadas ao desenvolvimento de *software*. Ao contrário do que alguns pensam, não se trata de um cargo ou de um conjunto de ferramentas, mas sim de uma visão organizacional de trabalho que tem o objetivo de automatizar o ciclo de desenvolvimento do *software* de modo que seja veloz, seguro

e integrado, com foco nas necessidades do usuário e *feedback* rápido. Práticas de DevOps, como integração e entrega contínua, são fundamentais para o sucesso de aplicações com arquitetura de microsserviços pelos benefícios que agrega ao time e ao desenvolvimento e operação da aplicação (GitLab Incorporated, 2022b).

Figura 3 – Ciclo DevOps



Fonte: Pennington (2020)

4.9 Organização de código

4.9.1 Monorepo

Monorepo é uma estratégia de organização de código onde usa-se apenas um repositório no sistema de controle de versionamento para gerenciar múltiplos projetos. Ela é usada por diversas grandes empresas como Google, Facebook e Microsoft para gerenciar inúmeros projetos que formam um repositório enorme. O benefício mais tangível dessa abordagem é a simplificação do gerenciamento do versionamento dos projetos - como todos ficam em apenas um repositório, é mais fácil entender o histórico de mudanças e acompanhar o estado da aplicação, também facilitando restaurações de estados anteriores (*rollbacks*). Segundo Brousse (2019), o uso dessa estratégia de organização de código também causa um impacto cultural nos times envolvidos

com os projetos, encorajando código consistente e de alta qualidade e melhorando a cognição e o trabalho em equipe deles. Por outro lado, essa estratégia é uma violação do fator I da metodologia 12-fatores e pode causar problemas, como pior desempenho da IDE e de ações como o *build* devido ao grande tamanho do repositório; sobrecarga do servidor de integração devido ao maior número de operações no repositório; aumento do acoplamento entre os projetos caso os desenvolvedores não sigam práticas adequadas para manter o acoplamento baixo; e aumento na complexidade da automação de processos relacionados a integração, entrega e implantação contínua. (FERNANDEZ; ACKERSON, 2022; SIWIEC, 2021; BROUSSE, 2019).

4.9.2 Multirepo

Multirepo, ou polirepo, é uma estratégia de organização de código onde usa-se múltiplos repositórios no sistema de controle de versionamento para o gerenciamento de múltiplos projetos. Na arquitetura de microsserviços isso geralmente significa ter um repositório para cada microsserviço e é a estratégia mais comum de organização de código. Suas vantagens em contraste com o monorepo incluem: tamanho razoável do repositório; escopo do repositório bem definido; definição de permissões diferentes para cada projeto; e operações no repositório têm melhor desempenho. Essa abordagem é mais adequada quando não há necessidade de gerenciar cuidadosamente a versão da aplicação como um todo e esse versionamento é transparente para os usuários (FERNANDEZ; ACKERSON, 2022).

4.10 Implantação em contêineres

Contêineres configuram isolamentos lógicos em uma máquina, sendo leves, altamente flexíveis e permitindo paradas, alterações e reinicios velozes. Depois de desenvolvido o microsserviço, é altamente recomendado que ele seja implantado em um contêiner, para assim favorecer sua padronização e portabilidade, também evitando interferências imprevistas com outros microsserviços. (Oracle Corporation, 2021).

4.11 Testes

Tradicionalmente, um *build* engloba tudo que é necessário para que um programa possa executar. Entretanto, só porque um programa executa não significa que ele fará o que é esperado. Para tanto, deve-se testar o programa, idealmente de forma automática e para toda funcionalidade, assim falhas e *bugs* podem ser descobertos antes de serem lançados. Existem inúmeros tipos e abordagens de testes de *software*, porém é impossível ter uma cobertura completa de testes. Trata-se, portanto, de quão testada é a aplicação - quanto mais bem testada, pelo uso de abordagens diversas e apropriadas, maior é a confiabilidade e a qualidade do sistema. Entretanto, é importante também não só automatizar a bateria de testes, mas também prezar pelo seu desempenho, pois

testes demorados tornam-se um obstáculo para a integração e *feedback* contínuos, portanto antes de sair criando vários testes complexos para a aplicação, deve-se avaliar se a confiabilidade resultante justifica a complexidade de implementação e execução. (FOWLER, 2006; NEWMAN, 2021)

Na arquitetura de microsserviços, o processo de testes torna-se mais abrangente, por haver mais pontos passíveis de falha, e mais complexo, por se tratar de um sistema altamente distribuido. Além disso, um microsserviço, de maneira isolada, também pode ser testado com tipos de testes comumente usados em aplicações monolíticas, como testes de unidade e de serviço.

4.11.1 Testes de unidade

Testes de unidade verificam funções ou métodos isoladamente, sem depender de serviços externos ou conexões de rede. Esse tipo de teste é voltado para desenvolvedores, ajudando-os a detectar *bugs* e facilitando a refatoração do código, por garantir que alterações estruturais não quebrem funcionalidades existentes. Além disso, esses testes têm custo muito baixo, tanto para implentar quanto para executar, então é muito difícil uma aplicação chegar a um ponto que tenha testes de unidade demais. Em específico para linguagens interpretadas, eles podem ser acionados automaticamente ao modificar arquivos, proporcionando ciclos de *feedback* mais rápidos (NEWMAN, 2021).

4.11.2 Testes de serviço

Testes de serviço são projetados para testar microsserviços diretamente, ignorando interfaces de usuário. Em aplicações monolíticas, isso envolveria testar classes que fornecem serviços à essa interface, enquanto em sistemas baseados em microserviços, cada teste de serviço foca nas capacidades individuais de um microsserviço. Esses testes aumentam a confiança no comportamento do serviço, contanto que o escopo seja isolado, isso é, qualquer falha detectada deve estar restrita ao microsserviço testado. Para garantir esse isolamento, muitas vezes é necessário substituir todas as dependências externas por simulações delas (*stubs*) (NEWMAN, 2021).

Alguns testes de serviço podem ser tão rápidos quanto testes unitários, mas a velocidade pode diminuir ao envolver bancos de dados reais ou conexões de rede com serviços simulados. Apesar de cobrirem um escopo maior que os testes unitários, tornando a identificação precisa de falhas mais difícil, eles ainda possuem menos variáveis do que testes de maior escala como os testes ponta a ponta, sendo assim mais confiáveis. (NEWMAN, 2021).

4.11.3 Testes *end-to-end* (ponta a ponta)

Testes ponta a ponta verificam o fluxo completo de uma determinada funcionalidade da aplicação, incluindo integrações externas, assim envolvendo múltiplos microsserviços. Newman

(2021) argumenta em seu livro "Building Microservices" que muitos desenvolvedores de microsserviços em escala dispensam testes ponta a ponta apesar do ótimo escopo que eles provêem, devido à grande quantidade de recursos que exigem para serem executados e por serem suscetíveis a problemas que não necessariamente tem relação com a funcionalidade testada, o que pode introduzir indeterminismo nos testes. É recomendado então usar testes ponta a ponta apenas quando o sistema não é muito grande, e a confiabilidade agregada por eles supera o custo de os manter. Conforme uma aplicação de microsserviços cresce, muitos desenvolvedores preferem aumentar a confiabilidade pelo uso de monitoramento avançado, estratégias de implantação cuidadosas e uso de contratos de dados orientados ao consumidor (CDCs) (NEWMAN, 2021).

4.11.4 Testes, testes e mais testes

Além de recomendar os tipos de testes comuns em monólitos, Familiar (2015) recomenda também testar os microsserviços conforme passam pelo *pipeline* de implantação, incluindo: **Testes internos**, que testam as funções internas do serviço, incluindo uso de acesso de dados, caching e relacionados; **Testes de serviço**, que testam serviços da API e seus modelos associados; **Testes de protocolo**, que testam o serviço a nível de protocolo, chamando a API usando o protocolo escolhido; **Testes de composição**, que testam o serviço em cooperação com outros serviços no contexto de uma solução; **Testes de escalabilidade e taxa de transferência**, que testam a escalabilidade e elasticidade do microsserviço implantado; **Testes de tolerância a falha**, que testam a capacidade do microsserviço de se recuperar após uma falha; e **Testes de penetração**, que consiste em envolver uma empresa terceirizada de segurança de *software* para realizar testes de penetração no sistema. (FAMILIAR, 2015)

4.12 Comunicação

No desenvolvimento de estruturas de comunicação entre diferentes processos, nota-se muitos produtos e abordagens que enfatizam o emprego de grande inteligência no próprio mecanismo de comunicação. Um exemplo disso é o Enterprise Service Bus (ESB), onde os mecanismos dessa abordagem geralmente incluem recursos sofisticados para roteamento, tratamento e transformação de mensagens e aplicação das regras de negócios (FOWLER; LEWIS, 2014).

A comunidade de microsserviços favorece uma abordagem alternativa - *endpoints* inteligentes e canais simples. Os microsserviços visam ser o mais desacoplados e coesos possível - eles possuem sua própria lógica de domínio e agem mais como filtros - recebendo uma solicitação, aplicando a lógica conforme apropriado e produzindo uma resposta. Isso geralmente é feito usando protocolos REST simples em vez de protocolos complexos como *Web Service Choreography* ou orquestração por uma ferramenta central (FOWLER; LEWIS, 2014).

O uso de APIs em conjunto com requisições HTTP é o método mais usado para realizar

comunicação síncrona na arquitetura de microsserviços. Uma requisição HTTP é feita por um cliente (ou consumidor) para um dado provedor em um endereço, com o propósito de acessar um recurso dele. Por usar o *Transmition Control Protocol* (TCP), é um método de comunicação confiável, mas não tão eficiente quanto poderia ser (FOWLER; LEWIS, 2014).

Para comunicação assíncrona, sistemas de mensagens são amplamente usados. Quando um serviço precisa enviar informações a outro de modo assíncrono, ele envia uma mensagem para uma fila de mensagens e ela será armazenada até ser processada ou excluída. As filas de mensagens podem ser usadas para dividir um processamento pesado, para armazenar trabalho em *buffers* ou lotes, ou para amenizar picos de cargas de trabalho (Amazon Web Services Incorporated, 2022).

Embora menos comum, chamada de procedimento remoto (RPC) também é utilizado para realizar comunicação síncrona ou assíncrona nos microsserviços. Uma chamada de procedimento remoto se dá quando um programa faz com que um procedimento ou uma sub-rotina execute em um espaço de endereço diferente, comumente em outra máquina numa rede compartilhada. Essa chamada é feita como se fosse um procedimento local, isso é, o programador não precisa explicitar que se trata de um procedimento remoto. (Microsoft Corporation, 2022b).

De acordo com Waseem et al. (2021), *API Gateway* e *Backend for frontend* são os padrões de projeto mais utilizados na implementação da comunicação entre microsserviços. Eles são padrões similares - a diferença é que no *Backend for Frontend* há um *gateway* para cada tipo de cliente ou serviço de ponta.

4.13 APIs

Considerando que APIs são uma parte crucial no desenvolvimento de microserviços, sendo responsável por grande parte da comunicação que se faz necessária para conectar tantos serviços separados e manter um funcionamento eficiente e livre de falhas, este trabalho apresentará diversas práticas no desenvolvimento de APIs.

4.13.1 Contratos de dados

Contratos de dados representam os acordos formais sobre a estrutura e formato das informações trocadas entre uma API e seus consumidores, bem como definem os campos, tipos de dados e regras de comunicação usadas, garantindo uma interação sem inconsistências. Esses contratos também implicam num compromisso de manter o serviço correspondente funcionando e inalterado. Entretanto, com a evolução de um serviço, surge a necessidade de serem introduzidas melhorias e mudanças que podem impactar nesses contratos (FOWLER; LEWIS, 2014).

Embora o versionamento de APIs seja uma solução tradicional para lidar com essa evolução, no contexto de microsserviços ele deve ser a última alternativa, pois tende a aumentar a

complexidade da manutenção deles. Em vez disso, é recomendável projetar contratos que sejam flexíveis e resilientes a mudanças e introduzir apenas modificações aditivas nos provedores, assim não comprometendo consumidores existentes (FOWLER; LEWIS, 2014).

Contratos de dados orientados ao consumidor (CDC) é uma abordagem para garantir que contratos de comunicação entre serviços estejam alinhados com as necessidades dos consumidores. Em vez de o provedor da API definir unilateralmente o contrato, os consumidores especificam suas expectativas, criando um contrato que o provedor formaliza por meio de testes automatizados a nível de serviço, permitindo que provedores validem continuamente se suas mudanças mantêm a compatibilidade com os consumidores sem aumentar a carga de testes consideravelmente, o que também reduz a necessidade de testes ponta a ponta, que tendem a ser custosos, como discutido na subseção 4.11.3 (ROBINSON, 2006; NEWMAN, 2021).

Ao projetar serviços de forma mais tolerante a mudanças e validar contratos dinamicamente, as equipes podem trabalhar de forma independente e lançar atualizações sem impactar outros sistemas. No entanto, para que essa abordagem funcione bem, é necessário uma boa colaboração entre as equipes e o uso de ferramentas adequadas que facilitem a criação e verificação automática desses contratos. Quando bem aplicado, o CDC promove maior estabilidade em sistemas distribuídos, permitindo que microsserviços evoluam de forma mais previsível e segura. (ROBINSON, 2006)

4.13.2 Formatos de dados

Atualmente JSON é um dos formatos mais populares para troca de dados na web, por ser facilmente lido tanto por humanos quanto por máquinas e não precisar de muitos metadados, como no formato XML. Em APIs, JSON é usado para enviar e receber requisições por meio do protocolo HTTP, sendo uma solução robusta para a comunicação entre cliente e servidor. Embora seja derivado do JavaScript, JSON também é suportado por muitas outras linguagens, seja nativamente ou por meio de bibliotecas. (BOURHIS; REUTTER; VRGOč, 2020)

Outra opção para trocas de dados é usar *Buffers* de protocolo, ou *Protocol buffers*, uma ferramenta de código aberto desenvolvida pelo Google que oferece um método de serialização de dados estruturados para envio de informações. Ela é neutra em linguagem e plataforma, é extensível e funciona como alternativa ao JSON na troca de dados. Essa ferramenta serializa os dados a serem enviados de modo a tornar o pacote mais leve e mais rápido, mas introduz uma complexidade extra. Por introduzir essa complexidade na troca de dados e não ser otimizada para quantidades de dados que excedem alguns megabytes, essa ferramenta não é recomendada para todo caso de uso (Google LLC, 2022).

4.13.3 Códigos de status de respostas HTTP

Esses códigos são números entre 100 e 599 que devem ser enviados com a resposta à requisição, cada um tendo um significado diferente, e cada centena sendo classificada em tipos diferentes de resposta. 100-199 representam respostas de informação, 200-299, respostas de sucesso, 300-399, tipos de redirecionamentos, 400-499, erros por parte do cliente e 500-599, erros por parte do servidor. Esse é um padrão definido na seção 10 da RFC 2616, e facilita o entendimento do cliente sobre o que aconteceu com a requisição à API. (NIELSEN et al., 1999; GUPTA, 2024)

4.13.4 API Gateway

Na arquitetura de microsserviços, muitas vezes a comunicação acontece de muitos pra muitos, podendo um microsserviço enviar e receber requisições para e de múltiplos outros microsserviços. Caso essa comunicação não seja gerenciada de forma adequada, a escalabilidade e segurança do sistema podem ser afetadas negativamente. Conforme mais microsserviços são criados e os já existentes evoluem, torna-se inviável gerenciar tantos *endpoints* em todo microsserviço que precisar usá-los. Uma solução para isso é usar um *API Gateway*. (NEWMAN, 2021)

Um *API Gateway* é um padrão de projeto onde tem-se um servidor que age como uma porta única de entrada para as APIs de um sistema, padronizando e controlando o acesso a elas. Esse *gateway* fica situado entre o consumidor e os microsserviços, e é responsável por redirecionar as requisições recebidas para os microsserviços apropriados, assim o gerenciamento das chamadas pode ser feita em apenas um lugar em vez de em cada consumidor. Além disso, nele também podem ser implementadas camadas de segurança, como autenticação, e de monitoramento, como *logging*.

Outra vantagem é que esse *API Gateway* pode agregar requisições, permitindo que o consumidor envie apenas uma requisição para o *API Gateway* para recuperar informações de diferentes microsserviços, o que normalmente exigiria múltiplas requisições. Nesse caso, quando recebida a requisição do consumidor, o *API Gateway* fica responsável por disparar as requisições correspondentes, agregar as respostas e as devolver ao consumidor.

Entretando, também há desvantagens no uso desse padrão: (1) cria-se um alto acoplamento entre os microsserviços e o *API Gateway*, (2) ele pode se tornar um ponto massivo de falha e (3) se não escalado adequadamente, esse *API Gateway* pode diminuir o de todas as requisições que passarem por ele (Microsoft Corporation, 2022a).

4.13.5 Segurança em APIs

Autenticação e autorização

Incluir autenticação em uma API consiste em exigir uma prova de autorização do uso daquele recurso. A autenticação nas APIs é altamente recomendada por aumentar a segurança de forma simples, e existem várias formas de implementá-la, sendo um dos mais comuns pelo uso de *JSON Web Tokens (JWT)*, definidos na RFC 7519 (NIELSEN et al., 1999).

Validação de entradas

Validar entradas significa verificar as requisições que chegam com o intuito de garantir que elas não contém dados impróprios, tais como injeções de SQL ou *scripting* (execução de uma determinada sequência de comandos) entre sites. Essa validação deve ser implementada tanto em nível sintático como em semântico, isso é, tanto impondo correção da sintaxe quanto impondo correção de valores (GUPTA, 2024).

Certificado Secute Socket Layer (SSL)

Usar um certificado SSL permite que o protocolo HTTPS seja usado em vez do HTTP, criptografando as informações que estão trafegando, aumentando a privacidade da informação trafegada. (GUPTA, 2024)

Limitação de taxa de requisições

Limitar a taxa de requisições é um jeito de proteger a infraestrutura do servidor nos casos de acontecerem grandes fluxos de requisições, tal como em um ataque de *DoS* (negação de serviço). Clientes terão seu acesso bloqueado caso enviem uma quantidade de requisições acima do limite determinado (GUPTA, 2024).

4.13.6 *Caching*

Às vezes referido como *cachear*, salvar informações em *cache* pode melhorar significativemente o tempo de busca da informação pelo cliente. Em uma API podem haver múltiplas requisições para a mesma informação em um curto intervalo de tempo, e para cada requisição será necessário buscar a informação. Entretanto, se a informação estiver salva no *cache*, não será necessário buscar essa informação, o que melhora o tempo de resposta da API, especialmente em *endpoints* que frequentemente retornam a mesma resposta (GUPTA, 2024).

4.13.7 Comprimir os dados

A transferência de cargas grandes pode diminuir a velocidade da API. Comprimir os dados auxilia nesse problema, diminuindo o tamanho da carga e aumentando a velocidade de

transferência. Uma possibilidade é usar *buffers* de protocolo (GUPTA, 2024).

4.13.8 Documentação

Uma API é apenas tão boa quanto sua documentação, e a falta de informações claras sobre seu uso pode ser um motivo suficiente para não utilizá-la. A documentação deve ser bem formatada e de fácil navegação, preferencialmente usando ferramentas populares para reduzir a curva de aprendizado dos desenvolvedores. Além disso, é importante incluir exemplos práticos de requisições e respostas, permitindo que os usuários testem facilmente a API. (GUPTA, 2024)

4.13.9 Paginar e filtrar

Em APIs, a paginação separa e categoriza resultados, enquanto a filtragem garante que apenas os resultados relevantes de acordo com os parâmetros da requisição são retornados. A paginação e a filtragem de resultados reduzem a complexidade da resposta e a quantidade de dados trafegados, assim poupando recursos (GUPTA, 2024).

4.14 Observabilidade e Monitoramento

A observabilidade se trata de permitir a observação do estado de um sistema por meio da externalização de seu comportamento, e possui 3 pilares - **registros, métricas e rastreamento**. O monitoramento engloba a observabilidade e se trata de acompanhar o estado de um sistema por meio de registros e ações que podem ser tomadas como resposta a eles. O monitoramento é fundamental para promover o funcionamento adequado de um sistema, especialmente os com arquiteturas distribuidas, como a de microsserviços. Com essas arquiteturas, o monitoramento se torna ainda mais importante e complexo, entretanto, traz diversos benefícios, como redução de tempo médio de detecção e reparo de incidentes e favorecimento do cumprimento do Acordo de Nível de Serviço (*Service Level Agreement* - SLA). Além disso, para que se possa ter um alto nível de automação, também é necessário haver um alto nível de monitoramento. As formas mais comuns de implementar monitoramento é por meio de *logs* (registros) e métricas.

Beyer et al. (2016) afirma que os aspectos mais importantes para se monitorar em sistemas distribuidos são latência (tempo de resposta do sistema), tráfego (demanda colocada no sistema), saturação (uso excedente de recursos do sistema) e erros occorridos, sejam erros explicítos (p. ex. erros 500 em uma requisição HTTP), implícitos (p. ex. respostas incorretas à requisição) ou por política (p. ex. tempo de resposta inadequado). O monitoramento desses aspectos muitas vezes são divididos em duas metodologias - RED e USE, que significam *Rate, Errors, Duration* (Taxa, Erros, Duração) e *Utilization, Saturation, Errors* (Utilização, Saturação, Erros), respectivamente. Enquanto RED foca na experiência do usuário, USE foca no funcionamento apropriado da infraestrutura, mas ambos são metodologias abrangentes e complementam um ao outro. (BEYER et al., 2016; DAM, 2018)

4.14.1 *Logs* (registros)

Um registro, ou *log*, descreve o que aconteceu em um dado momento em um dado processo, provendo informações rastreáveis sobre o estado e a saúde dele. Manter um histórico de registros de uma aplicação é uma forma simples e eficiente de se implementar monitoramento, e é fortemente indicado para qualquer sistema, especialmente os distribuidos.

Para favorecer uma eficiente escrita, leitura e operação dos registros, eles devem possuir ao menos: um código do evento ocorrido, uma mensagem descritiva, a condição tratada em caso de exceção ou erro e o período de retenção. Também deve-se padronizar o formato dos registros emitidos por todos os microsserviços e diferenciar entradas de erros, de avisos e de informação. Além disso, os registros devem ser agregados e organizados em um único lugar externo aos ambientes de execução da aplicação, para que possam ser facilmente consultados e para evitar perdas de informações, que podem acontecer especialmente em aplicações implantadas em contêineres e com escalamento horizontal. (BEYER et al., 2018)

4.14.2 Métricas

Uma métrica é uma medição de uma propriedade do sistema numa dada janela de tempo, e possui um objetivo específico, seja para questões de desenvolvimento, de infraestrutura ou mesmo de negócios e *business inteligence*. Disponibilizar a porcentagem de uso de CPU em uma máquina, por exemplo, tem o objetivo de informar o operador sobre esse uso para que ele possa decidir que ações precisam ser tomadas. Recomenda-se ter métricas para todo ponto de atenção do sistema, que, é claro, variam de acordo com o sistema e suas regras de negócio, porém geralmente incluem uso de recursos, tempo de resposta e quantidade de acessos.

Enquanto registros precisam ser desenvolvidos, métricas apenas precisam de instrumentação pois muitas ferramentas já possuem as próprias métricas ou já existem métodos consolidados para as obter. Nos servidores web mais populares, por exemplo, as informações básicas sobre uma requisição já são gravadas por padrão.

É recomendado usar paineis de controle de alto nível para melhorar a visualização e monitoramento do status da aplicação e diversas outras informações operacionais e de negócio a partir de suas métricas (FOWLER; LEWIS, 2014).

4.14.3 *Tracing* (rastreamento)

Em aplicações com arquitetura distribuida, como os microsserviços, uma única requisição de um usuário irá frequentemente interagir com múltiplos serviços antes de retornar uma resposta, assim havendo possibilidade de ocorrer problemas em múltiplos locais diferentes. Rastreamento trata-se de acompanhar um fluxo transacional (geralmente iniciado por uma requisição de um usuário), desde onde foi originado até onde terminou, atribuindo um identificador único a cada fluxo, que será propagado em cada serviço por onde o fluxo passar. Isso provê visibilidade ponta

a ponta sobre os fluxos no sistema, permitindo que desenvolvedores e operadores identifiquem problemas mais precisa e rapidamente, independentemente de onde eles ocorram.

5

Ferramentas para desenvolvimento de microsserviços

Este capítulo aprensenta ferramentas frequentemente usadas e que cumprem propósitos importantes no desenvolvimento de aplicações com arquitetura de microsserviços.

Quando se está procurando por ferramentas para o desenvolvimento de aplicações, a quantidade imensa de opções disponíveis pode ser opressiva. Perguntas como "Para que serve a ferramenta X"? "Qual a diferença entre a ferramenta X e a ferramenta Y", "Em qual cenário eu devo usar a ferramenta X?", "Qual ferramenta funciona melhor com a ferramenta X?"são muito comuns para iniciantes ou para pessoas experientes em poucas ferramentas. E apesar de nem sempre existirem respostas concretas para essas perguntas ou das respostas mudarem com o passar do tempo, esse capítulo provê uma orientação superficial para o desenvolvedor que procure entender o contexto de cada ferramenta e o que elas têm a oferecer.

5.1 *Frameworks* e linguagens de programação

Frameworks desempenham um papel fundamental no desenvolvimento de microsserviços, permitindo que os desenvolvedores criem sistemas modulares, escaláveis e de fácil manutenção de forma mais simples e evitando grande parte do código *boilerplate*. A escolha do *framework* e da linguagem pode dependender de diversos fatores, como os requisitos do sistema, a experiência da equipe e as preferências tecnológicas. Existem diversas opções disponíveis no mercado, cada uma com suas características, vantagens e desvantagens, sendo impossível abordar todas em detalhes. Dessa forma, aqui são discutidas apenas algumas das mais populares para o desenvolvimento de microsserviços.

5.1.1 Java e Spring Boot

Java é uma linguagem amplamente utilizada no desenvolvimento de microsserviços devido à sua robustez, escalabilidade e vasta gama de bibliotecas e ferramentas disponíveis. Spring Boot é um dos *frameworks* mais populares para construir microsserviços em Java por

simplificar o desenvolvimento de aplicações autônomas e de alto desempenho, permitindo que os desenvolvedores criem rapidamente serviços que podem ser facilmente integrados com outros sistemas. O Spring Boot oferece funcionalidades como configuração automática, injeção de dependência e integração com outros módulos do Spring, como o Spring Cloud, que facilita a implementação de microsserviços com capacidades de descoberta, balanceamento de carga e segurança. (VMware Tanzu, 2025)

A combinação de Java com Spring Boot é especialmente útil para empresas que já possuem uma base de código em Java ou que precisam de um ecossistema maduro com suporte a múltiplas ferramentas de monitoramento, escalabilidade e segurança. Além disso, a linguagem é fortemente tipada, o que ajuda a reduzir erros e a melhorar a manutenibilidade em sistemas grandes e complexos.

5.1.2 Python e Flask ou Django

Python é uma linguagem conhecida por sua simplicidade e rapidez de desenvolvimento, o que a torna uma escolha popular para prototipagem e desenvolvimento de microsserviços, especialmente quando o tempo é um fator crítico. Para a construção de microsserviços, *frameworks* como Flask e Django são frequentemente usados. O Flask é um *framework* minimalista e flexível, ideal para a criação de APIs simples e escaláveis. Ele fornece apenas os recursos essenciais, permitindo que os desenvolvedores escolham bibliotecas adicionais conforme necessário, o que dá flexibilidade para criar soluções sob medida. (PALLETS, 2025)

Por outro lado, Django é um *framework* mais completo e robusto, adequado para aplicações maiores que exigem uma estrutura mais rígida. Embora o Django seja tradicionalmente mais utilizado para aplicações monolíticas, ele pode ser adaptado para um estilo de arquitetura de microsserviços. Ele vem com uma série de funcionalidades, como mapeamento objeto-relacional (ORM), autenticação e gerenciamento de usuários, que podem ser valiosas em microsserviços que exigem manipulação de dados ou integração com bancos de dados relacionais. (Django Software Foundation, 2025)

5.1.3 Golang

Go, também conhecido como Golang, é uma linguagem desenvolvida pelo Google que se destaca pela sua alta performance e simplicidade. A principal vantagem do Go em microsserviços é a sua capacidade de lidar com concorrência de forma eficiente, com seu modelo de *goroutines* e *channels* (canais), que facilita o desenvolvimento de sistemas altamente concorrentes e escaláveis. O Go é especialmente popular em sistemas que exigem alta taxa de transferência e baixos tempos de latência, como sistemas de *streaming* ou microsserviços que lidam com grandes volumes de dados em tempo real. Além disso, o Go possui uma abrangente biblioteca padrão e ótimas bibliotecas desenvolvidas por terceiros, muitas vezes dispensando a necessidade de um *framework*

para a linguagem, tornando o microsserviço ainda mais simples. (Go Team, 2025)

5.1.4 JavaScript/TypeScript com Node.js e Express

Node.js também é muito popular para o desenvolvimento de microsserviços, especialmente em sistemas que exigem alta performance em *I/O* e processamento assíncrono. O Node.js usa um modelo de ciclo de eventos não bloqueante, que é ideal para microsserviços que precisam lidar com uma grande quantidade de requisições simultâneas, como APIs RESTful. Seu ecossistema vasto e a popularidade do JavaScript no desenvolvimento *frontend* também tornam o Node.js uma escolha atraente para equipes que desejam manter uma *stack* unificada em todo o sistema. Além disso, a enorme comunidade de desenvolvedores e a abundância de pacotes disponíveis facilmente instaláveis permitem que os microsserviços sejam desenvolvidos rapidamente e com uma ampla variedade de funcionalidades. O Express, um *framework* minimalista para Node.js, facilita o desenvolvimento de APIs simples e eficientes. (OpenJS Foundation, 2025a; OpenJS Foundation, 2025b)

5.1.5 C# e .NET

C# é uma das linguagens mais utilizadas no ecossistema Microsoft e tem se tornado uma escolha popular para o desenvolvimento de microsserviços, principalmente quando combinada com o *framework* .NET Core, que é uma plataforma de desenvolvimento de código aberto, alto desempenho e com suporte a diversos sistemas operacionais. O ASP.NET Core é um subconjunto do .NET focado em criar aplicações web, incluindo APIs RESTful, que são comumente usadas para comunicação entre microsserviços. Desenvolver microsserviços com .NET ou ASP.NET é especialmente adequado para empresas que já operam em ecossistemas Microsoft. Ambos *frameworks* têm uma comunidade ativa e suporte contínuo, o que os torna uma escolha confiável para o desenvolvimento de microsserviços. (Microsoft Corporation, 2025b; Microsoft Corporation, 2025a)

5.2 Servidores Web

Servidores web são responsáveis por receber e tratar requisições, agindo como um *middleware* (meio-termo) entre o consumidor e o provedor. Eles podem disponibilizar diversas funcionalidades úteis na comunicação por meio de requisições, tal como *caching*, compressão de dados, limitação de requisições e balanceamento de carga, assim não havendo necessidade de implementá-las manualmente. Além disso, servidores web também podem ser utilizados como *API Gateways*, técnica crucial para gestão da comunicação entre microsserviços, por centralizar o tráfego e se encarregar de redirecionar requisições para os microsserviços adequados.

5.2.1 Nginx e Apache HTTP Server

O Nginx é um servidor web projetado para lidar com um grande número de conexões simultâneas com consumo eficiente de recursos, tornando-o uma excelente escolha para aplicações escaláveis e dinâmicas. Sua arquitetura baseada em eventos permite melhor desempenho e escalabilidade em comparação com servidores baseados em *threads*, tornando-o ideal para aplicações modernas e de alto tráfego. Além disso, o Nginx pode ser facilmente utilizado como *API Gateway* e *proxy*, técnicas muito pertinentes para auxiliar na comunicação entre microsserviços. (F5 Incorporated, 2025)

O Apache HTTP Server, ou apenas Apache, é um servidor web antigo mas muito bem consolidado e muito usado até hoje. Seu suporte a módulos dinâmicos e seu modelo híbrido de processamento permitem ótima adaptação, mas podem resultar em maior consumo de recursos quando comparado ao NGINX em cenários de alto tráfego. Apesar disso, o Apache ainda é uma ótima opção para aplicações tradicionais ou que necessitem de compatibilidade com tecnologias legadas. (GARNETT; ELLINGWOOD, 2022)

5.3 Bancos de dados persistentes - SQL e NoSQL

Como mencionado na seção 4.6, cada microsserviço deve possuir seu próprio modelo de dados independente, podendo usar bancos de dados distintos. Portanto, a escolha do banco de dados adequado é um aspecto importante no projeto de um microsserviço, sendo importante considerar alguns fatores.

Os bancos de dados relacionais (SQL), como MySQL e PostgreSQL, armazenam dados em tabelas estruturadas, utilizando esquemas predefinidos. Eles são ideais para aplicações que requerem alta consistência e integridade dos dados, especialmente quando há muitas relações entre entidades. Esses bancos seguem o modelo ACID (Atomicidade, Consistência, Isolamento e Durabilidade), o que os torna indicados para cenários onde transações complexas e confiabilidade são essenciais. (MongoDB, 2024)

Por outro lado, os bancos de dados NoSQL, como MongoDB e Cassandra, oferecem maior flexibilidade ao lidar com dados não estruturados ou semi-estruturados. Eles diferem entre sí em como armazenam informações, podendo ser em documentos JSON (como o MongoDB), em *wide column store* (como o Cassandra), em grafos (como o Neo4j), ou vários outros, sendo a escolha desse tipo de armazenamento também importante. Em contraste com o formato de tabelas estruturadas, esses formatos tendem a facilitar a escalabilidade horizontal e a manipulação de grandes volumes de dados. Bancos de dados NoSQL são especialmente úteis para aplicações que exigem alta disponibilidade e rápida adaptação a mudanças nos dados. (MongoDB, 2024)

O modelo de dados deve ser avaliado para determinar se ele é altamente estruturado e relacional, favorecendo SQL, ou se é mais flexível e variável, onde NoSQL pode ser mais

apropriado. Além disso, a escalabilidade é um fator essencial: enquanto bancos SQL geralmente escalam melhor verticalmente (aumentando os recursos de um único servidor), bancos NoSQL são melhor projetados para escalabilidade horizontal (adicionando mais servidores conforme a necessidade). (MongoDB, 2024)

Outro ponto importante é a escolha entre consistência e disponibilidade. Bancos de dados relacionais enfatizam consistência rigorosa, facilitando que todas as transações sejam processadas corretamente antes de serem confirmadas. Já os bancos NoSQL são mais flexíveis nesse quesito. Por serem projetados para serem descentralizados, podem oferecer ou disponibilidade (caso do Cassandra) ou consistência de dados (caso do MongoDB), mas nunca os dois. Assim sendo, se um microsserviço precisa de transações complexas e forte integridade referencial, um banco de dados SQL tende a ser a melhor opção. (BROWN, 2020; MongoDB, 2024)

Ademais, a decisão entre bancos de dados SQL e NoSQL deve ser feita com base nos requisitos específicos de cada microsserviço. Algumas partes do sistema podem se beneficiar da confiabilidade e estrutura dos bancos relacionais, enquanto outras podem precisar da flexibilidade e escalabilidade dos bancos NoSQL. Uma análise cuidadosa desses aspectos ajudará a construir uma arquitetura de microsserviços mais eficiente e escalável, garantindo o desempenho e a confiabilidade da aplicação.

5.4 Bancos de dados em memória - Memcached e Redis

Outro tipo de banco de dados importante para microsserviços são os em memória, como o Memcached e o Redis, que são ideais para *caching*. O Memcached, apoiado pela Netflix, é simples e eficiente para armazenamento de chave-valor puro, sendo altamente escalável e consumindo pouca memória por não armazenar metadados complexos. No entanto, ele não oferece suporte a estruturas de dados avançadas, persistência, replicação nativa ou recursos como pub/sub. Já o Redis é mais versátil, suportando listas, conjuntos, hashes e permitindo também persistência dos dados no disco, além de replicação e *clustering*, tornando-se útil para aplicações além do cache, como filas e contadores. Enquanto o Memcached tende a ser mais rápido para o propósito puro de *cache* devido à sua leveza, o Redis é mais poderoso e flexível, mas pode exigir mais recursos dependendo do uso. (DORMANDO, 2018; REDIS, 2025)

5.5 Integração contínua e Entrega Contínua (CI/CD)

5.5.1 Sistemas de controle de versão

Um sistema de controle de versão é imprescindível para o desenvolvimento de *software* atualmente e é impossível se ter integração contínua sem um. Os sistemas de controle de versão podem ser categorizados em distribuidos (DVCS), como o Git, ou centralizados (CVCS), como o SVN. Os centralizados mantêm um repositório central onde todas as versões são armazenadas, e

os usuários precisam se conectar a ele para obter ou enviar alterações, enquanto nos distribuidos cada usuário possui uma cópia completa do repositório, permitindo trabalho *offline* e melhor gerenciamento de versões.

Atualmente os distribuidos são o padrão no desenvolvimento de *software*, e pouco se ouve falar dos centralizados. O **Git** é, empiricamente, o mais popular entre todos os sistemas de controle de versão, e é uma solução elegante e completa para o controle de versão no desenvolvimento de *software*, seja para um único desenvolvedor ou para equipes numerosas.

5.5.2 Plataformas para CI/CD

GitHub

O GitHub é uma plataforma parcial de CI/CD focada no gerenciamento de código-fonte e é extremamente popular, especialmente para projetos de código aberto. Ela oferece uma interface simples e fácil de usar, assim como diversas ferramentas de colaboração e gerenciamento de repositório. Entretanto, não oferece tantas ferramentas quanto o GitLab, então uma solução mais completa de CI/CD requeriria uso de outras ferramentas. Normalmente o GitHub pode ser usado apenas como serviço na nuvem, mas no plano *enterprise* há opções para auto-hospedagem.

GitLab

O GitLab é uma plataforma de CI/CD abrangente, oferecendo um conjunto mais completo de ferramentas para o ciclo de vida do desenvolvimento de *software*, desde ferramentas de controle de versão até ferramentas de implantação e monitoramento, simplificando a gestão e a integração de ferramentas por estarem todas em uma única plataforma. Além disso, o GitLab possui funcionalidades embutidas para trabalhar com as ferramentas Kubernetes e Docker, facilitando o uso delas e tornando a plataforma especialmente atraente para quem busca uma solução de CI/CD completa. O GitLab pode ser auto-hospedado ou usado como serviço na nuvem.

5.5.3 Servidores de integração

A escolha do servidor de integração para um microsserviço ou projeto afeta alguns fatores como facilidade de configuração, escalabilidade, suporte a *plugins* (extensões) e integração com outras ferramentas. Aqui são apresentadas 3 das opções mais populares, cada um tendo vantagens e desvantagens dependendo do contexto de uso.

GitHub Actions

O GitHub Actions se destaca pela integração nativa com repositórios do GitHub, tornando o processo de configuração bastante simples para projetos já hospedados no GitHub. Os *pipelines* para um repositório podem ser definidos diretamente nele, usando a linguagem YAML. Sua

maior vantagem está na facilidade de uso e na integração com o GitHub, permitindo a execução condicional e configurável do *pipeline* de acordo com eventos do repositório. No entanto, sua desvantagem principal é a dependência do ecossistema GitHub, o que requer uma migração de plataforma para projetos hospedados em outras plataformas de gerenciamento de repositórios.

Jenkins

O Jenkins é um dos sistemas de CI/CD mais bem estabelecidos e flexíveis do mercado. Sendo *open-source* e auto-hospedado, ele oferece uma grande variedade de *plugins* e suporte a diversas ferramentas. Os *pipelines* podem ser definidos para cada projeto usando arquivos jenkinsfile, que são baseados na linguagem Groovy. Sua principal vantagem é a personalização e flexibilidade, podendo ser configurado para praticamente qualquer *pipeline* de CI/CD. No entanto, essa configuração tende a ser mais complexa, exigindo maior conhecimento e manutenção por parte da equipe responsável, além da necessidade de gerenciar sua própria infraestrutura, o que aumenta a complexidade operacional.

GitLab CI/CD

Já o GitLab CI/CD oferece um meio-termo entre a simplicidade do GitHub Actions e a flexibilidade do Jenkins. Ele é integrado ao GitLab, mas não dependente, funcionando nativamente dentro da plataforma porém também podendo ser auto-hospedado, pelo uso da ferramenta GitLab Runner, o que constitui uma de suas maiores vantagens. Semelhante ao GitHub Actions, os *pipelines* podem ser definidos diretamente no repositório do GitLab usando a linguagem YAML. No entanto, para cenários com necessidades bem específicas, pode ser menos aplicável que o Jenkins.

5.6 Testes

A escolha de uma feramenta adequada para determinado teste depende do projeto em questão e do tipo de teste a ser executado. Embora muitos testes não exijam o uso de ferramentas além de uma linguagem de programação, elas podem diminuir significativamente o trabalho necessário. Contudo, como existem inúmeras coisas que podem ser testadas em um sistema, desde a eficácia de uma única função até uma funcionalidade abrangendo todo o sistema, apenas são listados aqui algumas das ferramentas para os testes mais comuns em microsserviços.

5.6.1 Testes de unidade

Testes de unidade são mais baixo-nível e as ferramentas usadas para os implementar dependem da linguagem de programação e *framework* em questão. No Java por exemplo, uma biblioteca muito comum para testes de unidade é o JUnit. No python, tem-se o pacote *unittest*.

No C#, tem-se o pacote *xUnit*. No Go, tem-se o pacote *testing*. No JavaScript tem-se a biblioteca *Jest*.

5.6.2 Testes de serviços

Testes de APIs

[EM PROGRESSO]

Postman, Insomnia, Thunder Client (integrado ao VSCode), cURL.

Testes de contratos

[EM PROGRESSO]

Pact, Spring Cloud Contract (para o *framework* Spring)

5.6.3 Testes *end-to-end* (ponta a ponta)

[EM PROGRESSO]

- Cypress: Uma poderosa ferramenta de teste de ponta a ponta para aplicativos da web que também pode testar APIs como parte do fluxo de trabalho de ponta a ponta.

- Selenium: Uma ferramenta para automatizar testes baseados em navegador para microsserviços que são expostos por meio de interfaces da web.

- Playwright: Uma biblioteca Node.js para automatizar navegadores Chromium, Firefox e WebKit, comumente usada para testes E2E de aplicativos da web.

- TestCafe: Uma ferramenta para automatizar testes da web, com suporte para testes entre navegadores e execução de testes paralelos.

- Appium: Uma estrutura para testar aplicativos móveis (nativos, híbridos ou web móvel).

5.7 Comunicação

5.7.1 RPC - gRPC

gRPC é uma ferramenta open-source desenvolvida pelo Google que permite o uso de Remote Procedure Call (RPC) entre sistemas distribuídos. Ele usa o formato *Buffers de protocolo* para trocas de dados com menor consumo de recursos. Uma característica importante do gRPC é seu suporte a *streaming* bidirecional e multiplexação, o que o torna ideal para cenários que exigem comunicação em tempo real e transferências de dados grandes entre serviços. Ele suporta tanto comunicação síncrona quanto assíncrona, oferecendo flexibilidade na interação entre os serviços. O gRPC também usa o HTTP/2, que permite recursos como *streams* de dados multiplexados,

compressão de cabeçalhos e gerenciamento eficiente de conexões, tornando-o mais performático do que as APIs REST tradicionais baseadas em HTTP, especialmente em ambientes com baixa latência e alto fluxo. Além disso, seu forte sistema de tipagem por meio de Protocol Buffers e suporte embutido para autenticação, balanceamento de carga e descoberta de serviços fazem do gRPC uma ferramenta poderosa para a construção de microsserviços de alto desempenho, escaláveis e seguros. (gRPC Authors, 2025)

5.7.2 Sistemas de mensagens

Sistemas de mensagens são plataformas que permitem a comunicação entre diferentes componentes ou serviços de um sistema distribuído por meio de mensagens. Eles facilitam a comunicação assíncrona entre sistemas, desacoplando produtores e consumidores de mensagens. Esses sistemas são essenciais para arquiteturas escaláveis, como microserviços, permitindo a transmissão eficiente de dados entre diferentes partes do sistema sem interdependências diretas. Aqui são apontados alguns sistemas de mensagens comumente usados em arquiteturas distribuidas, sendo a escolha entre eles dependente da forma das mensagens, requisitos de escalabilidade e se a solução é auto-hospedada ou com infraestrutura gerenciada.

RabbitMQ

O RabbitMQ é um sistema de mensagens baseado no protocolo AMQP que facilita a comunicação assíncrona entre aplicações, oferecendo funcionalidades de publicação e consumo de mensagens. Ele garante confiabilidade na entrega de mensagens e flexibilidade para diversos padrões de troca de mensagens. Contudo, RabbitMQ pode ter limitações de escalabilidade em comparação com sistemas como o Kafka, que são mais adequados para grandes fluxos de dados em tempo real.

Apache Kafka

O Kafka é uma plataforma de *streaming* distribuído de dados, projetada para lidar com grandes volumes de dados em tempo real. Ele é utilizado principalmente para publicar, armazenar e processar fluxos de dados como eventos e logs. O Kafka oferece alta escalabilidade, durabilidade e alta taxa de transferência, suportando casos de uso como rastreamento de atividades de websites, agregação de logs e processamento de fluxos de dados. Com recursos como particionamento e replicação, o Kafka é ideal para arquiteturas de microserviços e sistemas de processamento de dados distribuídos que têm grandes fluxos de dados.

5.7.2.1 Sistemas de mensagens em provedores na nuvem

Nos provedores de nuvem AWS, Azure e GCP, tem-se o SQS, Azure Service Bus e Google Pub/Sub, respectivamente. Esses são plataformas como serviços (PAAS) que oferecem sistemas de mensagens totalmente gerenciados, abstraindo assuntos de infraestrutura. Eles são

escaláveis e fornecem alta disponibilidade, mas podem oferecer menos opções de configuração em comparação com soluções auto-hospedadas como RabbitMQ e Kafka. Enquanto o RabbitMQ se destaca em enfileiramento de baixa latência e Kafka em *streaming* de eventos em tempo real, soluções em provedores na nuvem tendem a se concentrar em simplicidade, escalabilidade e fácil integração com outros serviços em seus respectivos ecossistemas.

5.8 Conteinerização

Docker

O Docker é uma ferramenta de conteinerização amplamente adotada devido à sua facilidade de uso, documentação abrangente e grande comunidade. Ele segue um modelo cliente-servidor, onde um *daemon* (processo em segundo plano) gerencia os contêineres. Os benefícios do Docker incluem configuração simples; implantação e escalamento de contêineres independentemente; grande diversidade de imagens de contêineres disponíveis; e integração com o Kubernetes, uma ferramenta popular e poderosa para gerenciamento de infraestrutura em contêineres. Tais benefícios fazem do Docker uma excelente escolha para o desenvolvimento e implantação de microsserviços.

Podman

O Podman adota uma arquitetura sem *daemon*, eliminando a necessidade de um serviço central para gerenciar os contêineres. Isso o torna mais seguro, pois permite a execução de contêineres sem privilégios de *root*. O Podman também oferece compatibilidade com a CLI do Docker, facilitando a migração de projetos. Além disso, sua capacidade de gerenciar pods - um grupo de contêineres compartilhando recursos - também o torna uma opção adequada para arquiteturas de microsserviços, especialmente em ambientes com requisitos elevados de segurança.

LXC

O LXC (Linux Containers) adota uma abordagem diferente, fornecendo contêineres em nível de sistema operacional, assim permitindo a criação de ambientes virtuais que se comportam como sistemas Linux completos. Isso o torna ideal para casos onde se deseja uma alternativa leve às máquinas virtuais ou um acesso altamente eficiente a recursos do *hardware*, como em aplicações com enormes volumes de dados ou cargas de processamento. Entretanto, não é uma opção comum para implantação de microsserviços, que se beneficiam de contêineres mais isolados e especializados, como os oferecidos pelo Docker e pelo Podman.

5.8.1 Orquestração de contêineres com Kubernetes

[EM PROGRESSO]Kubernetes

Além disso, o kubernetes abstrai a necessidade de um serviço de service discovery, pelo uso de services que já fazem todo o gerenciamento de DNS dos serviços

5.8.2 Orquestração de contêineres em provedores na nuvem

[EM PROGRESSO]

AWS EKS, Azure Kubernetes Service (AKS)

5.9 Observabilidade e Monitoramento

5.9.1 Métricas: Prometheus e Grafana

[EM PROGRESSO]

5.9.2 Logging: Grafana Loki

Grafana Loki, ou apenas Loki, é uma *stack* para agregação e indexação de *logs*, sendo composto por um conjunto de componentes independentes. Ele funciona recebendo um fluxo de *logs* a partir de um agente que os captura da aplicação, e em seguida faz a indexação apenas de metadados deles, como um *label* (rótulo), que apontam para os dados do *log*, que são compactados e armazenados como objeto, assim consumindo pouco armazenamento. Além de consumir pouco armazenamento, o Loki também faz uso eficiente de memória; tem possibilidade para multilocação, ou seja, consegue escutar múltiplas aplicações enviando logs ao mesmo tempo, o que é importante em sistemas distribuidos; é altamente escalável, permitindo diferentes configurações de implantação; e permite que diversas outras ferramentas de observabilidade se conectem com ele. Entretanto, ele usa a própria linguagem de consulta - *LogQL*, o que dificulta o aprendizado da ferramenta. Além disso, por indexar apenas metadados, a busca de *logs* por conteúdo é mais dificil, e é preciso que os *logs* sejam bem estruturados e rotulados para ser eficiente. (Grafana Labs, 2025)

5.9.3 Logging: Graylog

[EM PROGRESSO] Graylog é um sistema de gerenciamento de *logs* mais simples do que o Loki.

5.9.4 *Tracing* (rastremanento)

[EM PROGRESSO] Jaeger

5.10 Aplicações *serverless* (sem servidor)

[EM PROGRESSO] Claudia, Apache Openwhisk, Serverless, Kubeless, IronFunctions, AWS Lambda, OpenFaaS, Microsoft Azure Functions.

6

Aplicação exemplar com arquitetura de microsserviços

Este capítulo apresenta a aplicação exemplar com arquitetura de microsserviços desenvolvida.

A aplicação desenvolvida trata-se de um sistema web de e-commerce. Nela, um cliente da loja pode buscar e comprar produtos, enquanto um administrador pode gerenciar produtos e usuários cadastrados, tudo a partir de uma interface de usuário em um navegador. O código fonte está disponível no repositório do GitHub com link <https://github.com/Jp9910/microservices_project>

Os diagramas de classes, pacotes e componentes podem ser vistos na Figura 4, Figura 5 e Figura 6, respectivamente.

6.1 Divisão dos microsserviços

[EM PROGRESSO] Foram usados conceitos do domain-driven design para determinar os limites de cada microsserviço...

Falar sobre o Padrão MVC, CLEAN architecture e SOLID principles usados nos microsserviços

[EM PROGRESSO] Em especial houve uma dúvida muito grande quanto a qual microsserviço será responsável pela lógica de negócios dos pedidos e seu armazenamento. Foram considerados o microsserviço de carrinho (que viraria o microsserviço de pedidos) ou o microsserviço de loja (que sem os pedidos viraria o microsserviço de catálogo). No fim decidi implementar no microsserviço de loja porque...

Serviço de negócios - Loja

Esse microsserviço trata da lógica de negócios relacionada a produtos e pedidos.

Serviço de negócios - Carrinho

Esse microsserviço trata da lógica de negócios relacionada ao carrinho e realização da compra.

Serviço de negócios - Usuários

Esse microsserviço trata da lógica de negócios relacionada ao cadastro e autenticação de usuários.

Serviço de negócios - Financeiro

Esse microsserviço trata apenas do processamento (fictício) do pagamento de um pedido.

Serviço de ponta - Clientes

O microsserviço de ponta de clientes é o serviço acessado diretamente pelos clientes da loja para realizar todas as operações relevantes a eles a partir de uma interface de usuário, tal como ver e buscar produtos, adicionar produtos ao carrinho e realizar um pedido a partir de um carrinho.

Serviço de ponta - Administração

O microsserviço de ponta de administração é o serviço usado pelos administradores da loja para realizar todas as operações relevantes a eles a partir de uma interface de usuário, tal como gerenciar produtos e usuários.

6.2 Práticas e ferramentas usadas

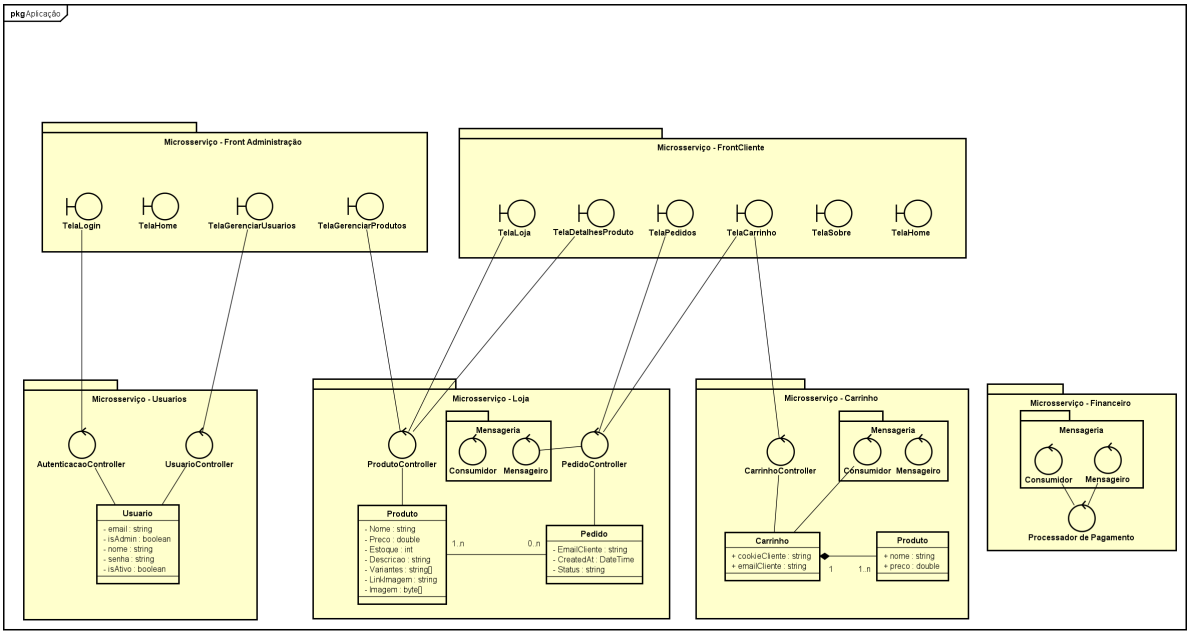
Frameworks e linguagens de programação

Para demonstrar a flexibilidade da arquitetura de microsserviços e como oportunidade para aprender novos *frameworks*, foram usados diferentes *frameworks* e linguagens de programação para os microsserviços. O microsserviço de usuários foi feito com Java e Spring Boot, o de loja com C# e .NET, o de carrinho e de financeiro com TypeScript e ExpressJS, o de ponta de cliente com ReactJS e o de ponta de administração com Angular.

Servidor web e Gateway

Como servidor web foi usado o Nginx nos serviços de ponta para servir os arquivos estáticos, por ser simples de configurar e oferecer diversas funcionalidades proveitosas para microsserviços. Além disso, ele também foi usado no componente de Gateway como um API

Figura 4 – Diagrama de classes da aplicação exemplar



Fonte: Autor

Gateway para o intermédio da comunicação entre os serviços de ponta e os serviços de negócio, redirecionando as requisições dos serviços de ponta para os serviços de negócio apropriados.

Bancos de dados e descentralização dos dados

Foram usados diferentes instâncias e tecnologias de bancos de dados na aplicação desenvolvida, assim proporcionando a descentralização dos dados. Nos microsserviços de loja e de usuários foi usado o banco de dados relacional PostgreSQL, tanto pela simplicidade de configuração quanto pela consistência de dados de operações mais importantes, como criação de um pedido ou de um usuário.

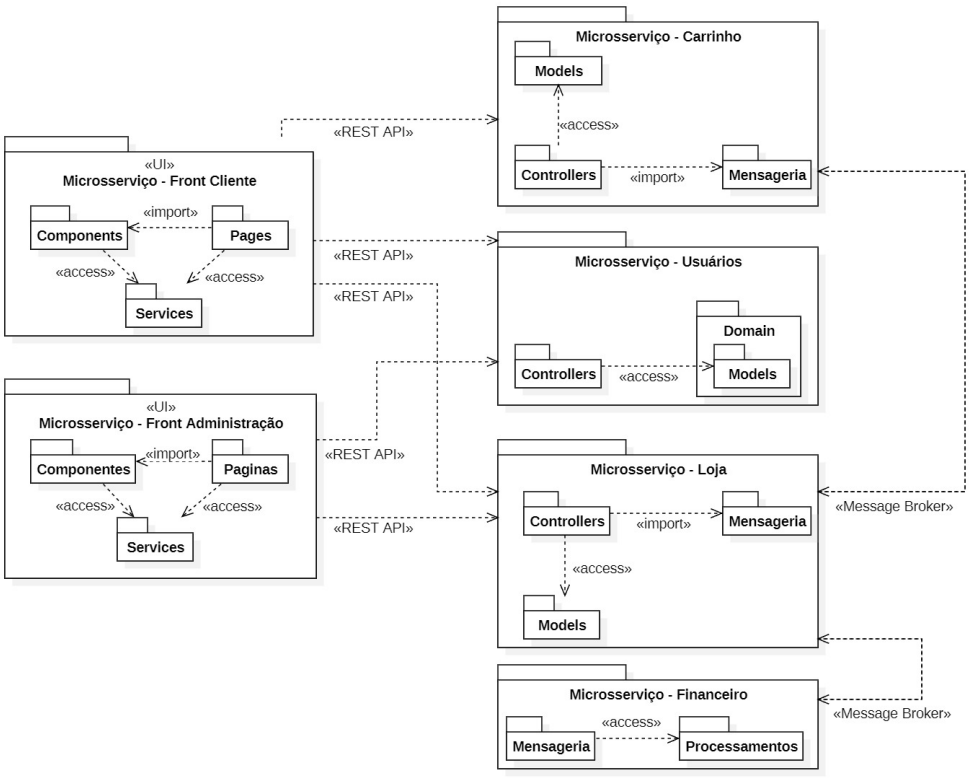
No microsserviço de carrinho foi usado o banco de dados NoSQL MongoDB porque um carrinho tem uma estrutura mais dinâmica, com itens podendo ter estruturas diferentes e sendo adicionados, alterados e removidos frequentemente.

Além disso, também foi usado o banco de dados em memória Memcached para realizar o *caching* da busca de produtos no microsserviço de loja, assim proporcionando menor latência e consumo de recurso nessa operação que é executada com alta frequência.

Metodologia 12-fatores

A maioria dos fatores da metodologia 12-fatores foram considerados e cumpridos no desenvolvimento da aplicação: I - Base de código única; II - Dependências portáteis e isoladas; III

Figura 5 – Diagrama de pacotes da aplicação exemplar



Fonte: Autor

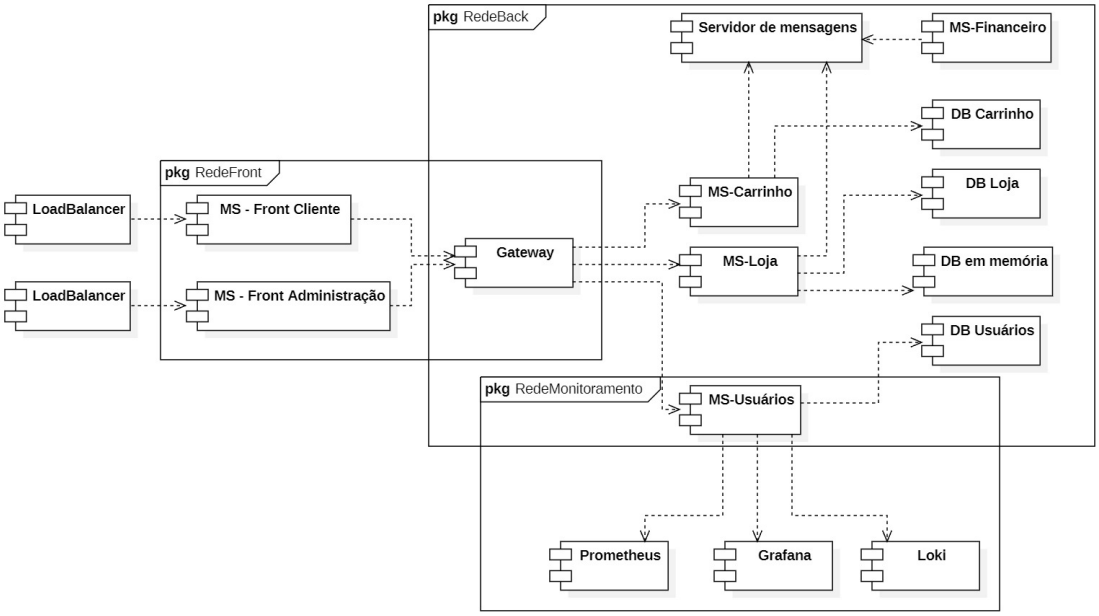
- Externalizar configurações; IV - Serviços de apoio são anexos; V - Separação entre construção, lançamento, e execução; VI - Processos sem estado (os microsserviços de ponta guardam a informação do usuário no navegador do cliente); VIII - simultaneidade; IX - Descartabilidade; X - Paridade de ambientes de execução; XI - *Logs*; e XII - Processos administrativos.

CI/CD

Foi usado o Git como sistema de controle de versão e o GitHub para gerenciamento de repositórios em todos os microsserviços, com configuração de proteção do ramo principal do repositório para evitar *commits* indevidos, assim sendo necessário a criação de um *pull request* e aprovação dele por um administrador do repositório para integração do novo código no ramo principal.

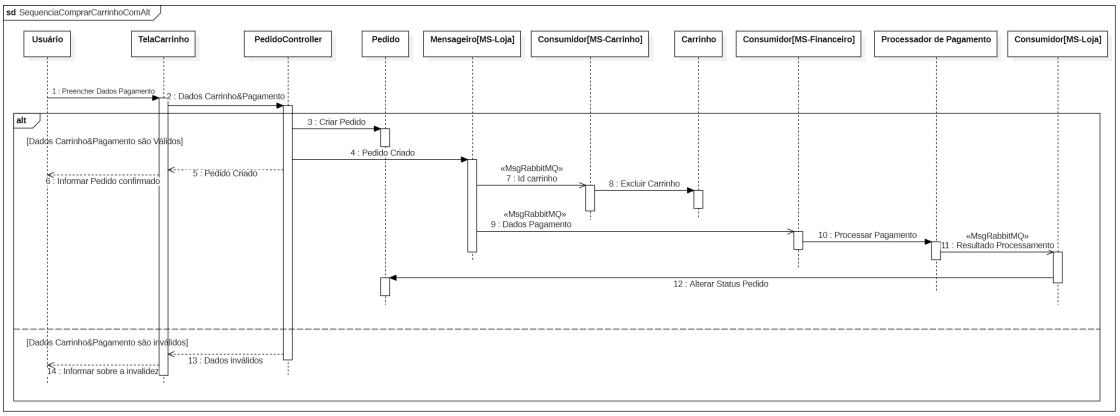
No microsserviço de ponta de administração também foi implementado um *pipeline* de CI/CD com 3 etapas sequenciais, com uso do GitHub Actions como servidor de integração para executar o *pipeline* automaticamente a cada novo *commit* recebido. A primeiro etapa executa o *linting* nos arquivos JavaScript com o ESLint e os testes de unidade com a biblioteca Karma; a segunda etapa faz a construção e upload do artefato já com o novo código. A terceira etapa

Figura 6 – Diagrama de componentes da aplicação exemplar



Fonte: Autor

Figura 7 – Diagrama de sequência de fazer uma compra no carrinho



Fonte: Autor

usa esse artefato para criar uma imagem de container Docker pronta para implantação, que é subida para o DockerHub, um repositório publico de imagens Docker, assim podendo ser baixada facilmente em qualquer máquina.

[anotação] Poderia ter um diagrama mostrando o pipeline

Organização do código - Multirepo

Para organização do código foi utilizada a técnica Multirepo com os repositórios no GitHub. Cada microsserviço possui um único repositório independente, porém também há um repositório principal que contém referências para os repositórios de cada microsserviço, reunindo tudo que é necessário para executar a aplicação inteira.

Contêinerização e orquestração de contêineres

[EM PROGRESSO]

Contêirezação com Docker e orquestração com docker-compose para desenvolvimento e Kubernetes para produção

Mencionar sobre a replicação e tudo mais do kubernetes.

O kubernetes vai buscar as imagens do repositório de imagens DockerHub e usá-las para configurar os containers para implantação

As configurações foram externalizadas, favorecendo a portabilidade. Todos os microsserviços podem ser iniciados em modo de desenvolvimento ou produção com mínimas alterações necessárias, apenas sendo necessário definir algumas variáveis de ambiente sensíveis, tal como a senha do banco de dados. Além disso, todos os microsserviços tem configurações de desenvolvimento e produção. Os serviços de ponta por exemplo, já usam essas configurações para definir o endereço dos microsserviços de negócio para onde vão fazer as requisições.

Testes

[EM PROGRESSO] Testes de unidade. Selenium?

APIs

[EM PROGRESSO]

RESTful,

Comunicação assíncrona

[EM PROGRESSO] RabbitMQ

Monitoramento

[EM PROGRESSO] Prometheus, grafana, loki,

O monitoramento foi feito apenas para o microsserviço de usuários por falta de tempo, mas todos os microsserviços podem ser configurados para utilizá-los, e novos *dashboards* podem ser criados para monitorá-los

7
Conclusão

[EM PROGRESSO]

Como pôde ser constatado, a escolha da arquitetura de uma aplicação não é uma decisão simples. Assim como quase tudo na computação, trata-se de *tradeoffs*, e para determinar se a arquitetura a ser escolhida é adequada, precisa-se entender e contextualizar seus benefícios, riscos, desvantagens e desafios. Apesar de não existir uma definição formal para a arquitetura de microsserviços, há muitas características que a diferencia de outras abordagens arquiteturais, tais como a componentização, a evolução, e a complexidade.

Foi observada uma ampla concordância entre pesquisadores e praticantes de microsserviços acerca do que é comum, do que é bem-visto, do que é considerado um anti-padrão, e de quais são os desafios no desenvolvimento de aplicações com arquitetura de microsserviços, assuntos os quais foram contextualizados e discutidos neste trabalho. Também foi descoberto um ponto em que há mais espaço para discussão e pesquisa - a prática de se começar uma arquitetura de microsserviços por uma arquitetura monolítica até que a aplicação e seus domínios já estejam bem definidos -, pois foi observado certo nível de discordância entre os autores das bibliográfias revisadas sobre o que é ou não necessário para sustentar uma arquitetura de microsserviços desde o início do desenvolvimento da aplicação, e quais seriam as razões para se adotar ou não essa arquitetura.

TODO: melhorar a conclusão

Referências

Amazon Web Services Incorporated. *Filas de mensagens*. 2022. Disponível em: <https://aws.amazon.com/pt/message-queue/>. Citado na página 39.

BEYER, B. et al. Site reliability engineering: How google runs production systems. In: ____. 1. ed. O'Reilly Media, 2016. cap. 6. ISBN 9781491929124. Disponível em: <https://sre.google/sre-book/monitoring-distributed-systems/>. Citado na página 43.

BEYER, B. et al. The site reliability workbook: Practical ways to implement sre. In: ____. O'Reilly Media, 2018. cap. 4. ISBN 9781492029458. Disponível em: <https://sre.google/workbook/monitoring/>. Citado na página 44.

BOURHIS, P.; REUTTER, J. L.; VRGOč, D. JSON: Data model and query languages. *Information Systems*, v. 89, p. 101478, Mar 2020. ISSN 03064379. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0306437919305307>. Citado na página 40.

BROUSSE, N. The issue of monorepo and polyrepo in large enterprises. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. New York, NY, USA: Association for Computing Machinery, 2019. (Programming '19). ISBN 9781450362573. Disponível em: <https://doi.org/10.1145/3328433.3328435>. Citado 2 vezes nas páginas 35 e 36.

BROWN, K. *Choosing the Right Databases for Microservices*. 2020. Disponível em: <https://www.ibm.com/think/insights/choosing-the-right-databases-for-microservices>. Citado na página 50.

DAM, J. *The RED Method: How to Instrument Your Services*. 2018. Disponível em: <https://grafana.com/blog/2018/08/02/the-red-method-how-to-instrument-your-services/>. Citado na página 43.

Django Software Foundation. *Django*. 2025. Disponível em: <https://www.djangoproject.com/>. Citado na página 47.

DORMANDO. *About Memcached*. 2018. Disponível em: <https://www.memcached.org/about>. Citado na página 50.

EVANS, E. *Domain-driven Design : Tackling Complexity in the Heart of Software*. 1. ed. Boston, Mass. ; Munich: Addison-Wesley, 2003. ISBN 9780321125217. Citado na página 25.

F5 Incorporated. *Beginner's Guide*. 2025. Disponível em: <https://nginx.org/en/docs/beginners_guide.html>. Acesso em: 15 Mar 2025. Citado na página 49.

FAMILIAR, B. What is a microservice? In: ____. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Berkeley, CA: Apress, 2015. p. 9–19. ISBN 978-1-4842-1275-2. Disponível em: <https://doi.org/10.1007/978-1-4842-1275-2_2>. Citado 7 vezes nas páginas 6, 15, 16, 18, 19, 20 e 38.

FERNANDEZ, T.; ACKERSON, D. *Release Management for Microservices*. 2022. Disponível em: <https://semaphoreci.com/blog/release-management-microservices>. Citado na página 36.

FOWLER, M. *Continuous Integration*. 2006. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 12 Nov 2022. Citado 3 vezes nas páginas 31, 32 e 37.

FOWLER, M. *Blue Green Deployment*. 2010. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/BlueGreenDeployment.html>. Acesso em: 15 Mar 2025. Citado na página 33.

FOWLER, M. *Frequency Reduces Difficulty*. 2011. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/FrequencyReducesDifficulty.html>. Acesso em: 24 Feb 2025. Citado na página 33.

FOWLER, M. *Continuous Delivery*. 2013. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/ContinuousDelivery.html>. Acesso em: 12 Nov 2022. Citado na página 32.

FOWLER, M. *Microservice Prerequisites*. 2014. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/MicroservicePrerequisites.html>. Acesso em: 06 Oct 2022. Citado 3 vezes nas páginas 23, 24 e 25.

FOWLER, M. *Microservice Tradeoffs*. 2015. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/microservice-trade-offs.html>. Acesso em: 13 Nov 2022. Citado 2 vezes nas páginas 12 e 21.

FOWLER, M. *Monolith first*. 2015. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/MonolithFirst.html>. Acesso em: 09 Nov 2022. Citado 3 vezes nas páginas 21, 23 e 24.

FOWLER, M.; LEWIS, J. *Microservices*. 2014. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 06 Nov 2022. Citado 9 vezes nas páginas 18, 19, 20, 28, 29, 38, 39, 40 e 44.

GARNETT, A.; ELLINGWOOD, J. *Apache vs Nginx: Practical Considerations*. 2022. Disponível em: <https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations>. Acesso em: 15 Mar 2025. Citado na página 49.

GitLab Incorporated. *What is CI/CD?* 2022. GitLab Topics. Disponível em: <https://about.gitlab.com/topics/ci-cd/>. Citado 3 vezes nas páginas 30, 31 e 32.

GitLab Incorporated. *What is DevOps?* 2022. GitLab Topics. Disponível em: <https://about.gitlab.com/topics/devops/>. Citado na página 35.

Go Team. *Go Programming Language*. 2025. Disponível em: <https://golang.org/>. Citado na página 48.

Google LLC. *Overview | Protocol Buffers*. 2022. Disponível em: <https://developers.google.com/protocol-buffers/docs/overview>. Citado na página 40.

Grafana Labs. *Loki overview | Grafana Loki documentation*. 2025. Disponível em: <https://grafana.com/docs/loki/latest/get-started/overview/>. Citado na página 56.

gRPC Authors. *What is gRPC? | Core concepts, architecture and lifecycle*. 2025. Disponível em: <https://grpc.io/docs/what-is-grpc/core-concepts/>. Citado na página 54.

GUPTA, L. *RESTful web API design*. 2024. Disponível em: <https://restfulapi.net/rest-api-best-practices/>. Acesso em: 16 Mar 2025. Citado 3 vezes nas páginas 41, 42 e 43.

Harness Incorporated. *Continuous Delivery vs. Continuous Deployment: What's the Difference?* 2021. Harness topics. Disponível em: <https://harness.io/blog/continuous-delivery-vs-continuous-deployment>. Citado na página 29.

HODGSON, P. *Feature Toggles (aka Feature Flags)*. 2017. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/feature-toggles.html>. Acesso em: 15 Mar 2025. Citado na página 34.

HUMBLE, J.; FARLEY, D. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 9780321601919. Citado 3 vezes nas páginas 31, 32 e 33.

IBM. *CAP Theorem*. 2022. Disponível em: <https://www.ibm.com/think/topics/cap-theorem>. Citado 2 vezes nas páginas 21 e 22.

LUMETTA, J. *Microservices for Startups: Should you always start with a monolith?* 2018. Disponível em: <https://buttercms.com/books/microservices-for-startups/should-you-always-start-with-a-monolith/>. Citado na página 24.

Microsoft Corporation. *The API gateway pattern versus the Direct client-to-microservice communication*. 2022. Disponível em: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>. Citado na página 41.

Microsoft Corporation. *gRPC*. 2022. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/architecture/cloud-native/grpc>. Citado na página 39.

Microsoft Corporation. *ASP.NET Core*. 2025. Disponível em: <https://dotnet.microsoft.com/en-us/apps/aspnet>. Acesso em: 14 Mar 2025. Citado na página 48.

Microsoft Corporation. *Microsoft .NET*. 2025. Disponível em: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>. Acesso em: 14 Mar 2025. Citado na página 48.

Middleware Lab. *What are Microservices? How Microservices architecture works*. Middleware Lab, 2021. Disponível em: <https://middleware.io/blog/microservices-architecture/>. Citado na página 10.

MongoDB. *NoSQL Vs SQL Databases*. 2024. Disponível em: <https://www.mongodb.com/resources/basics/databases/nosql-explained/nosql-vs-sql>. Citado 2 vezes nas páginas 49 e 50.

MONOLITHIC application. 2022. Page Version ID: 1118196758. Disponível em: <https://en.wikipedia.org/w/index.php?title=Monolithic_application&oldid=1118196758>. Citado na página 12.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. 2. ed. Sebastopol, Ca: O'reilly Media, 2021. ISBN 9781492034025. Citado 9 vezes nas páginas 6, 15, 17, 21, 22, 37, 38, 40 e 41.

NIELSEN, H. et al. *Hypertext Transfer Protocol – HTTP/1.1*. [S.l.], 1999. Disponível em: <https://datatracker.ietf.org/doc/rfc2616/>. Citado 2 vezes nas páginas 41 e 42.

OpenJS Foundation. *Express.js*. 2025. Disponível em: <https://expressjs.com/>. Acesso em: 14 Mar 2025. Citado na página 48.

OpenJS Foundation. *Node.js*. 2025. Disponível em: <https://nodejs.org/en/docs/>. Acesso em: 14 Mar 2025. Citado na página 48.

Oracle Corporation. topic, *Learn about architecting microservices-based applications on Oracle Cloud*. Oracle Corporation, 2021. Disponível em: <https://docs.oracle.com/pt-br/solutions/learn-architect-microservice>. Citado 5 vezes nas páginas 10, 14, 19, 29 e 36.

PALLETS. *Flask*. 2025. Disponível em: <https://flask.palletsprojects.com/en/2.0.x/>. Citado na página 47.

PENNINGTON, J. *The Eight Phases of a DevOps Pipeline*. 2020. Disponível em: <https://medium.com/taptuit/the-eight-phases-of-a-devops-pipeline-fda53ec9bba>. Citado na página 35.

Red Hat Incorporated. *What is CI/CD?* 2022. Red Hat Topics. Disponível em: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>. Citado 3 vezes nas páginas 29, 30 e 32.

REDIS. *Get Started with Redis Community Edition*. 2025. Disponível em: <https://redis.io/docs/latest/get-started/>. Citado na página 50.

RICHARDSON, C. *Microservices Pattern: Monolithic Architecture pattern*. 2018. Disponível em: <http://microservices.io/patterns/monolithic.html>. Citado 2 vezes nas páginas 13 e 14.

RICHARDSON, M. A. *Top 10 Challenges of Using Microservices for Managing Distributed Systems*. 2021. Disponível em: <https://www.spiceworks.com/tech/data-management/articles/top-10-challenges-of-using-microservices-for-managing-distributed-systems/>. Acesso em: 10 Nov 2022. Citado na página 21.

ROBINSON, I. *Consumer-Driven Contracts: A Service Evolution Pattern*. 2006. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/consumerDrivenContracts.html>. Acesso em: 14 Mar 2025. Citado na página 40.

RODRIGUES, R. de C. 2016. Disponível em: <https://www.fiap.com.br/2016/10/03/metodologia-12-fatores/>. Citado na página 26.

SATO, D. *Canary Release*. 2014. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/bliki/CanaryRelease.html>. Acesso em: 15 Mar 2025. Citado na página 34.

SIWIEC, D. *Monorepos for Microservices Part 1: Do or do not?* 2021. Disponível em: <https://danoncoding.com/monorepos-for-microservices-part-1-do-or-do-not-a7a9c90ad50e>. Citado na página 36.

TILKOV, S. *Don't start with a monolith*. 2015. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/dont-start-monolith.html>. Acesso em: 09 Nov 2022. Citado 2 vezes nas páginas 23 e 24.

VMware Tanzu. *Spring Boot*. 2025. Disponível em: <https://spring.io/projects/spring-boot>. Citado na página 47.

WASEEM, M.; LIANG, P.; SHAHIN, M. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software*, v. 170, p. 110798, 2020. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121220302053>. Citado 3 vezes nas páginas 6, 16 e 17.

WASEEM, M. et al. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, v. 182, p. 111061, 2021. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121221001588>. Citado 4 vezes nas páginas 6, 10, 17 e 39.

WIGGINS, A. topic, *The Twelve-Factor App*. 2017. Disponível em: <https://12factor.net/>. Acesso em: 21 Oct 2022. Citado na página 26.

XU, C. et al. CAOPLE: A Programming Language for Microservices SaaS. In: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. [S.l.: s.n.], 2016. p. 34–43. Citado 2 vezes nas páginas 10 e 21.