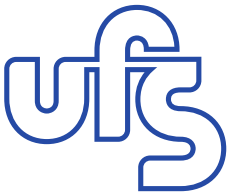


UNIVERSIDADE FEDERAL DE SERGIPE

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

2024

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): Admilson De Ribamar Lima Ribeiro

São Cristóvão – Sergipe

2024

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*. desenvolvimento de microsserviços. 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

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 o ramo do desenvolvimento de *software* provavelmente já ouviram 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 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 construir componentes leves e auto-contidos, que permitam que mudanças sejam desenvolvidas e implantadas rápida e independetemente. A partir disso originou-se um novo paradigma de desenvolvimento, chamado de "microsserviços", inspirado pela arquitetura orientada a serviços, e que tem ganho grande popularidade nos últimos anos (Middleware Lab, 2021; WASEEM et al., 2021).

Para projetar um aplicativo que seja multilíngue, facilmente escalável, fácil de manter e implantar, altamente disponível e que minimize falhas, considere a arquitetura de microsserviços. Esse tipo de arquitetura de *software* é amplamente considerado a melhor maneira de estruturar um sistema de *software* como um serviço atualmente. Entretanto, como quase tudo na computação, trata-se de uma troca (um *trade-off*), pois assim como há benefícios, também há riscos, desvantagens e desafios na adoção de uma arquitetura de microsserviços, os quais também serão discutidos neste trabalho (XU et al., 2016; Oracle Corporation, 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;

- Discutir práticas usadas no desenvolvimento de aplicações com arquitetura de microsserviços;

- Analisar a viabilidade da aplicação a ser desenvolvida no TCC 2

- Propor uma combinação de ferramentas para o desenvolvimento de aplicações com arquitetura de microsserviços;

- Contextualizar essas ferramentas e apontar os problemas que elas resolvem;

- Desenvolver uma aplicação de exemplo com arquitetura de microsserviços, usando a combinação de ferramentas proposta e algumas práticas discutidas;

- Discutir pontos positivos e negativos das ferramentas usadas nessa aplicação de exemplo.

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 mencionadas como imprescindíveis pelos autores e as que apareceram com mais frequência.

Para a proposta da combinação de ferramentas e a contextualização delas, será feita uma nova pesquisa bibliográfica para conhecer as mais comumente usadas em aplicações com arquitetura de microsserviços e entender como funcionam e os problemas que elas resolvem. Durante o desenvolvimento da aplicação de exemplo, serão reunidos e discutidos pontos positivos e negativos observados no uso das ferramentas.

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 esta última é o estilo mais comum de se desenvolver aplicações, 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. Os monólitos são autocontidos e totalmente independentes de outras aplicações. Eles são feitos não para uma tarefa em particular, mas sim para serem responsáveis por todo o processo para completar determinada função. Em outras palavras, as aplicações monolíticas têm problema de modularidade. Elas podem ser organizadas das mais variadas formas e fazer uso de padrões arquiteturais, mas são limitadas em muitos outros aspectos, discutidos 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 fácilmente entendida pelos seus desenvolvedores, fato que por sí só já melhora sua manutenibilidade.

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

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. Assim, os problemas dos monólitos começam a ficar evidentes (RICHARDSON, 2018).

2.1.2 Limitações

2.1.2.1 Crescimento, velocidade de desenvolvimento, e manutenção

Depois de chegar num 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 container 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

Escalar a aplicação pode ser um problema, pois ela só pode ser escalada em uma dimensão. É possível escalar o volume de operações executando mais cópias de um monólito, mas não é possível escalar 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/onde adequado (RICHARDSON, 2018).

2.1.2.3 Reutilização

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

2.1.2.4 Implantação

Realizar alterações em qualquer componente implica na necessidade de reimplantar toda a aplicação, mesmo as partes que não têm ligação com as alterações. Isso 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 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

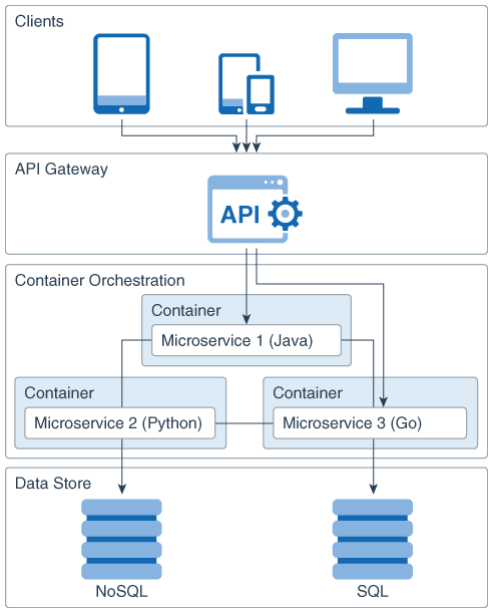
As escolhas de tecnologias são mais limitadas - um projeto tende a usar apenas um grupo de tecnologias porque realizar *upgrades* ou mudanças de tecnologias é uma tarefa complexa e pode causar problemas de compatibilidade (RICHARDSON, 2018).

2.1.2.7 Divisão de times

Quando a aplicação alcança determinado tamanho, é desejável dividir os desenvolvedores em times que têm foco em partes funcionais específicas. Entretanto, as aplicações monolíticas dificultam esse desenvolvimento independente porque com elas os times precisam coordenar o desenvolvimento e as implantações (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. Eles oferecem 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 eficientemente (FAMILIAR, 2015).

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* - que é reponsável por gerenciar as chamadas aos microsserviços e será melhor discutido na subseção 4.13.5 - fará as devidas requisições para os devidos microsserviços (localizados na camada *Container Orchestration*). Esses microsserviços então buscarão a informação necessária 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.

3
Características

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, e portanto microsserviços devem ser projetados para serem resilientes, isso é, tolerantes a falhas e ter um tempo de recuperação razoável quando algum problema 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 (FOWLER; LEWIS, 2014; FAMILIAR, 2015).

Um desafio advindo dessa distribuição é que a comunicação é complexa e falível. Geralmente ela depende de APIs e contratos de dados para definir como os microsserviços interagem, ou seja, os microsserviços são orientados-a-mensagens. A API define um conjunto

de *endpoints* acessíveis por rede, e o contrato de dados define a estrutura da mensagem que é enviada ou retornada (FAMILIAR, 2015).

3.2 Flexibilidade na escolha de tecnologias

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 possa escolher seu sistema operacional, linguagem e outras tecnologias. Além disso, eles podem ser desenvolvidos usando uma linguagem de programação e estrutura que melhor se adapte ao problema que ele é projetado para resolver. Embora essa flexibilidade seja algo positivo, usar muitas tecnologias diferentes aumenta a complexidade do sistema (Oracle Corporation, 2021; FAMILIAR, 2015).

3.3 Alta velocidade de desenvolvimento

Com um time responsável por cuidar do ciclo de desenvolvimento e sua automação, a velocidade com que microsserviços podem ser desenvolvidos é muito maior do que fazer 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 nesta arquitetura a maneira principal e mais eficiente para componentizar o *software* é justamente dividí-lo em microsserviços. 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 (FOWLER; LEWIS, 2014).

Quando uma aplicação usa múltiplas bibliotecas como componentes em um único processo, uma mudança em qualquer desses componentes resulta na necessidade de reimplantar a aplicação toda. Se essa mesma aplicação é decomposta em múltiplos serviços, é provável que uma mudança em um serviço só obrigaria a reimplantação do próprio serviço, assim tendo-se uma implantação mais simples e rápida. Ademais, considerando essa componentização e a independência dos microsserviços, eles são reusáveis por natureza e portanto torna-se mais fácil

criar soluções para o usuário final combinando 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 usar versionamento, 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 Flexibilidade no ambiente de execução

A implantação de microsserviços é altamente dependente de sua automação. Para garantir flexibilidade de ambiente de execução, essa automação deve incluir configuração de cenários diferentes de uso, não apenas para produção, mas também para desenvolvimento e testagem, possibilitando que o microsserviço tenha o melhor desempenho em diversos cenários. Para tanto, é necessário o uso de ferramentas que configurem essa flexibilidade (FAMILIAR, 2015).

3.6 Versionável e substituível

Apesar do versionamento não ser recomendado para microsserviços por dificultar o entendimento, quando necessário é possível manter versões diferentes de um mesmo serviço rodando ao mesmo tempo, assim proporcionando retrocompatibilidade e fácil migração. Além disso, serviços podem ser atualizados ou mesmo substituidos sem ocasionar indisponibilidade do serviço (FAMILIAR, 2015).

3.7 Evolução

Quanto maior e mais antigo o *software*, mais difícil é de lhe dar manutenção. Porém, com componentes modularizados e organizados adequadamente, uma aplicação com arquitetura de microsserviços tende a crescer muito mais horizontalmente do que verticalmente, possibilitando que ela nunca chegue em um estado em que fica inviável dar manutenção.

É possível decompor uma aplicação monolítica em uma arquitetura de microsserviços - tema de pesquisa muito recorrente na área de arquitetura de *software* - mas esse processo não faz parte do escopo deste trabalho.

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 tecnologias geralmente traz problemas por inexperiência dos desenvolvedores. Dessa forma, existem muitos desafios no desenvolvimento de aplicações com arquitetura de microsserviços que só conseguem ser compensados em aplicações mais complexas. E como em qualquer tipo de arquitetura, existem benefícios, riscos, desvantagens e desafios, e 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 desafio considerado crítico para o bom funcionamento da aplicação é a definição adequada dos limites de cada microsserviço. 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).

4
Práticas

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 divir 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 Provisionamento rápido

No contexto da computação, provisionamento significa disponibilizar um recurso, como uma máquina virtual por exemplo. Para produzir *software*, é necessário provisionar muitos recursos, tanto para os desenvolvedores quanto para o cliente. Naturalmente, o provisionamento é mais fácil na nuvem. Na AWS por exemplo, para conseguir uma nova máquina, basta inicar uma nova instância e acessá-la - um processo muito rápido quando comparado ao *on-premises*, onde precisaria-se comprar uma nova máquina, esperar chegar, configurá-la e só então ela estará pronta. Para alcançar um provisionamento rápido, é necessária automação (FOWLER, 2014).

4.1.2 Monitoramento básico

Muitas coisas podem dar errado em qualquer tipo de arquitetura, mas em especial nos microserviços pois cada serviço é fracamente acoplado, 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 para que problemas, especialmente os mais graves possam

ser detectados no menor tempo possível. Ademais, o monitoramento também pode ser usado para detectar problemas de negócio, como uma redução nos pedidos de um site de vendas, por exemplo (FOWLER, 2014).

4.1.3 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 A metodologia de 12 fatores

A metodologia de 12 fatores para o desenvolvimento de aplicações é um conjunto de regras e diretrizes para o desenvolvimento de aplicativos nativos da nuvem e *software* como um serviço. De acordo com ela, os microsserviços devem respeitar as seguintes orientações: (WIGGINS, 2017; Oracle Corporation, 2021; RODRIGUES, 2016)

I. Base de Código - Cada microsserviço deve ter uma base de código única e particular, com rastreamento utilizando controle de revisão, e devem ser criadas várias versões implantáveis;

II. Dependências - Cada microsserviço deve declarar e isolar suas dependências;

III. Configurações - Configurações de ambiente devem ser armazenadas fora do microsserviço, para que ele possa decidir a configuração apropriada a ser usada;

IV. Serviços de Apoio - Os microsserviços não devem fazer distinção entre serviços de terceiros e serviços locais;

V. Construir, lançar, e executar - Deve-se separar e distinguir cada etapa do processo de desenvolvimento. Na etapa de construção, o código é transformado em um executável. Na etapa de lançamento, o executável se combina com a configuração atual da implantação, seja teste, desenvolvimento ou produção. Na etapa de execução, o aplicativo é executado no ambiente adequado ao lançamento selecionado;

VI. Processos - Deve-se executar a aplicação como um ou mais processos que não armazenam estado, assim diminuindo o acoplamento e facilitando o escalamento;

VII. Vínculo de porta - Um microsserviço deve ser executado em um container e exposto por meio de portas;

VIII. Concorrência - Cada processo deve ser independente e executado separadamente, para se ter um melhor dimensionamento e possibilidade de executar mais ao mesmo tempo;

IX. Descartabilidade - Deve ser possível iniciar ou interromper a aplicação imediatamente sempre que necessário. Caso a aplicação pare de funcionar, deve ser capaz de iniciar novamente

sem perdas;

X. Paridade de desenvolvimento e produção - Deve-se manter os ambientes de desenvolvimento, teste e produção o mais semelhantes possível;

XI. Históricos (*Logs*) - Históricos devem ser tratatos como um fluxo de eventos;

XII. Processos administrativos - Tarefas de administração ou de gerenciamento devem ser executadas como processos únicos.

4.3 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, pois também terá que realizar o suporte aos usuários do produto. 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).

4.4 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[1] e o GitHub[2] 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, mas deixa a porta aberta para escolher uma

[1] Git: <https://git-scm.com/>
[2] GitHub: <https://github.com/>

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).

4.5 Descentralização dos dados

Para o gerenciamento de dados, há a possibilidade de compartilhar um banco de dados entre diferentes microsserviços, mas isso é visto como um anti-padrão. Uma aplicação com arquitetura de microsserviços tem melhor isolamento, segurança e disponibilidade quando cada serviço gerencia seu próprio banco de dados particular, inclusive tendo a possibilidade de ser instâncias diferentes da mesma tecnologia ou usar sistemas de banco de dados totalmente diferentes. 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 do serviço e do seu banco pode ser feita em conjunto. 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 atualizações. Geralmente a abordagem para se garantir consistência nas atualizações é pelo uso de transações quando atualizando múltiplos recursos. Contudo, o uso de transações resultada em um acoplamento temporal, o que é problemático quando se tem muitos serviços. Além disso, transações distribuidas são notoriamente difíceis de implementar, e portanto arquiteturas de microsserviços realizam coordenação sem transações entre serviços, com reconhecimento claro de que consistência pode ser apenas consistência eventual e que problemas serão lidados pela compensação de operações (FOWLER; LEWIS, 2014).

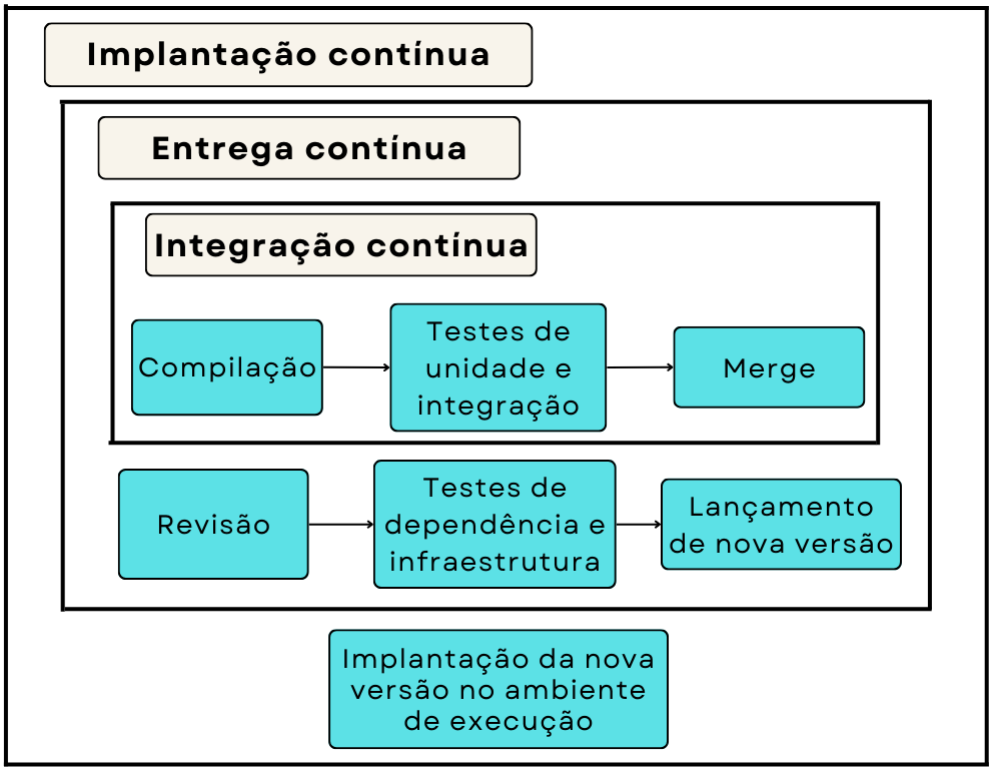
4.6 CI/CD

CI/CD é um método para entregar aplicações e mudanças nelas aos clientes com frequência. CI é um acrônimo de *Continuous Integration* (integração contínua), e diz respeito à automação de como o novo código feito pelo desenvolvedor chega no repositório principal e se transforma em um artefato lançável. 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 os artefatos se transformam em novas versões lançadas que são implantáveis num ambiente de execução a qualquer hora. Implantação contínua diz respeito a automação de como as novas versões lançadas são de fato implantadas no ambiente de execução. A Figura 2 ilustra as definições e limites da integração, entrega e implantação contínua. Apesar de determinar essas definições para o presente trabalho, foram observados conflitos na bibliografia revisada 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. No fim, não é importante se ater à semântica - apenas deve-se lembrar 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 desenvolvimento de aplicações (Red Hat Incorporated, 2022; Harness Incorporated, 2021; GitLab Incorporated, 2022a).

CI/CD auxilia profissionais de desenvolvimento e de operações a trabalhar mais eficiente e efetivamente, por diminuir tarefas manuais lentas e processos de aprovação antiquados. Além disso, faz com que os processos sejam previsíveis e repetíveis enquanto diminui o espaço para erro humano. Considerando que o ciclo de vida de um microsserviço deve ser tão automatizado quanto possível, CI/CD é uma prática de extrema importância (GitLab Incorporated, 2022a).

Figura 2 – CI/CD



Fonte: Autor

4.6.1 CI

CI (*Continuous Integration* - Integração contínua) é uma prática no desenvolvimento de *software* onde os desenvolvedores integram o código em um repositório compartilhado múltiplas vezes no dia. 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 compilação do código desenvolvido. A integração contínua por sí só requer apenas uma ferramenta de controle de versão (como Git[3]) para ser feita, mas existem práticas bem consolidadas na indústria do desenvolvimento de *software* que auxiliam e incrementam esse processo (FOWLER, 2006).

4.6.1.1 Manter apenas um repositório fonte

Deve-se ter um repositório, compartilhado por toda a equipe desenvolvedora, que contém tudo aquilo que é necessário para a construção do projeto, para que seja possível se ter um novo ambiente de desenvolvimento funcional rapidamente. 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).

4.6.1.2 Automatizar testes de novas compilações

As integrações de cada novo *commit* no repositório precisam ser compiladas e testadas, o que é uma tarefa árdua e demorada se feita manualmente. Desse modo, deve-se elaborar uma bateria de testes, e executá-la de forma automática a cada nova compilação, para assim detectar falhas rapidamente e aumentar a qualidade do código (GitLab Incorporated, 2022a; FOWLER, 2006).

4.6.1.3 Otimizar a compilação

Compilações lentas afetam negativamente a integração contínua, atrasando integrações e diminuindo a frequência do *feedback*. Além disso, todas as etapas da compilação devem ser simples de executar, idealmente por meio de um único comando (FOWLER, 2006).

4.6.1.4 Servidor de integração

Às vezes não é possível realizar todos os testes necessários na máquina local do desenvolvedor, seja por questões de tempo ou pelo projeto ser muito complexo para ser testado integralmente. Nesse caso, será preciso providenciar um local que será capaz de testar todo o projeto continuamente. Tal local é chamado de *CI Daemon* ou servidor de integração. Esse servidor será responsável por compilar o código a cada *commit*, realizar a bateria de testes e disponibilizar informações e relatórios sobre os passos executados, assim mantendo a qualidade do código e possibilitando a compilação contínua (FOWLER, 2006).

[3] Git: <https://git-scm.com/>

4.6.1.5 Consertar compilações quebradas imediatamente

O principal ponto de usar a integração contínua é que a equipe sempre estará trabalhando a partir de uma base estável. Se há uma base instável, é tarefa de toda a equipe resolver o problema o mais rápido possível, pois nenhum novo *commit* poderá ser feito em cima daquela base até que esteja estável e confiável novamente. Geralmente o melhor meio de resolver esse problema é reverter os *commits* problemáticos, trazendo o sistema de volta a um ponto funcional. Não deve-se tentar *debugar* no ramo principal do repositório (FOWLER, 2006).

4.6.2 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 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 pensada, pois para alguns negócios é preferível uma taxa de implantações mais baixa (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).

4.7 DevOps

Por um lado, as equipes de desenvolvimento tentam ser o mais eficientes possível, entregando tarefas completas sempre que possível. Por outro lado, a equipe de operações valoriza a estabilidade, considerando cada alteração como uma possível causa de problemas. Assim, tem-se diferenças importantes no foco do trabalho - uma equipe preza pela velocidade em novas funcionalidades e outra pela estabilidade. Com isso em mente, as equipes precisam aprender a trabalhar em conjunto para conseguir alcançar uma entrega contínua e ao mesmo tempo manter um *software* funcional no ambiente de execução.

DevOps é um movimento cultural que visa a integração e otimização do processo de aprendizagem e cooperação entre os integrantes de equipes relacionadas ao desenvolvimento de *software*. Não se trata de um cargo, mas sim 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. Para tanto, precisa-se de ferramentas para facilitar as automações e monitoramento, mas é importante lembrar que DevOps trata-se mais de um movimento cultural do que uma aparelhagem técnica. Práticas de DevOps, como integração e entrega contínua, são essenciais para o sucesso de aplicações com arquitetura de microsserviços pelos benefícios que traz ao time e ao desenvolvimento da aplicação (GitLab Incorporated, 2022b).

4.8 Organização de código

4.8.1 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 também pode causar problemas como: pior desempenho da IDE e de ações como a compilação 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).

4.8.2 Polirepo

Polirepo (também conhecido como multirepo) 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).

4.9 Implantação em containers

Containers configuram isolamentos lógicos em uma máquina. Eles são leves e altamente flexíveis, permitindo serem parados, alterados e reiniciados rapidamente. Depois de construido o microsserviço, recomenda-se lançá-lo dentro de um container para torná-los padronizados, leves, seguros e evitar interferências com outros microsserviços. Uma opção de mecanismo de gerenciamento de containers é o Docker[4] (Oracle Corporation, 2021).

4.10 Testes

Tradicionalmente uma compilação engloba tudo que é necessário para que um programa possa executar. Entretanto, só porque um programa executa não significa que ele fará o que se esperava. Para isso deve-se testar o programa, idealmente de forma automática e para toda funcionalidade, assim falhas e *bugs* podem ser descobertos antes de serem lançados. Em uma arquitetura de microsserviços, o processo de testes deve englobar várias estratégias diferentes de testes. Essas estratégias podem incluir testes funcionais, como um teste de unidade, ou testes não-funcionais, como um teste de desempenho. Usar múltiplas estratégias de testes aumenta as chances da aplicação operar com sucesso em ambientes e plataformas diferentes. De acordo com Waseem et al. (2021), testes de unidade e testes fim-a-fim são as estratégias de testes mais usadas no desenvolvimento de microsserviços (FOWLER, 2006; FAMILIAR, 2015).

Além de usar os métodos mais comuns de testes, deve-se também testar os microsserviços conforme passam pelo *pipeline* de implantação. Isso inclui: (FAMILIAR, 2015)

- Testes internos: Testar as funções internas do serviço, inclusive uso de acesso de dados, e caching;

- Teste de serviço: Testar a a implementação de serviço da API. Essa é uma implementação privada da API e seus modelos associados;

- Teste de protocolo: Testar o serviço no nível de protocolo, chamando a API sobre o determinado protocolo (geralmente HTTP);

- Teste de composição: Testar o serviço em cooperação com outros serviços no contexto de uma solução;

- Teste de escalabilidade/taxa de transferência: Testar a escalabilidade e elasticidade do microsserviço implantado;

- Teste de tolerância a falha: Testar a capacidade do microsserviço de recupera-se após uma falha;

- Teste de penetração: Trabalhar com uma empresa terceirizada de segurança de *software* para realizar testes de penetração no sistema.

[4] Docker: <https://www.docker.com/>

4.11 Comunicação

Na construção 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 aplicativos criados a partir de 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 é feito usando protocolos REST simples em vez de protocolos complexos como *Web Service Choreography* ou orquestração por uma ferramenta central (FOWLER; LEWIS, 2014).

Requisições HTTP em conjunto com uma API de recursos é o método mais usado para realizar comunicação síncrona na arquitetura de microsserviços. Uma requisição HTTP é feita por um cliente para um dado *host* em um servidor, com o propósito de acessar um recurso nesse servidor. 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, filas de mensagens são amplamente usadas. Quando um serviço precisa enviar informações a outro de modo assíncrono, ele envia uma mensagem para a fila de mensagens, e ela será armazenada até ser processada ou excluída. Cada mensagem é processada uma única vez, por um único consumidor. 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. gRPC[5] é uma ferramenta moderna com alto desempenho que tem ganhado grande popularidade em meio aos praticantes de microsserviços e é considerada um "projeto de incubação"pela Cloud Native Computing Foundation (Microsoft Corporation, 2022b).

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

[5] gRPC: <https://grpc.io/>

Esses padrões são discutidos na subseção 4.13.5.

4.12 Monitoramento

Em qualquer aplicação o monitoramento é importante para garantir um bom funcionamento. Entretanto, na arquitetura de microsserviços o monitoramento se torna indispensável e mais complexo.

4.12.1 Históricos

Um histórico, também conhecido como *logs*, descreve o que aconteceu em determinado sistema e provê informações sobre o estado e a saúde dele. Manter um histórico do microsserviço é uma forma simples e eficiente de se implementar monitoramento, e é fortemente indicado. Para manter históricos, pode-se desenvolver um serviço dedicado a isso, para ser reutilizado onde necessário. Outra possibilidade é utilizar bibliotecas, próprias ou desenvolvidas por terceiros.

Para facilitar a escrita, leitura e operação dos históricos, deve-se padronizar o formato deles em todos os microsserviços e diferenciar entradas de erros, de avisos e de informação. Além disso, eles devem ser agregados e organizados em um único lugar para que possam ser facilmente consultados.

4.12.2 Métricas

Métricas nos permitem saber o que está acontecendo em qualquer momento, e decidir que ações devem ser tomadas a partir disso. Por exemplo, escalar um serviço que recebe muitas requisições ou diminuir um que não. Métricas podem inclusive servir para questões de negócios e de *business inteligence*.

Enquanto históricos 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).

4.13 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 explorará diversas práticas no desenvolvimento de APIs.

4.13.1 Códigos de status de respostas HTTP

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

4.13.2 Troca de dados com JSON

Atualmente JSON é um dos formatos mais populares para troca de dados na web, pelo fato de ser facilmente lido tanto por humanos quanto por máquinas. 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)

4.13.3 Troca de dados com *Buffers* de Protocolo

Buffers de protocolo, ou *Protocol buffers* é 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).

4.13.4 Contratos de dados e versionamento

Uma API depende de contratos de dados, isso é, uma definição dos dados que serão recebidos e retornados. Esse contrato implica num compromisso de manter o serviço correspondente funcionando e inalterado. Entretanto, é possível desenvolver melhorias ou adicionar funcionalidades sem quebrar o contrato. Para tanto, deve ser feito um versionamento da API e deve ser mantida a transparência com os clientes que a usam. Por exemplo, sempre que algo for alterado é preciso atualizar a documentação.

Para fazer o versionamento, pode-se usar o processo de versionamento de *software* para representar os estados da API. Nesta técnica, usa-se três números para representar a versão, por exemplo "2.3.7". O primeiro representa a versão maior, o segundo, a versão menor, e o terceiro, o patch (pequena atualização para consertar ou melhorar algo) (SOFTWARE. . . , 2022).

Em uma API, para cumprir o contrato de dados, apenas modificações aditivas podem ser feitas, tais como novos endpoints ou novos campos opcionais em algum recurso. Quando isso é feito, a versão da API muda de 2.3.7 para 2.4.0 ou para 2.3.8 dependendo do tamanho da mudança. Quando é necessário realizar alterações que descumprem o contrato, deve-se alterar a versão maior da API, por exemplo passando de 2.3.7 para 3.0.0. Nesses casos, é importante manter a versão anterior funcionando e inalterada, criando uma nova rota para acessar a versão nova, para que clientes usando a versão anterior não apresentem falhas.

4.13.5 API Gateway

Numa arquitetura de microsserviços, os clientes geralmente consomem funcionalidades de mais de um microsserviço. Se esse consumo é feito do cliente diretamente para o microsserviço, o cliente precisa lidar com várias requisições aos *endpoints*. Quando os microsserviços mudam ou mais microsserviços surgem frequentemente, fica inviável tratar tantos endpoints por parte do cliente. Uma solução para isso é usar um *API Gateway*.

Um *API Gateway* é um padrão de projeto e funciona como uma porta única de entrada para as APIs de cada microsserviço, padronizando e controlando o acesso aos microsserviços e APIs. Esse gateway fica situado entre o cliente 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 API de cada microsserviço. Além disso, nele também podem ser implementadas camadas de segurança e de monitoramento.

Outra vantagem é que esse *API Gateway* pode agregar requisições, permitindo que o cliente 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 cliente, o *API Gateway* fica responsável por disparar as requisições correspondentes, agregar as respostas e as devolver ao cliente.

Entretando, também há desvantagens no uso desse padrão: (1) cria-se um alto acoplamento entre os microsserviços e o *API Gateway*, (2) surge um possível ponto massivo de falha nesse *API Gateway* e (3) se não escalado adequadamente, esse *API Gateway* pode se tornar um gargalo (Microsoft Corporation, 2022a).

4.13.6 Segurança em APIs

Autenticação

Incluir autenticação em uma API consiste em exigir uma prova de autorização do uso da API. A autenticação nas APIs é indispensável para aumentar a segurança, e existem formas diferentes de implementá-la.

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 (RapidAPI, 2022).

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, o que adiciona uma camada de segurança (RapidAPI, 2022).

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 (RapidAPI, 2022).

Compartilhar o mínimo possível

Compartilhar o mínimo possível é uma medida de segurança genérica que pode ser adotada em qualquer microsserviço. Especificamente nas APIs, deve-se retornar estritamente apenas os dados necessários para o cliente. Muitas ferramentas usadas para implementar APIs incluem por padrão informações como se fossem marcas d'água, mas que podem ser removidas, tal como headers "X-Powered-By", que vazam informações do servidor que podem auxiliar usuários mal-intencionados (RapidAPI, 2022).

4.13.7 Testes isolados

Testar uma API isoladamente serve para determinar se ela atende a parâmetros pré-definidos ou não. Tais parâmetros podem ser o cumprimento da funcionalidade, a confiabilidade, a latência, o desempenho, e a segurança. Quando um teste de API falha, deverá ser possível saber

precisamente onde o problema se encontra, assim aumentando a velocidade de desenvolvimento e a qualidade do produto.

4.13.8 Salvar a resposta no *cache*

Às vezes referido como *cachear*, salvar informações no *cache* melhora o tempo de busca da informação. 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 (RapidAPI, 2022).

4.13.9 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 (discutido na subseção 4.13.3) (RapidAPI, 2022).

4.13.10 Paginar e filtrar

A Paginação separa e categoriza resultados, enquanto a filtragem retorna apenas os resultados relevantes de acordo com os parâmetros da requisição. A paginação e filtragem de resultados reduz a complexidade da resposta e facilitam o uso da API (RapidAPI, 2022).

4.13.11 PATCH ou PUT

Quando é necessário modificar um recurso em uma API, usa-se os métodos HTTP PUT ou PATCH. Enquanto PUT atualiza o recurso inteiro, PATCH atualiza apenas uma parte específica do recurso, assim usando uma carga de dados menor. Portanto, quando possível deve-se usar PATCH em vez de PUT para modificar um recurso (RapidAPI, 2022).

5

Análise de viabilidade

Este capítulo analisa a viabilidade da aplicação a ser desenvolvida no TCC 2.

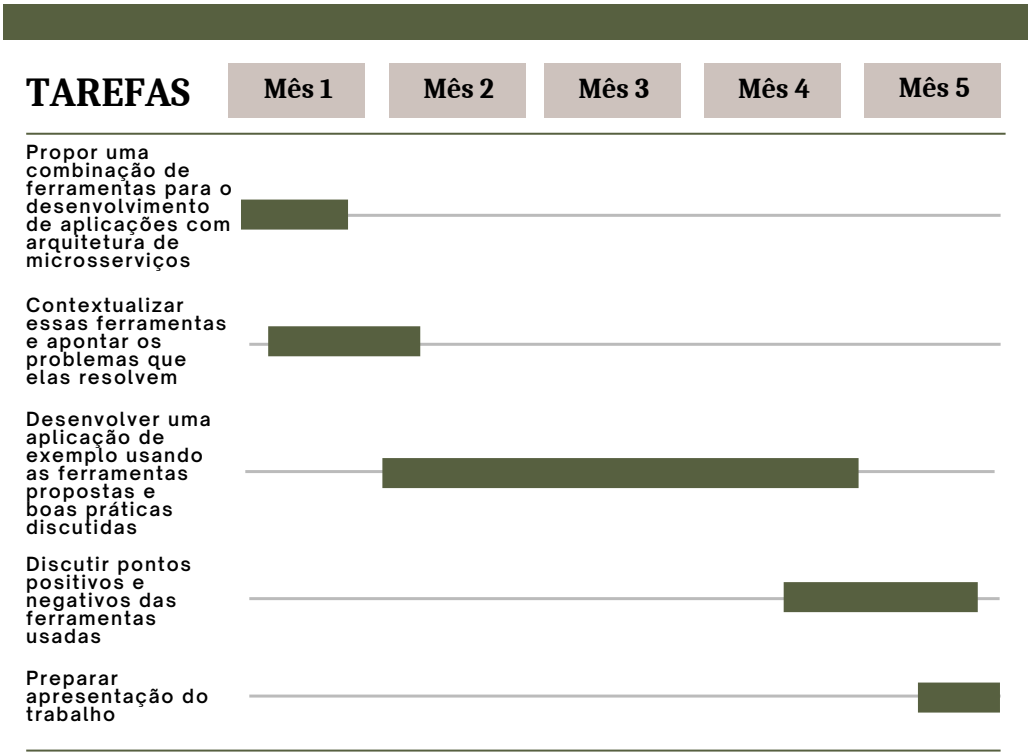
5.1 Ferramentas *open-source*

6

Plano de Continuidade

Esta foi a primeira etapa do trabalho. Na próxima etapa, planeja-se: (1) propor uma combinação de ferramentas para o desenvolvimento de aplicações com arquitetura de microsserviços, (2) contextualizar essas ferramentas e apontar os problemas que elas resolvem, (3) desenvolver uma aplicação de exemplo com arquitetura de microsserviços usando as ferramentas propostas e algumas práticas discutidas e (4) discutir pontos positivos e negativos das ferramentas usadas nessa aplicação de exemplo. O cronograma de atividades previstas para serem realizadas na disciplina de Trabalho de Conclusão de Curso 2 é apresentado na Figura 3.

Figura 3 – Cronograma de atividades que serão desenvolvidas.



Fonte: Autor

7

Conclusão

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.

A proposta de uma combinação de ferramentas que podem ser usadas no desenvolvimento de aplicações com arquitetura de microsserviços e suas contextualizações ficaram adiadas para serem exploradas na próxima etapa deste trabalho. Também ficaram para a próxima etapa o desenvolvimento de uma aplicação exemplar com arquitetura de microsserviços usando a combinação proposta e a discussão acerca dos pontos positivos e negativos observados no uso dessas ferramentas.

Referências

Amazon Web Services Incorporated. *Filas de mensagens*. 2022. Disponível em: <https://aws.amazon.com/pt/message-queue/>. Citado na página 31.

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 33.

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 na página 29.

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, 14, 15, 17, 18, 19 e 30.

FERNANDEZ, T.; ACKERSON, D. *Release Management for Microservices*. 2022. Disponível em: <https://semaphoreci.com/blog/release-management-microservices>. Citado na página 29.

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 27, 28 e 30.

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 28.

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 21, 22 e 23.

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 11 e 20.

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 20, 21 e 22.

FOWLER, M.; LEWIS, J. *Microservices*. 2014. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 06 Nov 2022. Citado 7 vezes nas páginas 17, 18, 19, 24, 25, 31 e 32.

GitLab Incorporated. *What is CI/CD?* 2022. GitLab Topics. Disponível em: <https://about.gitlab.com/topics/ci-cd/>. Citado 3 vezes nas páginas 26, 27 e 28.

GitLab Incorporated. *What is DevOps?* 2022. GitLab Topics. Disponível em: <https://about.gitlab.com/topics/devops/>. Citado na página 29.

Google LLC. *Overview | Protocol Buffers*. 2022. Disponível em: <https://developers.google.com/protocol-buffers/docs/overview>. Citado na página 33.

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 26.

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 22.

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 34.

Microsoft Corporation. *gRPC*. 2022. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/architecture/cloud-native/grpc>. Citado na página 31.

Middleware Lab. *What are Microservices? How Microservices architecture works*. Middleware Lab, 2021. Disponível em: <https://middleware.io/blog/microservices-architecture/>. Citado na página 9.

MONOLITHIC application. 2022. Page Version ID: 1118196758. Disponível em: <https://en.wikipedia.org/w/index.php?title=Monolithic_application&oldid=1118196758>. Citado na página 11.

NIELSEN, H. et al. *Hypertext Transfer Protocol – HTTP/1.1*. [S.l.], 1999. Disponível em: <https://datatracker.ietf.org/doc/rfc2616/>. Citado na página 33.

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 6 vezes nas páginas 9, 13, 18, 23, 25 e 30.

RapidAPI. Mídia Social. 2022. Disponível em: <https://twitter.com/Rapid_API>. Acesso em: 25 Oct 2022. Citado 2 vezes nas páginas 35 e 36.

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 2 vezes nas páginas 26 e 28.

RICHARDSON, C. *Microservices Pattern: Monolithic Architecture pattern*. 2018. Disponível em: <http://microservices.io/patterns/monolithic.html>. Citado 2 vezes nas páginas 12 e 13.

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 20.

RODRIGUES, R. de C. 2016. Disponível em: <https://www.fiap.com.br/2016/10/03/metodologia-12-fatores/>. Citado na página 23.

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 29.

SOFTWARE versioning. 2022. Page Version ID: 1116804459. Disponível em: <https://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=1116804459>. Citado na página 34.

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 21 e 22.

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, 15 e 16.

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 5 vezes nas páginas 6, 9, 16, 30 e 31.

WIGGINS, A. topic, *The Twelve-Factor App*. 2017. Disponível em: <https://12factor.net/>. Acesso em: 21 Oct 2022. Citado na página 23.

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 9 e 20.