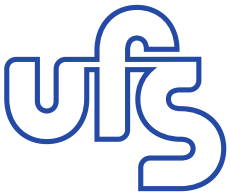


UNIVERSIDADE FEDERAL DE SERGIPE

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*. 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

Figura 1 – Aplicação com arquitetura de microsserviços 15
Figura 2 – Exemplo de *pipeline* de CI/CD 31
Figura 3 – Ciclo DevOps 34
Figura 4 – Cronograma de atividades que serão desenvolvidas. 54

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 11

1.1 Objetivos 11

1.1.1 Objetivo geral 11

1.1.2 Objetivos específicos 12

1.2 Metodologia 12

2 Fundamentação teórica 13

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

2.1 As aplicações monolíticas 13

2.1.1 Benefícios 13

2.1.2 Limitações 14

2.1.2.1 Crescimento, velocidade de desenvolvimento, e manutenção . 14

2.1.2.2 Escalabilidade 14

2.1.2.3 Reutilização 14

2.1.2.4 Implantação 14

2.1.2.5 Confiabilidade e resiliência 15

2.1.2.6 Flexibilidade de tecnologias 15

2.1.2.7 Divisão de times 15

2.2 Os microsserviços 15

2.2.1 Tipos de microsserviços 16

2.2.1.1 Serviço de dados (*data service*) 16

2.2.1.2 Serviço de negócio (*business service*) 16

2.2.1.3 Serviço de tradução (*translation service*) 17

2.2.1.4 Serviço de ponta (*edge service*) 17

2.3 Trabalhos relacionados 17

2.3.1 Microservices, IoT and Azure - capítulo 2: What is a microservice, por Familiar (2015) 17

2.3.2 A Systematic Mapping Study on Microservices Architecture in DevOps, por Waseem, Liang e Shahin (2020) 17

2.3.3 Design, monitoring, and testing of microservices systems: The practitioners' perspective, por Waseem et al. (2021) 18

2.3.4 Building Microservices: Designing Fine-Grained Systems, por Newman (2015) 18

3 Características 19

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

3.1 Sistema distribuído 19
3.2 Flexibilidade na escolha de ferramentas 20
3.3 Alta velocidade de desenvolvimento 20
3.4 Componentização 20
3.5 Portabilidade 21
3.6 Versionável e substituível 21
3.7 Menor erosão de *software* 21
3.8 Complexidade e desafios 22
3.8.1 Teorema CAP 22

4 Práticas 24

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

4.1 Começar pela arquitetura monolítica 24
4.1.1 Requisito dos microsserviços: Provisionamento rápido 25
4.1.2 Requisito dos microsserviços: Monitoramento básico 25
4.1.3 Requisito dos microsserviços: Implantação rápida 26
4.2 *Domain-Driven Design* (DDD) 26
4.3 A metodologia 12-fatores 26
4.4 Produtos, não projetos 28
4.5 Desenvolver e compartilhar ferramentas 28
4.6 Descentralização dos dados 29
4.7 CI/CD 29
4.7.1 CI 30
4.7.1.1 Uso de apenas um repositório fonte 30
4.7.1.2 *Build* rápido 31
4.7.1.3 Automação de testes em novas integrações 32
4.7.1.4 Uso de servidor de integração 32
4.7.1.5 Estabilidade do ramo principal 32
4.7.2 CD 32
4.7.2.1 Anti-padrão: gerenciamento manual de ambientes 33
4.7.2.2 Anti-padrão: implantação manual 33
4.7.2.3 Anti-padrão: baixa frequência de implantação 33
4.7.2.4 Estratégias de implantação 33
4.8 DevOps 33
4.9 Organização de código 35
4.9.1 Monorepo 35

4.9.2 Polirepo . . . 35
4.10 Externalização/parametrização de configurações . . . 35
4.11 Implantação em containers . . . 36
4.12 Testes . . . 36
4.13 Comunicação . . . 37
4.13.1 Descoberta de serviços (*Service Discovery*) . . . 38
4.14 APIs . . . 38
4.14.1 Códigos de status de respostas HTTP . . . 38
4.14.2 Formatos de dados . . . 38
4.14.3 Contratos de dados e versionamento . . . 39
4.14.4 API Gateway . . . 39
4.14.5 Segurança em APIs . . . 40
4.14.6 Testes isolados . . . 41
4.14.7 Salvar a resposta no *cache* . . . 41
4.14.8 Comprimir os dados . . . 41
4.14.9 Paginar e filtrar . . . 41
4.14.10 PATCH ou PUT . . . 42
4.15 Observabilidade e Monitoramento . . . 42
4.15.1 *Logs* (registros) . . . 42
4.15.2 Métricas . . . 43
4.15.3 Rastreamento . . . 43

5 Análise de viabilidade . . . 44
Este capítulo analisa a viabilidade da aplicação a ser desenvolvida no TCC 2.
5.1 Ferramentas *open-source* . . . 44

6 Desenvolvimento da aplicação . . . 45
Este capítulo aprensenta ferramentas que podem ser usadas no desenvolvimento de aplicações com arquitetura de microsserviços
6.1 Resource Management Problems . . . 45
6.2 Linguagens de programação . . . 46
6.3 Frameworks . . . 46
6.4 Servidor Web . . . 46
6.5 Bancos de dados . . . 46
6.5.1 Tipos de bancos de dados . . . 46
6.5.2 Escalamento . . . 46
6.5.2.1 CQRS . . . 46
6.5.2.2 Replicação . . . 47
6.5.3 Relacionais . . . 47
6.5.4 Not-only SQL . . . 47

6.5.5 Caching . . . 47
6.6 Integração contínua e Entrega Contínua (CI/CD) . . . 47
6.6.1 Sistema de controle de versão . . . 48
6.6.2 Gerenciamento de repositórios . . . 48
6.6.3 Automação de testes . . . 48
6.6.4 Servidor de integração . . . 48
6.7 Testes . . . 49
6.7.1 Testes Manuais . . . 49
6.7.2 Testes unitários . . . 49
6.7.3 Testes de integração . . . 49
6.7.4 Testes de API . . . 49
6.8 Comunicação . . . 49
6.8.1 RPC . . . 49
6.8.2 Mensageria . . . 49
6.8.3 Streaming de Dados . . . 49
6.8.4 APIs . . . 50
6.8.4.1 API Gateway . . . 50
6.8.4.2 Tipos de API . . . 50
6.8.4.2.1 GraphQL . . . 50
6.8.4.2.2 REST . . . 50
6.8.4.3 Documentação . . . 50
6.9 Provisionamento . . . 50
6.9.1 de máquinas virtuais . . . 50
6.9.2 de containers . . . 50
6.9.3 na nuvem . . . 50
6.9.4 automático . . . 50
6.10 Orquestração de Containers . . . 51
6.10.1 Kubernetes . . . 51
6.11 Observabilidade e Monitoramento . . . 51
6.11.1 Métricas: Prometheus e Grafana . . . 51
6.11.2 Logging: Grafana Loki . . . 51
6.11.3 Logging: Graylog . . . 52
6.12 Conjunto de ferramentas . . . 52
6.13 Framework arquitetural . . . 52
6.14 Aplicações *serverless* (sem servidor) . . . 52

7 Plano de Continuidade . . . 53

8 Conclusão . . . 55

Referências **56**

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 serão 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 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 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

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

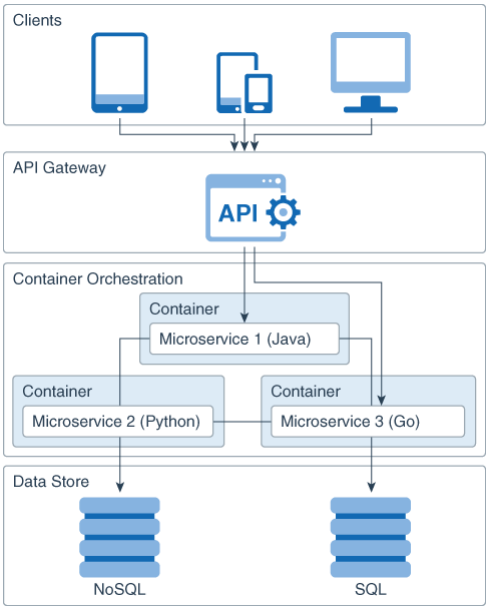
As escolhas de tecnologias para novas funcionalidades são mais limitadas - um projeto tende a usar apenas um certo grupo de tecnologias porque realizar *upgrades* ou mudanças de tecnologias é uma tarefa complexa e pode causar problemas de compatibilidade (RICHARDSON, 2018).

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, 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* (subseção 4.14.4) 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 (2015)

Newman (2015) 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 (2015) não discute ferramentas para a implementação dos conceitos e práticas discutidas.

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, 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 4.13 e subseção 4.14.3 (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 e serviços de apoio. 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

Ter portabilidade em uma aplicação significa permitir que ela seja executada facilmente em diferentes plataformas. Apesar de lista-la como uma característica dos microsserviços, a portabilidade não os é inerente - ela deve ser alcançada por meio de automação e externalização de configurações, para assim os microsserviços serem implantados de forma fácil e eficiente. (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, 2015)

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 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 precisaria-se 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 *Domain-Driven Design* (DDD)

TODO:

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; Um contrato claro e limpo com o sistema operacional, oferecendo maior portabilidade de ambiente de execução; Adequação à implantação em plataformas na núvem, 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. (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. 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* (discutida na subseção 4.9.1) constitui uma violação dessa diretiva. [Anotação: CUMPRIDO POR GIT E Branches no repositório]

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 núvem 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, discutida na seção 4.7;

XI. Registros (*Logs*) - Registros, discutidos na subseção 4.15.1 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. 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, 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 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. Contudo, o uso de transações resulta em um acoplamento temporal, o que é problemático quando se tem muitos serviços. Além disso, transações distribuidas são notoriamente difíceis de implementar, portanto arquiteturas de microsserviços 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. 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 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)

Além disso, 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 da aplicação e dos times que a desenvolvem. A Figura 2 ilustra uma possível estrutura de um *pipeline* de CI/CD. 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, por diminuir tarefas manuais lentas e processos de aprovação antiquados. Ademais, faz com que os processos sejam previsíveis e repetíveis enquanto diminui o espaço para erros humanos. Considerando que o ciclo de vida de um microsserviço deve ser tão automatizado quanto possível, CI/CD é uma prática de extrema importância no desenvolvimento desses (Red Hat Incorporated, 2022; GitLab Incorporated, 2022a).

4.7.1 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 (como Git[1]) para ser feita, mas existem práticas bem consolidadas na indústria do desenvolvimento de *software* que incrementam esse processo, discutidas a seguir. (FOWLER, 2006).

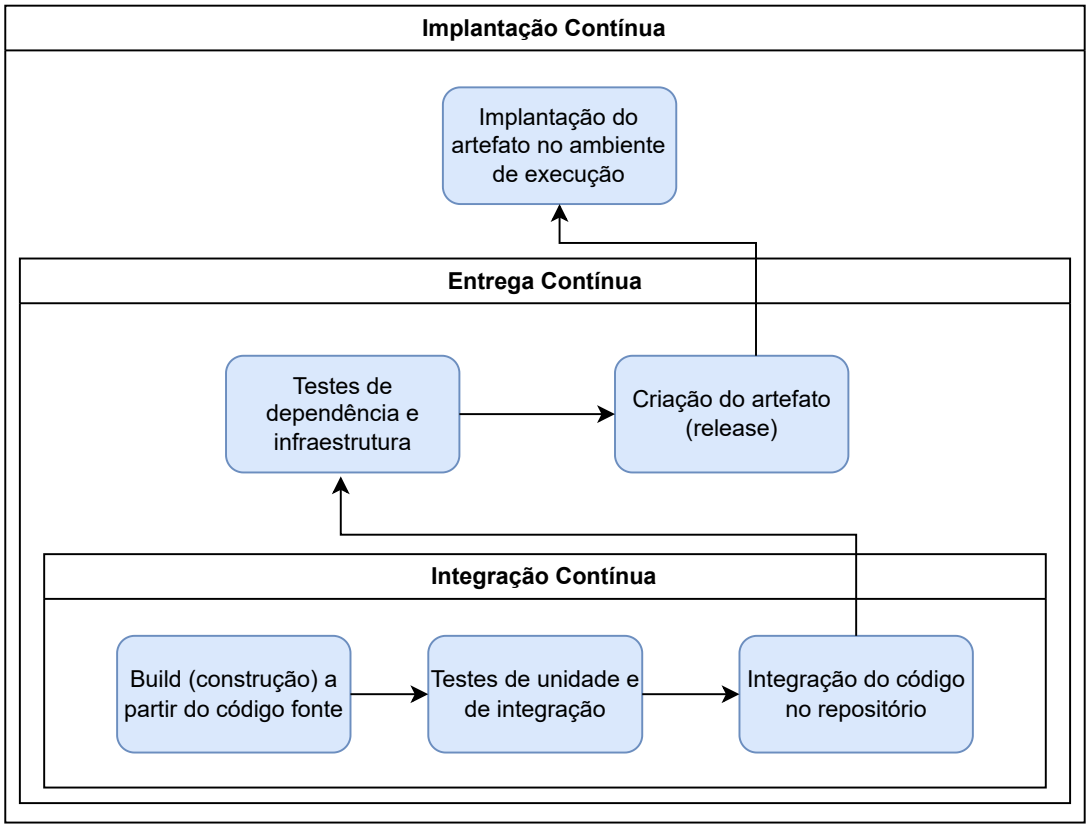
Aplicação dessa prática no projeto: PIPELINE NO GITHUB ACTIONS

4.7.1.1 Uso de apenas um repositório fonte

Deve-se ter apenas um repositório por projeto, que é compartilhado por toda a equipe desenvolvedora e que contém tudo aquilo que é necessário para uma instalação rápida e funcional

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

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



Fonte: Autor

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).

Aplicação dessa prática no projeto: DOCKER e DOCKER-COMPOSE

4.7.1.2 *Build* rápido

Um *build*, ou construção, da aplicação trata-se da transformação dos arquivos no repositório em um programa iniciável. Construções lentas afetam negativamente a integração contínua, atrasando integrações e diminuindo a frequência do *feedback*. Além disso, todas as etapas da construção 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 Uso de 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, garantindo que apenas o código que cumpra tais condições seja integrado, assim 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 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).

4.7.2.1 Anti-padrão: 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)

4.7.2.2 Anti-padrão: 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)

4.7.2.3 Anti-padrão: 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)

4.7.2.4 Estratégias de implantação

TODO: Elaborar sobre Blue-green deployment, feature toggles, canary release.

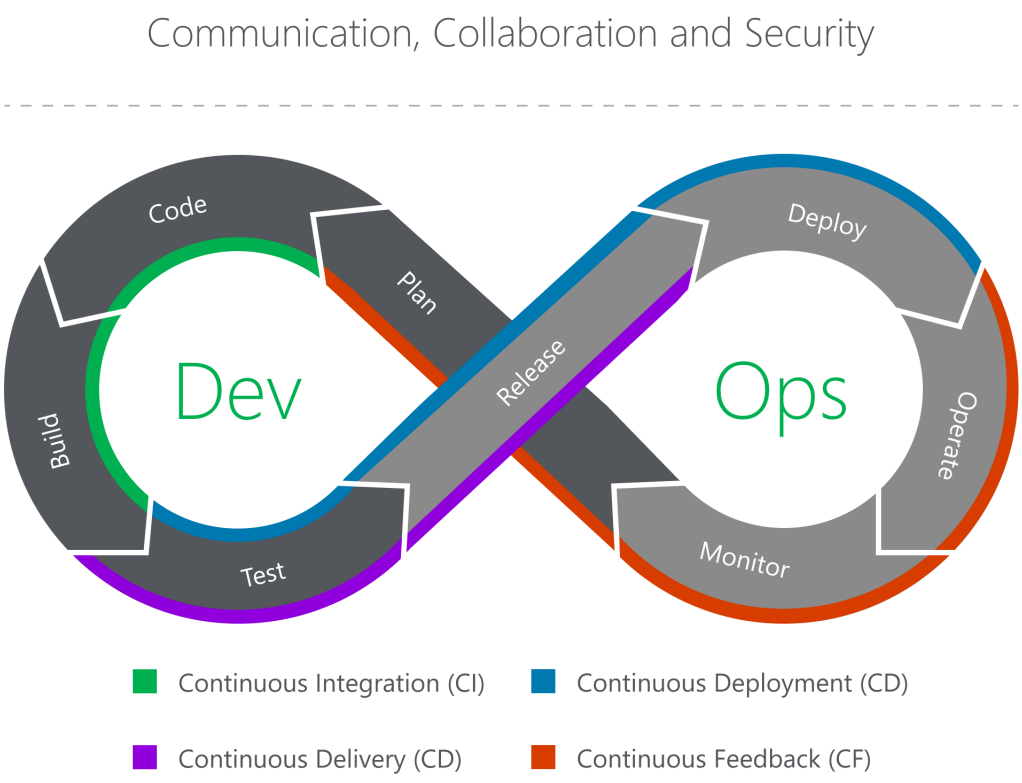
4.8 DevOps

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 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 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).

4.10 Externalização/parametrização de configurações

TODO: Falar sobre externalização da configuração do ambiente. Traz mais portabilidade ao código. Fazer paralelo com o 12-factor app fator 3

4.11 Implantação em containers

TODO: Elaborar melhor sobre orquestração de containers.

Containers configuram isolamentos lógicos em uma máquina. Eles são leves e altamente flexíveis, permitindo serem parados, alterados e reiniciados rapidamente. Depois de desenvolvido o microsserviço, recomenda-se implantá-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[2] (Oracle Corporation, 2021).

4.12 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 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;

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

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

4.13 Comunicação

TODO: Elaborar sobre o FastCGI. (geralmente usado para servidores de aplicação de linguagem interpretada como uma alternativa ao http)

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 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[3] 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. Esses padrões são discutidos na subseção 4.14.4.

4.13.1 Descoberta de serviços (*Service Discovery*)

TODO: Elaborar sobre descoberta de serviços

A descoberta de serviço é uma parte muito importante da comunicação entre microsserviços (NEWMAN, 2015; CUSIMANO, 2022).

4.14 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.

4.14.1 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 (NIELSEN et al., 1999), e facilita o entendimento do cliente sobre o que aconteceu com a requisição à API.

4.14.2 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)

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

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).

4.14.3 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.

4.14.4 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).

4.14.5 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).

4.14.6 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.

4.14.7 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).

4.14.8 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 4.14.2) (RapidAPI, 2022).

4.14.9 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).

4.14.10 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).

4.15 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)

4.15.1 *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 containers e com escalamento horizontal. (BEYER et al., 2018)

4.15.2 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).

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

5

Análise de viabilidade

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

5.1 Ferramentas *open-source*

6

Desenvolvimento da aplicação

Este capítulo aprensenta ferramentas que podem ser usadas 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 mesmo para pessoas experientes, mas 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.

Ferramentas de código aberto ou com opções gratuitas estão seguidas de um asterisco (*). Expor uma boa quantidade de ferramentas tal que supra as necessidades mais frequentes de uma arquitetura de microsserviços, com espaço para alternativas.

»> Manter em mente: Focar em soluções de infraestrutura, pois As soluções para os requisitos/problemas de **negócio** podem ser iguais tanto na arquitetura monolítica quanto na de microsserviços, ou >seja, podem ser satisfeitos/solucionados nas duas arquiteturas.

6.1 Resource Management Problems

This category provides the mapping of problems and solutions for different types of resources required to implement MSA in DevOps. Study (S01) recommends the virtualization of applications, infrastructures, and platforms resources as a solution for addressing resource management problems. Study (S09) suggests using containers and VMs for microservices in DevOps to get the desired level of efficiency in resource utilization. Study (S03) proposes the HARNESS approach (i.e., a DevOps based approach) that provides a cloud-based platform for bringing together commodity and specialized resources (e.g., skilled people). Study (S19)

introduces an MSA based SONATA NFV platform with DevOps to address resource management problems by providing a set of tools (e.g., GitHub, Jenkins, Docker). The SONATA NFV platform can also create the CI/CD pipeline to automate steps in software delivery process. Study (S09) argued that dedicated access to the host's hardware can be increased either by giving extra privileges to microservices or by enhancing the capability of containers to access the host resources.

asdfasdf. (STEEN; TANENBAUM, 2023).

Steen e Tanenbaum (2023) afirma que...

6.2 Linguagens de programação

Java, JavaScript, Python, Elixir. [1]

6.3 Frameworks

TODO: Para microsserviços, ...

6.4 Servidor Web

TODO: nginx, apache. Importante distinguir servidores web de servidores de aplicação...

6.5 Bancos de dados

TODO:

6.5.1 Tipos de bancos de dados

6.5.2 Escalamento

6.5.2.1 CQRS

Command Query Responsibility Segregation (CQRS) - Segregação de Responsabilidade de Consulta de Comando.

Trata-se de usar um modelo para escrita e outro para leitura. Adicionar complexidade.

"CQRS stands for Command Query Responsibility Segregation, which is a software design pattern that separates the operations for reading data (queries) from those for writing data

[1] Uma linguagem de programação não pode, por sí só, ser caracterizada como código aberto ou não, mas uma implementação de uma linguagem pode. Para o escopo deste trabalho, todas as implementações mais comuns das linguages de programação citadas são válidas.

(commands). This separation allows each operation to be optimized independently, improving performance and scalability in complex applications."

- ler: https://martinfowler.com/bliki/CQRS.html

6.5.2.2 Replicação

- ler: https://atlan.com/what-is/database-replication/

- ler: https://release.com/blog/syncing-databases-how-to-do-it-and-best-practices

- Aplicação: https://stackoverflow.com/questions/58399450/keeping-databases-in-sync-after-write-update-across-regions-zones

- ver se tem algum serviço assim na AWS

6.5.3 Relacionais

MySQL*, PostgreSQL*, MariaDB*

6.5.4 Not-only SQL

MongoDB*, Couchbase*, Redis*, Amazon DynamoDB

6.5.5 Caching

Memcached (*), Redis

6.6 Integração contínua e Entrega Contínua (CI/CD)

GitHub Actions ou GitLab...

re-ler: https://martinfowler.com/bliki/ContinuousDelivery.html

Gitlab: <https://about.gitlab.com/topics/ci-cd/> e <https://docs.gitlab.com/ci/> e <https://gitlab.com/microsservicos1/TesteGitLab/-/learn_gitlab>

- Reler: https://martinfowler.com/articles/continuousIntegration.html

Automatizar testes e builds para cada commit em qualquer branch permite feedback rápido, garantindo que o código esteja em um estado consistente em todas as branches, promovendo uma prática de CI eficiente. (Fonte: curso do vinicius)

Many software tools are available to support implementing CI/CD practices. These CI/CD tools range from repository management such as Github and Bitbucket, Jenkins for build automation, and Selenium and Katalon Studio for test automation.

What are some common CI/CD tools? CI/CD tools can help a team automate their development, deployment, and testing. Some tools specifically handle the integration (CI) side, some manage development and deployment (CD), while others specialize in continuous testing or related functions. One of the best known open source tools for CI/CD is the automation server Jenkins. Jenkins is designed to handle anything from a simple CI server to a complete CD hub. Deploying Jenkins on Red Hat OpenShift

Tekton Pipelines is a CI/CD framework for Kubernetes platforms that provides a standard cloud-native CI/CD experience with containers. Deploying Jenkins on Red Hat OpenShift

Beyond Jenkins and Tekton Pipelines, other open source CI/CD tools you may wish to investigate include: Spinnaker, a CD platform built for multicloud environments. GoCD, a CI/CD server with an emphasis on modeling and visualization. Concourse, "an open-source continuous thing-doer."Screwdriver, a build platform designed for CD.

Teams may also want to consider managed CI/CD tools, which are available from a variety of vendors. The major public cloud providers all offer CI/CD solutions, along with GitLab, CircleCI, Travis CI, Atlassian Bamboo, and many others. Additionally, any tool that's foundational to DevOps is likely to be part of a CI/CD process. Tools for configuration automation (such as Ansible, Chef, and Puppet), container runtimes (such as Docker, rkt, and cri-o), and container orchestration (Kubernetes) aren't strictly CI/CD tools, but they'll show up in many CI/CD workflows. (Red Hat Incorporated, 2022)

6.6.1 Sistema de controle de versão

Existem sistemas de controle de versão distribuidos (DVCS) ou centralizados (CVCS), porém atualmente os distribuitos são o padrão no desenvolvimento de *software*, e pouco se ouve falar dos centralizados. O **Git** é de longe o mais famoso entre todos os sistemas de controle de versão, e é uma solução elegante e completa para o controle de versão no desenvolvimento de *software*.

6.6.2 Gerenciamento de repositórios

GitHub, Bitbucket, GitLab, amazon S3

6.6.3 Automação de testes

Selenium para navegadores

6.6.4 Servidor de integração

Como mencionado na subseção 4.7.1.4... Normalmente a responsabilidade de executar a pipeline de integração é delegada para um servidor de integração, em vez de confiar no

desenvolvedor para executá-la manual e localmente sempre antes de fazer algum commit. Assim tem-se a garantia de que a pipeline será de fato executada e o desenvolvedor não precisa ficar aguardando ela ser executada na máquina dele.

Mostrar o exemplo do pipeline CI rodando no GitHub Actions, com proteções de branch e requerimento de revisão de código

<https://about.gitlab.com/topics/ci-cd/continuous-integration-server/>

<https://www.ibm.com/docs/en/integration-bus/10.1?topic=environment-integration-servers>

<https://www.ibm.com/docs/en/app-connect/11.0.0?topic=overview-integration-servers-integration-

CruiseControl*, Jenkins, GitHub Actions, GitLab CI, etc

6.7 Testes

Existem diversos tipos de teste, Teste Manual vs Automatizado vs Contínuo. Mas não é o foco do trabalho

6.7.1 Testes Manuais

6.7.2 Testes unitários

6.7.3 Testes de integração

6.7.4 Testes de API

Postman*, Hoppscotch*, Thunder Client (VSCode) programar os testes em uma linguagem de programação cURL postman solução integrada ao VSCode - thunderclient, rapidAPI

6.8 Comunicação

6.8.1 RPC

gRPC (<https://grpc.io/>)

6.8.2 Mensageria

RabbitMQ*, Azure Service Bus, Amazon Simple Queue Service, Google Cloud Pub/Sub.

6.8.3 Streaming de Dados

Apache Kafka*

6.8.4 APIs

6.8.4.1 API Gateway

nginx (free and paid version), Tyk*, Amazon API Gateway

6.8.4.2 Tipos de API

6.8.4.2.1 GraphQL

GraphQL

6.8.4.2.2 REST

REST is not a protocol or a standard, it is an architectural style. During the development phase, API developers can implement REST in a variety of ways.

Like the other architectural styles, REST also has its guiding principles and constraints. These principles must be satisfied if a service interface has to be referred to as RESTful.

A Web API (or Web Service) conforming to the REST architectural style is called a REST API (or RESTful API).

6 princípios: https://www.ibm.com/think/topics/graphql-vs-rest-api

6.8.4.3 Documentação

Swagger (free and paid version)

6.9 Provisionamento

6.9.1 de máquinas virtuais

VirtualBox*

6.9.2 de containers

Docker*, LXC (Linux containers)

6.9.3 na nuvem

AWS EC2

6.9.4 automático

AWS Launch Templates

6.10 Orquestração de Containers

Docker Swarm*, Kubernetes*, Conductor*, AWS EKS, Azure Kubernetes Service (AKS)

6.10.1 Kubernetes

6.11 Observabilidade e Monitoramento

Modern tracing tools, such as Jaeger, Zipkin, and OpenTelemetry, help organizations implement distributed tracing efficiently. These tools collect, process, and visualize traces, often integrating with dashboards and monitoring platforms like Grafana or Prometheus for deeper insights. OpenTelemetry, in particular, is gaining traction as a vendor-neutral, open-source standard that unifies telemetry data across logs, metrics, and traces, making it easier to achieve full observability. As distributed systems continue to grow in complexity, tracing has become an essential practice for ensuring system reliability, minimizing downtime, and optimizing application performance.

6.11.1 Métricas: Prometheus e Grafana

Prometheus is an open-source monitoring and alerting tool designed for collecting and analyzing time-series data (metrics). It is widely used for monitoring infrastructure, applications, and services, especially in cloud-native and Kubernetes environments. (Prometheus, 2025)

6.11.2 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)

6.11.3 Logging: Graylog

Graylog é um sistema de gerenciamento de *logs* mais simples do que o Loki.

Para manter registros, pode-se desenvolver um serviço dedicado a isso ou utilizar bibliotecas, para poder ser reutilizado onde necessário.

Logstash*, Sentry, Middleware, Elastic Stack, Graylog*

6.12 Conjunto de ferramentas

Seneca*, Google Cloud Functions,

6.13 Framework arquitetural

goa*, Kong*

6.14 Aplicações *serverless* (sem servidor)

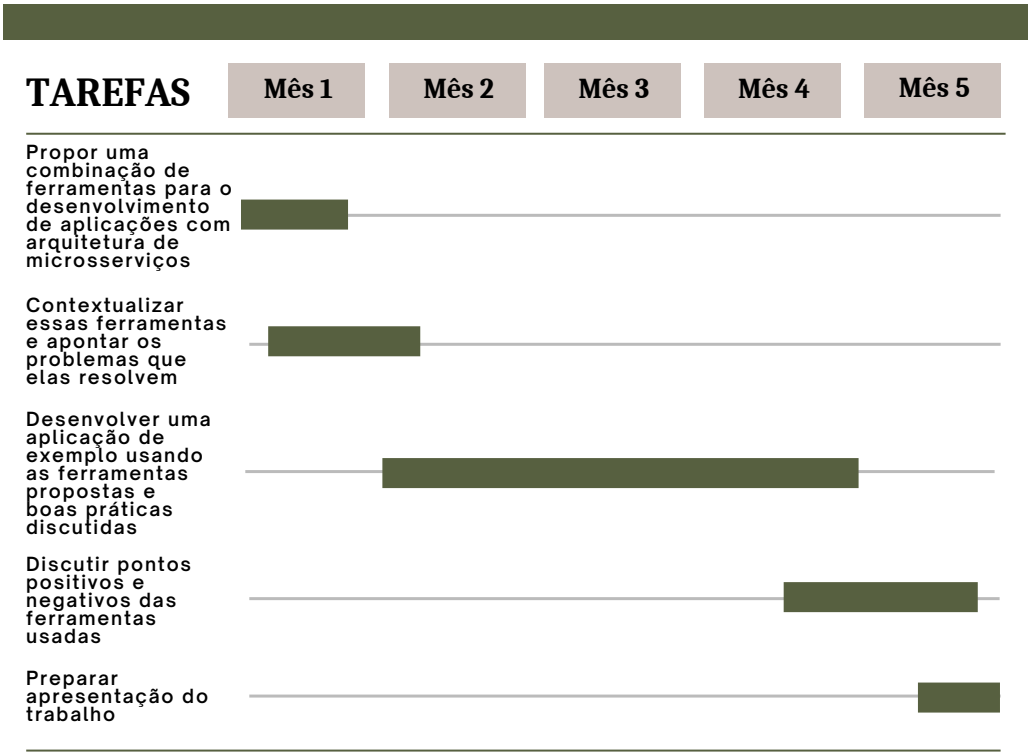
Claudia, Apache Openwhisk, Serverless, Kubeless, IronFunctions, AWS Lambda, OpenFaaS, Microsoft Azure Functions.

7

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 4.

Figura 4 – Cronograma de atividades que serão desenvolvidas.



Fonte: Autor

8
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 37.

BEYER, B. et al. Site reliability engineering: How google runs production systems. In: ____. First edition. O'Reilly Media, 2016. cap. 6. ISBN 9781491929124. Disponível em: <https://sre.google/sre-book/monitoring-distributed-systems/>. Citado na página 42.

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 43.

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 38.

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 35.

CUSIMANO, S. *Service Discovery in Microservices*. 2022. Disponível em: <https://www.baeldung.com/cs/service-discovery-microservices>. Citado na página 38.

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 42.

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, 16, 17, 19, 20, 21 e 36.

FERNANDEZ, T.; ACKERSON, D. *Release Management for Microservices*. 2022. Disponível em: <https://semaphoreci.com/blog/release-management-microservices>. Citado na página 35.

FOWLER, M. *Continuous Integration*. 2006. Blog do Martin Fowler. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 12 Nov 2022. Citado 4 vezes nas páginas 30, 31, 32 e 36.

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 33.

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 24, 25 e 26.

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 13 e 22.

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 22, 24 e 25.

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 19, 20, 21, 28, 29, 37 e 43.

GitLab Incorporated. *What is CI/CD?* 2022. GitLab Topics. Disponível em: <https://about.gitlab.com/topics/ci-cd/>. Citado 4 vezes nas páginas 30, 31, 32 e 33.

GitLab Incorporated. *What is DevOps?* 2022. GitLab Topics. Disponível em: <https://about.gitlab.com/topics/devops/>. Citado na página 34.

Google LLC. *Overview | Protocol Buffers*. 2022. Disponível em: <https://developers.google.com/protocol-buffers/docs/overview>. Citado na página 39.

Grafana Labs. *Loki overview | Grafana Loki documentation*. 2025. Disponível em: <https://grafana.com/docs/loki/latest/get-started/overview/>. Citado na página 51.

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 30.

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 2 vezes nas páginas 32 e 33.

IBM. *CAP Theorem*. 2022. Disponível em: <https://www.ibm.com/think/topics/cap-theorem>. Citado 2 vezes nas páginas 22 e 23.

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 25.

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 40.

Microsoft Corporation. *gRPC*. 2022. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/architecture/cloud-native/grpc>. Citado na página 38.

Middleware Lab. *What are Microservices? How Microservices architecture works*. Middleware Lab, 2021. Disponível em: <https://middleware.io/blog/microservices-architecture/>. Citado na página 11.

MONOLITHIC application. 2022. Page Version ID: 1118196758. Disponível em: <https://en.wikipedia.org/w/index.php?title=Monolithic_application&oldid=1118196758>. Citado na página 13.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. First edition. Sebastopol, Ca: O'reilly Media, 2015. ISBN 9781491950357. Citado 6 vezes nas páginas 6, 16, 18, 22, 23 e 38.

NIELSEN, H. et al. *Hypertext Transfer Protocol – HTTP/1.1*. [S.l.], 1999. Disponível em: <https://datatracker.ietf.org/doc/rfc2616/>. Citado na página 38.

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 11, 15, 20, 29 e 36.

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 34.

Prometheus. *Overview | What is Prometheus*. 2025. Disponível em: <https://prometheus.io/docs/introduction/overview/>. Citado na página 51.

RapidAPI. Mídia Social. 2022. Disponível em: <https://twitter.com/Rapid_API>. Acesso em: 25 Oct 2022. Citado 3 vezes nas páginas 40, 41 e 42.

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 30, 33 e 48.

RICHARDSON, C. *Microservices Pattern: Monolithic Architecture pattern*. 2018. Disponível em: <http://microservices.io/patterns/monolithic.html>. Citado 2 vezes nas páginas 14 e 15.

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 22.

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

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 35.

SOFTWARE versioning. 2022. Page Version ID: 1116804459. Disponível em: <https://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=1116804459>. Citado na página 39.

STEEN, M. van; TANENBAUM, A. *Distributed Systems*. Fourth edition, version 4.01 (january 2023). Erscheinungsort nicht ermittelbar: Maarten van Steen, 2023. ISBN 978-90-815406-4-3. Disponível em: <https://www.distributed-systems.net/index.php/books/ds4/>. Citado na página 46.

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 24 e 25.

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, 17 e 18.

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, 11, 18, 36 e 38.

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 11 e 22.