Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Problemas e dificuldades na migração de sistemas monolíticos para microsserviços

Thiago Cunha Ferreira

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisora: Profª. Drª. Ana Cristina Vieira de Melo

São Paulo
2021

Aos meus pais, que sempre estiveram ao meu lado, compartilhando felicidades e dando apoio em momentos difíceis.

Agradecimentos

No one knows what the future holds. That's precisely why,
just as this reunion demonstrates, the possibilities are infinite.
— Okabe Rintarō

Aos amigos que fiz durante a faculdade, sem os quais esses anos teriam sido bem mais difíceis, e à Profª Ana Cristina, por ter me auxiliado na escrita deste trabalho.

Resumo

Thiago Cunha Ferreira. **Problemas e dificuldades na migração de sistemas monolíticos para microsserviços**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Nos últimos anos, microsserviços vêm se tornando uma forma bem popular de estruturar aplicações na indústria, com diversas vantagens e oportunidades identificadas por grandes empresas como Netflix, Uber e Amazon. Dessa forma, um grande número de organizações vem se interessando em realizar a transição de seus sistemas monolíticos, muitas vezes já defasado, para esse modelo distribuído. Contudo, tanto a adoção da arquitetura de microsserviços quanto o próprio processo de migração possuem uma gama de problemas e empecilhos que dificultariam a sua aplicação na prática. Este trabalho tem o objetivo de analisar e contextualizar algumas dessas complicações, identificadas em diversos relatos da indústria e da academia. Após o desenvolvimento desses pontos, fica evidente que prosseguir com a migração de aplicações monolíticas para microsserviços nem sempre é caminho mais adequado a todos os contextos. Além disso, mesmo quando esse não for o caso, ainda é importante reconhecer essas dificuldades e considerar soluções viáveis para garantir uma melhor expressividade dos benefícios dos microsserviços, bem como uma qualidade geral do sistema resultante.

Palavras-chave: Microsserviços. Sistemas monolíticos. Migração. Problemas. Dificuldades.

Abstract

Thiago Cunha Ferreira. **Problems and difficulties in migrating from monolithic systems to microservices**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

In recent years, microservices have become a very popular way for structuring applications in the industry, with several advantages and opportunities identified by large companies such as Netflix, Uber and Amazon. Thus, a large number of organizations have been interested in moving ahead their monolithic systems, often outdated, to this distributed model. However, both the adoption of the microservices architecture and the migration process itself have a range of problems and obstacles that would make it difficult to apply them in practice. This work aims to analyze and contextualize some of these complications, identified in several reports from industry and academia. After developing these points, it is evident that continuing with the migration from monolithic applications to microservices is not always the most suitable path for all contexts. Furthermore, even when this is not the case, it is still important to recognize these difficulties and consider viable solutions to ensure a better expressiveness of the benefits of microservices, as well as an overall quality of the resulting system.

Keywords: Microservices. Monolithic systems. Migration. Problems. Obstacles.

Lista de Abreviaturas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
BD	Banco de Dados
CAP	*Consistency, Availability, and Partition Tolerance*
CDN	*Content Delivery Network*
CI/CD	*Continuous Integration/Continuous Delivery*
DDD	*Domain-Driven Design*
DNS	*Domain Name System*
FaaS	*Function as a Service*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTTP	*HyperText Transfer Protocol*
IaaS	*Infrastructure as a Service*
JSON	*JavaScript Object Notation*
REST	*REpresentational State Transfer*
RPC	*Remote Procedure Call*
SGBD	Sistema Gerenciador de Banco de Dados
SOA	*Service-Oriented Architecture*
SOC	*Service-Oriented computing*
XML	*eXtensible Markup Language*

Sumário

1 Introdução **1**
1.1 Problema . . . 2
1.2 Objetivo e motivação . . . 3
1.3 Metodologia . . . 3
1.4 Estrutura . . . 3

2 Sobre microsserviços **5**
2.1 Introdução . . . 5
2.2 Características de microsserviços . . . 7
2.2.1 Serviços independentes . . . 7
2.2.2 Integração pela rede . . . 8
2.2.3 Mudanças na estrutura organizacional . . . 10
2.3 Benefícios . . . 11
2.3.1 Heterogeneidade tecnológica . . . 12
2.3.2 Maior resiliência . . . 12
2.3.3 Escalabilidade mais eficaz . . . 13
2.3.4 Melhor manutenibilidade . . . 14
2.3.5 Implantações separadas . . . 15
2.4 Desvantagens . . . 16
2.4.1 Complexidade . . . 16
2.4.2 Dados distribuídos . . . 17
2.4.3 Aumento nos custos . . . 18

3 Do monolito para microsserviços **19**
3.1 Antes da migração . . . 20
3.1.1 Necessidade e capacidade de migração . . . 20
3.1.2 Objetivos . . . 21
3.1.3 Movimentação organizacional . . . 22
3.1.4 Análise da estrutura monolítica . . . 24

3.1.5 Modelagem de serviços 24
3.2 Durante a migração 26
3.2.1 Transformando o monolito em microsserviços 27
3.2.2 Ajustando operações 32
3.3 “Depois” da migração 36

4 Conclusão **37**

Referências **41**

Capítulo 1

Introdução

A arquitetura de sistemas computacionais, apesar de ser um tópico tão antigo quanto a própria computação, veio a se consolidar como um assunto de enorme importância na engenharia de *software* somente no começo do século XXI. Essa é uma área que hoje possui suas bases bem desenvolvidas nos quesitos de disponibilidade ferramental, padrões de uso replicáveis e dispersão de conhecimento entre desenvolvedores. Porém, isso não significa que o assunto se tornou trivial (Garlan, 2014).

Atualmente, com a necessidade de integração entre serviços de formas não previsíveis, a construção de *software* sobre a computação em nuvem, o advento de "novos" modelos organizacionais de times e processos (como *DevOps*), dentre outros fatores, há um grande interesse por parte de empresas em investir na organização de seus sistemas computacionais de acordo com essas novas tendências, aproveitando-se dos benefícios e possibilidades trazidos por esses movimentos (Dragoni, Giallorenzo *et al.*, 2017). Um modelo arquitetural que se tornou popular no mercado nos últimos anos (Loukides e Swoyer, 2020) e que facilita a adaptação de sistemas a esses requisitos modernos é o de **microsserviços**.

Dragoni, Giallorenzo *et al.* (2017) define a arquitetura de microsserviços como uma aplicação distribuída onde todos os seus módulos são microsserviços, esses sendo processos independentes, coesos e que interagem entre si por mensagens. Caso um sistema opte por seguir esse padrão e, dependendo da forma como a sua implementação é feita, há a possibilidade de que uma série de características não funcionais sejam melhor alcançáveis, como manutenibilidade ou disponibilidade do sistema. Essas e outras vantagens possibilitadas pela arquitetura, juntamente da demonstração de sua efetividade como base para aplicações reais e massivas, como Uber e Netflix (Haddad, 2015; Izrailevsky *et al.*, 2016), e de seu amadurecimento técnico e ferramental (AWS, 2021; Netflix, 2021), são algumas das razões por trás da fama dessa arquitetura.

A popularidade dos microsserviços também está associada a um outro tipo de estrutura muito comum em sistemas computacionais na indústria: a **arquitetura monolítica**. Podendo um monolito, nesse contexto, ser definido como "[...] uma aplicação de *software* cujos módulos não podem ser executados independentemente" (Dragoni, Giallorenzo *et al.*,

2017, tradução nossa)[1], essa é uma maneira de organizar aplicações de forma relativamente simples e direta, onde as tarefas de desenvolvimento, implantação (em inglês, *deployment*), construção de testes e escalonamento do sistema são bem menos complexos (Richardson, 2021b).

Porém, a medida que uma aplicação cresce e sua complexidade aumenta, ter ela reunida sob uma única unidade central (o monolito) pode trazer uma série de desvantagens e complicações, como implantações mais demoradas e arriscadas, além da tendência a acoplamento de módulos (Newman, 2015). Essas e outras limitações podem, no entanto, ser solucionadas pela arquitetura de microsserviços (Richardson, 2021b): por exemplo, características como a capacidade de implantação independente de serviços ou separação de bases de código em múltiplas unidades distintas podem ajudar a mitigar esses problemas encontrados em sistemas monolíticos.

1.1 Problema

Considerando as diversas vantagens possibilitadas pela arquitetura, sua maturação no mercado e como ela é vista como uma solução aos problemas de longa data de aplicações monolíticas, o alto interesse da indústria na ideia de migrar seus sistemas para microsserviços não é nada inesperado. No entanto, apesar desses fatores, um número significativo de relatos apontam que a transição nem sempre termina de forma positiva.

Diversos agentes relatam uma série de problemas e desafios que podem acometer implementações reais de microsserviços, como o uso de um número excessivo de ferramentas e tecnologias, uma maior fragilidade do sistema e dificuldades em adaptar a organização ao novo modelo (Wolff, 2019). Apesar de muitas dessas complicações serem resultado do modo como o projeto de migração é conduzido, a própria arquitetura traz outras, como as envolvendo distribuição de dados. Além disso, esse padrão arquitetural nem sempre é uma opção tão compatível com certos contextos (como em escalas muito reduzidas). Isso faz com que microsserviços não seja realmente uma opção viável para todas as aplicações (Etheredge, 2020; Velepucha e Flores, 2021).

Dessa forma, o conflito entre a visão otimista e a realista dos microsserviços pode se tornar um problema. Uma organização, tendo que arcar com seus sistemas monolíticos defasados, poderia ver nesse novo modelo uma forma de solucionar essas questões e "se adaptar às técnicas de ponta da indústria". Porém, em algum momento desse processo, o projeto pode começar a "desandar", seja por más decisões de implementação [que pode ocorrer por falta de experiência com microsserviços (Fritzsch *et al.*, 2019)], seja pela falta de capacidade em se lidar com as complicações da arquitetura ou do próprio processo de migração. Ao aumentar essa discrepância, cada vez mais aplicações passariam por esse ciclo.

[1] No original: "[...] a software application whose modules cannot be executed independently."

1.2 Objetivo e motivação

Este trabalho de conclusão de curso tem por objetivo explorar as dificuldades e complicações existentes no processo de transição de sistemas monolíticos para sistemas estruturados em microsserviços. Isso envolve tanto as complicações que surgem pela própria natureza complexa da arquitetura quanto pelo processo não trivial de reestruturar a aplicação (ou parte dela) de acordo com o novo modelo. Além do mais, pretende-se que esses tópicos sejam apresentados de forma contextualizada.

Espera-se que com este trabalho, um leitor que faça parte de um time que pretende realizar essa transição possa ficar mais ciente do contexto ao redor da arquitetura de microsserviços, das motivações e dificuldades associadas ao processo e, por fim, considerar se essa mudança é realmente algo necessário ou até mesmo vantajoso para o futuro de sua aplicação. Caso decida por prosseguir com a migração, também espera-se que as noções apresentadas ao longo do trabalho o auxiliem nesse longo processo.

1.3 Metodologia

O conteúdo deste trabalho é baseado, em grande parte, na exploração de relatos (através de artigos científicos, transcrições de entrevistas ou até mesmo de postagens em blogues) feitos por profissionais da área de engenharia de *software*, empresas e pesquisadores que tiveram contato teórico e/ou prático com microsserviços. A importância dessas fontes para a composição deste trabalho está ligada à quantidade e à variedade de problemas e dificuldades que podem ser encontrados durante a refatoração de sistemas de escala "corporativa". Nesses cenários, a necessidade de atender às expectativas de clientes reais e de definir uma organização propriamente dita introduz uma série de complicações adicionais de difícil replicação em projetos didáticos, como as alterações na estrutura organizacional.

Além disso, um pouco de conhecimentos práticos sobre sistemas distribuídos e microsserviços, originados de breves contatos em ambiente acadêmico e empresarial, foram utilizados ao lado desses relatos para melhor relativizá-los. As características de implementações de sistemas em microsserviços podem variar significativamente de caso a caso (Osses *et al.*, 2018), e essa pequena experiência prática permitiu uma melhor compreensão dessa diversidade ao reunir e relativizar esses relatos.

1.4 Estrutura

Este trabalho está estruturado em duas partes centrais:

1. **Apresentação da arquitetura de microsserviços**: Conceituação, descrição de características e introdução às possíveis vantagens e desvantagens dos microsserviços;
2. **O processo de transição de monolito para microsserviços**: Contextualização de passos envolvidos em projetos de migração para microsserviços, acompanhados da apresentação de problemas e dificuldades que podem acometer as etapas desse processo;

3. **Conclusão**: Resumo do conteúdo apresentado até então e conclusão do trabalho.

Capítulo 2

Sobre microsserviços

Para uma discussão mais embasada sobre os problemas e complicações associados ao processo de transição de sistemas monolíticos para microsserviços, é fundamental que primeiro haja uma caracterização geral da arquitetura.

Enquanto conceitualmente, no nível mais básico, essa é uma forma de estruturar aplicações que possa aparentar ser relativamente simples, diversas dificuldades trazidas por aspectos mais específicos dos microsserviços serão evidenciadas ao aplicá-la em ambientes do "mundo real". Considerando ainda que não existe uma fórmula exata que todas as implementações devam seguir, há uma infinidade de problemas em potencial, nem sempre tão óbvios. Mais especificamente, o estado final do sistema acaba sendo guiado por uma série de escolhas influenciadas por fatores como o contexto da aplicação, quais benefícios associados aos microsserviços serão priorizados e como se lida com as dificuldades da arquitetura (Richardson, 2021a; Ramchand *et al.*, 2021).

Dessa forma, este capítulo irá expor algumas das principais facetas dos microsserviços, além de alguns pontos positivos e negativos associados a eles. Isso permitirá não só uma melhor contextualização do assunto, mas também a introdução de algumas complicações associadas à própria arquitetura que podem se fazer presente durante o processo de migração.

2.1 Introdução

A arquitetura de microsserviços é, em alto nível, uma forma de organizar aplicações como um conjunto de *microsserviços*, componentes (comumente chamados somente de "serviços") independentes e que se comunicam através de rede de computadores, formando assim o seu corpo de funcionalidades. Essa comunicação é feita através de *Application Programming Interfaces* (APIs) expostas por cada unidade. Cada serviço possui uma base de código separada e relativamente pequena, onde seu escopo é geralmente delimitado conforme os domínios da aplicação, utilizando técnicas analíticas como *Domain-Driven Design* (DDD). A independência entre microsserviços se manifesta através da separação de processos computacionais, do particionamento do armazenamento de dados entre serviços, da possibilidade de construção de *pipelines* para implantações individuais e, por fim, de

uma melhor capacidade de divisão das responsabilidades entre múltiplos times, cada um responsável por partes específicas da aplicação. Esse último fator permite a formação de equipes multidisciplinares, de menor escala e com posse completa sobre os processos internos de um serviço (desenvolvimento, testes, implantação e suporte técnico) (Lewis e Fowler, 2014; Thones, 2015; Newman, 2015; Dragoni, Giallorenzo *et al.*, 2017).

A ideia básica por trás dos microsserviços surge antes mesmo da concepção do termo. Desde o começo dos anos 2000, conceitos como *Service-Oriented Computing* (SOC) e *Service-Oriented Architecture* (SOA) aparecem no mercado com o objetivo de explorar certas vantagens trazidas pela separação das responsabilidades de um sistema em unidades autônomas e distribuídas e, mais tarde, pela organização dessas unidades ao redor de subdomínios (Dragoni, Giallorenzo *et al.*, 2017). Alguns fatores impediram uma adaptação bem-sucedida do modelo SOA original em aplicações "reais", como especificações convolutas e a tendência a uma centralização de complexidade nas conexões entre serviços (Newman, 2015), bem como em dificuldades relacionadas ao descobrimento de serviços (Mazzara *et al.*, 2018). Entretanto, algumas das ideais centrais a esses padrões arquiteturais seriam elementos centrais à construção do que eventualmente seria conhecido como microsserviços.

A partir de 2011, grupos de engenheiros e arquitetos de *software* passaram a compartilhar suas experiências com padrões arquiteturais que vinham sendo explorados no mercado e a apresentar casos de estudos de aplicações com certas características, servindo como ponto de origem para o cunho do termo "microsserviços" como é conhecido atualmente (Lewis e Fowler, 2014). Para alguns, essa arquitetura seria uma implementação do modelo SOA onde serviços são menores e realizam comunicações mais simples e flexíveis (Dragoni, Giallorenzo *et al.*, 2017; Velepucha e Flores, 2021). Para outros, microsserviços seriam um estilo arquitetural distinto do SOA (Pautasso *et al.*, 2017). De qualquer forma, a disponibilidade de ferramentas e conhecimento prático relacionado ao assunto evoluiu consideravelmente desde então, especialmente com a contribuição da indústria (Jamshidi *et al.*, 2018).

Deve-se ater, no entanto, que essa é apenas uma caracterização superficial de conceitos comumente associados à arquitetura de microsserviços. Sendo ela uma estrutura bastante flexível, nem todos os elementos descritos são efetivamente aplicados em cenários reais e o que se entende por "microsserviços" pode variar: algumas implementações podem, por exemplo, decidir manter um único banco de dados, ter múltiplos times sobre um único serviço ou ainda manter uma unidade monolítica que hospeda funcionalidades vitais à aplicação, e ainda poderem ser vistos, por certos grupos, como um sistema "baseado em microsserviços".

Sendo um estilo arquitetural surgido diretamente de práticas do mercado, o importante não é necessariamente seguir à risca todas as características costumeiramente associadas a ele, mas sim organizar a aplicação de maneira mais conveniente aos interesses internos e tentar evitar os percalços da arquitetura, considerando o contexto ao seu redor. Isso não significa, no entanto, que não há práticas consideradas "antipadrões", como o uso de persistência compartilhada, nem que essas escolhas não possam levar a consequências indesejadas no futuro (Taibi *et al.*, 2019).

2.2 Características de microsserviços

Nesta seção, serão apresentadas algumas das características comumente associadas à arquitetura de microsserviços.

2.2.1 Serviços independentes

Como mencionado anteriormente, uma das principais características vistas em sistemas baseados microsserviços é a decomposição de suas funcionalidades e bases de código entre diversos serviços independentes. Cada um deles pode ser considerado como uma "aplicação" totalmente separada, ou seja, com processos separados, possivelmente em máquinas distintas (utilizando ou não mecanismos de virtualização), responsáveis por uma parte específica do sistema. E é através da comunicação de dados, comandos ou avisos entre microsserviços que se forma o seu corpo de funcionalidades.

Assim como em sistemas monolíticos, a execução de certas tarefas pode necessitar da interação de múltiplos módulos. Porém, ao contrário desse modelo, que só permite a separação lógica dos módulos, a arquitetura de microsserviços possibilita que essa divisão seja "física" também, de forma que funções pertencentes a um mesmo grupo sejam executadas separadamente de outras partes do sistema.

Formas de divisão

A separação dos microsserviços segue princípios similares aos módulos vistos em linguagens de programação, fundamentados pela noção de *single-responsibility principle* descrita por Martin (2014). Mais especificamente, é do interesse de ambos agrupar um conjunto de funções e estruturas de dados suficientemente próximas (seguindo alguma métrica) para minimizar o número de "locais" afetados por alterações internas de serviços. Adicionalmente, espera-se que o acesso às suas funcionalidades seja restrito a uma interface que exponham o mínimo possível de detalhes internos, de forma a evitar que agentes externos (consumidores) criem dependências com a sua implementação interna e assim impeçam que sua evolução seja feita independentemente deles. Essas características, melhor representadas pelos conceitos de "alta coesão" e "baixo acoplamento", são vantajosas ao caráter de independência e especialização dos microsserviços (Newman, 2015; Dragoni, Giallorenzo *et al.*, 2017; Wolff, 2019).

Uma das principais estratégias para particionamento dos microsserviços utiliza conceitos de DDD para definir os limites entre serviços. Desenvolvido por Evans (2004), a técnica de *domain-driven desing* permite agrupar os requisitos de um sistema num modelo baseado em seus domínios, de forma a modelar os conhecimentos e relações entre eles (Munezero *et al.*, 2018). Nesse padrão, o conceito de *bounded context*, que contém os detalhes de um único domínio, é formado ao redor das capacidades de negócio, ao invés de uma divisão focada somente no código ou na reutilização de funcionalidades (Shadija *et al.*, 2017b), e é através desses *bounded contexts* que serão definidos os limites iniciais dos microsserviços. No entanto, trabalho adicional será necessário para propriamente modelar bons microsserviços (Munezero *et al.*, 2018; Rademacher, Sorgalla *et al.*, 2018).

No entanto, DDD não é a única maneira de dividir as responsabilidades dos serviços.

Existem também técnicas automáticas (e semiautomáticas) feitas de diferentes formas, como através da análise estática (e possivelmente dinâmica) de código, da captura dos relacionamentos entre dados e/ou do uso algoritmos de *machine learning* (Ren *et al.*, 2018; Abdullah *et al.*, 2019; Selmadji *et al.*, 2020).

Tamanho, número e granularidade de serviços

Como o nome do padrão arquitetural sugere, um dos fatores marcantes dos microsserviços é o seu tamanho diminuto. Porém, o que exatamente significa um serviço ser "micro" e quão pequeno ele deveria ser não são noções tão simples de serem estabelecidas.

Considerando que as partes da aplicação estariam associadas a processos computacionais separados, seria esperado que cada um desses processos tivesse uma quantidade menor de código e menos funcionalidades do que se estivessem todos reunidos numa só "entidade" (monolito). Porém, isso não implica que eles são absolutamente "pequenos": alguns fatores, como uma linguagem de programação que exija mais código *boilerplate* ou serviços naturalmente complexos, podem fazer com que microsserviços não sejam tão reduzidos assim. O seu tamanho "ideal" estaria mais ligado ao contexto em que ele está inserido (Newman, 2015; Newman, 2019).

A tendência a serviços de menor escopo também reduz sua granularidade, conceito que, segundo Kulkarni e Dwivedi (2008, tradução nossa), se "[...] refere ao tamanho do serviço e ao escopo das funcionalidades que um serviço expõe"[1]. Inversamente, isso tenderá a aumentar o número de microsserviços (Chen, 2018). Se a ideia transmitida pelo prefixo "micro" for extrapolada, esse seria o caminho a ser seguido por aplicações modeladas nessa arquitetura.

No entanto, nem sempre é vantajoso construir muitos serviços da menor forma possível (Richardson, 2019b; Etheredge, 2020), uma vez que certas qualidades de sistemas baseados em microsserviços são influenciadas pelo nível de granularidade escolhido (Vera-Rivera *et al.*, 2021) e por certas características da aplicação. Por exemplo, a introdução de um grande número de serviços pode levar a impactos na *performance*, uma vez que mais chamadas através da rede serão necessárias para cumprir uma funcionalidade, mas também pode ser interessante se a reutilização de funcionalidades for priorizada. Por outro lado, times menores são mais facilmente associados a serviços menores, mas a complexidade adicional que será introduzida ao escolher a arquitetura (Chen, 2018) pode não ser justificável em pequenas aplicações (até certo ponto) (Shadija *et al.*, 2017a).

2.2.2 Integração pela rede

Em qualquer aplicação de médio e grande porte, é esperado que certas funcionalidades sejam construídas sobre interações de múltiplos componentes lógicos (módulos internos, bibliotecas externas, sistemas auxiliares, etc.). Apesar do mesmo se aplicar a sistemas estruturados em microsserviços, a comunicação entre eles se torna um elemento central na formação da arquitetura. Assim, é interessante haver uma maior consideração sobre como essas conexões são formadas para garantir um certo nível de qualidade ao sistema distribuído.

[1] No original: "[...] refers to the service size and the scope of functionality a service exposes."

Uma vez que os componentes da arquitetura de microsserviços são executados em processos separados, possivelmente até em máquinas diferentes, a comunicação entre eles não poderia ser baseada no acesso a uma região comum da memória, como através da simples chamada de funções ou métodos. Nesses casos, faz-se necessária uma forma de comunicação entre processos, o que pode ser alcançado através do uso de redes de computadores (locais ou a própria Internet) como meio de interligar os microsserviços. Logicamente, a arquitetura preza para que essas conexões sejam feitas utilizando protocolos mais simples e de forma a evitar a concentração de lógica da aplicação em elementos que não sejam os próprios serviços (Pautasso *et al.*, 2017; Velepucha e Flores, 2021) [como *proxies* ou servidores *Domain Name System* (DNS), por exemplo].

Em essência, existem dois padrões de comunicação proeminentes na indústria que descrevem formas distintas de estruturar a comunicação entre múltiplos serviços: **requisição-resposta** (*request-response*, em inglês) e **orientado a eventos** (*event-driven*). A primeira se baseia em uma troca de mensagens, onde o processo "remetente" envia uma mensagem contendo uma "requisição" [representada em uma linguagem específica, como *JavaScript Object Notation* (JSON), *eXtensible Markup Language* (XML), etc.] a um "destinatário", mais especificamente a uma API exposta por esse último agente, através de tecnologias e protocolos de comunicação acordados [*HyperText Transfer Protocol* (HTTP), *Remote Procedure Call* (RPC), *REpresentational State Transfer* (REST), etc.]. Quando a requisição for recebida, fica a cargo do destinatário retornar ao remetente uma "resposta" de acordo. Esse processo pode ocorrer de forma síncrona (as atividades do remetente são interrompidas até que uma resposta seja recebida) ou assíncrona (o remetente não espera pela resposta).

Já o padrão de comunicação orientado a eventos se baseia no compartilhamento de eventos ocorridos no sistema (um usuário foi cadastrado, uma compra foi efetivada, etc.) para que qualquer microsserviço interessado neles possam reagir de acordo (enviar um email de confirmação de conta, processar cobrança de uma compra, etc.). Esse mecanismo é usualmente implementado através do padrão *publish/subscribe*, onde serviços enviam eventos a agentes independentes (na figura dos *message brokers*, por exemplo) através da "publicação" desse evento em "tópicos", e esses agentes serão responsáveis por distribuí-los às unidades que demonstraram interesse em certos tópicos' ao se "inscrever" neles.

Em geral, essa integração dos microsserviços através de redes expõe algumas complicações adicionais. Pela própria natureza física dos meios de transmissão utilizados por redes de computadores, problemas associados à velocidade e a disponibilidade das conexões entre serviços podem afetar o funcionamento da aplicação de forma parcial ou até mesmo completa, caso medidas de resiliência não sejam implementadas. Além disso, tanto a escolha do padrão de comunicação quanto as decisões necessárias para sua implantação [utilização de mensagens assíncronas, API estruturada em REST, uso da restrição *Hypermedia As The Engine Of Application State* (HATEOAS), etc.] possuem impactos positivos e negativos para a aplicação. Por exemplo, uma comunicação baseada em eventos pode reduzir acoplamento entre módulos, mas também pode tornar mais complicadas a compreensão geral da aplicação, se utilizada demasiadamente (Newman, 2015).

2.2.3 Mudanças na estrutura organizacional

No contexto de microsserviços, existe uma abundância de conhecimento e materiais relativos a aspectos mais práticos da arquitetura. Um grande número de técnicas, tecnologias e *frameworks* foram desenvolvidas ou adaptadas para melhor acomodar aplicações organizadas em microsserviços (Newman, 2015; Balalaie, Heydarnoori, Jamshidi *et al.*, 2018; Richardson, 2021a), porém outro aspecto menos técnico também é um importante fator na formação e manutenção desses sistemas: a configuração da estrutura organizacional ao seu redor.

Em seu livro "*How Do Committees Invent*", Conway (1968) observa, no que costuma ser caracterizado como "lei de Conway", que a maneira como uma organização é estruturada tende a afetar o *design* de seus sistemas. Considerando isso, ao tentar manter uma estrutura corporativa que não se "encaixe" bem com a arquitetura de microsserviços, o *design* do sistema tenderá a ser modelado com algumas características conflitantes. Pensando num modelo organizacional que costuma suportar grandes aplicações monolíticas, com times extensos e separados por camadas técnicas, com a necessidade de troca de artefatos entre equipes, dentre outras características, haverá a tendência da implementação dos microsserviços ser feita de modo a melhor acomodar esses aspectos da empresa, e não ao que seria mais benéfico para a aplicação em si. Dessa forma, para melhor construir e integrar sistemas baseados em microsserviços a um certo ambiente (empresarial), alguns ajustes à estrutura da organização podem ser necessários.

Uma dessas alterações estaria ligada à forma dos times. Primeiramente, há um incentivo à formação de equipes menores e focadas em setores mais reduzidos da aplicação, configuração que tornaria a sua gestão mais simples e permitiria uma maior agilidade (Balalaie, Heydarnoori e Jamshidi, 2016a; Haden, 2021). Também há um incentivo para que elas sejam multidisciplinares, de forma que diversos níveis de especialização sejam reunidos sobre unidades mais próximas, o que permitiria uma maior autonomia das equipes, ao menos no quesito de capacidade técnica (Lewis e Fowler, 2014; Balalaie, Heydarnoori e Jamshidi, 2016a). Essa formação é mais viável quando os serviços são mais "enxutos", permitindo que uma equipe consiga gerenciar todos os aspectos de um microsserviço.

Outra mudança necessária para melhor acomodar a arquitetura de microsserviços seria uma melhor distribuição de responsabilidades entre os times. Dessa forma, equipes passam a ser responsáveis pela gestão completa do ciclo de vida dos serviços a quais são atribuídas, reduzindo assim a dependência para com agentes centrais da organização em relação à necessidade de coordenação antes que qualquer alteração no sistema possa ser efetivada. Similarmente, essas responsabilidades devem ser separadas de forma a evitar a necessidade de manter comunicação constante entre múltiplos times (o que pode ocorrer quando há uma divisão indevida dos serviços ou quando eles são mantidos por mais de uma equipe). Isso, no entanto, não quer dizer que elas devam operar completamente isoladas: a comunicação entre módulos distintos ainda é a base do funcionamento da aplicação, o que implica num certo nível de contato entre as equipes, e a existência de uma figura central ainda é necessária para direcionar a evolução do sistema sob uma perspectiva de negócios e para definir padrões arquiteturais de maior escopo (Newman, 2015; Balalaie, Heydarnoori e Jamshidi, 2016a).

A introdução dessas "novas" formas de estruturar organizações faz parte do movimento de tornar microsserviços mais autônomos. Sob uma perspectiva técnica, enquanto serviços forem construídos como processos completamente separados e dependentes de outros módulos da aplicação somente através de interfaces bem definidas, é possível isolar a implementação interna de forma a permitir sua evolução independentemente de agentes externos. Porém, de nada adianta ter essa capacidade se não for possível colocar isso em prática: se times tiverem que "responder" a outras unidades antes de poder prosseguir com qualquer alteração, perde-se boa parte da potencial independência dos serviços.

No entanto, a profundidade dessas alterações organizacionais deve variar. Em casos de empresas que ainda seguem modelos costumeiramente "monolíticos", um certo nível de mudanças na estrutura organizacional é bem-vindo, porém o nível de fragmentação de times e quanta autonomia é desejável atribuir-lhes dependerá do contexto em que a aplicação esteja inserida (Haden, 2021). É possível, por exemplo, que obrigações legais incentivem a centralização de algumas questões relativas ao desenvolvimento e implantação de alterações em serviços (segurança em sistemas bancários seria um caso) ou que times maiores sejam formados por uma decisão de manter serviços maiores. Dessa forma, fica a cargo daqueles que desejam seguir com o modelo de microsserviços julgar quais alterações introduzir e como implementá-las em seus contextos específicos.

2.3 Benefícios

Uma das razões por trás da popularização dos microsserviços está associada a uma série de benefícios possibilitados pela utilização dessa arquitetura em certos casos. Especialmente entre aplicações que encontraram as limitações do modelo monolítico, como redução na velocidade de desenvolvimento ou em dificuldades no seu escalonamento (Richardson, 2021b), ao atuar em largas escalas (muitos desenvolvedores, trabalhando sobre um domínio complexo, com altas exigências de disponibilidade e atualizações constantes, etc.), a sua disposição como uma "malha" de serviços pequenos e independentes permite (ou facilita) a expressividade de algumas características que podem ser interessantes, especialmente em sistemas computacionais complexos.

No entanto, é importante frisar que o nível de expressividade de cada uma delas irá variar de caso a caso, sendo até mesmo possível que o efeito inverso ocorra (um sistema que tenderia à resiliência se torna frágil, por exemplo). Isso ocorre devido à variedade de formas como uma aplicação baseada em microsserviços pode ser formada (Richardson, 2021a), onde, por uma série de escolhas que priorizam certas características em detrimento de outras e que devem ser feitas durante sua construção (como dividir o domínio em serviços, como a comunicação entre serviços será realizada, como organizar os times, etc.), surge uma "versão" do sistema com certos aspectos qualitativos mais ou menos pronunciados.

Com isso em mente, esta seção abordará alguns dos pontos positivos trazidos pelo modelo de microsserviços.

2.3.1 Heterogeneidade tecnológica

Visto que microsserviços se comunicam exclusivamente através da rede, com suas bases de código efetivamente separadas, abre-se a possibilidade de cada serviço ser implementado utilizando tecnologias distintas. Ao se estruturar a comunicação utilizando protocolos e padrões que não estejam atrelados a uma linguagem de programação ou *framework* específicos, como HTTP, *Advanced Message Queuing Protocol* (AMQP), REST, etc., e ao estabelecer APIs agnósticas a tecnologias, há uma clara separação entre os ambientes externo e internos aos microsserviços: a implementação da comunicação naquele espaço pode ser feita de formas padronizadas a nível de aplicação, enquanto a implementação da lógica nesse pode utilizar a *stack* de tecnologia mais conveniente para seu contexto. Dessa forma, desde que a comunicação externa esteja relativamente padronizada, como o serviço é construído pouco importa para sua integração com o resto do sistema.

Um dos interesses por trás da utilização de tecnologias distintas entre múltiplos serviços é a possibilidade de escolher ferramentas que melhor sirvam seus propósitos (Rademacher, Sachweh *et al.*, 2019). Por exemplo, dois microsserviços podem escolher utilizar tipos distintos de banco de dados, baseando-se em qual deles melhor combina com os tipos de dados que cada um vai tratar (a natural interconectividade suportada por bancos de dados em grafos pode ser mais interessante para um deles, enquanto a estruturação rigorosa de bancos relacionais pode ser para outro), ou podem escolher entre duas linguagens de programação diferentes, a depender das prioridades de cada um (um microsserviço escrito na linguagem Python pode estar focado em tarefas de aprendizado de máquina, enquanto outro escrito em C prioriza tarefas de alto desempenho).

Apesar de ser possível ter múltiplas *stacks* de tecnologias diferentes, geralmente exagerar nessa quantidade pode trazer algumas desvantagens (Taibi *et al.*, 2019). Essa variedade dificultaria a introdução de novos desenvolvedores nas equipes (o que inclui a transição de membros entre times) e complicaria a criação e suporte a ferramentas criadas e utilizadas internamente (como bibliotecas ou *templates* para auxiliar na implantação dos serviços), dado que seria necessário adaptá-las para integração com diferentes tecnologias (Newman, 2015).

2.3.2 Maior resiliência

Resiliência, no contexto de sistemas computacionais, é definido por Florio (2013) como o nível de confiança dado a um sistema de *software* em se adaptar de maneira a absorver e tolerar os impactos causados por falhas, ataques e mudanças de origem interna e externa. Em ocasiões onde essa resiliência não se faz presente, pode-se experienciar um aumento no tempo de indisponibilidade do sistema ou um maior atraso no cumprimento de suas funcionalidades. Para aplicações que não possam conviver com esses problemas (por causas contratuais, de segurança ou pelas exigências de seus clientes), garantir um bom nível de resiliência é fator fundamental.

A arquitetura de microsserviços permite que uma série de mecanismos e padrões sejam estabelecidos para melhor garantir a resiliência do sistema (Mendonca *et al.*, 2020). É possível que falhas em serviços ou na conexão entre eles sejam isoladas do resto da aplicação (caso elas não afetem o acesso a componentes centrais), para não torná-la completamente

indisponível (através da degradação de certas funcionalidades, por exemplo). Para tanto, algumas técnicas podem ajudar, entre elas: o ajuste de *timeouts* sensatos para a espera de respostas a requisições, implantação de *circuit breakers* para que a indisponibilidade de serviços consumidos seja percebida mais rapidamente ou simplesmente garantir um bom isolamento de microsserviços através de *fallbacks*, onde a falha de um deles não impacta o funcionamento de seus consumidores (Newman, 2015). Em contraste, aplicações monolíticas possuem certa dificuldade em conseguir o mesmo nível de resiliência, uma vez que toda a lógica está sendo executada sob um mesmo processo computacional e qualquer falha fatal que ocorrer em qualquer parte da aplicação tornaria a instância da aplicação inteiramente indisponível.

Porém, em sua forma "pura" de sistema distribuído, sem uma consideração prévia pela resiliência do sistema, a arquitetura de microsserviços não é inerentemente resiliente. Pelo contrário: diversas questões ligadas aos níveis de rede, *hardware* e aplicação a tornam frágil, como as comuns falhas de comunicação entre serviços (que podem ocorrer até mesmo por uma sobrecarga da capacidade de transferência de mensagens). Dessa forma, para minimizar esses impactos, serão necessários maiores considerações sobre esse atributo durante a implantação da aplicação (Jamshidi *et al.*, 2018).

2.3.3 Escalabilidade mais eficaz

Apesar de ser possível compreender escalabilidade nesse contexto como a capacidade da aplicação em se expandir "logicamente" (número de microsserviços, de times ou funcionalidades), o termo está associado aqui com a sua expansão "física", ou seja, como ela lida com variações no tráfego de usuários e na demanda por recursos computacionais, medindo assim sua capacidade de se adequar à essas flutuações da melhor forma possível. Considerando essa segunda definição, pode-se dizer que a arquitetura de microsserviços permite que aplicações tenham um maior controle sobre como a sua escalabilidade, permitindo que ela seja feita de forma desigual entre partes distintas do sistema. Essa desigualdade se manifesta das seguintes formas (Newman, 2015; Coulson *et al.*, 2020):

- Cada serviço pode ter um número diferente de instâncias sendo executadas a qualquer momento, ou seja, eles podem ser escalados independentemente dos outros. Dessa maneira, é possível melhor ajustar a infraestrutura da aplicação às suas reais necessidades (módulos responsáveis por funcionalidades mais utilizadas necessitariam de maior poder computacional total, o que pode ser obtido através de mais instâncias, por exemplo);
- Como cada microsserviço pode estar localizado em máquinas com recursos diferentes (sejam elas virtuais ou reais), a arquitetura também permite uma escalabilidade vertical independente: serviços que necessitem de maior poder computacional podem ser alocados para máquinas mais robustas, e serviços que precisem de menos poder, para máquinas menos robustas;

Em uma arquitetura monolítica, um redimensionamento da aplicação dessa forma não é algo realmente possível, uma vez que apenas os grandes componentes que a constituem podem ser escalonadas (Dragoni, Lanese *et al.*, 2017; Coulson *et al.*, 2020). Dessa forma, todas as partes do sistema "crescem" ou "diminuem" em uníssono, podendo resultar

em investimentos excessivos na capacidade da infraestrutura. Especialmente considerando o movimento atual na indústria de estruturar aplicações utilizando tecnologias de computação em nuvem (Esposito *et al.*, 2016; Yu *et al.*, 2019), através de serviços como "*Infrastructure as a Service*" (IaaS), que permite o provisionamento elástico e configurável da infraestrutura de um sistema computacional, ou "*Function as a Service*" (FaaS), que abstrai por completo a construção e o gerenciamento de tal infraestrutura, e levando em conta a sua compatibilidade com a arquitetura de microsserviços (Esposito *et al.*, 2016; Malla e Christensen, 2019), surge um incentivo para a adoção de microsserviços.

2.3.4 Melhor manutenibilidade

Uma das vantagens mais procuradas por times que cogitam migrar seus sistemas para a arquitetura de microsserviços está ligada à possibilidade de uma maior manutenibilidade da aplicação (Fritzsch *et al.*, 2019). Através dessa característica, durante a concepção de novas funcionalidades ou alteração das antigas, os processos de planejamento e implementação, bem como a atividade de identificar a origem de *bugs*, podem ser realizados de formas mais simples e mais isolada do restante da aplicação. A expressividade desse fator, no entanto, dependerá da busca ativa por duas qualidades: uma boa divisão dos serviços e maior independência entre times.

O primeiro fator está associado a uma das características da arquitetura de microsserviços: o incentivo à modularização "natural" do sistema. Através de um *design* da aplicação e da organização ao redor de capacidades de negócio, acaba-se por facilitar o alcance de melhor modularidade (Vural e Koyuncu, 2021). Algumas das vantagens trazidas por esse fator são (Newman, 2015; Bogner *et al.*, 2020; Velepucha e Flores, 2021):

- Bases de código menores e que, individualmente, tendem a uma menor complexidade e uma maior compreensibilidade;
- Separação clara entre estruturas internas e interfaces externas dá margem a maior abstração e impede acoplamentos diretamente pelo de código ou estruturas do banco de dados (apesar disso ainda ser possível através de outros meios, como compartilhamento excessivo de detalhes internos pela API dos exposta por serviços);
- Reimplementação completa de serviços se torna mais viável o quão menor ele for;
- Alta coesão de funcionalidades reduz a necessidade de alterações em múltiplos lugares.

Já o segundo fator que auxilia na manutenibilidade da aplicação é uma boa divisão de responsabilidades entre times, onde ter somente uma equipe completamente responsável por cada serviço traz certas vantagens. Comparando a atribuição de múltiplas equipes a um mesmo serviço com a atribuição de apenas uma, o tipo de comunicação e o nível esperado de dispersão de conhecimento técnico diferem.

No primeiro caso, bloqueios organizacionais ou geográficos podem gerar uma certa distância entre equipes, com comunicações menos frequentes e de maior granularidade, assim dificultando a coordenação de processos internos ao serviço (desenvolvimento das estruturas internas, por exemplo) e possivelmente gerando conflitos de implementações. No segundo, o contato mais próximo entre membros do grupo evita essas complicações.

Deve-se notar, no entanto, que essa distância entre as equipes pode ser melhor lidada quando apenas coordenações mais superficiais e menos frequente são necessárias, como as durante a formação de comunicações entre serviços, naturalmente mais abstratas pela menor exposição de detalhes internos através das APIs (Newman, 2015; Chen, 2018).

É importante mencionar que, apesar desses atributos que permitem uma melhor manutenibilidade do sistema estarem associados aos microsserviços, eles são de longe características exclusivas dessa arquitetura. É inteiramente possível presenciar grande parte das consequências descritas em sistemas monolíticos através da compartimentalização da aplicação em módulos orientados ao seu domínio ou através da divisão de equipes de maneira funcional. Em geral, a vantagem da arquitetura de microsserviços nesse quesito está no maior incentivo para adoção dessas práticas. Uma aplicação monolítica teria maior dificuldade em garantir essas práticas: uma boa modularização seria difícil de ser garantida a longo prazo nesse contexto, já que há uma maior facilidade em violar os limites de módulos e gerar acoplamentos indevidos através de código, e a organização de times de forma funcional nem sempre seria a estrutura mais natural (se considerarmos a lei de Conway, o modelo comum de separar aplicações web monolíticas em grandes camadas de apresentação, lógica de negócios e base de dados incentivará a formação de times estruturados de acordo com essas especialidades técnicas) (Newman, 2015; Newman, 2019; Coulson *et al.*, 2020).

2.3.5 Implantações separadas

Como consequência dos microsserviços serem uma espécie de "aplicação" separada e independente, é possível, em teoria, que cada um deles seja implantado separadamente. Dessa forma, um serviço pode ser criado ou atualizado por uma equipe sem necessariamente impactar no funcionamento de outras unidades, assim não sendo necessárias coordenações tão profundas com outros times. Isso implica na possibilidade de entregas mais frequentes e em menores escalas, práticas características de *Continuous Delivery* [facilitadas pela arquitetura de microsserviços (Chen, 2018)]. Algumas das vantagens trazidas por essas características são: redução de riscos associados a grandes *releases* (muitas modificações publicadas de uma vez só dão margem a problemas de maior complexidade), maior facilidade em localizar a origem de erros (menos modificações reduz a área de busca), e possibilidade de rápida reversão de funcionalidades ao estado anterior, caso necessário (Fowler, 2013; Newman, 2015; Taibi *et al.*, 2018; Velepucha e Flores, 2021).

No entanto, seguir a prática de implantar microsserviços de forma independente exige considerações adicionais. Primeiramente, mesmo que isso seja possível de uma perspectiva técnica, o nível de independência entre serviços pode impedir que isso seja feito na prática. Um alto acoplamento entre serviços significa que há uma maior chance de alterações em um deles afetar o funcionamento de outro e uma baixa coesão implica que alterações em funções conceitualmente próximas deverão ser aplicadas entre múltiplos serviços. Com essas características, alterações que deveriam estar isoladas em um único serviço irão afligir outros de forma a necessitarem modificações simultâneas e, consequentemente, implantações simultâneas.

Em segundo lugar, a automação tem um importante papel na viabilidade de implantações independentes. Em ecossistemas de *softwares* empresariais, é comum haver uma

série de ambientes computacionais distintos, cada um focado em certas etapas do ciclo de vida da aplicação, como os de desenvolvimento, testagem e validação, até chegar ao de produção, no qual o usuário final terá acesso. Ter que realizar o processo de implantação em cada um deles manualmente pode envolver um conjunto de passos intermediários específicos a cada ambiente, potencialmente levando à introdução de errors. Por exemplo, o desenvolvimento local de um microsserviço pode exigir que múltiplos outros componentes (incluindo seus bancos de dados próprios) tenham que ser lançados simultaneamente e que outras condições especiais ou configurações específicas sejam aplicadas (menos instâncias rodando em paralelo, inclusão ou remoção de componentes arquiteturais como *Service Discovery*, etc.). Em vista dessas complicações e considerando que elas se tornam mais proeminentes quanto maior for o número de microsserviços, a automatização desse processo é um passo importante a ser tomado, até mesmo como forma de concentrar mais esforços na entrega de funcionalidades e menos na manutenção da infraestrutura da aplicação.

Por fim, ao prosseguir com essa forma independente de atualizar os microsserviços, algumas dificuldades relacionadas à evolução independente de suas interfaces e a versionamento podem surgir (Balalaie, Heydarnoori e Jamshidi, 2016b; Taibi *et al.*, 2019). No momento em que um serviço consumido por outros é atualizado, nem sempre é possível forçar que serviços consumidores se adaptem a qualquer tipo de alteração das funcionalidades ou da API, ao menos de forma imediata. Isso impede que serviços evoluam de forma completamente livre, devendo evitar mudanças que afetariam a integridade dos consumidores (especialmente quando esses são agentes externos, como quando essa API é disponibilizada publicamente). Caso esse tipo de alteração seja inevitável, algumas práticas adicionais podem impedir a quebra dos consumidores, como garantir a coexistência de múltiplas versões da funcionalidade, da API ou até mesmo do próprio serviço, de forma que serviços possam continuar dependendo de uma versão antiga e, com o tempo, irem atualizando para a nova versão (Newman, 2015; Larrucea *et al.*, 2018; Akbulut e Perros, 2019).

2.4 Desvantagens

Em geral, a arquitetura de microsserviços é percebida como uma forma alternativa de estruturar aplicações que tenta solucionar problemas que costumam surgir em sistemas computacionais monolíticos quando a sua complexidade cresce além de um certo limiar (Taibi *et al.*, 2017; Richardson, 2021b). Porém, a sua escolha como forma principal de estruturar aplicações não vem sem custos. Além das considerações e complicações necessárias para se obter níveis satisfatórios dos benefícios apresentados na seção passada, a própria arquitetura traz consigo algumas desvantagens. Nessa seção, serão exploradas algumas delas.

2.4.1 Complexidade

Um dos fatores negativos mais proeminentes da arquitetura de microsserviços é a sua tendência a aumentar a complexidade geral de um sistema (Kromhout, 2018; Reisz *et al.*, 2020). Ao introduzir um modelo distribuído e de granularidade fina para aplicações,

além da complexidade em manter uma série de processos computacionais independentes (microsserviços), mas que devem coexistir e se relacionar de formas complexas, e uma "malha" de conexões entre os serviços, a arquitetura possui um enorme leque de técnicas, padrões arquiteturais e tecnologias distintas que podem ser incluídas para suportar aspectos específicos da aplicação (Balalaie, Heydarnoori e Jamshidi, 2016b; Richardson, 2021a). De repente, considerações mais estritas sobre limites e dimensões de serviços se tornam fatores importantes à qualidade da aplicação, mecanismos de resiliência passam a ser importantes para a estabilidade do sistema, fluxos para monitoramento serão feitos de forma distribuída, o controle sobre um alto número de instâncias de serviços deverá ser feito por ferramentas de orquestração, dentre muitos outros fatores. Isso tudo também implica num aumento da complexidade operacional desse sistema computacional.

Apesar da conotação negativa, essa complexidade adicional pode ser algo tolerável e até justificável em certos ambientes, enquanto traz mais complicações do que benefícios em outros. Em cenários onde combinam-se o desejo de obter os benefícios possibilitados pela arquitetura de microsserviços com uma organização capaz e disposta a seguir uma boa implementação, o balanço entre essas vantagens e a maior complexidade (além dos outras desvantagens) pode pender para aquele. O mesmo não pode ser dito de outros casos: o *overhead* operacional adicional pode não ser justificável em cenários onde os aspectos positivos trazidos pela arquitetura mal seriam notados (como *startups* em sua fase inicial, onde o escopo e a relevância comercial da aplicação são incertos) ou onde não haveria capacidade ou disposição para garantia de certas competências, como monitoramento básico ou uma cultura de *DevOps* (Fowler, 2014). Nesses casos, a escolha de uma estrutura monolítica, ao menos até que o sistema possa maturar o suficiente, pode fazer mais sentido.

2.4.2 Dados distribuídos

Apesar de ser possível manter um único banco de dados servindo a todos os serviços da aplicação que necessitem de persistência, tal abordagem costuma não ser recomendada em grande parte dos casos (Taibi *et al.*, 2019), uma vez que essa abordagem traz diversas consequências negativas à qualidade final da aplicação (maiores detalhes na Subsubseção 3.2.1). Em seu lugar, o armazenamento de dados acompanha a natureza distribuída dos microsserviços: múltiplos bancos de dados independentes são mantidos, cada um servindo a alguns microsserviços (ou, no máximo, a cada um deles) de forma que a persistência das estruturas de dados de um serviço seja feita separadamente de outro. Em outras palavras, seriam formados banco de dados (BDs) específicos para servirem um (ou mais) microsserviços, assim restringindo o acesso direto dos BDs por agentes externos e somente permitindo um acesso indireto a suas operações através das interfaces expostas pelos serviços.

Contudo, a mera introdução de estados nos microsserviços traz uma série de problemas relacionados à construção de funcionalidades que necessitam do acesso a esses múltiplos bancos. Como apresentado em detalhes na Subsubseção 3.2.1, existem problemas de consistência de dados, quebras de integridade e relacionamentos nos bancos, complicações nas transações envolvendo múltiplos serviços, dentre outros. Por causa dessas questões, uma série de práticas e padrões arquiteturais podem ser aplicados para resolvê-las ou amenizá-

las, ou pode-se até mesmo optar por manter um BD maior como forma de contornar esses problemas de dados distribuídos (Taibi *et al.*, 2018; Newman, 2019).

2.4.3 Aumento nos custos

O uso do modelo de microsserviços traz alguns custos adicionais. Alguns deles estão relacionas à própria natureza distribuída da arquitetura, outros são resultados de escolhas de implementação inapropriadas ao contexto da aplicação. Entre esses possíveis custos, estão:

- ***Performance***: Uma vez que a execução de certas funcionalidades do sistema pode depender da interação entre múltiplos serviços em diversos processos separados, potencialmente em máquinas diferentes, através de redes de computadores (meio bem mais lentas que a memória RAM), o tempo necessário para completar essas operações pode acabar sendo significativamente maior do que quando elas eram realizadas majoritariamente num único processo (Amaral *et al.*, 2015). No entanto, isso nem sempre irá ocorrer, já que outros fatores podem compensar esses atrasos de rede, como o uso interno de tecnologias mais eficientes ou paralelização de passos computacionais custosos em máquinas distintas (se eles forem isolados em serviços próprios, seria possível alocar recursos adicionais somente para essa função);
- **Conhecimento técnico**: A complexidade da arquitetura de microsserviços torna o processo de desenvolvimento nesse ambiente uma experiência também mais complexa. Com as complicações envolvidas na arquitetura dos serviços (comunicação, replicação, *deployment*, etc.), e considerando uma quebra de barreiras técnicas entre as equipes, um nível maior de conhecimento técnico passa a ser requisito para uma boa evolução do projeto. Isso afeta até mesmo ao se pensar no escopo local dos serviços, onde seus desenvolvedores devem estar inteirados ao contexto dos microsserviços para melhor ajustes aos "novos" tipos de trabalho a serem desenvolvidos, como o gerenciamento da transição de dados entre módulos, criação de funcionalidades "degradáveis", dentro outros (Newman, 2015; Newman, 2019);
- **Incompatibilidade do modelo**: Nem todas as organizações estão dispostas a enfrentar as complicações e "pré-requisitos" necessários para uma boa implementação de microsserviços, como um fluxo de monitoramento básico ou rápido provisionamento de novos servidores (Fowler, 2014). Prosseguir a implementação da arquitetura de forma a seguir aplicando antipadrões implicará em diversos custos à qualidade final do sistema (Taibi *et al.*, 2019).

Capítulo 3

Do monolito para microsserviços

Alterar uma estrutura tão básica quanto a arquitetura de uma aplicação já é, por natureza, um projeto não trivial (Balalaie, Heydarnoori e Jamshidi, 2016a). Com tantos fatores complexos e interligados nesse processo, é de se esperar que a migração para microsserviços não seja completamente suave e que ela apresente diversos empecilhos ao longo do caminho, alguns deles relativos a questões técnicas, outros a organizacionais (Larrucea *et al.*, 2018; Fritzsch *et al.*, 2019; Velepucha e Flores, 2021).

Apesar das dificuldades causadas por essas atividades não serem nem universais, nem se manifestarem com a mesma intensidade entre todos os processos de migração para microsserviços (Fritzsch *et al.*, 2019), é importante ao menos estar ciente das formas como elas podem se manifestar, bem como maneiras de solucioná-las (ou ao menos atenuá-las). Com esses conhecimentos, organizações que planejam fazer a migração de seus sistemas para a nova arquitetura estarão melhor capacitadas a prosseguir (ou não) com o projeto de forma menos problemática e, potencialmente, apresentando melhores resultados. Dessa forma, este capítulo terá como foco a apresentação de algumas das considerações, dificuldades e problemas que podem surgir ao longo desse processo.

Ele será aqui dividido em dois momentos principais: um "antes" e "durante" a migração, correspondendo, a grosso modo, a instantes de planejamento e de execução da refatoração do sistema, respectivamente, além de uma pequena discussão sobre o conceito de "depois" da migração no fim do capítulo. Essa divisão, no entanto, não é feita de forma precisa e, em alguns aspectos, pode não ser completamente coerente. Isso porque a forma não-linear como esse processo costuma ser feita (decisões arquiteturais e organizacionais sendo, em teoria, constantemente revistas e aprimoradas) permite, por exemplo, que atividades de "planejamento" como a delimitação de serviços, seja feita gradativamente e possivelmente reformulada em estágios mais avançados do projeto. Além disso, os fatores que demarcariam um possível "fim" da migração podem variar significativamente (como quantidade de módulos ou funcionalidades extraídas, alcance dos objetivos planejados com o projeto, percepção abstrata, etc.) ou até mesmo nem existirem. Assim, essa divisão tem caráter apenas didático, não possuindo representatividade intrínseca em casos reais.

3.1 Antes da migração

Antes mesmo do início da extração de microsserviços e da aplicação de outros padrões arquiteturais auxiliares, uma série de contrapontos podem ser identificados na etapa de planejamento e preparação da migração. Algumas das tarefas associadas a esse momento são: analisar a necessidade e a viabilidade em prosseguir com a transição para microsserviços, estabelecer objetivos concretos para o projeto, aplicar ajustes à estrutura organizacional, analisar a situação atual da aplicação sobre múltiplas perspectivas (tecnologias, fluxos das funcionalidades, processos internos, etc.) e estabelecer a delimitações entre microsserviços.

Como mencionado anteriormente, várias dessas atividades não estão associadas unicamente a um momento específico, sendo possível, por exemplo, que serviços sejam divididos em partes maiores ou menores em qualquer ponto da migração, ou ainda que a forma como as equipes se organizam mude conforme a coleta de experiência "em campo".

3.1.1 Necessidade e capacidade de migração

Com a popularização dos microsserviços e considerando a sua boa integração (ao menos em teoria) com outros conceitos e práticas "modernas" no mercado para sistemas computacionais (Balalaie, Heydarnoori e Jamshidi, 2016a; Verified Market Research, 2021), a migração de sistemas monolíticos para essa arquitetura se tornou uma tendência no mercado (Auer *et al.*, 2021), sendo possível encontrar casos mais extremos onde acredita-se que utilizá-los resolveria "magicamente" os problemas acumulados nas estruturas monolíticas (Richardson, 2019e). No entanto, essa visão otimista dos microsserviços nem sempre considera que transicionar para esse novo modelo traz diversas repercussões ao ciclo de vida da aplicação.

Os impactos que podem ser notados com a reestruturação em microsserviços são diversos, afetando tanto aspectos técnicos e quanto organizacionais da aplicação. Por exemplo, sendo essa arquitetura um tipo de sistema distribuído, o desenvolvimento dos serviços e processos ao seu redor, como testes fim a fim, implantações, monitoramento, segurança, etc., irão precisar ser revistos para se adequarem ao novo modelo e suas "deficiências". Já a organização terá, em teoria, que remoldar o formato e as responsabilidades atribuídas aos times, o que poderá gerar atritos internos (quebras de contrato entre serviços, dependência de outras equipes para implementar funcionalidades consumidas, etc.) (Newman, 2015; Fritzsch *et al.*, 2019).

Apesar da transição gradual para microsserviços permitir que, através de iterações menores, esses impactos sejam identificados e solucionados mais rapidamente (ver Subsubseção 3.2.1), o próprio processo possui custos adicionais, como o desvio de tempo e esforços para o projeto de reestruturação. Considerando esses e os comprometimentos anteriores, é fundamental que, logo de início, haja considerações sobre dois pontos: a real necessidade dessa migração e a capacidade da organização em cumpri-la efetivamente.

Estabelecer esse primeiro ponto significa ponderar se o estado atual do monolito possui certas características indesejáveis, ou não possui algumas desejáveis, e se elas poderiam ser alcançadas ou eliminadas através da adoção de microsserviços como base da aplicação. Isso

poderia se manifestar, por exemplo, quando o custo de manutenção de monolitos com bases de código gigantes se torna proibitivamente caro, de forma que pequenas modificações em um módulo pode afetar outros lugares insuspeitos da aplicação. A modularidade e tamanho reduzido associados aos microsserviços são fatores que poderiam ajudar na resolução desses problemas. Deve-se ter em mente, no entanto, que muitas dessas características dos microsserviços não vêm necessariamente "de graça": assim como melhor explorado no capítulo anterior, muitas das qualidades finais do sistema dependerão de escolhas tomadas durante o processo de migração, de forma que somente a adoção de microsserviços não é suficiente para suprir essas expectativas.

Sobre esse segundo ponto, nem todos os contextos conseguem se adaptar suficientemente bem a esse modelo. Em alguns casos, um domínio da aplicação instável pode resultar em acoplamentos entre serviços por causa de uma má identificação de seus limites (antipadrão *wrong cuts*). Em outros, a introdução da complexidade extra dos microsserviços não é algo viável. Também pode acontecer do projeto ser direcionado de forma incorreta e acabar num estado insatisfatório, como quando ideias pré-concebidas, como a correlação entre mais microsserviços e melhor qualidade da aplicação, são aplicadas cegamente (Richardson, 2019b; Newman, 2019; Etheredge, 2020).

Considerando esses fatores, pode fazer mais sentido, antes de prosseguir com uma migração com indícios de poder ser mais problemática que benéfica, permanecer com um modelo monolítico e tentar aplicar outras práticas e padrões arquiteturais como forma de solucionar deficiências ou aprimorar o estado atual da aplicação. Por exemplo, a procura por uma maior autonomia de times não depende necessariamente de uma separação literal do sistema, podendo ser alcançada também através de uma melhor modularização do monolito e aplicação de políticas organizacionais (como criação de equipes funcionais). Também é possível que complicações relacionadas à escalabilidade possam ser resolvidas, até certo ponto, por métodos convencionais de escala horizontal e vertical (múltiplas instâncias do monolito e máquinas com mais recursos, respectivamente). Essas alternativas nem sempre apresentarão resultados satisfatórios, porém são mais simples de serem instituídas do que um redesenho da aplicação em microsserviços, o que as tornam boas alternativas até que se provem ineficazes (Newman, 2019; Etheredge, 2020).

3.1.2 Objetivos

Após analisar a viabilidade da transição para microsserviços e identificar deficiências e problemas atrelados à estrutura monolítica da aplicação, é importante que haja uma consolidação dos objetivos desse projeto. Ela irá guiar, em parte, o tipo de abordagem (tanto técnica quanto organizacional) que deverá ser tomada durante esse processo para que os benefícios da arquitetura possam ser alcançados em níveis expressivos. Por exemplo, em situações onde muito valor é colocado sobre a resiliência a falhas, práticas e mecanismos específicos para evitar congestionamento dos canais de comunicação ou para provisionar novas instâncias de serviços (em caso de queda dos *hosts*) podem ser de grande interesse. Em outra situação, caso a auditoria do sistema seja um fator de alta relevância, padrões arquiteturais como *event sourcing* (Fowler, 2005) podem ser úteis.

Os objetivos para a transição para microsserviços são variados e cada caso pode ser motivado por um conjunto diferente deles. Alguns dos principais observados na indústria

são: maior manutenibilidade, melhor escalabilidade dos serviços, uma maneira mais eficiente de dividir times e o trabalho pelo qual cada um deles seria responsável, e a capacidade de entregar alterações mais frequentes (Taibi *et al.*, 2017).

No entanto, alguns pontos podem ser levantados sobre a abrangência desses objetivos: tanto não definir nenhum, quanto escolher muitos, tem suas próprias desvantagens. No primeiro caso, não atentar às necessidades reais da aplicação e prosseguir com a mentalidade de que os microsserviços são o objetivo em si próprios pode ser algo prejudicial ao seu futuro. Ao se concentrar unicamente em "ter microsserviços" (ou seja, como implementar tecnicamente a estrutura de microsserviços), outras questões tão importantes para o processo de migração passariam a ser desconsideras, como alterações necessárias à estrutura organizacional ou a adequação da aplicação a práticas de desenvolvimento de *software* básicas. Isso poderia afetar negativamente o projeto (Richardson, 2019a; Richardson, 2019d).

Já no segundo caso, estabelecer um grande número de objetivos poderá trancar os microsserviços como única alternativa viável (Newman, 2019), ou ainda levar a um gasto desnecessário de esforços em objetivos de menor importância para o contexto da aplicação. Por exemplo, Leppanen *et al.* (2015), através de entrevistas com empresas de tecnologia, conclui que práticas como *deploys* contínuos e frequentes, práticas facilitadas pelos microsserviços, nem sempre são possíveis ou mesmo desejáveis.

3.1.3 Movimentação organizacional

Como discutido anteriormente na Subseção 2.2.3, a migração para uma arquitetura de microsserviços não envolve somente aspectos técnicos: a estrutura organizacional de empresas ou grupos responsáveis por esses sistemas computacionais também precisa se adequar às particularidades do padrão arquitetural. Dessa forma, há a necessidade de se transicionar de modelos organizacionais tradicionalmente "monolíticos", onde equipes são mantidas em "silos" técnicos (equipes separadas entre desenvolvimento, testes, *deployment*, etc.) (Balalaie, Heydarnoori e Jamshidi, 2016a; Kalske *et al.*, 2018) e alterações nos serviços requerem coordenação constante entre múltiplos times (Richardson, 2019c), para modelos que melhor acompanhem a estrutura dos microsserviços.

Porém, iniciar e desenvolver essas mudanças organizacionais não são tarefas simples. Os problemas e dificuldades específicos que poderiam surgir dessas alterações irão variar conforme o contexto: em alguns casos, a empresa já está estruturada de tal forma que a transição para a nova arquitetura não necessitará, ao menos no quesito organizacional, de grandes etapas adicionais (cultura de DevOps bem estabelecida, divisão funcional de times, etc.), enquanto em outros, alterações mais profundas podem ser necessárias, como a substituição de metodologias de desenvolvimento incompatíveis com a arquitetura (modelo *waterfall*, por exemplo) (Fritzsch *et al.*, 2019).

Além disso, a grande variedade de particularidades que cada empresa possui pode interagir de formas não óbvias com os modelos de microsserviços, causando dificuldades e problemas específicos para organizações diferentes. Apesar disso dificultar uma generalização das possíveis complicações que uma organização pode experienciar, abaixo estão algumas razoavelmente comuns (Fritzsch *et al.*, 2019; Newman, 2019):

- **Adaptação de times**: Em níveis maiores ou menores, a mudança para microsserviços traz fardos adicionais sobre as equipes. Considerando as práticas organizacionais estabelecidas anteriormente ao início do processo de migração, pode haver uma certa dificuldade para a adoção de novos modelos de trabalho. Por exemplo, em ambientes onde prevalece a cultura de "linha de produção", com um artefato sendo passado de equipe em equipe com propósitos diferentes (de desenvolvedores para times de teste, depois para setor de garantia de qualidade, depois para equipe de *deployment*), a transição para microsserviços e um novo modelo organizacional onde um time reúne todas essas responsabilidades podem requirir algum tempo de ajuste para indivíduos e processos internos. Nesse sentido, a colaboração entre times mais autônomos também pode ser desafiadora, considerando que esse é um tipo diferente de interação se comparado ao modelo de trabalho anterior;

  Outra complicação nesse sentido seria a mudança de mentalidade, especificamente sobre aceitar que, num ambiente complexo e nem sempre previsível dos microsserviços, erros e falhas provavelmente irão ocorrer. A divisão dos serviços, a adoção de tecnologias e práticas, as mudanças nos processos internos de desenvolvimento, dentre as outras diversas atividades envolvidas na migração para microsserviços, nem sempre podem ser cumpridas da melhor forma, até mesmo porque não existe uma forma universalmente boa de prosseguir com essa transição (Etheredge, 2020). Para mitigar esse problema, às vezes é possível reduzir o escopo dos impactos desses erros através de uma migração gradual para microsserviços (como apresentado na Subsubseção 3.2.1);

- **Descontrole de visão global da aplicação**: Com a descentralização de responsabilidades e da governança de aplicações entre diversos microsserviços, problemas podem surgir ao tratá-las num escopo mais global. Um deles está relacionado à compreensão geral do sistema: dependendo do número de serviços e da complexidade de comunicação envolvida na formação de funcionalidade, o entendimento do funcionamento interno em alto nível pode ser prejudicado (por exemplo, elementos como autenticação, autorização e processamento de transações distribuídas entre serviços pode envolver fluxos não triviais). Também é possível que a comunicação de decisões e de visões necessárias para ações coordenadas entre múltiplos serviços seja algo de difícil propagação, a depender do nível de autonomia dada às equipes;

- **Convencer organização a adotar microsserviços**: Ao mesmo tempo em que algumas empresas decidem migrar para microsserviços sem compreender a escala dos impactos finais ao produto, outros sistemas que poderiam se beneficiar desse modelo não o fazem. Várias razões poderiam ser dadas: inércia a mudanças a nível empresarial, percepção de que o aumento nos custos e na complexidade da aplicação "não valem a pena", escolha no investimento de outras "soluções" para problemas encontrados no monolito, dentre outras. Porém, é possível que alguns vejam na migração para microsserviços um potencial para uma mudança positiva de um *status quo* problemática com o monolito. Dessa forma, seria necessário um trabalho adicional para convencer a organização desse potencial e liderar as mudanças necessárias.

  Isso, no entanto, teria suas próprias dificuldades. Primeiramente, investir na migração

para microsserviços implicaria na alocação de uma certa quantia de trabalho adicional na extração dos componentes e ajustes de processos internos, desviando parte do foco no desenvolvimento funcional da aplicação. Além disso, dependendo de como o projeto é guiado, pode levar um certo tempo até que os benefícios da arquitetura possam ser apreciados. Para tanto, Newman (2019, p. 47-52), através da aplicação do modelo detalhado por Kotter (1996) envolvendo microsserviços, apresenta como o processo de adoção da arquitetura pode ser conduzido para implementar mudanças necessárias na organização.

3.1.4 Análise da estrutura monolítica

Considerando que o processo de migração para microsserviços envolve a alteração de vários aspectos do sistema monolítico, torna-se evidente a importância de um bom entendimento da implementação atual. Para solidificar o processo de migração, enquanto o "caminho" a ser percorrido é guiado pelos objetivos selecionados anteriormente, a sua "construção" será guiada pelo estado atual da aplicação, influenciada por elementos como as interfaces de comunicação entre módulos internos, implementação da base de código, fluxos de execução (de funcionalidades, segurança, *reporting*, etc.) em alto e baixo níveis, processo de *deployment*, escolhas tecnológicas, dentre vários outros.

Apesar das várias informações que podem ser extraídas do sistema monolítico e que podem ser úteis à migração, nem todos esse detalhes precisam necessariamente serem apresentados de antemão. É possível tomar uma abordagem mais gradual e modular, onde detalhes sobre a implementação atual (de alto e baixo níveis) são desvendadas e aprofundadas ao longo do tempo, de acordo com as necessidades da atividade atualmente em foco. Por exemplo, em momentos inicias da migração, onde deseja-se delinear superficialmente o estado atual da aplicação como forma de nivelar esse conhecimento ao longo da organização, é possível apenas focar na compreensão, a nível macro, de características como as tecnologias utilizadas, os componentes existentes e o processo de *deployment* já estruturados (Balalaie, Heydarnoori, Jamshidi *et al.*, 2018). No entanto, durante a extração de um microsserviço, informações técnicas mais específicas àquela parte do sistema seriam necessárias.

No entanto, obter essas informações nem sempre é algo tão simples. A dificuldade na extração de certos níveis de informações sobre a aplicação e processos correlatos pode variar consideravelmente entre implementações específicas. Listar as tecnologias utilizadas ou descrever o processo de *deploy* são tarefas relativamente diretas, enquanto compreender tecnicamente as interações entre certas partes do sistema ou as possíveis cadeias de execução de certas funcionalidades podem ser tarefas mais árduas. Nesses últimos casos, é possível, por exemplo, que uma série de antipadrões e *bad smells* tenham se acumulado ao longo tempo, reduzindo assim a compreensibilidade daquela seção.

3.1.5 Modelagem de serviços

Antes que a extração de serviços do monolito possa ser iniciada, é necessário haver uma etapa preparatória que consistirá na modelagem dos microsserviços a serem criados. Isso envolve primeiramente definir os seus limites operacionais, ou seja, delimitar quais

funções são de responsabilidade de qual serviço (através de divisões lógicas na aplicação monolítica ou em seu domínio). Também faz-se necessário a formação da interface de comunicação com outros serviços, permitindo o contato entre microsserviços de uma forma relativamente padronizada e abstraída de seus detalhes internos. Após essas duas tarefas serem completas, os microsserviços modelados estariam prontos para serem implementados e integrados a outros serviços (Pautasso *et al.*, 2017).

No entanto, obter uma modelagem satisfatória nem sempre é uma tarefa trivial. A seguir, serão apresentados dois fatores de complicação para o processo de modelagem, bem como dois problemas causados por más modelagens:

- **Granularidade**: Um dos fatores que influenciam no escopo dos microsserviços é o nível desejado de granularidade do sistema, onde se estabelece um balanço entre a especialização e o número total de serviços (Subsubseção 2.2.1). A escolha por um ou outro tem suas vantagens e desvantagens. Por exemplo, ter um número menor de microsserviços abordando porções maiores do domínio da aplicação pode ser mais interessante em certos cenários, como quando a falta de conhecimento sobre as especificidades do domínio levariam a problemas de "cortes errados" (decomposição errada, como explorada a seguir), ou quando a organização, talvez por inexperiência com a arquitetura, decide evitar as complicações relacionadas a gerenciar um grande número de microsserviços (Subsubseção 3.2.2). Porém, essa escolha também traz pontos negativos, como prejuízos em sua manutenibilidade, causada pelo aumento da base de código, e em sua independência, visto que uma maior parte da aplicação estaria funcionando sobre o mesmo "bloco", impedindo, por exemplo, que partes menores possam ser implantadas separadamente.

  Optar pela outra alternativa também traz outros compromissos, como a troca de serviços mais enxutos e (possivelmente) mais independentes por um maior *overhead* arquitetural e uma possível maior latência das funcionalidades, fruto de atrasos na comunicação pela rede;

- **Priorização de serviços**: Quando o processo de migração é feita de forma incremental, é esperado que apenas uma pequena parte do sistema monolítico seja extraída para a nova arquitetura de cada vez, assim coexistindo, por um longo período, duas macroestruturas: um conjunto de microsserviços que já foram extraídos e um monolito com os serviços que ainda não foram. Dessa forma, considerando essa abordagem iterativa (o que nem sempre é aplicável, como apresentado adiante na Subsubseção 3.2.1), faz-se necessária uma ordem de extração dos microsserviços;

  A priorização dos serviços é uma tarefa que dependerá de diversos fatores, inclusive da disposição dos componentes no monolito (Newman, 2019, p. 58-60). De forma mais genérica, Newman (2019, p. 60-62) apresenta dois desses fatores a serem maximizados: a facilidade e os benefícios trazidos pela decomposição dos microsserviços. Nem sempre é possível ter ambos, sendo assim necessário que a priorização seja feita através do balanço desses fatores, juntamente de outras características e objetivos. Por exemplo, enquanto pode ser interessante escolher serviços menos problemáticos de serem extraídos como forma de melhor introduzir equipes mais inexperientes aos microsserviços, priorizar os que trarão maior valor imediato pode servir para impulsionar a migração através da demonstração de sua efetividade;

- **Decomposição errada**: Sejam quais forem as técnicas utilizadas para decomposição de uma aplicação em possíveis candidatos a serviços, nem sempre essa divisão será feita da melhor forma possível. Em métodos "manuais", como divisões funcionais através das técnicas de DDD, é possível chegar a múltiplos resultados do particionamento, visto que eles estão ligados às conclusões extraídas de (um grupo de) indivíduos. Nos automáticos [como em Mazlami *et al.* (2017)], o processamento algorítmico nem sempre consegue capturar todos os fatores de interesse da organização (por exemplo, o desejo por uma menor complexidade arquitetural). Nos semi-manuais, um pouco dos problemas de ambos pode ocorrer.

  Em alguns casos, isso pode levar a uma divisão indevida dos microsserviços, onde dois ou mais deles são atribuídos a responsabilidades próximas o suficiente para que essa separação seja vista, de certa forma, como problemática. Alguns indícios de que isso estaria afetando certos serviços são: um alto acoplamento entre eles, expresso por níveis inadequados de compartilhamento de suas estruturas internas com agentes externos, e uma baixa coesão, expressa pela necessidade constante de alterações síncronas entre múltiplos serviços quando a modificação de detalhes (supostamente) internos afeta a operação de unidades externas. Assim, algumas das consequências dessa má divisão podem ser notadas na redução de independência dos microsserviços afetados, o que leva à necessidade de maiores e mais frequentes coordenações entre os responsáveis pelos serviços. O custo, por exemplo, de ter que realizar alterações em múltiplos serviços por causa desse problema pode ser significativo (Newman, 2015; Kalske *et al.*, 2018);

- **Decomposição sem considerar interfaces**: Uma boa modelagem de serviços não depende somente de uma boa partição do domínio (ou dos dados) da aplicação, mas também precisa de uma interface de comunicação entre microsserviços bem estruturada. Sem que esse passo seja feito juntamente da repartição do monolito, problemas relacionados a quebra de consumidores podem surgir (Pautasso *et al.*, 2017).

3.2 Durante a migração

Após atividades da etapa de planejamento, inicia-se "de fato" a transição da arquitetura monolítica para a de microsserviços. O momento de considerar a viabilidade do projeto já ocorreu e, em situações ideais, foram estudadas e iniciadas (talvez em uma escala menor) atividades como alterações na estrutura organizacional, análise geral da aplicação, priorização dos serviços a serem extraídos e modelagem dos primeiros microsserviços. Após isso, além de um possível retorno dessas atividades sob uma escala maior ou de formas mais eficazes, o sistema computacional monolítico passará por uma série de modificações em sua estrutura para adaptá-lo ao padrão arquitetural de microsserviços.

No entanto, assim como durante a "fase de planejamento", as mudanças vistas durante essa etapa são acompanhadas por uma série de problemas e dificuldades próprias. Seja a migração de código e de dados, a reestruturação de processos (como os de *deployment* e testes) ou ainda o redesenho de fluxos da aplicação (como monitoramento e segurança), redesenhar o sistema monolítico em microsserviços na prática envolve uma variedade de

tarefas nem sempre triviais. Dessa forma, nessa seção, serão apresentados algumas dessas atividades e suas complicações.

3.2.1 Transformando o monolito em microsserviços

De todos os passos envolvidos nesse longo processo de migração para microsserviços, a extração de funcionalidades do monolito e a sua subsequente realocação em serviços independentes são duas das principais atividades. Nele, serão identificadas as seções da base de código e do(s) banco(s) de dados que devem ser extraídas da estrutura monolítica e "realocadas" para os microsserviços modelados durante a fase de planejamento (Subseção 3.1.5). Assim, de forma gradual, as responsabilidades do monolito serão distribuídas entre os novos microsserviços e (parte das) funcionalidades da aplicação passarão a funcionar num contexto distribuído.

Na prática, para realizar essa reestruturação, existem uma série de estratégias diferentes, cada uma melhor aplicável em certos contextos e de acordo com a implementação do monolito, que podem auxiliar nesse processo. Porém, mesmo com esses "guias", dificuldades e problemas relacionadas à extração dos microsserviços ainda podem surgir, seja pelo uso indevido dessas estratégias, seja por compromissos que elas naturalmente impõem.

Big Bang ou incremental?

Existem duas vias pelas quais a migração para microsserviços pode ocorrer: utilizando o modelo *Big Bang* ou o incremental. No primeiro caso, a aplicação é recriada "do zero", de forma completamente separada da versão monolítica (que continua a servir os clientes finais) e com as funcionalidades que já existiam sendo replicadas através de uma reescrita completa. No segundo, a aplicação é progressivamente alterada em direção à arquitetura de microsserviços, onde funcionalidades antes implementadas no monolito passam a ser implementadas em serviços independentes, o que, com o tempo, formará a nova "versão" do sistema.

Em geral, a estratégia de migração "*Big Bang*" costuma ser desencorajada em grande parte dos casos (Janes e Russo, 2019; Newman, 2019). Isso porque ela traz consigo, especialmente em contextos empresariais, algumas dificuldades indesejáveis, como problemas na replicação exata de funcionalidades da aplicação, um distanciamento gradual entre as versões antiga (monolito) e nova (microsserviço) e um potencial para danos à organização (causados por mudanças abruptas de processos internos entre as versões) (Finnigan, 2018; Opus Software, 2020). Porém, em algumas circunstâncias não usuais, a escolha desse método pode fazer mais sentido, como em situações onde a base de código tem tamanho reduzido ou a transição para microsserviços é vista como algo urgente (Bozan *et al.*, 2021).

Em seu lugar, a estratégia incremental possui diversas vantagens que a tornam mais adequada para grande parte dos contextos. Por exemplo, ela possibilita um aprendizado gradual do modelo de microsserviços e como ele se comporta em um ambiente "real" (como o de produção), permite que os pontos positivos da migração sejam observados mais rapidamente (e não somente após o "fim" da transição, como na estratégia anterior) e reduz a quantidade de discrepâncias entre funcionalidades nas duas versões (enquanto a migração

está sendo feita, a versão do monolito pode evoluir) (Newman, 2019; Opus Software, 2020). Assim, apesar desse modelo ainda apresentar diversas complicações, explorados ao longo deste capítulo, esses e outros benefícios tornam essa estratégia interessante para diversos contextos.

Primeiro partição de código ou banco de dados?

No processo de criar um microsserviço a partir do monolito existente, é possível destacar dois macrocomponentes que precisam ser trabalhados: a base de código e o banco de dados. Subdivisões subsequentes, como a separação da base de código em *front-end* e *back-end*, podem ser feitas e possuem um certo valor, visto que, por exemplo, existem alguns padrões arquiteturais próprios da camada de apresentação (Newman, 2015; Jackson, 2019). No entanto, este trabalho irá focar nessa primeira divisão, uma vez que cada uma apresenta vastas diferenças entre o tipo de trabalho e quais complicações podem ser percebidas. Enquanto atividades relacionadas à codificação da lógica dos microsserviços estão focadas na replicação de funcionalidades, nos ajustes das comunicações entre serviços (ou entre eles e o monolito) e em lidar com os desafios vindos da distribuição da lógica da aplicação, as relacionadas ao armazenamento de dados devem lidar com a separação lógica (e possivelmente física) de BDs (ou seus *schemas*) e à resolução de questões associadas à distribuição dos dados (como sincronização entre as versões antiga e nova do BD e quebras de garantias transacionais).

Considerando a importância da eventual separação do BD (Subsubseção 3.2.1), existem três formas de prosseguir com a migração para microsserviços: extrair a lógica da base de código e então separar o banco de dados apropriadamente, começar com a divisão do banco de forma alinhada com os limites definidos entre serviços e então propriamente codificá-lo, ou fazer ambos simultaneamente. Cada uma dessas abordagens possui vantagens e desvantagens próprias: a primeira costuma ser mais simples e apresentar resultados mais imediatos, mas pode resultar num adiamento indefinido da divisão do BD; a segunda evita impactos aos consumidores da aplicação (assumindo que não há acesso direto ao BD por agentes externos) e reduz o número de problemas futuros relacionados a dependências de serviços com as estruturas do banco monolítico, mas introduz complicações associadas à distribuição de dados logo no início do projeto de migração; e a última apresenta vantagens dos dois outros métodos, mas pode ser um passo grande demais, levando mais tempo para que cada microsserviços seja completo. Dessa forma, a decisão por qual estratégia seguir dependerá do contexto local, incluindo fatores como prioridades, expectativas e nível de preparo dos times (Newman, 2019).

Separação da base de código

O primeiro passo da separação do monolito é escolher qual será o próximo serviço a ser extraído. Como discutido na Subseção 3.1.5, essa priorização pode ser feita pelo balanço de dois fatores associados aos microsserviços: valor de retorno esperado e facilidade de extração. Sob uma perspectiva mais técnica, fatores como as dependências entre diferentes partes da aplicação e a qualidade de código podem também ser inclusas nessa priorização. Por exemplo, é recomendável que se priorize a repartição de serviços relativamente desacoplados (por serem mais fáceis de extrair sem muita disrupção) e que minimizem

a dependência com partes do monolito que ainda não foram extraídas (de forma a evitar maiores acoplamentos) (Newman, 2019). Em relação à extração de componentes mais centrais ao funcionamento da aplicação, é interessante focar naqueles que servem uma multitude de propósitos em diversos subdomínios. Um exemplo seria o conceito de "sessão" de um usuário, potencialmente associado a uma série de elementos do sistema, como carrinho de compras ou páginas visitas (Dehghani, 2018).

Uma outra preocupação ao tentar extrair parte da base de código em seus próprios microsserviços é escolher como as funcionalidades removidas do monolito serão importadas nos novos serviços. Duas formas distintas podem ser utilizadas: migração de parte da base de código para os novos microsserviços, seguido de sua refatoração para adaptá-lo à nova estrutura, ou então reescrita da funcionalidade "do zero". O uso de um modelo ou outro dependerá do estado atual da implementação: caso as partes envolvidas na migração sejam de qualidade (boas práticas de desenvolvimento, um nível aceitável de *code smells*, etc.), realocar o código entre monolito e microsserviço pode ser a melhor opção, assim economizando esforços e evitando as complicações de garantir paridade entre as duas versões. Caso contrário, reescrever a funcionalidade diretamente nos novos serviços pode ser mais interessante. Além desses, outro método viável seria refatorar o código no monolito e então transferi-lo a um novo microsserviço (Dehghani, 2018; Newman, 2019).

Por fim, temos as complicações de escolher e utilizar padrões de migrações apropriadamente. A estratégia incremental, apresentada na Subseção 3.1.5, pode, na verdade, ser realizada através de diferentes técnicas. Essencialmente, elas descrevem procedimentos gerais que podem ser aplicados para extração de serviços de forma gradual.

Visto que cada uma delas opera melhor em certos contextos e servem propósitos distintos, é razoável que múltiplos padrões sejam utilizados ao longo da migração do sistema. Por exemplo, uma dos mais conhecidos, o padrão "*Strangler Fig*", onde funcionalidades extraídas são mantidas por um tempo ao lado das implementações no monolito e a transição entre uma versão e a outra se baseia no redirecionamento de chamadas, é melhor utilizado para migração de serviços autocontidos (sem nenhuma dependência com outros módulos) ou de vários simultaneamente (abrangendo uma cadeia de chamadas), e pode ser utilizado em situações onde o código-fonte do monolito é inacessível a alterações. Em contraste, o padrão "*Branch by Abstraction*", que se baseia no uso de interfaces como forma de rapidamente trocar entre as implementações do monolito e do microsserviço, é melhor aplicado na reestruturação de módulos que estão numa posição profunda na cadeia de dependências do sistema, com múltiplos consumidores diretos e/ou indiretos que ainda não foram extraídos. Já o padrão "*Parallel Run*" dita como a execução em paralelo de ambas as versões pode ser estruturada para comparar os seus resultados (Newman, 2019).

Separação do banco de dados

Em sistemas classificados como monolíticos, é comum que, além da lógica, o armazenamento de dados também seja feito de forma monolítica (Velepucha e Flores, 2021). Isso se manifesta através de um único BD, geralmente do tipo relacional, que serve todos os módulos do sistema, ou seja, todas as operações de acesso e modificação de dados que diferentes partes da aplicação podem necessitar são executadas sobre uma mesma unidade lógica do banco de dados central.

O principal problema com essa organização é que, uma vez que os detalhes de implementação das estruturas internas do BD (como as colunas de tabelas) são acessíveis e podem ser referenciadas por qualquer agente interno e/ou externo ao banco, há uma maior tendência a acoplamento de diferentes partes do sistema com detalhes internos do banco (Newman, 2019). Por exemplo, um serviço pode ter acesso direto a tabelas "fora de sua jurisdição" (ou seja, que não estejam diretamente relacionadas a sua lógica) ou uma tabela pode estar diretamente vinculada a outras através de restrições de chaves estrangeiras. Em ambos os casos, diferentes partes da aplicação passam a depender de estruturas internas ao BD, tornando mais provável que alterações nelas quebrem múltiplos módulos da aplicação. Mesmo em situações onde mais de um banco de dados seja utilizado (outros tipos, como os orientados a documentos, podem ser utilizados em contextos mais específicos), as complicações desse acoplamento continuarão a surgir enquanto não houver restrições de acesso às estruturas de dados internas.

Ao realizar a migração para microsserviços, a recomendação é que esse BD seja repartido em múltiplas instâncias lógicas (e possivelmente físicas), acompanhando as delimitações dos serviços (Lewis e Fowler, 2014). Dessa forma, em teoria, cada unidade que necessitasse de persistência teria sua própria "instância" de um BD, responsável por armazenar dados relativos ao serviço em questão, e o acesso a operações (de consulta e modificação) sobre dados externos estaria limitado ao que for exposto pela interface de outros serviços. Dessa forma, a estrutura interna do banco pode evoluir independentemente, uma vez que modificações irão afetar somente o serviço ao qual o BD esteja associado.

Contudo, realizar de fato essa transição está longe de ser um processo simples. Sem considerar os inúmeros problemas e dificuldades que surgem como consequência da distribuição de dados, o trabalho de repartir gradualmente a estrutura de um banco de dados possui um número de desafios próprios. Dessa forma, uma possibilidade seria simplesmente continuar a migração para microsserviços somente da parte lógica dos serviços, mantendo um banco de dados compartilhados entre esses serviços (padrão *Shared Database*). Porém, ao seguir por esse método, os problemas de acoplamento associados a banco de dados monolíticos continuarão a evoluir, atrapalhando na busca do princípio de maior independência dos microsserviços. Sendo esse padrão até mesmo ser visto como um antipadrão (Taibi *et al.*, 2019), poucos cenários se beneficiariam dessa prática.

No entanto, essa repartição nem sempre é viável a curto prazo. Ela pode levar um bom tempo para ser completa, especialmente ao levar em consideração que algumas áreas da aplicação são de tamanha importância para o funcionamento básico do negócio (e, por vezes, implementadas de maneira tão frágil a mudanças) que uma atividade complexa e tão propensa a erros como a distribuição gradual do BD central (envolvendo quebras de relacionamentos entre entidades, migração dos dados já existentes, garantias de sincronização entre versões durante desenvolvimento, etc.) deve ser feita com o maior cuidado. Assim, em certas situações, alguns padrões "paliativo" podem ser aplicados para reduzir a "superfície" de acoplamento entre serviços e o banco de dados monolítico: pode-se utilizar o conceito de "visões" como ponto de acoplamento, ao invés da implementação real (padrão *Database View*), ou um novo serviço pode ser criado como *proxy* entre todos os serviços e o BD central (padrão *Database Wrapping Service*), ou ainda, em casos onde serviços necessitem de acesso direto a uma estrutura de banco de dados, é possível realizar um mapeamento da implementação atual para um novo *schema*, de acordo com as necessidades dos serviços

consumidores (padrão *Database-as-a-Service Interface*) (Newman, 2019).

Também é importante ter em mente alguns dos problemas originados da distribuição de dados que afetam a operação normal da aplicação. Alguns deles são (Newman, 2019):

- **Chaves estrangeiras fracas**: Um relacionamento entre entidades, antes modelada no próprio "Sistema Gerenciador de Banco de Dados" (SGBD) e possivelmente cumprindo com certas restrições (de integridade, por exemplo), passaria a ser expresso sem essas restrições automaticamente, de forma que a codificação desse relacionamento ficaria apenas implícita. Por exemplo, um serviço que registra as atividades dos usuários durante a utilização de uma aplicação poderia ter que acessar os dados referentes a usuários em outro serviço, porém ele pode ter sido deletado do BD em algum momento. A forma de lidar com isso dependerá da implementação de lógica nos microsserviços, sendo algumas das opções: permitir a existência de dados indefinidos, copiar localmente campos de interesse ou forçar restrições entre serviços;
- **Possível aumento de latência**: Ao distribuir múltiplos bancos de dados de forma que somente um microsserviço tenha acesso direto a ele, a realização de operações que no BD monolítico necessitariam do acesso a várias entidades (tabelas), como em consultas complexas evolvendo múltiplos operadores relacionais *join*, agora poderia necessitar da comunicação entre vários microsserviços com o objetivo de acessar indiretamente bancos externo, assim iterativamente obtendo parte dos dados ou realizando parte das atividades que fazem compõem uma funcionalidade do sistema. Uma vez que essa comunicação é feita através da rede, ao invés de ser realizada no mesmo processo, seria esperado um aumento na latência em certas funcionalidades da aplicação. Algumas medidas podem ser tomadas para tentar mitigar isso, como o investimento em mecanismos de *caching*, assim reduzindo o tempo de acesso a dados, ou até mesmo a não separação de certas partes do banco de dados, evitando assim a necessidade de realizar uma comunicação através da rede;
- **Teorema *Consistency, Availability, and Partition Tolerance* (CAP)**: Com a transformação da aplicação monolítica em um sistema distribuído, uma série de complicações relacionados aos dados podem ser observadas tanto durante sua transição entre microsserviços (como parte da execução de uma funcionalidade), quanto no momento em que são armazenados nos bancos de dados. Uma delas é consequência do teorema CAP, que força qualquer sistema distribuído a priorizar duas de três características: consistência, disponibilidade e tolerância a partições (onde, no contexto dos microsserviços, o sacrifício desse último elemento não faria sentido, pois implicaria na união deles em um único processo).

  Para qualquer das escolhas, compromissos deverão ser estabelecidos, com as consequências negativas se manifestando principalmente durante falhas de serviços ou da comunicação entre eles. Sacrificar a consistência de uma operação implicaria num maior tempo de disponibilidade das funcionalidades, porém introduz a possibilidade de que dados inconsistentes com a "fonte da verdade" sejam utilizados por um tempo, até que eventualmente eles sejam padronizados ao resto do sistema. Enquanto isso, sacrificar disponibilidade significa que dados serão consistentes entre todos os módulos, mas implicando na interrupção temporária de funcionalidades quando

certos serviços não forem acessíveis.

Qual seria a opção mais adequada dependerá do quão aceitável, no contexto da operação, seria ter um modelo de consistência eventual. Além disso, dado que padrões arquiteturais que priorizam consistência sobre disponibilidades, como o algoritmo de *commits* em duas fases (*two-phase commits*, em inglês), são complexos e ainda imperfeitos, uma outra alternativa seria simplesmente evitar a separação de serviços que poderiam se encontrar nesse cenário, caso a consistência de dados a tomo momento seja fator crítico;

- **Perda de atomicidade de transações**: Algumas funcionalidades de sistemas complexos podem ser constituídas de múltiplos passos intermediários que abrangem vários subdomínios diferentes, realizando operações sobre dados persistentes e que precisam ser feitas em uma ordem específica. Um exemplo seria a realização de compras de produtos em lojas virtuais, tarefa que poderia envolver atividades em microsserviços de pagamento, usuário, inventário interno, etc. Numa estrutura monolítica, esse tipo de funcionalidade poderia ser construída ao redor de uma operação transacional, aproveitando de suas propriedades "Atomicidade, Consistência, Isolamento e Durabilidade" (ACID) e assim evitando um número de potenciais problemas. No entanto, ao distribuir os passos intermediários entre microsserviços, essas propriedades são perdidas.

  Mais especificamente sobre a atomicidade perdida, isso é uma consequência da possibilidade de falhas independentes em microsserviços responsáveis por algumas das tarefas intermediárias da transação. Dessa forma, alguns dos passos que constituem uma funcionalidade abrangendo múltiplos serviços podem não ser completos, enquanto outros seriam. Isso pode ser um problema grave, uma vez que, caso haja mudanças de estado em alguma das etapas, a funcionalidade geral até pode cancelar a operação que falhou e as que seriam iniciadas adiante, mas os serviços que já concluíram suas atividades e "commitaram" as alterações para seus BDs não podem naturalmente revertê-las, como seria possível fazer dentro de uma operação transacional em um único banco de dados. Ao deixar inacabado o processamento da funcionalidade, problemas de inconsistências ao longo da aplicação podem começar a surgir. Dito isso, existem padrões que tentam evitar esse problema, como as transações distribuídas (*two-phase commits* seria um exemplo), ou outros que tentam minimizar os danos do cenário descrito, como o modelo de "sagas" e suas transações compensatórias.

3.2.2 Ajustando operações

Durante a migração para microsserviços, a simples realocação de funcionalidades do sistema em vários módulos distribuídos não é característica suficiente para compor uma arquitetura de microsserviços, ainda mais uma que possa tirar proveito dos seus benefícios "prometidos". Com a sua introdução, a forma como vários processos e práticas associados ao ciclo de vida da aplicação, como a utilização de testes ou fluxos de monitoramento interno, são estruturados pode divergir significativamente dos modelos voltados a aplicações monolíticas. Dessa forma, serão apresentados alguns dos problemas relativos a mudanças nas operações internas que podem ser notadas durante essa transição.

Incorporando tecnologias e melhores práticas

Uma das consequências do particionamento da aplicação monolítica em microsserviços é a crescente necessidade do uso de "novas" tecnologias. Antes do início desse projeto, uma *stack* de ferramentas já foi selecionada, configurada e utilizada, possivelmente por um longo tempo, na versão monolítica. Porém, em vista das diferenças arquiteturais entre ambas as versões, novas tecnologias podem se fazer necessárias para melhor acomodar a aplicação à estrutura de microsserviços. Por exemplo, mecanismos de comunicação assíncrona por mensageria (como Kafka ou RabbitMQ) podem ser empregados em parte das integrações entre serviços, a replicação de serviços entre ambientes pode ser facilitados pelo uso de *containers* como Docker, a gestão das instâncias dos serviços pode ser feita com Kubernetes, o monitoramento e visualização de métricas do sistema podem ser suportados pelo Prometheus e Grafana, dentre muitas outras ferramentas aplicáveis em diversos setores, como *pipelines* para implantação, balanceamento de carga, testes fim a fim, etc.

De forma similar, a maneira como o sistema é desenvolvido requer certa atenção. Ao introduzir uma comunicação distribuída e dependente de um meio de comunicação menos confiável, algumas práticas adicionais podem ser necessárias para tentar facilitar a evolução da aplicação ou para evitar algumas complicações. Por exemplo, técnicas de transações compensatórias e *circuit breakers* podem ser utilizadas para aumentar a resiliência dos sistemas, enquanto componentes como *API Gateway* e *Service Discovery* ajudariam na simplificação da comunicação entre *front* e *back-end* e na visibilidade de instâncias de serviços na rede, respectivamente.

No entanto, introduzir várias dessas tecnologias em um ambiente já consolidado pode se mostrar um desafio considerável. Primeiramente, a sua adoção exigiria uma série de reconfigurações e reajustes para adequação ao contexto atual e suas exigências (tempo de funcionamento, carga de uso, normas de segurança e privacidade, etc.). Em segundo lugar, a falta de familiaridade ou experiência de equipes com a utilização das novas tecnologias e práticas pode levar a más implementações e complicações futuras. Por fim, faz-se necessária a presença de entidades centrais na organização que imponham certas restrições e regulamentações para a construção de microsserviços, evitando a adoção de muitas *stacks* de tecnologias diferentes no mesmo ambiente (Newman, 2019), e para a aplicação de padrões arquiteturais que envolvem a aplicação inteira.

Overhead operacional

A distribuição dos componentes de uma aplicação em microsserviços também afeta como processos associados ao ciclo de vida da aplicação são organizados. Alguns dos *overheads* operacionais que podem surgir com o projeto de migração estão ligados aos seguintes processos:

- **Automação**: Enquanto um sistema monolítico tem que lidar com a gestão de um número reduzido de processos computacionais [usualmente, um servidor *front-end*, um *back-end* e um banco de dados, além de outros serviços relacionados à infraestrutura, como um *load balancer* ou uma *Content Delivery Network* (CDN)], a divisão em microsserviços aumenta significativamente esse trabalho, não só pela necessidade de realizar atividades similares para cada serviço criado (configuração

de ambientes, aplicação de testes, realização de *deployments*, etc.), mas também pela maior complexidade arquitetural que deve ser mantida (novos fluxos de autenticação e autorização, localização das instâncias dos serviços na rede, testes fim a fim, resiliência a falha de serviços consumidos, etc.).

Enquanto poucos microsserviços forem extraídos, reduz-se o número de vezes nas quais essas atividades terão que ser repetidas e a complexidade de gestão do sistema distribuído não será tão expressiva (por exemplo, menos serviços significa menos possíveis pontos de falha individuais), possibilitando assim que essas tarefas sejam feitas manualmente. Contudo, a expansão do número de serviços e da complexidade arquitetural da aplicação tornará o custo de gestão manual proibitório, de forma que mais tempo e esforços das equipes serão gastos em atividades associadas à manutenção de processos e da infraestrutura do sistema, ao invés do cumprimento de requisitos funcionais, além de facilitar a introdução de erros de configuração, visto que a propensão trazida pela sua natureza repetitiva.

Dessa forma, é interessante considerar desde cedo a automatização de processos como a implantação de serviços, através de *pipelines* de *Continuous Integration/Continuous Delivery* (CI/CD), bem como o uso de de *templates* para geração de novos microsserviços (Newman, 2019);

- **Testes**: Apesar de testes unitários não possuírem complicações adicionais na arquitetura de microsserviços, o mesmo não pode ser dito dos outros níveis. Primeiramente, testes de serviços com um todo podem ter que falsear a comunicação com outros módulos através de chamadas *stub* e objetos *mock*, mas também tendo a possibilidade de criar outros serviços "falsos" para testar os mecanismos de comunicação. O nível acima desse, testes fim a fim, carrega outras possíveis complicações (Newman, 2019):
  - Alguns testes podem acabar sendo frágeis, no sentido de que eles podem falhar ou passar de maneira incerta. Isso ocorre pela presença do fator distribuído dos microsserviços permitindo que razões externas atrapalhem a reprodutibilidade dos testes, como falhas de uma unidade específica durante uma cadeia de chamadas;
  - A atribuição dos responsáveis pela escrita dos testes pode não ser tão simples (se cada equipe é responsável por um conjunto de serviços, não é há uma escolha óbvia de quem fará testes envolvendo o sistema inteiro);
  - O fato desses testes serem feitos sobre um conjunto de serviços em versões específicas pode induzir à formação de versionamento da aplicação inteira. Com o tempo, isso tende a reduzir a capacidade de atualizações independentes de microsserviços.
- **Monitoramento**: A migração para um modelo distribuído introduz certas complicações às tarefas de monitoramento da aplicação, como a coleta e a estruturação de *logs* das atividades e de métricas internas. Na arquitetura monolítica, esses processos eram facilitados pela concentração de grande parte desses registros (possivelmente com uma separação entre os do *front-end* e os do *back-end*) em um único processo, o que permite a extração de informações de forma bem mais direta. Por exemplo, *stack traces* e *loggers* são de fácil obtenção e organização quando a execução ocorre

sobre um mesmo processo, já que a correlação entre múltiplas partes do sistema pode ser feita diretamente na memória.

Porém, no modelo de microsserviços, esses dados estão localizados em múltiplas fontes (cada instância de cada serviço) e irão conter somente informações locais aos microsserviços, como a utilização de recursos das máquinas, a quantidade de requisições por *endpoint* ou *logs* implementados na lógica interna, sem necessariamente conseguir correlacionar registros externos como parte da execução de uma mesma funcionalidade.

Em cenários onde o monitoramento da versão monolítica costumava ser feito manualmente, manter essa prática num ambiente de microsserviços introduz diversas complicações, como (Newman, 2019):

- Necessidade de acesso direto a uma quantidade potencialmente enorme de instâncias de serviços para extração dos dados de monitoramento, processo que pode ser um pouco lento;
- Em casos onde a comunicação entre serviços não possui uma forma de correlacionar *logs* entre múltiplos serviços [como através de IDs de correlação], a identificação da causa de certos *bugs* pode ser mais complexa;
- Instâncias executadas na *cloud* públicas e que podem ser criadas e removidas dinamicamente podem ter os dados de monitoramento excluídos durante o seu decomissionamento;
- A visão sobre o estado atual da aplicação se torna limitada, sem uma correlação mais profunda dos dados, de forma a priorizar informações mais simples e nem sempre tão úteis para a tomada de decisões, sejam elas funcionais (análise da interação de usuários com a aplicação) ou não funcionais (como pontos de gargalo em certas funcionalidades).

Dessa forma, a implementação de processos automáticos de monitoramento é algo essencial num ambiente de microsserviços.

Outras dificuldades

- **Atritos entre equipes**: Dependendo das escolhas feitas durante a migração, a integração entre equipes e entre unidades organizacionais pode ser afetada de forma a gerar certos atritos (Fritzsch *et al.*, 2019). As maneiras como isso pode se manifestar variam de caso a caso, sendo algumas delas: um serviço pode necessitar de alguma funcionalidade ou dado de responsabilidade externa, assim dependendo da velocidade de entrega de outra equipe; múltiplos times trabalhando sob um mesmo serviço podem divergir na estruturação dos detalhes internos (padrões de codificação, escolhas de tecnologias, etc.) e interferir nas alterações uns dos outros (ou ter um maior trabalho de coordenação); independência dada a times pode dificultar no planejamento e aplicação de medidas envolvendo múltiplas áreas da aplicação, dentre outras.
- **Mal preparo de equipes**: Conduzir uma transformação para microsserviços é uma tarefa complexa e que não está isenta de uma variedade de falhas e armadilhas.

> Para melhor dirigir esse projeto e poder se recuperar mais facilmente dos possíveis contratempos, é de interesse da organização investir na introdução de conhecimento e experiência na área de microsserviços (e correlatas, como *DevOps* e computação em nuvem). Isso pode ser feito através de treinamentos de funcionários, contratação de pessoal, formação de órgãos centrais que dão auxílio aos times, contratação de serviços de consultoria, etc. (Newman, 2019).

3.3 "Depois" da migração

Considerando uma transição gradual da estrutura monolítica para uma de microsserviços, definir uma marca para servir como "fim" do projeto de migração não é uma tarefa simples. Isso porque a transição para a nova arquitetura, devido as mais diversas complexidades exploradas neste capítulo e muitas outras não mencionadas, raramente resulta num produto que satisfaça para sempre e de forma completa as necessidades ou ambições da organização, seja por causa de decisões que se provariam menos que ideais, implementações técnicas indevidas ou simples mudanças de foco ou na escala da empresa. Dessa forma, é de se esperar que a estrutura da aplicação continue a ser retrabalhada muito após o início da migração para microsserviços: tecnologias sendo alteradas, padrões arquiteturais sendo repensados, procedimentos sendo reescritos, serviços sendo redesenhados, times sendo reordenados, etc. Portanto, se o fim do projeto de migração for considerado quando o sistema estiver estruturado da melhor forma possível, ele não acabaria na prática.

Porém, mesmo contando com melhorias indefinidas, eventualmente a noção de que a aplicação é majoritariamente organizada em microsserviços será consolidada. Em um momento, o *software* e a organização ao seu redor são apresentados como "em transição" e, após um certo limiar (não necessariamente bem definido), eles podem estar consolidados ao redor de microsserviços. Individualmente, esse limite entre os dois estados pode variar significativamente: ele pode ser quando os esforços ativos em fazer alterações na arquitetura diminuíram significativamente, a extração de todos (ou dos principais) componentes selecionadas para migração foi completa, as mudanças nas estruturas organizacional e de processos foram estabilizadas, os problemas que motivaram a proposta desse projeto foram suficientemente amenizadas ou solucionadas, dentre muitas outras possibilidades.

Na prática, a não ser que esse projeto tenha sido feito no modelo *Big Bang* e o seu fim seja alinhado com a sua implantação para uso geral de seus clientes, tal separação pode ser vista como fútil, uma vez que as ações tomadas durante esse momento "após" a migração são uma extensão das atividades realizadas ao longo de todo o processo, com um refinamento gradual da implementação existente e da adição de novos suportes arquiteturais a medida que a aplicação se expande.

Capítulo 4

Conclusão

Ao longo da última década, a arquitetura de microsserviços passou de um conjunto de práticas vistas esporadicamente na indústria de tecnologia para ser popularizada como "a mais nova forma de organizar aplicações". Com o desenvolvimento da área, provou-se que o modelo de microsserviços é capaz de trazer uma série de benefícios e resolver diversas dificuldades normalmente associadas a arquiteturas de caráter mais monolítico, como a alta dificuldade de manutenção ou escalabilidade ineficaz. Isso vêm progressivamente atraindo grande interesse da indústria.

Por outro lado, também há a percepção de que seguir esse caminho está longe de ser algo simples. As alterações envolvidas no processo de migração nem sempre são fáceis, ou mesmo possíveis (em certos contextos), de serem implementadas da forma "correta". Isso pode causar uma série de problemas e complicações que colocam em risco atributos qualitativos do sistema, uma vez que eles estão atrelados à forma como a estrutura baseada em microsserviços é construída na prática. Por exemplo, a resiliência de aplicações em microsserviços pode ser maior que a vista em aplicações monolíticas, mas somente se um conjunto de práticas for bem adotado: caso contrário, pode-se acabar com um sistema bem mais frágil do que o original.

Dessa forma, é interessante que haja uma boa compreensão, por parte das organizações que pretendam seguir essa jornada, não somente das etapas envolvidas para se chegar à nova arquitetura, mas também de quais problemas e dificuldades poderão ser encontrados ao longo da migração para microsserviços. Essa seria, em essência, a proposta deste trabalho.

A primeira parte dele é dedicada a uma caracterização geral do assunto, iniciando pela apresentação, em alto nível, de algumas das características associadas à arquitetura e como elas costumam ser compostas, como a divisão de responsabilidades da aplicação entre os serviços, o estabelecimento de comunicação entre eles e os possíveis ajustes na estrutura organizacional ao seu redor. Através dessa contextualização, coloca-se em perspectiva a origem de certas complicações que serão exploradas ao longo do texto.

Também foram introduzidos certos benefícios proporcionados, em teoria, pelos microsserviços, como melhor manutenibilidade da aplicação, escalabilidade mais eficiente e independência de implantações. Nessa parte, além da apresentação de como esses pontos

positivos costumam se manifestar, também são introduzidas certas complicações envolvidas na obtenção de níveis satisfatórios desses benefícios, visto que, em vários dos casos, se fazem necessárias novas tecnologias e técnicas, além de considerações adicionais.

Por fim, são trazidas algumas das desvantagens que acompanham, em níveis variados, o uso dessa arquitetura: uma maior complexidade geral da aplicação (comunicação e configuração de serviços, por exemplo) e de seu ciclo de vida (testagem, segurança, etc.), complicações com dados distribuídos (como consistências de dados entre módulos independentes) e aumento nos custos.

Já em um segundo momento, foca-se menos na arquitetura de microsserviços e mais na própria migração (apesar de haver intersecção entre ambos os tópicos). Apresenta-se então alguns dos problemas e dificuldades encontradas nesse processo, de acordo com o período no qual eles podem se manifestar e contextualizadas de acordo com certas atividades da transição.

De início, são expostas dificuldades relativas a estágios de preparação. Isso envolve considerações sobre a validade do projeto de migração (se realmente existe a necessidade e a capacidade para a adoção de microsserviços), sobre mudanças na estrutura organizacional (como times se organizam, distribuição das responsabilidades, etc.), sobre análise da implementação atual (fluxos de funcionalidades, relação entre módulos, etc.) e sobre como microsserviços são divididos (partição do domínio). Em seguida, são apresentadas questões ligadas ao estágio de implementação. Isso envolve tarefas como a divisão da base de código e banco de dados em serviços independentes, incorporação de novas técnicas e tecnologias, reorganização operacional, dentre outras.

Após analisar todos esses pontos, algumas conclusões podem ser tomadas sobre o processo de migração para microsserviços:

- Ele é moldado por uma série de escolhas, técnicas ou não, que ditam os atributos finais do sistema, sendo cada uma delas acompanhada de pontos positivos e negativos. O melhor caminho a ser seguido pode variar de acordo com as intenções por trás da escolha dessa arquitetura e com o contexto da aplicação, o que pode torná-lo um pouco incerto;
- A natureza distribuída, modular e independente dos microsserviços são grande parte das fontes de complexidades que dificultam uma boa adoção generalizada da arquitetura;
- Ele introduz alterações possivelmente disruptivas ao ciclo de vida da aplicação. Além disso, caso ele seja feito iterativamente, a lenta extração de serviços e a necessidade de integração com o monolito apresentam complicações próprias.

Dessa forma, apesar da arquitetura de microsserviços ser uma forma de estruturar aplicações que, nas condições certas e implementada da forma "correta", permita um crescimento bem melhor que dentro dos confins de um monolito, nem todos os sistemas realmente conseguem se basear, ou mesmo necessitam, de microsserviços. Se o objetivo por trás dessa transição for escapar dos problemas associados aos monolitos, talvez seja mais interessante primeiramente procurar por soluções alternativas e com uma menores barreira de entrada. Caso isso não seja suficiente, a adoção do padrão arquitetural deve ser

feita de forma prudente, evitando ultrapassar os limites viáveis ao contexto.

Referências

[Abdullah *et al.* 2019] Muhammad Abdullah, Waheed Iqbal e Abdelkarim Erradi. "Unsupervised learning approach for web application auto-decomposition into microservices". Em: *Journal of Systems and Software* 151 (mai. de 2019), pgs. 243–257. doi: 10.1016/j.jss.2019.02.031 (citado na pg. 8).

[Akbulut e Perros 2019] Akhan Akbulut e Harry G. Perros. "Software Versioning with Microservices through the API Gateway Design Pattern". Em: *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*. IEEE, jun. de 2019. doi: 10.1109/acitt.2019.8779952 (citado na pg. 16).

[Amaral *et al.* 2015] Marcelo Amaral *et al.* "Performance Evaluation of Microservices Architectures Using Containers". Em: *2015 IEEE 14th International Symposium on Network Computing and Applications*. IEEE, set. de 2015. doi: 10.1109/nca.2015.49 (citado na pg. 18).

[Auer *et al.* 2021] Florian Auer, Valentina Lenarduzzi, Michael Felderer e Davide Taibi. "From monolithic systems to Microservices: An assessment framework". Em: *Information and Software Technology* 137 (set. de 2021), pg. 106600. doi: 10.1016/j.infsof.2021.106600 (citado na pg. 20).

[AWS 2021] AWS. *Microsserviços*. url: https://aws.amazon.com/pt/microservices/ (acesso em 13/12/2021) (citado na pg. 1).

[Balalaie, Heydarnoori e Jamshidi 2016a] Armin Balalaie, Abbas Heydarnoori e Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". Em: *IEEE Software* 33.3 (mai. de 2016), pgs. 42–52. doi: 10.1109/ms.2016.64 (citado nas pgs. 10, 19, 20, 22).

[Balalaie, Heydarnoori e Jamshidi 2016b] Armin Balalaie, Abbas Heydarnoori e Pooyan Jamshidi. "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report". Em: *Communications in Computer and Information Science*. Springer International Publishing, 2016, pgs. 201–215. doi: 10.1007/978-3-319-33313-7_15 (citado nas pgs. 16, 17).

[Balalaie, Heydarnoori, Jamshidi *et al.* 2018] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri e Theo Lynn. "Microservices migration patterns". Em: *Software: Practice and Experience* (jul. de 2018). doi: 10.1002/spe.2608 (citado nas pgs. 10, 24).

[Bogner *et al.* 2020] Justus Bogner, Adrian Weller, Stefan Wagner e Alfred Zimmermann. "Exploring Maintainability Assurance Research for Service- and Microservice-Based Systems: Directions and Differences". Em: *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*. Ed. por Luís Cruz-Filipe *et al.* Vol. 78. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:22. isbn: 978-3-95977-137-5. doi: 10.4230/OASIcs.Microservices.2017-2019.3. url: https://drops.dagstuhl.de/opus/volltexte/2020/11825/ (citado na pg. 14).

[Bozan *et al.* 2021] Karoly Bozan, Kalle Lyytinen e Gregory M. Rose. "How to transition incrementally to microservice architecture". Em: *Communications of the ACM* 64.1 (jan. de 2021), pgs. 79–85. doi: 10.1145/3378064. url: https://dl.acm.org/doi/fullHtml/10.1145/3378064 (citado na pg. 27).

[Chen 2018] Lianping Chen. "Microservices: Architecting for Continuous Delivery and DevOps". Em: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, abr. de 2018, pgs. 39–397. doi: 10.1109/icsa.2018.00013 (citado nas pgs. 8, 15).

[Conway 1968] Melvin E. Conway. "How do committees invent". Em: *Datamation* 14.4 (1968), pgs. 28–31 (citado na pg. 10).

[Coulson *et al.* 2020] Nathan Cruz Coulson, Stelios Sotiriadis e Nik Bessis. "Adaptive Microservice Scaling for Elastic Applications". Em: *IEEE Internet of Things Journal* 7.5 (mai. de 2020), pgs. 4195–4202. doi: 10.1109/jiot.2020.2964405 (citado nas pgs. 13, 15).

[Dehghani 2018] Zhamak Dehghani. *How to break a Monolith into Microservices*. Abr. de 2018. url: https://martinfowler.com/articles/break-monolith-into-microservices.html (acesso em 13/12/2021) (citado na pg. 29).

[Dragoni, Giallorenzo *et al.* 2017] Nicola Dragoni, Saverio Giallorenzo *et al.* "Microservices: Yesterday, Today, and Tomorrow". Em: *Present and Ulterior Software Engineering*. Springer International Publishing, 2017, pgs. 195–216. doi: 10.1007/978-3-319-67425-4_12 (citado nas pgs. 1, 6, 7).

[Dragoni, Lanese *et al.* 2017] Nicola Dragoni, Ivan Lanese *et al.* "Microservices: How To Make Your Application Scale". Em: (fev. de 2017). arXiv: 1702.07149 [cs.SE] (citado na pg. 13).

[Esposito *et al.* 2016] Christian Esposito, Aniello Castiglione e Kim-Kwang Raymond Choo. "Challenges in Delivering Software in the Cloud as Microservices". Em: *IEEE Cloud Computing* 3.5 (set. de 2016), pgs. 10–14. issn: 2325-6095. doi: 10.1109/mcc.2016.105 (citado na pg. 14).

[Etheredge 2020] Justin EtheredgeJustin Etheredge. *Gasp! You Might Not Need Microservices*. Mai. de 2020. url: https://www.simplethread.com/gasp-you-might-not-need-microservices/ (acesso em 21/12/2021) (citado nas pgs. 2, 8, 21, 23).

[Evans 2004] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. Boston: Addison-Wesley, 2004. isbn: 9780132181266 (citado na pg. 7).

[Finnigan 2018] Ken Finnigan. *Enterprise Java Microservices*. MANNING PUBN, nov. de 2018. 272 pp. isbn: 1617294241 (citado na pg. 27).

[Florio 2013] Vincenzo De Florio. "On the Constituent Attributes of Software and Organizational Resilience". Em: *Interdisciplinary Science Reviews* 38.2 (jun. de 2013), pgs. 122–148. doi: 10.1179/0308018813z.00000000040. eprint: https://doi.org/10.1179/0308018813Z.00000000040. url: https://doi.org/10.1179/0308018813Z.00000000040 (citado na pg. 12).

[Fowler 2005] Martin Fowler. *Event Sourcing*. Dez. de 2005. url: https://martinfowler.com/eaaDev/EventSourcing.html (acesso em 13/12/2021) (citado na pg. 21).

[Fowler 2013] Martin Fowler. *ContinuousDelivery*. Mai. de 2013. url: https://martinfowler.com/bliki/ContinuousDelivery.html (acesso em 13/12/2021) (citado na pg. 15).

[Fowler 2014] Martin Fowler. *MicroservicePrerequisites*. Ago. de 2014. url: https://martinfowler.com/bliki/MicroservicePrerequisites.html (acesso em 13/12/2021) (citado nas pgs. 17, 18).

[Fritzsch *et al.* 2019] Jonas Fritzsch, Justus Bogner, Stefan Wagner e Alfred Zimmermann. "Microservices Migration in Industry: Intentions, Strategies, and Challenges". Em: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, set. de 2019. doi: 10.1109/icsme.2019.00081 (citado nas pgs. 2, 14, 19, 20, 22, 35).

[Garlan 2014] David Garlan. "Software architecture: a travelogue". Em: *Future of Software Engineering Proceedings*. ACM, mai. de 2014. doi: 10.1145/2593882.2593886 (citado na pg. 1).

[Haddad 2015] Einas Haddad. *Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow*. Uber. Set. de 2015. url: https://eng.uber.com/service-oriented-architecture/ (acesso em 15/12/2021) (citado na pg. 1).

[Haden 2021] Jeff Haden. *When Jeff Bezos's 2-Pizza Teams Fell Short, He Turned to the Brilliant Model Amazon Uses Today*. Fev. de 2021. URL: https://www.inc.com/jeff-haden/when-jeff-bezoss-two-pizza-teams-fell-short-he-turned-to-brilliant-model-amazon-uses-today.html (acesso em 13/12/2021) (citado nas pgs. 10, 11).

[Izrailevsky *et al.* 2016] Yury Izrailevsky, Stevan Vlaovic e Ruslan Meshenberg. *Completing the Netflix Cloud Migration*. Fev. de 2016. URL: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration (acesso em 13/12/2021) (citado na pg. 1).

[Jackson 2019] Cam Jackson. *Micro Frontends*. Jun. de 2019. URL: https://martinfowler.com/articles/micro-frontends.html (acesso em 23/12/2021) (citado na pg. 28).

[Jamshidi *et al.* 2018] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis e Stefan Tilkov. "Microservices: The Journey So Far and Challenges Ahead". Em: *IEEE Software* 35.3 (mai. de 2018), pgs. 24–35. DOI: 10.1109/ms.2018.2141039 (citado nas pgs. 6, 13).

[Janes e Russo 2019] Andrea Janes e Barbara Russo. "Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices". Em: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, out. de 2019. DOI: 10.1109/issrew.2019.00067 (citado na pg. 27).

[Kalske *et al.* 2018] Miika Kalske, Niko Mäkitalo e Tommi Mikkonen. "Challenges When Moving from Monolith to Microservice Architecture". Em: *Current Trends in Web Engineering*. Springer International Publishing, 2018, pgs. 32–47. DOI: 10.1007/978-3-319-74433-9_3 (citado nas pgs. 22, 26).

[Kotter 1996] John Kotter. *Leading change*. Boston, Mass: Harvard Business School Press, 1996. ISBN: 9780875847474 (citado na pg. 24).

[Kromhout 2018] Bridget Kromhout. "Containers will not fix your broken culture (and other hard truths)". Em: *Communications of the ACM* 61.4 (mar. de 2018), pgs. 40–43. DOI: 10.1145/3162086 (citado na pg. 16).

[Kulkarni e Dwivedi 2008] Naveen Kulkarni e Vishal Dwivedi. "The Role of Service Granularity in a Successful SOA Realization A Case Study". Em: *2008 IEEE Congress on Services - Part I*. IEEE, jul. de 2008, pgs. 423–430. DOI: 10.1109/services-1.2008.86 (citado na pg. 8).

[Larrucea *et al.* 2018] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios e Christof Ebert. "Microservices". Em: *IEEE Software* 35.3 (mai. de 2018), pgs. 96–100. DOI: 10.1109/ms.2018.2141030 (citado nas pgs. 16, 19).

[Leppanen *et al.* 2015] Marko Leppanen *et al.* "The highways and country roads to continuous deployment". Em: *IEEE Software* 32.2 (mar. de 2015), pgs. 64–72. DOI: 10.1109/ms.2015.50 (citado na pg. 22).

[Lewis e Fowler 2014] James Lewis e Martin Fowler. *Microservices - a definition of this new architectural term*. Mar. de 2014. URL: https://martinfowler.com/articles/microservices.html (acesso em 13/12/2021) (citado nas pgs. 6, 10, 30).

[Loukides e Swoyer 2020] Mike Loukides e Steve Swoyer. *Microservices Adoption in 2020*. Rel. técn. O'Reilly, jul. de 2020. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/ (acesso em 13/12/2021) (citado na pg. 1).

[Malla e Christensen 2019] Sulav Malla e Ken Christensen. "HPC in the cloud: performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)". Em: *Internet Technology Letters* 3.1 (dez. de 2019), e137. DOI: 10.1002/itl2.137 (citado na pg. 14).

[Martin 2014] R. C. Martin. *The single responsibility principle*. Mai. de 2014. URL: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html (acesso em 13/12/2021) (citado na pg. 7).

[Mazlami *et al.* 2017] Genc Mazlami, Jurgen Cito e Philipp Leitner. "Extraction of Microservices from Monolithic Software Architectures". Em: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, jun. de 2017. DOI: 10.1109/icws.2017.61 (citado na pg. 26).

[Mazzara *et al.* 2018] Manuel Mazzara *et al.* "Microservices: Migration of a Mission Critical System". Em: *IEEE Transactions on Services Computing* 14.5 (dez. de 2018), pgs. 1464–1477. DOI: 10.1109/tsc.2018.2889087. eprint: 1704.04173 (cs.SE) (citado na pg. 6).

[Mendonca *et al.* 2020] Nabor C. Mendonca, Carlos M. Aderaldo, Javier Camara e David Garlan. "Model-Based Analysis of Microservice Resiliency Patterns". Em: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, mar. de 2020, pgs. 114–124. DOI: 10.1109/icsa47634.2020.00019 (citado na pg. 12).

[Munezero *et al.* 2018] Immaculée Josélyne Munezero, Doreen-Tuheirwe Mukasa, Benjamin Kanagwa e Joseph Balikuddembe. "Partitioning Microservices: A Domain Engineering Approach". Em: *2018 IEEE/ACM Symposium on Software Engineering in Africa (SEiA)*. 2018, pgs. 43–49 (citado na pg. 7).

[Netflix 2021] Netflix. *Netflix Open Source Software Center*. URL: https://netflix.github.io/ (acesso em 13/12/2021) (citado na pg. 1).

[Newman 2015] Sam Newman. *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 9781491950357 (citado nas pgs. 2, 6–10, 12–16, 18, 20, 26, 28).

[Newman 2019] Sam Newman. *Monolith to Microservices*. O'Reilly UK Ltd., dez. de 2019. ISBN: 1492047848 (citado nas pgs. 8, 15, 18, 21, 22, 24, 25, 27–31, 33–36).

[Opus Software 2020] Opus Software. *Migrating From Monoliths to Microservices*. Set. de 2020. URL: https://www.opus.software/migrating-from-monoliths-to-microservices/ (acesso em 13/12/2021) (citado nas pgs. 27, 28).

[Osses *et al.* 2018] Felipe Osses, Gastón Márquez e Hernán Astudillo. "Exploration of academic and industrial evidence about architectural tactics and patterns in microservices". Em: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, mai. de 2018. DOI: 10.1145/3183440.3194958 (citado na pg. 3).

[Pautasso *et al.* 2017] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis e Nicolai Josuttis. "Microservices in Practice, Part 1: Reality Check and Service Design". Em: *IEEE Software* 34.1 (jan. de 2017), pgs. 91–98. DOI: 10.1109/ms.2017.24 (citado nas pgs. 6, 9, 25, 26).

[Rademacher, Sachweh *et al.* 2019] Florian Rademacher, Sabine Sachweh e Albert Zundorf. "Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture". Em: *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, mar. de 2019, pgs. 21–30. DOI: 10.1109/icsa.2019.00011 (citado na pg. 12).

[Rademacher, Sorgalla *et al.* 2018] Florian Rademacher, Jonas Sorgalla e Sabine Sachweh. "Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective". Em: *IEEE Software* 35.3 (mai. de 2018), pgs. 36–43. DOI: 10.1109/ms.2018.2141028 (citado na pg. 7).

[Ramchand *et al.* 2021] Mahir Ramchand, A. Capiluppi, F. J. Blaauw e V. Degeler. *A Systematic Mapping of Microservice Patterns*. Bachelor Thesis. Bachelor. https://fse.studenttheses.ub.rug.nl/, jul. de 2021. URL: https://fse.studenttheses.ub.rug.nl/25671/ (citado na pg. 5).

[Reisz *et al.* 2020] Wesley Reisz, Yan Cui, Leif Beaton e Nicky Wrightson. *Reviewing the Microservices Architecture: Impacts, Operational Complexity, and Alternatives*. Dez. de 2020. URL: https://www.infoq.com/articles/reviewing-microservices-architecture-operational-complexity/ (acesso em 14/12/2021) (citado na pg. 16).

[Ren *et al.* 2018] Zhongshan Ren *et al.* "Migrating Web Applications from Monolithic Structure to Microservices Architecture". Em: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. Internetware '18. Beijing, China: Association for Computing Machinery, set. de 2018. ISBN: 9781450365901. DOI: 10.1145/3275219.3275230 (citado na pg. 8).

[Richardson 2019a] Chris Richardson. *Anti-pattern: microservices as the goal*. Jan. de 2019. URL: https://chrisrichardson.net/post/antipatterns/2019/01/14/antipattern-microservices-are-the-goal.html (acesso em 13/12/2021) (citado na pg. 22).

[Richardson 2019b] Chris Richardson. *Microservices adoption anti-pattern: More the merrier*. Mai. de 2019. URL: https://chrisrichardson.net/post/antipatterns/2019/05/21/antipattern-more-the-merrier.html (acesso em 23/12/2021) (citado nas pgs. 8, 21).

[Richardson 2019c] Chris Richardson. *Microservices adoption anti-pattern: Red flag law*. Jun. de 2019. URL: https://chrisrichardson.net/post/antipatterns/2019/06/07/antipattern-red-flag-law.html (acesso em 13/12/2021) (citado na pg. 22).

[Richardson 2019d] Chris Richardson. *Microservices adoption anti-pattern: Trying to fly before you can walk*. Abr. de 2019. URL: https://chrisrichardson.net/post/antipatterns/2019/04/09/antipattern-flying-before-walking.html (acesso em 16/12/2021) (citado na pg. 22).

[Richardson 2019e] Chris Richardson. *Microservices adoption anti-patterns: microservices are a magic pixie dust*. Jan. de 2019. URL: https://chrisrichardson.net/post/antipatterns/2019/01/07/microservices-are-a-magic-pixie-dust.html (acesso em 13/12/2021) (citado na pg. 20).

[Richardson 2021a] Chris Richardson. *A pattern language for microservices*. URL: https://microservices.io/patterns/index.html (acesso em 13/12/2021) (citado nas pgs. 5, 10, 11, 17).

[Richardson 2021b] Chris Richardson. *Pattern: Monolithic Architecture*. URL: https://microservices.io/patterns/monolithic.html (acesso em 13/12/2021) (citado nas pgs. 2, 11, 16).

[Selmadji *et al.* 2020] Anfel Selmadji *et al.* "From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach". Em: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, mar. de 2020, pgs. 157–168. DOI: 10.1109/icsa47634.2020.00023 (citado na pg. 8).

[Shadija *et al.* 2017a] Dharmendra Shadija, Mo Rezai e Richard Hill. "Microservices: granularity vs. performance". Em: *Companion Proceedings of The 10th International Conference on Utility and Cloud Computing*. UCC '17 Companion. Austin, Texas, USA: Association for Computing Machinery, dez. de 2017, pgs. 215–220. ISBN: 9781450351959. DOI: 10.1145/3147234.3148093. URL: https://doi.org/10.1145/3147234.3148093 (citado na pg. 8).

[Shadija *et al.* 2017b] Dharmendra Shadija, Mo Rezai e Richard Hill. "Towards an understanding of microservices". Em: *2017 23rd International Conference on Automation and Computing (ICAC)*. IEEE, set. de 2017, pgs. 1–6. DOI: 10.23919/iconac.2017.8082018 (citado na pg. 7).

[Taibi *et al.* 2017] Davide Taibi, Valentina Lenarduzzi e Claus Pahl. "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation". Em: *IEEE Cloud Computing* 4.5 (set. de 2017), pgs. 22–32. DOI: 10.1109/MCC.2017.4250931 (citado nas pgs. 16, 22).

[Taibi *et al.* 2018] Davide Taibi, Valentina Lenarduzzi e Claus Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study". Em: *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science e Technology Publications, 2018. DOI: 10.5220/0006798302210232 (citado nas pgs. 15, 18).

[Taibi *et al.* 2019] Davide Taibi, Valentina Lenarduzzi e Claus Pahl. "Microservices Anti-patterns: A Taxonomy". Em: *Microservices*. Springer International Publishing, dez. de 2019, pgs. 111–128. DOI: 10.1007/978-3-030-31646-4_5 (citado nas pgs. 6, 12, 16–18, 30).

[Thones 2015] Johannes Thones. "Microservices". Em: *IEEE Software* 32.1 (jan. de 2015), pgs. 116–116. DOI: 10.1109/MS.2015.11 (citado na pg. 6).

[Velepucha e Flores 2021] Victor Velepucha e Pamela Flores. "Monoliths to microservices - Migration Problems and Challenges: A SMS". Em: *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*. IEEE, mar. de 2021. DOI: 10.1109/ici2st51859.2021.00027 (citado nas pgs. 2, 6, 9, 14, 15, 19, 29).

[Vera-Rivera *et al.* 2021] Fredy H. Vera-Rivera, Carlos Gaona e Hernán Astudillo. "Defining and measuring microservice granularity—a literature overview". Em: *PeerJ Computer Science* 7 (set. de 2021), e695. DOI: 10.7717/peerj-cs.695 (citado na pg. 8).

[Verified Market Research 2021] Verified Market Research. *Cloud Microservices Market 2020 Trends, Market Share, Industry Size, Opportunities, Analysis and Forecast by 2026*. Rel. de pesq. Market Reports, fev. de 2021. URL: https://www.instanttechnews.com/technology-news/2021/02/23/cloud-microservices-market-2020-trends-market-share-industry-size-opportunities-analysis-and-forecast-by-2026/ (acesso em 13/12/2021) (citado na pg. 20).

[Vural e Koyuncu 2021] Hulya Vural e Murat Koyuncu. "Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?" Em: *IEEE Access* 9 (2021), pgs. 32721–32733. DOI: 10.1109/access.2021.3060895 (citado na pg. 14).

[Wolff 2019] Eberhard Wolff. *Why Microservices Fail:An Experience Report*. Rel. técn. innoQ Deutschland GmbH, 2019. URL: https://www.conf-micro.services/2019/papers/Microservices_2019_paper_18.pdf (acesso em 21/12/2021) (citado nas pgs. 2, 7).

[Yu *et al.* 2019] Guangba Yu, Pengfei Chen e Zibin Zheng. "Microscaler: Automatic Scaling for Microservices with an Online Learning Approach". Em: *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, jul. de 2019, pgs. 68–75. DOI: 10.1109/icws.2019.00023 (citado na pg. 14).