Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Método de migração de sistemas monolíticos legados para a arquitetura de microsserviços

Taylor Rodrigues Lopes

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr. André Luiz Peron Martins Lanna

Brasília
2021

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Rodrigues Lopes, Taylor
RL864m Método de migração de sistemas monolíticos legados para a arquitetura de microsserviços / Taylor Rodrigues Lopes; orientador André Luiz Peron Martins Lanna. -- Brasília, 2021.
151 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2021.

1. arquitetura de software. 2. microsserviço. 3. monolítico. 4. migração. I. Luiz Peron Martins Lanna, André, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Método de migração de sistemas monolíticos legados para a arquitetura de microsserviços

Taylor Rodrigues Lopes

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr. André Luiz Peron Martins Lanna (Orientador)
CIC/UnB

Prof.a Dr.a Edna Dias Canedo
CIC/UnB

Prof. Dr. Ricardo Terra Nunes Bueno Villela
DCC/UFLA

Prof. Dr. Sérgio Antônio Andrade de Freitas
PPCA/UnB

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 29 de janeiro de 2021

Dedicatória

Dedico este trabalho à vida dos meus avós “Henrique & Percília Gomes” e “Jesué & Ilda Lopes”, por me trazer à memória tudo aquilo que me dá esperança - Lm 3:21.

(*in memorian*)

Agradecimentos

Meu primeiro agradecimento não poderia deixar de ser para o Criador, Deus. "*Sem Ele, nada do que existe teria sido feito*" (Jo 1:3). Obrigado, Senhor, pela saúde e força para superar os desafios deste mestrado e por conduzir toda minha vida.

Aos meus queridos pais, Jesué e Maria, agradeço o amor, ensinamentos e investimentos na minha formação desde criança, ainda que com recursos escassos. Vocês são minha referência neste mundo, expressão da verdadeira sabedoria. Aos meus irmãos, Hudson e Kayla e cada um dos meus cunhados, sobrinhos, tios e primos pelas orações e afeto.

A minha amada esposa Cláudia e meu filho Matheus, não sei se peço obrigado ou desculpas. Perdoem minha ausência, pelo longo tempo que estive debruçado nos estudos. Agradeço a paciência e suporte em todos os momentos. Não foi fácil, nunca foi, mas conseguimos. Que essa conquista possa render frutos a todos nós. Amo vocês.

Ao meu orientador, Prof André Lanna, um especial obrigado por estar ao meu lado nesta caminhada, contribuindo com sua experiência acadêmica e mais, sempre amigo, proporcionando meios para que eu chegasse confiante ao final deste trabalho.

Ao Prof. Paulo Merson, profundo conhecedor da arquitetura de software, responsável por me despertar o interesse em microsserviços e contribuir revisando parte deste trabalho.

Ao estimado Prof. Sérgio Freitas, meu primeiro professor no PPCA, pelo aprendizado em sala e conselhos que servirão para a vida. Ao Prof. Marcelo Ladeira, coordenador, e Prof.a Edna Canedo pelo auxílio na escolha do tema, e também por compor a minha banca de qualificação e defesa de mestrado junto ao Prof. Gibeon Aquino (UFRN) e Ricardo Terra (UFLA), respectivamente, que muito contribuíram neste processo de formação. Enfim, a todos os professores do PPCA a qual tenho muito respeito e orgulho, como Prof. Márcio Victorino, Prof. Alexandre Zaghetto e tantos outros.

À Universidade de Brasília (UnB) e ao PPCA pela oferta e condução deste mestrado em alto nível, realizando não apenas uma aspiração pessoal, mas devolvendo a minha organização um profissional mais capacitado.

Ao Exército Brasileiro, na pessoa do Cel Fernando e Cap João de Deus, que me incentivaram e deram apoio irrestrito desde o início, minha eterna gratidão. Também aos companheiros de trabalho que torciam por mim, todos sintam-se representados.

Resumo

Atualmente, grande parte das organizações dependem de Sistemas de Informação (SI). Em geral, estes sistemas são construídos com base na *"arquitetura monolítica"*, tendo a execução centralizada em um único servidor. Ao longo dos anos, porém, as constantes mudanças para atender necessidades de negócio e o acúmulo da dívida técnica, têm tornado estes sistemas cada vez maiores e complexos, dificultando aspectos como manutenibilidade e escalabilidade. Essa difícil realidade vivida por muitas organizações motivou o presente trabalho a investigar uma nova tendência arquitetural denominada *"microsserviços"*. Popularizado por empresas como Netflix e Amazon, os microsserviços podem ser uma alternativa para a modernização de sistemas legados, propiciando mínimo *downtime* e impacto ao usuário final. Nessa nova arquitetura, o software é decomposto em pequenas partes que funcionam de modo independente e autônomo, trazendo algumas melhorias em termos de atributos de qualidade de software. Contudo, há também desafios e *tradeoffs*: adotar microsserviços tende a ser um processo difícil e não raramente malsucedido, sobretudo, em razão da carência de métodos para conduzir o processo de migração. Nesse sentido, fundamentado em estudos científicos, este trabalho apresenta um método de migração intitulado *Microservice Full Cycle* - MFC, inspirado no ciclo de vida de desenvolvimento de software e em estratégias *DevOps*. O objetivo é auxiliar sistemas de software legados a gradualmente evoluírem orientados por um conjunto de etapas e atividades comuns à arquitetura de microsserviços. A validação do método MFC é feita por meio de uma simulação em uma aplicação real, tendo evidenciado vantagens tais como códigos mais coesos e desacoplados, independência tecnológica, agilidade em *build*, teste e *deploy* (automação), escalabilidade sob alta demanda, maior interoperabilidade e integração, capacidades geodistribuídas, além de monitoramento e *feedback* em tempo real.

Palavras-chave: arquitetura de software, microsserviço, monolítico, migração

Abstract

Currently, most organizations depend on Information Systems. In general, these systems are built based on monolithic architecture, with centralized execution on a single server. Over the years, however, the constant changes to meet business needs and the accumulation of technical debt, have made these systems increasingly larger and complex, making aspects such as maintainability and scalability difficult. This difficult reality experienced by many organizations motivated the present work to investigate a new architectural trend called microservices. Popularized by companies like Netflix and Amazon, microservices can be an alternative for the modernization of legacy systems, providing low downtime and imperceptible impact to the end user. In this new architecture, the software is broken down into small parts that work independently and autonomously, bringing some improvements in terms of software quality attributes. However, there are also challenges and tradeoffs: adopting microservices tends to be a difficult process and not rarely unsuccessful, mainly due to the lack of methods to conduct the migration process. In this sense, based on scientific studies, this work presents a migration method entitled Microservice Full Cycle - MFC, inspired by the software development life cycle and DevOps strategies. The goal is to assist legacy software systems to gradually evolve guided by a set of steps and activities common to the microservice architecture. The validation of the MFC method is done through an experiment in a real application, showing advantages such as more cohesive and uncoupled codes, technological independence, agility in build, test and deploy (automation), scalability under high demand, greater interoperability and integration, geodistributed capabilities, plus realtime monitoring and feedback.

Keywords: software architecture, microservice, monolithic, migration

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Programming Interface.

CI Integração Contínua.

DDD Domain-Driven Design.

DISP Diretoria de Seleção de Pessoal.

DSRM Design Science Research Methodology.

EB Exército Brasileiro.

GQM Goal Question Metric.

MFC Microservice Full Cycle.

MVC Model-View-Controller.

PHP Hypertext Preprocessor.

QA Atributos de Qualidade.

REST Representational State Transfer.

SI Sistemas de Informação.

SISP Sistema de Seleção de Pessoal.

Capítulo 1

Introdução

Netflix, Amazon, IBM, eBay, Spotify, PayPal, Uber, LinkedIn e muitas outras grandes corporações estão migrando para a arquitetura de microsserviços [2, 3, 8, 9, 10, 11, 12, 13], isso porque, migrar de uma arquitetura monolítica para microsserviços traz muitos benefícios [1, 14, 15, 16, 17]. Essas mudanças estratégicas por parte de empresas privadas sugerem que a arquitetura de software é, de fato, uma vantagem competitiva. Para Taibi et al. [2], o que leva um desenvolvedor de software a escolher uma arquitetura em detrimento a outra são os resultados de experiências anteriores ou os benefícios percebidos da nova arquitetura. A arquitetura exerce influência direta sobre o ciclo de vida do software e os atributos de qualidade de software [18, 19] podendo, por exemplo, determinar quão fácil é modificar um software ou quão rápido ele deve ser para atender a expectativa do usuário. É a arquitetura que possibilita melhor compreender as estruturas de software, seus elementos e o modo como eles se relacionam [20].

Dentre as tradicionais arquiteturas de software, a arquitetura monolítica tem sido a mais amplamente utilizada pela maioria dos sistemas devido a sua simplicidade [12, 21, 22, 23]. Uma típica aplicação monolítica contém toda a lógica e dados para implementar uma funcionalidade em uma única base de código, geralmente utilizando o padrão arquitetural MVC (do inglês *Model-View-Controller*) [9], conforme ilustrado pela Figura 1.1(a). Um monolítico é mais simples de desenvolver, fácil de testar e mais produtivo em estágios iniciais de projeto de software [12, 24]. Monolítico não é sinônimo de legado e pode ser uma boa escolha escolha arquitetural dependendo do contexto [16]. Contudo, essa arquitetura tem suas limitações [24], pois à medida que os sistemas crescem em tamanho, começam a aparecer problemas como dificuldades de entender o código, alto acoplamento entre módulos, aumento do tempo de *deploy* e dependência de tecnologia a longo prazo [12, 25, 26]. Apesar dos investimentos em sistemas empresariais monolíticos, há uma tendência de desenvolver aplicações usando a arquitetura de microsserviços [27, 28].

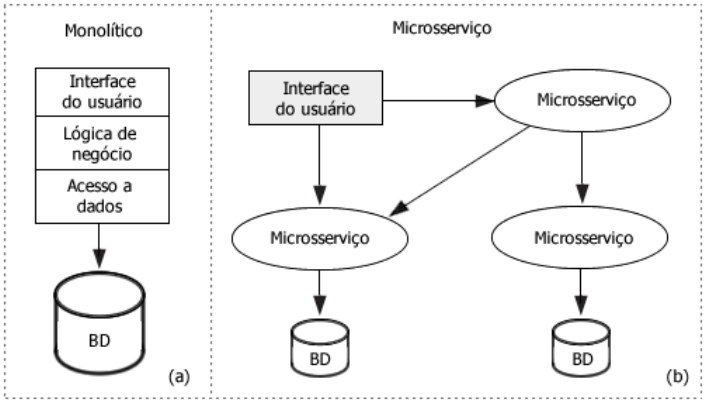


Figura 1.1: Representação arquitetural monolítica e de microsserviços.
(Fonte: própria)

A arquitetura de microsserviço é uma abordagem para desenvolver uma única aplicação como um conjunto de pequenos serviços construídos em torno de capacidades de negócios [8], em geral tendo seus próprios repositórios de dados [21] a exemplo da Figura 1.1(b). Caracteriza-se fundamentalmente pelo *deploy* independente e automatizado contendo um único serviço ou apenas alguns serviços coesos [29]. Microsserviços vêm ganhando popularidade [2, 22, 30, 31] por promover agilidade, escalabilidade, manutenibilidade [26] e outras significativas melhorias em atributos de qualidade de software [14, 28]. Contudo, há também *tradeoffs*: microsserviços têm associados a si toda a complexidade de sistemas distribuídos o que implica esforço adicional ao lidar com *deployment*, testes, monitoramento, além da desvantagem de propensas transações distribuídas [32]. Ainda assim, microsserviços surgem como resposta arquitetural a muitos dos problemas de software e oferecem como benefícios a entrega contínua, virtualização sob demanda, automação de infraestrutura, pequenas equipes independentes e sistemas escaláveis [32, 33].

Nas últimas décadas, inúmeras organizações começaram a desenvolver seus próprios sistemas corporativos baseados na arquitetura monolítica [3, 34], muitos deles hoje tendo se tornado uma *grande bola de lama* (do inglês *Big Ball of Mud*) [35, 36, 37]. Os efeitos colaterais dessa indesejável condição têm levado várias empresas a migrar seus sistemas monolíticos para microsserviços [2, 38]. Em geral, organizações que ingressam no processo de migração para microsserviços partem de grandes e complexas aplicações monolíticas que começam a sinalizar problemas [13, 27, 39], geralmente devido à dificuldade de realizar mudanças funcionais [2, 40], optando por introduzir gradualmente novos serviços capazes de substituir pequenas partes de seus sistemas corporativos até que predominantemente os sistemas estejam sob a arquitetura de microsserviços [19, 41, 42, 43]. Esse processo de migração controlado e contínuo passou a ser popularmente chamado de estrangulamento do monolítico com base no *Strangler pattern* descrito por Fowler [44, 45].

Migrar sistemas de uma arquitetura monolítica para microsserviços, no entanto, não é uma tarefa trivial [1, 3, 46] e requer determinados conhecimentos que podem frustrar até mesmo os mais experientes profissionais. Embora diversos estudos venham colaborando para tornar o processo de migração mais democrático e acessível por meio de táticas[1] arquiteturais e *patterns* já estabelecidos [47, 48, 49], pouco ainda se sabe sobre como arquitetos e desenvolvedores lidam com a migração a partir de monolíticos [3, 41]. Nesse contexto, o presente trabalho se propôs a examinar diferentes estudos, identificando as etapas e atividades comumente adotadas durante a migração. Os achados foram consolidados em um método denominado *Microservice Full Cycle* - MFC, de modo a reduzir a subjetividade e melhor compreender as nuances do processo de migração.

Dentre os motivos de escolha da arquitetura de microsserviços estão (i) a possibilidade de explorar os benefícios da computação em nuvem e (ii) a capacidade de modernizar incrementalmente aplicações monolíticas legadas [50, 51]. Por operar nativamente em ambiente de nuvem [1], microsserviços se alinham perfeitamente aos objetivos estratégicos do Exército Brasileiro (EB), âmbito de realização deste estudo. Além de iniciativas de nuvem privada como o *EBCloud* [52], o EB incentiva *"adotar serviços em nuvem, utilizando-se das melhores práticas já estabelecidas a nível mundial"*, conforme publicação do Boletim do Exército nº 46, de 17 NOV 2017 [53]. O principal interesse deste estudo, entretanto, está no fato de que microsserviços podem contribuir para a modernização de sistemas legados do EB, mitigando conhecidos problemas de software. A Figura 1.2 representa a expectativa do processo de migração utilizando o método MFC proposto, na qual um monolítico legado é, aos poucos, transformado em modernos microsserviços.

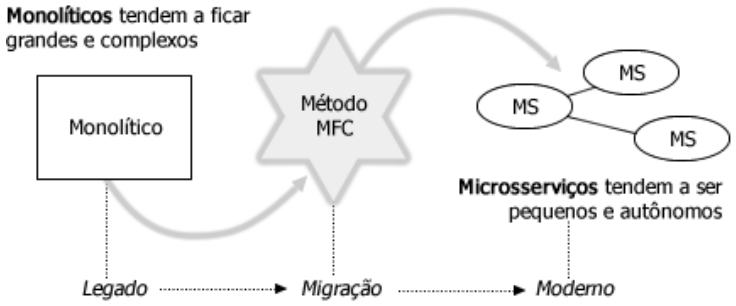


Figura 1.2: Expectativas do processo de migração.
(Fonte: própria)

[1]Táticas são técnicas de *design* que permitem alcançar um requisito de atributo de qualidade específico. Bass et al. [20], livro *"Software Architecture in Practice"*

O trabalho é conduzido segundo a metodologia de pesquisa *Design Science Research Methodology* - DSRM, descrita no Capítulo 4. Para efeitos práticos, o método MFC é submetido a uma simulação utilizando um sistema real que, por restrições de sigilo, será referenciado nesta dissertação pelo nome fictício SISP, termo para um genérico *Sistema de Seleção de Pessoal*. SISP é hoje o principal ativo de informações de uma importante Organização Militar (OM) do EB chamada pelo pseudo-nome DISP, uma alusão à *Diretoria de Seleção de Pessoal*. Cabe à DISP, dentre outras missões, realizar o processo seletivo de militares para cargos, cursos e estágios do Exército. Para isto, conta com o SISP, uma aplicação Web desenvolvida em 2008 na linguagem de programação PHP, utilizando framework *Codeigniter* e banco de dados *Oracle*. O SISP possui também um módulo especial chamado SISPNet de acesso exclusivo pela Internet para inscrições e acompanhamento online, conforme ilustra a Figura 1.3.

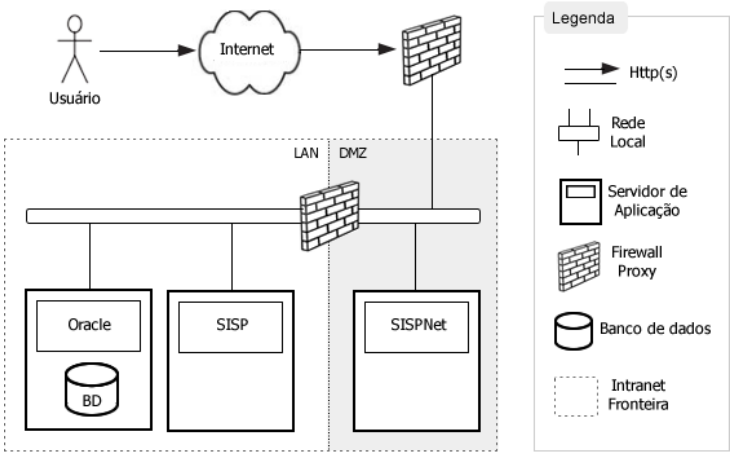


Figura 1.3: Diagrama de *deployment* - SISP.
(Fonte: própria)

1.1 Definição do Problema

No contexto deste trabalho, identificam-se dois **problemas: (i) a necessidade de modernização do legado[2]; e (ii) a falta de um método de migração**. Alguns pesquisadores [1, 2, 3, 4, 5, 6] tentaram traçar uma solução, mas não há consenso. Em geral, os métodos de migração existentes possuem características *"ad hoc"*, ou seja, cada qual é criado ou adaptado com a finalidade de resolver o problema específico de uma realidade. Abordagens muito específicas podem ser problemáticas quando (i) limitam-se apenas às

[2]Entende-se por *modernização* o refatoramento de sistemas legados a fim de alinhá-los às modernas necessidades de negócios [5]

partes sem oferecer uma visão do *todo*, (ii) tornam-se dependentes de ferramentas e tecnologias e (iii) denotam incapacidade de reúso em diferentes projetos. Por outro lado, um método totalmente genérico pode se mostrar impreciso em resolver um problema específico, bem como não é garantia de servir a todo e qualquer contexto. De fato, não existe um método universal que possa ser usado em todas as situações [54, 55].

Do ponto de vista arquitetural, o SISP é um sistema monolítico baseado no *pattern* Model-View-Controller (MVC) que não observa a rigor as restrições do *design* em camadas. Com mais de 12 anos de existência e sucessivas manutenções e expansões não-gerenciadas ao longo do tempo, o SISP tem resultado em indesejados alto acoplamento e baixa coesão de módulos (conforme avaliado pela ferramenta *SonarQube*), dificultando a refatoração ou criação de qualquer nova funcionalidade. Complexo e sem perspectiva de evolução sustentável estima-se que o SISP consuma 80% do tempo dos desenvolvedores em correções de *bugs* ou modificações *hardcoded* de regras de negócio e cubra somente 65% das necessidades internas (seções) da organização. Essa baixa efetividade fez com que silos de pequenas outras aplicações satélites (PHP, MS Access, Excel, Oracle Forms) surgissem para atender as demandas, complicando ainda mais o cenário. Em decorrência disso, alguns problemas de sistema de software surgiram e estão sintetizados na Tabela 1.1.

Tabela 1.1: Caracterização do problema *vs.* Atributos de Qualidade (QA).

Problema	**QA**
Código-fonte complexo, difícil de entender, modificar ou criar novas funcionalidades	Manutenibilidade
Lentidão em momentos de alta demanda com episódios de indisponibilidade	Desempenho
Ausência de comunicação com outros sistemas, retendo informações de interesse compartilhado	Interoperabilidade

Conforme se observa, os problemas do SISP não exatamente têm origem nos requisitos funcionais, mas em atributos de qualidade de software (requisitos não-funcionais). Cada problema afeta diretamente um atributo de qualidade de software, existindo diferentes táticas arquiteturais que podem ser utilizadas para contornar as limitações postas [20]. Assim, infere-se a partir das alegações supracitadas que o principal questionamento para o problema de pesquisa está em responder: ***"a transformação de sistemas monolíticos em microsserviços por meio do método Microservice Full Cycle - MFC é capaz de mitigar os problemas encontrados em aplicações legadas como o SISP, especialmente em relação a manutenibilidade, desempenho e interoperabilidade?"***

1.2 Justificativa

O SISP é hoje a principal fonte de informação sobre processo seletivo do pessoal de carreira do EB, tendo vital importância nos trabalhos internos desenvolvidos diariamente pela DISP. Em seu distinto papel social, o SISP opera com aproximadamente 63 mil usuários ativos e afeta todos os anos a vida de mais de 15 mil militares e suas famílias que, por força de lei e ofício, são selecionados e transferidos para diferentes regiões do Brasil. Toda essa logística e operacionalidade visa a capacitação e rotatividade da Força Terrestre inclusive em áreas remotas de fronteiras e operações militares como a *Intervenção Federal no Rio de Janeiro* (2018) e isto requer, sobretudo, Sistemas de Informações (SI) que operem com a máxima eficiência e economicidade de recursos públicos.

A relevância do SISP no âmbito do EB e seus problemas sistêmicos abordados na Seção 1.1, evidenciam a necessidade de modernizá-lo. Uma possível solução para lidar com a depreciação do legado é transformá-lo em microsserviços. Contudo, para maximizar os resultados, é importante que a pretendida modernização arquitetural seja alcançada metodologicamente. Segundo Balalaie et al. [1], a ausência de uma metodologia bem elaborada tende a resultar em um mero esforço de *"tentativa e erro"* que pode não apenas desperdiçar muito tempo, mas também levar a uma solução errada. Diversos autores [3, 4, 12, 46] constatam, no entanto, ser escassos os métodos que facilitem a migração de sistemas monolíticos para microsserviços. Esse possível *gap* de pesquisa [56] vem a confirmar que estudos acadêmicos relacionados a microsserviços estão em fase inicial e ainda não atingiram a maturidade [4], o que faz deste trabalho e a proposta de um novo método (MFC) necessária e justificável, sendo uma oportunidade singular para contribuir com os anseios da comunidade científica, do Exército Brasileiro (EB) e das demais organizações que buscam uma alternativa para modernizar sistemas legados.

Finalmente, pode-se justificar a viabilidade técnica desta pesquisa com base em casos de sucesso vivenciados por outros pesquisadores [2, 3, 12, 23, 41, 57, 58, 59]. Em especial, Luz et al. [41] relatam as lições aprendidas no processo de migração para microsserviços em três instituições governamentais brasileiras. Segundo consta, parte da motivação em migrar a arquitetura legada veio do excessivo custo de manutenção de software causado pelas frequentes correções de *bugs* e a difícil implementação de novas funcionalidades. Os resultados do estudo evidenciaram diversos desafios, como (i) a falta de entendimento ao decompor o monolítico em serviços, (ii) o forte acoplamento entre os componentes do sistema legado e (iii) a radical mudança requerida no processo de desenvolvimento. Contudo, os benefícios foram compensadores, propiciando (i) a redução do tempo e risco de desenvolvimento (*time-to-market*), (ii) o aumento da disponibilidade do software, e (iii) oportunidade de experimentar diferentes tecnologias.

1.3 Contribuição

Este trabalho contribui sob dois aspectos: acadêmico e institucional. Do ponto de vista acadêmico, o método MFC vem a minimizar o atual *gap* de pesquisa relatado por diversos autores [3, 4, 12, 46] no que diz respeito à escassez de métodos sistemáticos que suportem a migração de tradicionais sistemas monolíticos para microsserviços. A maioria das publicações sobre microsserviços se limita a tratar da motivação, benefícios ou desafios encontrados no processo de migração. Nos casos em que os estudos apresentam um método, *framework* ou processo para lidar com a migração, ficam restritos à etapa de decomposição, isto é, basicamente exploram formas de quebrar monolíticos em microsserviços. Contudo, sabe-se que a migração tende a ser bem mais abrangente, razão pela qual o *Microservice Full Cycle* - MFC traz em sua proposta um conjunto de práticas para evoluir o legado, preservando a natureza reprodutível metodologicamente (Capítulo 5).

Quanto à *contribuição institucional*, o MFC tem resgatado a capacidade de evolução de sistemas de software legado, fato constatado na simulação com o SISP, cujos resultados alcançados evidenciaram ganhos em (i) *manutenibilidade*: o sistema segmentado em pequenos "pedaços" facilita a compreensão do código-fonte, favorecendo o desenvolvedor alterar ou criar novas funcionalidades; (ii) *desempenho*: serviços autônomos e independentes contribuem para a escalabilidade seletiva de partes sobrecarregadas da aplicação, dando ganho às requisições demandadas; e (iii) *interoperabilidade*: a exposição de *interfaces*/contratos (API) possibilita a interação entre serviços e outros sistemas, propiciando que o software seja pensado e arquitetado a nível institucional, e não apenas setorialmente. Também é intrínseco ao MFC, os benefícios da arquitetura de microsserviços, tais como códigos mais coesos e desacoplados, pequenas equipes, independência tecnológica, automação de *build*, teste e *deploy* (CI/CD), escalabilidade sob alta demanda, maior interoperabilidade/integração e monitoramento/feedback em tempo real.

1.4 Objetivos

1.4.1 Objetivo Geral

O presente trabalho tem como principal objetivo **prover um método de migração de sistemas monolíticos legados para a arquitetura de microsserviços**. Em termos práticos, o método *Microservice Full Cycle* - MFC é constituído por um conjunto de etapas e atividades relevantes ao processo de migração consolidado a partir de diversas publicações científicas (Apêndice A). A pesquisa é validada em uma simulação [60] voltada a um módulo do SISP, em especial para aprimorar os atributos de qualidade de

software relativos a manutenibilidade, desempenho e interoperabilidade. Não é pretensão deste trabalho refazer por completo todo o sistema, mas verificar se a partir da estratégia *"Strangler Application"* [44] é possível que um novo sistema possa, aos poucos, ser construído em torno do antigo até que o legado seja totalmente *"estrangulado"*.

1.4.2 Objetivos Específicos

Para alcançar o objetivo geral, são necessários atender aos seguintes objetivos específicos:

1. Compreender as nuances do ciclo de desenvolvimento de microsserviços, baseando-se em trabalhos correlatos que tenham empregado métodos para migração;
2. Executar simulação para validar o método de migração proposto (MFC), adotando métricas e análise de resultado que atestem sua eficácia.

1.5 Estrutura do texto

Este documento está estruturado em sete capítulos. O Capítulo 2 apresenta a fundamentação teórica sobre o tema, discutindo diferentes conceitos e elementos das arquiteturas monolíticas e de microsserviços. No Capítulo 3, é feita uma revisão do estado da arte comparando diferentes métodos de migração para microsserviços, assimilando os aspectos mais relevantes da atualidade. O Capítulo 4 descreve a metodologia empregada na condução da pesquisa (DSRM). O Capítulo 5 descreve o método *Microservice Full Cycle* - MFC, definindo etapas e atividades propostas para guiar o processo de migração. O Capítulo 6 apresenta os resultados e análises da pesquisa, relatando os achados e impedimentos encontrados na simulação. O Capítulo 7 faz as considerações finais, descrevendo as limitações e oportunidades de melhorias a serem desenvolvidas em trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Esse capítulo apresenta a revisão dos principais conceitos teóricos relacionados a arquitetura de microsserviços, tendo como objetivo fundamentar as decisões e as práticas desenvolvidas durante o processo de migração. Os diversos tópicos abordados são majoritariamente baseados em artigos científicos revisados por pares, dentre os quais se discutem os mais diversos temas ligados à arquitetura de software monolítica e de microsserviços.

2.1 Arquitetura de Software

Arquitetura de software diz respeito à *tradeoffs* [61], isto é, a habilidade de decidir entre aspectos antagônicos, mediando compensações, custos e benefícios. Clements et al. [20, 62] definem a arquitetura de software como sendo um conjunto de estruturas necessárias para compreender os elementos de software, as relações entre eles e suas propriedades. A arquitetura vai além da formalidade de diagramas expressos por meio de textos ou representações visuais [49]; ela exerce influência direta sobre os atributos de qualidade de software (requisitos não-funcionais) [63, 64]. A necessidade de entrega de serviços para um número cada vez maior de usuários *online* e em tempo real, fez rapidamente crescer o desenvolvimento baseado em sistemas distribuídos [65, 66]. Isto tem sido possível graças a evolução de redes de comunicação de alta disponibilidade [67], levando a próxima geração de software a ser altamente escalável e capaz de interagir com ambientes de nuvem [68].

2.2 Arquitetura monolítica

Aplicações monolíticas são ainda o *modus operandi* de muitos sistemas corporativos [41]. Um monolítico é uma aplicação onde toda a lógica é executada em um único servidor [68, 69], isto é, cada serviço é *deployed* como uma única solução [70]. Neste tipo de abordagem, é comum as aplicações serem decompostas em arquitetura de camadas, tal como

o Model-View-Controller (MVC) [68] e terem vários módulos se comunicando [71]. De certa maneira, sistemas classificados como "monolíticos" podem ser também distribuídos, haja vista que uma simples aplicação local pode ler dados em diferentes bancos remotos e apresentar as informações para o cliente no *web browser*, o que significa pelo menos três computadores envolvidos se comunicando via rede [16].

Conforme mostra a Figura 2.1, um monolítico ainda é a melhor opção para iniciar um novo projeto, mesmo tendo a certeza de que futuramente a aplicação será grande o suficiente a ponto de justificar o uso de microsserviços [46, 69]. No artigo *"Monolith First"*, Fowler [72] afirma que (i) quase todas as histórias bem-sucedidas de microsserviço começaram com um monólito que ficou grande demais e foi dividido; e (ii) quase todos os casos de um sistema que foi construído como microsserviço a partir do zero, acabou com sérios problemas. Isto porque monolíticos são menos complexos de início se comparados aos microsserviços, principalmente devido à curva de aprendizado menor requerida [73].

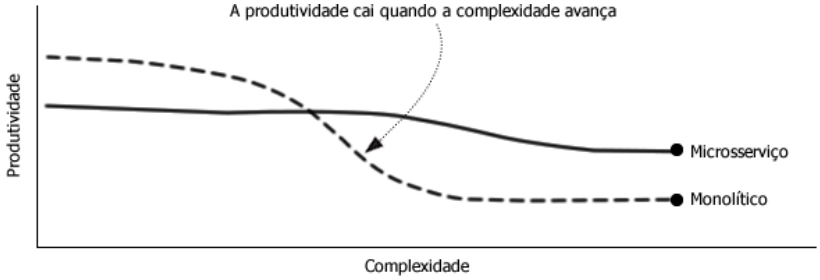


Figura 2.1: Complexidade vs produtividade entre monolítico e microsserviço.
(Fonte: baseado em Fowler [73])

Para Shimoda & Sunada [34] a estratégia chamada de *"monolith-first"* tem atraído atenção, pois aplicar microsserviços a um novo negócio que ainda não é bem compreendido para só então depois decompô-lo em serviços é uma tarefa difícil e arriscada. *"Monolith-first"* permite melhor explorar a complexidade do sistema, seus componentes e fronteiras e, à medida que a complexidade aumenta, introduz-se gradualmente os microsserviços [72]. Essa abordagem parece também seguir a *Lei de Gall* que afirma que *"um sistema complexo que funciona, invariavelmente tende a ser evoluído de um sistema simples que funcionou"* [74].

Contudo, há controvérsias [75]. Parafraseando Simon Brown [76], *"se você não pode construir um bem estruturado monolítico, o que o faz pensar que poderá construir um bem estruturado conjunto de microsserviços?"*. Para Stefan Tilkov [77] criar um monolítico primeiro é na verdade exatamente a coisa errada a se fazer. Isto porque se é difícil dividir um monolítico em microsserviços, então começar com um monolítico fará com que os serviços fiquem tão fortemente acoplados uns aos outros quanto de fato é um monolítico, além de que muitas partes dependem de bibliotecas e dados compartilhados.

Para ele, se você é capaz de criar um monólito bem estruturado, provavelmente não precisa de microsserviços. Em outras palavras, evite adicionar a complexidade de sistemas distribuídos em sua aplicação se não tiver um bom motivo para isso [73]. Neste sentido, David Linthicum [75] ressalta ainda que nem toda aplicação monolítica pode ser facilmente migrada para a arquitetura de microsserviço. Às vezes, é mais economicamente benéfico reconstruir a aplicação do zero do que refatorá-la, sobretudo, se a aplicação (i) é muito antiga e usa linguagem e banco de dados obsoletos; (ii) possui um *design* ruim; e (iii) é fortemente acoplada ao banco de dados [43].

Newman [16] classifica uma aplicação como sendo monolítica quando todas as funcionalidades do sistema podem ser *deployed* juntas. Três tipos de monolíticos são basicamente identificados: (i) *monolítico de processo único*: representa a maioria dos monolíticos, quando todo o código é *deployed* em um único processo - Figura 2.2(a). Uma variação deste tipo é o *monolítico modular*, quando existe no monolítico a divisão em módulos independentes, mas ainda sim faz parte de um único *deployment* - Figura 2.2(b). Um outro tipo de monolítico modular é o *monolítico modular com banco de dados decomposto*, talvez o mais desafiador em termos de *deploy* - Figura 2.2(c); (ii) *monolítico distribuído*: consiste em múltiplos serviços separados, mas não importa a razão, o sistema inteiro precisa ser *deployed* junto. Isto pode ser algo como um SOA-monolítico; o monolítico distribuído possui todas as desvantagens de sistemas distribuídos e todas as desvantagens do monolítico de processo único - Figura 2.2(d); por fim (iii) *monolítico caixa preta de terceiros*: são softwares desenvolvidos por outros, sem acesso ao código-fonte, como softwares de prateleiras com *deploy* na própria infraestrutura. Pode ser, ainda, um produto consumido, tipo *SaaS* (do inglês *Software-as-a-Service*). Neste caso, existe a dependência de *deploy* realizado por terceiros - Figura 2.2(e).

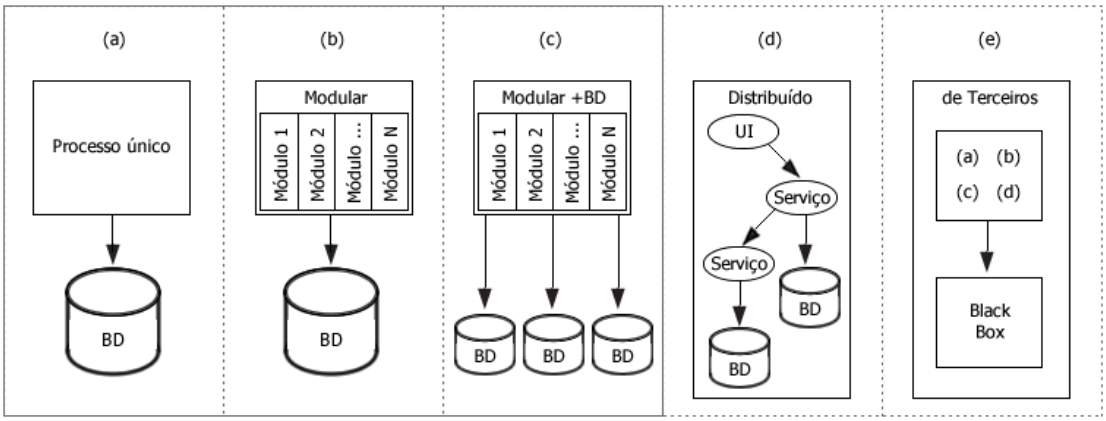


Figura 2.2: Tipos de monolíticos.
(Fonte: baseado em Newman [16])

Monolíticos também têm seus infortúnios. Uma aplicação monolítica é caracterizada por ser uma base de código única, quase sempre grande e complexa, o que implica uma série de desafios [69, 78]. Nestas condições, monolíticos não são facilmente mantidos ou evoluídos [79], sobretudo, pela dificuldade que novos desenvolvedores encontram para se familiarizar com o código-fonte e ingressar à equipe em projetos [68, 69]. Por depender de recursos compartilhados, monolíticos não são independentemente executáveis [6, 79, 80], cada módulo é parte integrante de toda aplicação. Isto dificulta a escalabilidade do módulo separadamente quando surgem gargalos na aplicação. Monolíticos também restringem o uso ou escolhas de diferentes tecnologias [11, 81], motivo pela qual não é comum encontrar nos monolíticos mais que uma linguagem de programação, plataforma de *deployment*, sistema operacional ou tipo de banco de dados e assim por diante [16]. Para superar estas e outras limitações monolíticas, a arquitetura de microsserviços tem emergido [82, 83]. Um estudo pioneiro da IBM [69] destaca algumas características essenciais da arquitetura monolítica em comparação aos microsserviços (Tabela 2.1), o que basicamente demonstra que microsserviços são o oposto a monolíticos [2].

Tabela 2.1: Comparação da arquitetura monolítica e de microsserviços.

Categoria	**Monolítico**	**Microsserviços**
Código	Base única de código para toda a aplicação.	Múltiplas bases de códigos. Cada microsserviço tem sua própria base de código.
Facilidade de entender	Frequentemente confuso e difícil de manter.	Maior legibilidade e mais fácil de manter.
Deployment	Complexo *deployment* com janela de manutenção e *downtime* (interrupção) programado.	Simples *deployment*, cada serviço pode ser *deployed* individualmente, com mínimo ou nenhum *downtime*.
Linguagem	Tipicamente é desenvolvido em uma única linguagem de programação.	Cada microsserviço pode ser desenvolvido em uma diferente linguagem de programação.
Escalabilidade	Exige escalar toda a aplicação ainda que o gargalo seja localizado.	Permite escalar apenas serviços com gargalos, sem escalar toda a aplicação.

2.3 Arquitetura de microsserviços

A arquitetura de microsserviços surgiu como uma nova alternativa ao estilo monolítico para projetar grandes e complexos sistemas de software [3, 84, 85]. Assim como no Desenvolvimento Baseado em Componentes (do inglês *Component-based Development* - CBD), microsserviços se objetivam a reduzir a complexidade do processo de desenvolvimento de software, facilitar a manutenção e dar suporte às operações de TI [31]. A nomenclatura

"Microsserviço" como a conhecemos hoje é ainda anterior às primeiras publicações em 2014 por James Lewis e Martin Fowler [8, 33] e na verdade remonta 2012 quando arquitetos de softwares pós *workshop* em Veneza a usaram para descrever um estilo arquitetural comum ao que muitos deles estavam explorando na ocasião [8, 79].

2.3.1 Conceito

Microsserviço é um estilo arquitetural para desenvolver uma única aplicação como uma suíte de pequenos serviços, cada um executando em seu próprio processo e se comunicando, geralmente, por meio de HTTP/REST API [1, 8]. Apesar desta ser a mais clássica definição de microsserviços [57], ainda não existe um consenso a respeito [30]. A IBM [69], por exemplo, define microsserviço como um estilo arquitetural em que grandes e complexas aplicações são compostas por um ou mais serviços. Para Tserpes [86], o termo "microsserviços" se refere a um conceito de arquitetura de software pelo qual a funcionalidade de uma aplicação é decomposta em funções menores e independentes residindo em recursos próprios de infraestrutura. Na prática, microsserviços envolvem diferentes conceitos que emergiram de boas práticas [33] e da comunidade de desenvolvedores ágeis [12], estando fortemente conectados à abordagem *DevOps* [19, 31].

2.3.2 Características

Autores em diferentes estudos publicados descrevem inúmeras características gerais sobre microsserviços. Na perspectiva arquitetural, Merson [29] considera que microsserviços pertencem majoritariamente à visão de *deployment* de modo que cada unidade de *deployment* deve conter apenas um único serviço ou apenas alguns serviços coesos. Para ele, esta "restrição de *deployment*" é o fator diferencial que caracteriza e distingue a arquitetura de microsserviços. Dragoni et al. [79] e Alshuqayran et al. [15] destacam como principais características da arquitetura de microsserviços a sua natureza modular e distribuída. Para Balalaie et al. [1], microsserviços contribuem especialmente na entrega mais frequente de novas *features* e em prover escalabilidade. Friedrichsen [87] e Ghani & Shanudin [4] citam que a principal característica dos microsserviços é produzir um serviço escalável, coeso, de baixo acoplamento, distribuído e descentralizado [4, 87]. Já de acordo com Alshuqayran et al. [88] escalabilidade, independência, manutenibilidade, *deployment*, monitoramento e modularidade são os atributos mais mencionados sobre microsserviços na literatura. A seguir, são descritos os mais relevantes fatores inerentes aos microsserviços.

Custo de implementação. Hasselbring e Steinacker [83] alertam para os custos que a arquitetura de microsserviço possui, principalmente por se tratar de um sistema dis-

tribuído. Segundo Taibi et al. [2], estimar o tempo de desenvolvimento de um sistema baseado em microsserviços é menos preciso do que em monolíticos. Em início de projeto, relatos mostram que microsserviços exigem um esforço adicional entre 20% e 30% a mais em comparação ao esforço necessário para desenvolver uma solução monolítica. Microsserviços também requerem maquinaria extra, o que pode elevar os custos substancialmente. É preciso levar em consideração que a adoção de microsserviços implica uma infraestrutura *DevOps*, e isto significa muito mais esforço em termos de curva de aprendizado, desenvolvimento e uso de ferramentas. Os desenvolvedores precisam ser mais experientes, capazes de decompor o sistema e desenvolver novos serviços autônomos dissociados do monolítico, o que também inclui migrar os dados e prover novos mecanismos de comunicação e orquestração entre os serviços. As bibliotecas existentes não podem simplesmente serem reutilizadas, mas devem ser idealmente convertidas em um ou mais microsserviços [2]. É importante destacar que microsserviços levam mais tempo para obter o retorno do investimento (do inglês *Return on Investment* - ROI), isto, porém, é compensado a longo prazo pela melhor manutenibilidade [2].

Cultura organizacional. Taibi et al. [2] ressaltam que modificar arquiteturas existentes costuma ser um grande problema para vários desenvolvedores, principalmente devido a mentalidade das pessoas. Desenvolvedores mais antigos nem sempre acreditam em tecnologias recentes. Além disso, muitos deles consideram o sistema legado como sua própria criação e relutam em aceitar uma mudança tão substancial no software que eles escreveram. Escolher uma nova tecnologia que refaz por completo todo o atual sistema de uma organização geralmente é percebida como arriscada e, como esperado, desenvolvedores e arquitetos mais antigos tendem a ser mais conservadores do que os mais novos ao adotar recentes tecnologias [2].

Responsável pelo serviço (*ownership*). Newman [16] defende que todo serviço deve ter um dono ou responsável, a quem se submete qualquer proposta de mudança. Havendo mudança no contrato do serviço, às partes que consomem o serviço precisam ser comunicadas de modo que não venham a quebrar. No caso de existirem poucos serviços em produção pode até fazer sentido que o código do microsserviço seja de responsabilidade coletiva, porém, isto não é aconselhável para equipes maiores. Uma equipe responsável por um serviço que seja orientado ao domínio do negócio tende a estar mais focada e especializada sobre aquele negócio, facilitando eventuais manutenções [16]. Para Taibi et al. [2], como os microsserviços possuem poucas dependências externas, eles podem ser desenvolvidos por diferentes equipes de forma independente, reduzindo a sobrecarga de comunicação e a necessidade de coordenação entre as equipes [2]. Cada equipe possui sua própria base de código e pode ser responsável pelo desenvolvimento de cada serviço. A

distribuição de responsabilidades claras e independentes permite dividir grandes equipes de projeto em várias equipes pequenas e mais eficientes. Além disso, como os microsserviços podem ser desenvolvidos com diferentes tecnologias e com uma estrutura interna totalmente diferente, muitas decisões técnicas podem ser tomadas pelas próprias equipes, deixando apenas decisões técnicas de alto nível para serem coordenadas entre equipes [2].

Segurança. Proteger monolíticos é relativamente mais fácil que microsserviços. Um sistema baseado em microsserviço requer que os microsserviços se comuniquem entre si pela rede e isso expõe mais dados e informações (*endpoints*) sobre o sistema e, assim, expande a superfície de ataque. Além disso, mensurar a segurança em microsserviços também é mais desafiador, pois, por exemplo, proteger igualmente todos os serviços pode causar uma sobrecarga no desempenho e diminuir a disponibilidade do sistema [46]. É preciso melhorar a segurança dos dados dos usuários e regular a privacidade de dados, a exemplo do que prevê o *Regulamento Geral sobre a Proteção de Dados* (RGPD), da União Europeia - UE [89] ou a Lei brasileira Nº 13.709, de 14 de agosto de 2018.

Tecnologia. Microsserviços são tecnologicamente agnósticos [16]. Sua natureza "poliglota" permite o uso de distintas tecnologias, seja a nível de ambiente operacional, persistência (banco de dados) ou linguagem de programação [24, 90]. Embora microsserviços aceitem distintas tecnologias, organizações maduras frequentemente limitam a quantidade de tecnologias utilizadas [16]. Segundo Newman [16], é preciso ter uma visão mais ampla e analisar se pelo lado da organização é desejável manter múltiplas tecnologias, até porque, será necessário depender de novas habilidades ou pagar por variadas licenças em nome da otimização de determinados serviços. Para ele, talvez seja melhor adotar apenas um tipo de linha de ação (ou tecnologia) aceitando que isto possa até não ser perfeito para todos os serviços e equipes, mas que é bom o suficiente para a maioria.

DevOps. O termo *DevOps* surgiu da combinação de dois termos ágeis intimamente relacionados: "operações" (infraestrutura) e "desenvolvimento"[91]. *DevOps*, no entanto, não significa que desenvolvedores fazem todas as operações, substituindo os profissionais de operações, mas se trata de um movimento cultural que busca quebrar as barreiras e melhorar a comunicação, colaboração e integração entre desenvolvedores de software (Dev) e profissionais de operações de TI (Ops) [16, 83], promovendo um relacionamento eficaz e de ajuda recíproca entre as equipes por meio de *feedbacks* [1]. A ideia é facilitar o alinhamento e entendimento entre as pessoas envolvidas na entrega do software, não importando qual papel ou responsabilidades tenham [16], a fim de reduzir o tempo de lançamento do software no mercado (do inglês *time-to-market*) e trazer agilidade a todas as fases do ciclo de vida de desenvolvimento de software [23].

A arquitetura de microsserviços está fortemente conectada ao *DevOps*, que herda seus princípios básicos de metodologias ágeis [19, 31]. Para Balalaie et al. [1], microsserviços se tornaram popular quando as práticas *DevOps* ganharam impulso na indústria de software. Pode-se dizer, portanto, que *DevOps* é suportado por microsserviços [2]. As técnicas *DevOps* são comumente usadas para automatizar o processo de desenvolvimento e operação através da integração contínua (do inglês *Continuous Integration* - CI), entrega contínua (do inglês *Continuous Delivery* - CD) e *deployment* contínuo (do inglês *Continuous Deployment* - CDE) [3]. A automação é a chave para o sucesso *DevOps* [92].

CI, CD e CDE. No âmbito *DevOps*, pode-se distinguir CI, CD e CDE da seguinte maneira: a *integração contínua* - CI é uma prática de desenvolvimento de software na qual os desenvolvedores frequentemente integram seu trabalho, em geral, diariamente [93], começando pelo *merge* do novo código em uma *branch* principal [94, 95]. Segundo Balalaie et al. [49], CI automatiza o processo de *build* e teste mantendo a base de código (repositório) pronta para ser enviada à produção, o que somente é feito por intervenção humana [48]. Primeiro, cada serviço deve estar separado em seu próprio repositório. Então, uma CI deve ser criada para cada serviço. Sempre que o código no repositório mudar, um *job* no "servidor CI" é disparado e tem a responsabilidade de buscar o novo código no repositório e executar os devidos testes [32, 49]. Neste momento, outras operações automatizadas também podem ser realizadas, como a geração de artefatos e *benchmarks* de performance [92]. Ao final do processo de CI, a equipe de desenvolvimento deve ser informada de possíveis erros. Uma regra simples na CI é que novas mudanças não devem prejudicar a estabilidade do sistema, sendo obrigatório passar em todos os testes predefinidos [49]. Quanto antes testar o código, mais cedo os problemas são identificados [93]. CI torna-se mais fácil com pequenas equipes e menores bases de códigos [94].

CI geralmente é acompanhado pela *entrega contínua* - CD, que se refere ao *release* de todos *builds* que passaram nos testes e verificações de qualidade e estão prontos para o ambiente de produção [93]. Prática de CD requer, portanto, prática de CI [96], conforme mostra a Figura 2.3. Com CD a entrega de software é feita automaticamente para um ambiente similar à produção (*stage*, *release* ou *test*, por exemplo) [96, 97], sem no entanto efetivar o *deploy* que ocorre apenas quando é manualmente acionado [48, 96]. Assim, antes de ir para produção o código pode ainda passar pelo crivo de outros usuários, como Testers/QA/QC [95], sendo submetido a teste de aceitação, testes manuais, dentre outros. Esta prática oferece benefícios como risco reduzido, menor custo e *feedback* do usuário mais rápido [32, 96]. Em resumo, CD garante que todas as mudanças em uma aplicação foram seriamente testadas e estão prontas para serem *deployed* [98].

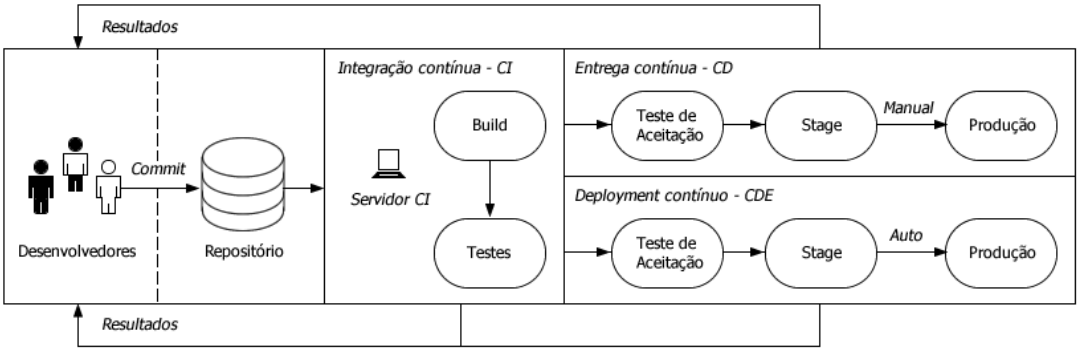


Figura 2.3: Relação entre CI, CD e CDE.
(Fonte: Adaptado de Shahin et al. [96])

Toda mudança (*check-in*) é um candidato à *release* [32]. Sempre que uma mudança é *committed* no controle de versão, a expectativa é que ela passe em todos os testes e seja liberada para produção [99]. O *deployment* contínuo - CDE é um *pipeline* completo neste sentido, colocando as mudanças do software em produção sem intervenção humana [48]. Basicamente isto significa que CDE vai um passo além do CD e automaticamente efetiva o *deploy* da aplicação, desde que os critérios de qualidade tenham sido atendidos [98]. É importante notar que a prática de CDE implica prática de CD, mas o inverso não é verdadeiro [99]. O que diferencia CD e CDE é, portanto, o ambiente de produção (ou seja, clientes reais usando a aplicação), sendo CDE consequentemente uma continuação do CD [96]. Na maioria dos casos, os pipelines de CDE fazem o *deploy* em um pequeno subconjunto de sistemas de produção usando o *canary deployment pattern*, faseados para outros sistemas depois que o código se mostrar estável em produção [48]. A ideia é entregar valor de negócio o mais rápido possível e não acumular código novo em *stage* [95]. Contudo, dado o nível de automação, é possível que CDE seja adequado apenas para certos tipos de organizações ou sistemas com processos mais maduros [96, 99].

API. Todo microsserviço precisa se comunicar [2] e, para isto, tipicamente utilizam API (do Inglês *Application Programming Interface*) e o estilo arquitetural REST (do inglês *Representational State Transfer*) [22, 23, 32]. Microsserviços adotam APIs como canais para atender aos usuários. É similar a uma "função" na programação, que é composta pelo nome da função, parâmetros de entrada e a saída a ser gerada [4]. APIs também funcionam como contratos entre fornecedor e consumidor. Segundo Newman [16], cada funcionalidade exposta pelo microsserviço deve ser pensada em termos de contrato. A recomendação é que o contrato seja explícito para quem o consome, tomando o cuidado de não quebrá-lo quando o serviço for modificado. Ter interfaces (API) estáveis entre serviços é essencial se o objetivo é manter a capacidade de *deploy* independente (do inglês

independent deployability). Uma vez exposta a API, qualquer mudança na interface irá repercutir em outros serviços que precisarão mudar também. Por isto é importante expor tão pouco quanto possível uma fronteira de um microsserviço, pois uma vez que algo se torna parte de uma interface, é difícil voltar atrás. Contudo, deve-se estar atento, pois há fortes evidências de que microsserviços tendem a aumentar a personalização do software a pedido de usuários [100]. Não sendo possível evitar a quebra do contrato, deve-se fazer a mudança de tal forma que não impacte o consumidor ou, em último caso, que se minimize o impacto comunicando ao consumidor o novo contrato e dando a ele tempo para migrar. Isto basicamente pode ser feito de duas formas: (i) criando duas versões distintas do microsserviço ou (ii) manter uma única versão do microsserviço com suporte a ambos contratos, o que parece ser uma melhor opção. Caso a mesma equipe responda tanto pelo serviço modificado quanto pelo serviço consumidor, basta que ambos ganhem uma nova versão do *release* e *deploy*, embora esta prática não deva ser um hábito [16].

Em termos de API, microsserviços mal projetados aumentam significativamente a complexidade de comunicação do sistema, razão pela qual é essencial encontrar a granularidade correta dos microsserviços [83]. Uma abordagem conhecida como "*outside-in*" sugere inicialmente pensar a interface do serviço do ponto de vista dos consumidores do serviço para só então se planejar quanto ao contrato do serviço. Muitas equipes ao desenhar o serviço levam em conta modelos de dados e detalhes internos de implementação, enquanto o melhor seria perguntar: o que os consumidores do serviço precisam? O contrato de serviço com o mundo exterior pode ser construído como uma interface/tela de usuário. Portanto, se para desenhar uma interface é comum a interação com o cliente para saber o que ele quer, também o contrato de serviço assim deve ser [16].

Coesão. Um sistema deve ser projetado de modo que cada módulo seja responsável por apenas uma "funcionalidade de negócio" [32]. O objetivo disso é alcançar a alta coesão mantendo juntas coisas que mudam por uma mesma razão [101], o que equivale dizer *"o código que muda junto, permanece junto"* [16]. Para sistemas baseados em microsserviços, a alta coesão é alcançada agrupando processos de negócios comuns, de modo que, se o desenvolvedor precisar alterar uma *feature*, apenas um único microsserviço seja alterado [32, 102]. Portanto, a ideia é que cada microsserviço venha a ser desenhado para fazer uma única tarefa, e fazê-la muito bem, princípio esse herdado da filosofia Unix [13, 21, 74]. Segundo Killalea [103], o mais fundamental dos benefícios por trás dos microsserviços é justamente a clara *"separação de conceitos"*[1], fazendo com que cada serviço concentre a atenção em algum aspecto bem definido da aplicação; isso torna o desenvolvimento bastante simplificado [2].

[1]Separação de conceitos (do inglês "Separation of concerns") é um termo cunhado por Edsger W. Dijkstra [104] em 1974 no *paper "On the role of scientific thought"*

Para Ahmed et al. [28], a técnica de agrupamento (do inglês *clustering*) permite identificar em um sistema os componentes com responsabilidades correlacionadas. Assim, sistemas grandes devem ser decompostos em blocos menores e gerenciáveis de modo que as entidades que possuem similaridades entre si fiquem agrupadas no mesmo subsistema, enquanto as entidades com diferenças entre si, sejam classificadas em subsistemas diferentes. Técnicas de agrupamento como *Hierarchical Agglomerative Clustering* (HAC) e *K-means clustering* podem ajudar a descobrir o número de *clusters* ou segmentos que causam transtornos, além de contribuir para que a alta coesão e o baixo acoplamento sejam alcançados [28]. Embora bastante presente na literatura, a coesão é difícil de medir devido à sua essência semântica. Uma possível forma de mensurá-la, entretanto, é através das interfaces expostas e da semelhança entre os tipos de dados dos parâmetros declarados, bem como da quantidade de operações disponíveis por serviço [19].

Acoplamento. Um serviço é fracamente acoplado quando sua atualização funcional não requerer mudanças em outro serviço [105]. Acoplamento, portanto, define o grau de dependência entre componentes ou módulos de uma aplicação [6]. Por exemplo, se A é acoplado a B em termos de implementação, quando B mudar, A também mudará [16]. Para Newman [16], o conceito mais importante ao criar microsserviços é o da capacidade de *"deploy independente"* (do inglês, "*independent deployability*"). É preciso assegurar que os serviços sejam fracamente acoplados, pois fazer mudanças em dois serviços é mais oneroso que fazer a mesma mudança em um único serviço (ou monolítico) [16]. Apesar de indesejável, o acoplamento não pode ser totalmente eliminado porque os serviços precisam se comunicar através de suas interfaces. O grande problema é quando os desenvolvedores tornam esse acoplamento muito pior do que deveria ser. Na prática, um serviço pouco acoplado sabe o mínimo necessário sobre os serviços com os quais colabora. Por esta razão, o ideal é que os desenvolvedores tentem limitar o número de chamadas de um serviço para outro, sempre que possível [32, 106]. A lei que rege uma colaboração mais desacoplada deveria seguir a Lei de *Postel*: *"seja conservador no que faz [ou envia], seja liberal no que aceita dos outros"* (Jon Postel) [106]. Diversas técnicas podem contribuir para evitar o acoplamento, definir a correta granularidade dos serviços (*contexto delimitado*) por meio do *Domain-Driven Design* - DDD é uma delas [107]. Além disso, serviços devem se comportar influenciados exclusivamente pelo seu contrato, isto é, pela API. Definir adequadamente uma API permite que mudanças de implementação não afetem os demais serviços, tornando-os autônomos e independentes inclusive de tecnologias [21, 32, 108]. Dentro do possível, cada serviço deve ser operacionalmente independente um dos outros, sendo a única forma de comunicação entre os serviços as suas interfaces publicadas [109].

O acoplamento é, portanto, representado pelas dependências e conexões de um microsserviço para com os outros e, havendo dependências circulares, estas devem ser evitadas.

Na realidade, microsserviços propensos à referência circular devem se tornar candidatos a uma fusão [19]. A grande vantagem de arquiteturas independentes onde nada é compartilhado (do inglês *shared-nothing architectures*) é a excelente escalabilidade horizontal e a melhor tolerância a falhas. Isto porque, se dois componentes não estão compartilhando nada, eles obviamente são incapazes de ter impacto negativo um no outro [83]. O grau de acoplamento pode ser calculado pelo número de chamadas para um microsserviço dividido pelo número de chamadas que o microsserviço está fazendo em relação a outros. Neste caso, microsserviços com forte acoplamento podem ser candidatos à refatoração. Um método popular para testar o acoplamento entre microsserviços é usar o *Docker Swarm*, que fornece isolamento total e desacoplamento aos *containers*, juntamente com um analisador de protocolo de rede (Ex: *WireShark*) para identificar requisições e dependências [19].

Em resumo, o fato é que o conceito de independência encoraja tanto o fraco acoplamento quanto a alta coesão [109], até porque, em termos de arquitetura, uma estrutura é estável se o acoplamento é baixo e a coesão é alta (Larry Constantine) [16]. Cabe destacar que *acoplamento* tem mais a ver com o quanto mudar uma coisa requer uma mudança em outra; e *coesão* diz mais sobre agrupar código relacionado [16]. Estes conceitos condizem com o *"princípio da responsabilidade única"* definido por Robert C. Martin [101]: *"Reúna as coisas que mudam pela mesma razão. Separe as coisas que mudam por diferentes razões"* [110]. Assim, a recomendação é que aplicações construídas a partir de microsserviços devem ser o mais desacopladas e coesas possível [32, 106].

Tamanho. Em microsserviços tamanho é uma medida relativa [74] e não existe atualmente um consenso do tamanho desejado de um microsserviço [57]. Para Cojocaru et al. [57] o tamanho, também conhecido por granularidade, é o atributo de qualidade mais controverso de um microsserviço e que constantemente enseja na questão: quão grande (ou pequeno) deve ser um microsserviço? A resposta não é conclusiva, mas o DDD pode fornecer alguns *insights*, basicamente dando subsídios para alcançar o *"business case"* com o menor número possível de dependências [33]. No geral, a noção mais aceita sobre o tamanho de um serviço é que ele deva ser capaz de fazer "uma coisa" corretamente. Contudo, a noção de "uma coisa" não é muito clara, isto é, não se sabe até que ponto as capacidades de negócios devem ser divididas para definir um serviço [81]. Para Chris Richardson [47] um microsserviço deve ser tão pequeno quanto possível. Neste caso, pequeno o bastante não significa ser o menor possível [32]. Nota-se, porém, que o termo "pequeno" é também semanticamente subjetivo. Em razão disto, conclui-se que o conceito de tamanho é na verdade uma das coisas que menos importa [16]. Ao invés de pensar em tamanho, o melhor seria se preocupar em manter os vários serviços independentes, principalmente se forem centenas deles [1, 6].

De qualquer modo, o tamanho representa um conceito crucial para microsserviços e

traz grandes benefícios em termos de manutenibilidade e extensibilidade do serviço. Na prática, se um serviço for muito grande, ele deve ser refinado em dois ou mais serviços, preservando a granularidade e mantendo o foco em prover uma única capacidade de negócio [109]. Microsserviços menores são mais fáceis de manter e mais tolerantes a falhas [102]. Novas arquiteturas tendem a diminuir ainda mais o tamanho dos serviços, com os chamados "nanosserviços" (conforme citado nas arquiteturas *Microsoft Azure Functions* e *lambda/serverless*) [57]. Dentre as métricas usadas para mensurar o tamanho de microsserviços estão [111, 112]: (i) o número de linhas de código (LoC) [19, 57]; (ii) a capacidade de reescrever um microsserviço dentro do período de 2 a 6 semanas; (iii) a composição de uma equipe *"two-pizza"*, quantidade esta que seria suficiente para alimentar toda a equipe [32]; (iv) o número de interfaces expostas [19]; (v) o consumo de recursos do servidor; e (vi) o número de *features* [57].

Monitoramento. Segundo Newman [16], o modo de lidar com o monitoramento e a solução de problemas precisa mudar à medida que a arquitetura de microsserviço cresce. Em um estágio avançado, tudo tende a ser mais complexo, ficando difícil prever quando o sistema vai quebrar e deixar de operar. Para prevenir possíveis danos, é importante ter mecanismos que façam a coleta de dados, ações de monitoramento e emissão de alertas quando algo parecer errado com o sistema, como por exemplo: (i) *agregação de logs*: um sistema central de *logs* permite capturar diferentes eventos e gerar alertas quando necessário, a exemplo de ferramentas como o *ELK* ou *Elastic stack* (*Elasticsearch*, *Logstash* e *Kibana*) [113]; (ii) *rastreamento*: útil para analisar uma sequência de chamadas entre microsserviços que falharam ou causaram latência, por exemplo. Para isto, todas as chamadas que chegam aos microsserviços devem receber inicialmente um ID (*correlation identifier*), que pode ser gerado por um API *gateway* ou serviço específico. Esse ID é então retransmitido adiante a outros serviços, se for o caso, via cabeçalho HTTP ou mensagem *payload*, de modo que, a cada nó o ID é persistido nos *logs*, possibilitando posterior rastreabilidade; (iii) *testes*: em geral, testes funcionais automatizados nos dão o *feedback* da qualidade do software antes do *deploy* da aplicação. É possível testar o comportamento esperado da aplicação simulando transações de usuários *fakes* e emitindo alertas de conformidade, se necessário [16].

Aplicações em microsserviço valorizam muito o monitoramento em tempo real, o que geralmente pode ser feito coletando (i) *"métricas técnicas"* como por exemplo a quantidade de solicitações por segundo que o banco de dados está recebendo, e (ii) *"métricas de negócios"* relevantes, como a quantidade recebida de pedidos por minuto [83]. Este tipo de monitoramento altamente influenciado por práticas *DevOps* permite que a arquitetura de microsserviço obtenha o *feedback* necessário para aperfeiçoar o sistema de modo mais rápido e fácil [3]. O monitoramento de cada microsserviço individualmente permite se

antecipar a eventuais falhas e anomalias em tempo de execução [83]. Um das formas de detectar estas anomalias é utilizando um modelo estatístico treinado, baseando-se nos dados de monitoramento *on-line* de conhecidas situações normais. Em seguida, cada novo dado recebido é calculado por um módulo de detecção de anomalias indicando possíveis discrepâncias (*outliers*) [1]. Para ter capacidade de monitoramento, cada serviço deve possuir uma interface que entregue as informações de monitoramento. Um jeito inicialmente simples e muito importante de implementar isto é prover o monitoramento da saúde do microsserviço por meio do status "*ok*" e "*broken*", permitindo que outros serviços verifiquem a disponibilidade e evitem chamadas a serviços momentaneamente interrompidos [33]. O gerenciamento da saúde (do inglês *Health Management*) é um atributo de qualidade que descreve a capacidade de um microsserviço de lidar com falhas. Um microsserviço é resiliente a falhas, sobretudo, quando é capaz de salvar o estado interno e reiniciar automaticamente recuperando o estado mais atualizado antes da falha [19]. Atualmente existem várias ferramentas para monitorar aplicações em nuvem, como por exemplo *Docker Stats*, *cAdvisor*, *DataDog*, *Amazon cloudWatch*, CLAMS, entretanto, a maioria dessas estruturas é específica de um determinado provedor de nuvem e não consegue satisfazer os requisitos de monitoramento de desempenho de microsserviços complexos implantados em vários data centers [114].

Quando não usar microsserviços. Uma análise feita por Newman [16] destaca quatro situações em que nem sempre microsserviços é a solução mais adequada: (i) *em domínios não claros*: definir erroneamente as fronteiras dos serviços pode custar caro, tornando os componentes excessivamente acoplados. Decompor prematuramente um sistemas em microsserviços pode ser perigoso, especialmente se o domínio é algo novo; (ii) *em startups*: *startups* como Netflix e Airbnb apenas fizeram uso da arquitetura de microsserviços a medida que evoluíam. *Startups* estão frequentemente experimentando várias ideias até encontrar uma que caia no gosto do consumidor. Isto implica constantes mudanças no produto explorado e, consequentemente, um domínio instável. Microsserviços podem ser mais indicado para resolver problemas quando já se tenha inicial sucesso nos negócios. Começar com monolítico pode ser uma forma de melhor compreender as fronteiras do sistemas, ganhando tempo para amadurecer os conceitos do ponto de vista operacional; (iii) *em softwares instalados e gerenciados pelo cliente*: microsserviços colocam muita complexidade dentro do domínio de negócio. Equipes que empregam microsserviços precisam compensar os desafios buscando novas habilidades e tecnologias, como *cluster Kubernets*, e isto não é o tipo de coisa que se pode esperar de um usuário final; (iv) *em casos que não se tenha uma boa razão*: ao adotar microsserviços é preciso ter uma clara ideia do que exatamente está se tentando alcançar, não use microsserviços apenas porque todo mundo está usando [16].

Automação. Mazzara et al. [109] ressaltam que tanto em uma arquitetura monolítica quanto de microsserviços, é possível haver um gerenciamento manual da aplicação e dos *hosts* (máquinas). No entanto, à medida que o sistema é dimensionado, o número de *hosts* tende a aumentar, tornando-se difícil de manter. Microsserviços tipicamente estão espalhados por vários *hosts*, cada um executando vários serviços. Gerenciar manualmente esses serviços ou mesmo adicionar um novo serviço seria uma grande sobrecarga em termos de configuração, testes, *deployment* e outras tarefas sensíveis ao crescimento que consomem muito tempo [109]. Para Newman [16], quanto mais instâncias de serviços são adicionadas, mais indivíduos são necessárias para gerenciar o *deployment*, que muitas vezes depende de processo manual de configuração e gerenciamento. Neste sentido, é importante ter mecanismos para especificar o local do serviço e assegurar que o *estado desejável* da aplicação (do inglês *desired state management*) é mantido ao longo do tempo. No mundo dos microsserviços, *Kubernetes* [115] emergiu como principal escolha para conteinerização de serviços, permitindo o *deployment* de serviços entre múltiplos *hosts* e assegurando melhor escalabilidade e balanceamento de carga. Em nuvens públicas como AWS [116] e Azure [117] é possível encontrar diferentes opções para manipular *deployment* de microsserviço, incluindo aquelas oferecidas pelos *Kubernetes*. Contudo, uma sugestão é tentar não começar usando *Kubernetes* ou plataformas similares. *Kubernetes* não é pré-requisito para adoção de microsserviços; ele ajuda a gerenciar múltiplos processos e, portanto, isso pode não ser produtivo até que se tenha vários serviços.

Equipe de desenvolvimento. Organizar a equipe de desenvolvimento é vital para o sucesso da adoção de microsserviços [83]. Historicamente a TI foi estruturada em torno de competências, formando por exemplo equipes Java, equipes de testadores, equipes de DBAs, etc. Entretanto, criar software requer múltiplas habilidades e forte entrosamento entre equipes. Esses silos tem sido quebrados e equipes antes dedicadas agora estão se integrando para trabalhar em conjunto [16]. Balalaie et al. [1] descreve que os métodos tradicionais de desenvolvimento de software tendem a formar equipes dividindo os membros do projeto com base nas funções exercidas, o que geralmente cria equipes com *segmentação horizontal* do tipo *Dev*, *QA* e *Ops*, como mostra a Figura 2.4(a). Nesta ilustração, tem-se três atores distintos da qual *Ops* responde pelas operações envolvendo infraestrutura e automação, *QA* (do inglês *Quality Assurance*) trata da garantia de qualidade, sobretudo, realizando testes e *Dev* atua no desenvolvimento do software propriamente dito. Este arranjo não é o mais interessante para lidar com os microsserviços, pois no cenário de microsserviços cada equipe é responsável por seus próprios serviços, desde o desenvolvimento a sua operação [46, 79, 82]. Nesta nova dinâmica, a decomposição do sistema em partes menores facilita o gerenciamento dos serviços por equipes individuais [43, 118], melhora a compreensão do código-fonte, a manutenibilidade do sistema e a inclusão ou

transição de novos membros à equipe devido a curva de aprendizado menor. Alinhado a isso, *DevOps* sugere a *segmentação vertical* entre os membros do projeto formando equipes multidisciplinares, isto é, pessoas com diferentes habilidades que cooperam desde o início do projeto para criar mais valor a um serviço em particular, fazendo entregas software com maior frequência.

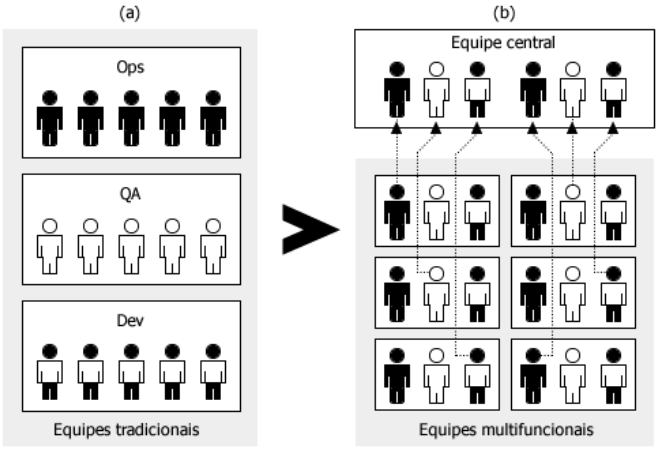


Figura 2.4: Estrutura da equipe de desenvolvimento.
(Fonte: baseado em Balalaie et al. [1])

No modelo de equipe da Figura 2.4(b) proposto por Balalaie et al. [1] são constituídas pequenas equipes multifuncionais (do inglês *cross-functional teams*), cada qual focada em um ou mais serviços. Também é formada uma equipe central (do inglês *core-team*) composta por representantes da equipe de cada serviço. Essa equipe possui uma visão geral de como os serviços interagem no sistema e é responsável por qualquer decisão crítica de arquitetura, incluindo refatorações entre serviços e regras aplicadas ao servidor de borda (API *Gateway*). Em todo caso, equipes que desenvolvem microsserviços costumam ser organizadas com base nas capacidades de negócios, não em capacidades técnicas [43, 118]. As equipes também devem ser altamente independentes e seus membros possuírem todas as habilidades necessárias para criar e manter os microsserviços. Embora os microsserviços tenham uma estrutura modular promissora para equipes maiores, desmembrar as equipes é tão relevante quanto desmembrar os módulos de software [83]. Se o propósito é tornar a equipe mais proficiente ao lidar com o ciclo de vida do software, as habilidades da equipe também precisam evoluir em termos de capacitação [16]. Em um meio arquitetural tão heterogêneo e poliglota quanto microsserviços, Newman [16] ressalta a importância de que as escolhas e decisões que afetem os serviços sejam as mais acertadas possíveis, haja vista que algumas dessas decisões são irreversíveis. Um simples mecanismo para lidar com isto é ter ao menos um líder técnico de cada equipe fazendo parte de um grupo (do inglês *cross-cutting concern*) onde os problemas possam ser discutidos, de modo que

soluções a nível global possam repercutir melhor a nível local, em um contexto específico [16]. Contudo, alguns problemas não necessariamente precisam de mais pessoas para serem resolvidos [16]. Adicionar mais pessoas irá apenas melhorar a velocidade de entregas se o trabalho puder ser dividido em partes [119], o que remete a autonomia da equipe e a consequente distribuição de responsabilidades [16].

Versionamento. Segundo Newman [16], é um grande desafio mudar o contrato de um serviço de tal modo que os consumidores não sejam afetados. Definir versões para os serviços pode ajudar neste sentido, embora torne o processo de *deployment* de cada serviço ainda mais complexo [1]. De qualquer modo, basicamente existem duas formas habituas para fazer isto. A primeira é rodar duas versões do microsserviço. Neste caso, dois serviços distintos funcionam simultaneamente, cada um expondo diferentes *endpoints*, cuja escolha fica a critério do consumidor. Essa abordagem exige (i) mais infraestrutura para rodar o serviço extra, (ii) a compatibilidade dos dados entre os serviços, e (iii) que as correções sejam feitas em todas as versões que estão operando. A segunda opção é ter uma única versão do serviço rodando, mas com suporte a ambos contratos. Neste modelo, pode ser preciso expor duas APIs por meio de diferentes portas ou redirecionamento, além de introduzir mais complexidade no microsserviço. Em todo o caso, antiga e nova versões devem coexistir até que todos os consumidores façam a mudança para a versão mais recente [33]. De acordo com Balalaie et al. [1], o versionamento de serviço pode não ser o ideal a se fazer. Técnicas como o *Tolerant Reader* [106] e *Consumer-driven contracts* [120] são mais recomendáveis para evitar o versionamento do serviço. A nível de código-fonte (e não do serviço), uma das abordagens utilizadas é o Versionamento Semântico[2] que sugere padronizar a versão/numeração do software em função da compatibilidade entre a versão antiga e atual, dada uma mudança.

Escalabilidade. À medida que o tamanho do sistema começa a crescer, problemas como a dificuldade de dimensionar (escalar) aplicações sob alta demanda começam a aparecer. Microsserviços ajudam a superar este e outros desafios [23, 47]. Em uma situação de sobrecarga de um microsserviço, é possível replicá-lo mais facilmente quantas vezes forem necessárias para alcançar escalabilidade e suprir a demanda [11, 84]. Na verdade a escalabilidade pode ser não apenas (i) *horizontal*, quando o microsserviço é replicado; mas também (ii) *vertical*, quando se aumenta a quantidade de recursos alocados a um microsserviço [121]. Sistemas escaláveis devem reagir às mudanças de cargas de trabalho automaticamente por meio da capacidade elástica que possuem e conforme oferecido pelas infraestruturas de nuvem [83]. Tratando-se de microsserviços é possível tanto escalar aumentando os recursos de apenas parte do processo que está sobrecarregado (*scale up*),

[2]https://semver.org/lang/pt-BR/

quanto também escalar reduzindo os microsserviços subutilizados (*scale down*), inclusive tirando-os do ar se não forem necessários [16].

Para pequenos sistemas, a arquitetura monolítica pode ser a solução mais apropriada e se tornar altamente disponível e escalável usando mecanismos simples de balanceamento de carga. Executar múltiplas cópias de um monolítico por trás de mecanismos como balanceador de cargas ou filas pode ser efetivo, entretanto, não adianta se o gargalo for um banco de dados que não suporte a escalabilidade [16]. Para Ahmadvand & Ibrahim [46], serviços monolíticos tradicionais não possuem uma escalabilidade eficiente quando uma determinada tarefa dentro do serviço é altamente demandada. A Figura 2.5 mostra que para escalar um monolítico, é preciso escalar toda a aplicação, pois seus módulos funcionais fazem parte de um único bloco. Por outro lado, em microsserviços as funcionalidades estão desmembradas, sendo possível escalar parcialmente a aplicação, criando ou reduzindo as instâncias do serviço conforme o alto (ou baixo) volume de requisições, o que proporciona a uma otimização dos recursos.

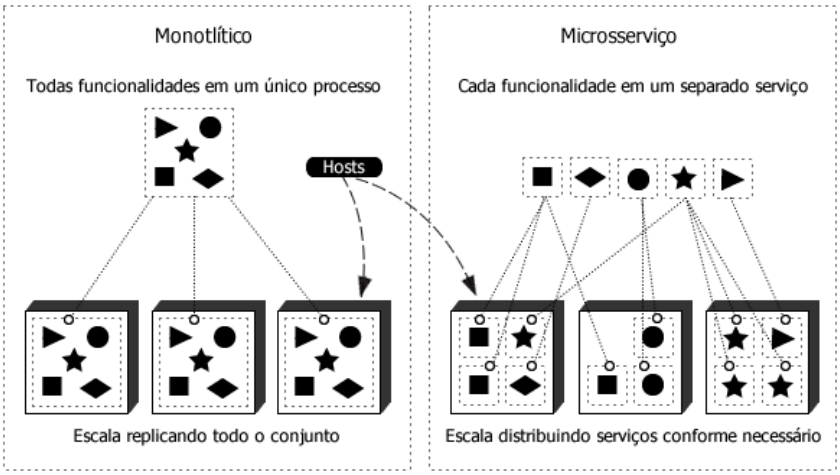


Figura 2.5: Escalabilidade de monolíticos e microsserviços.
(Fonte: baseado em Lewis e Fowler [8])

No passado, muitos sistemas legados foram migrados para a nuvem, mas mantendo a mesma arquitetura em uma nova infraestrutura [6]. Colocar uma aplicação em um *container* virtualizado e chamá-lo de aplicação em nuvem não faz com que essa aplicação obtenha os benefícios do ambiente de nuvem [23]. Uma aplicação escalável requer uma arquitetura escalável [23] a exemplo dos microsserviços, que é uma arquitetura nativa da nuvem [1]. Em nuvem, no entanto, falhas podem ocorrer a qualquer momento e aplicações neste ambiente devem ser projetadas para resistir as incertezas [23]. Segundo Mazzara et al. [109], há quem defenda que a arquitetura de microsserviços por si só aumente a escalabilidade do sistema. No entanto, para que a arquitetura alcance plenamente seu

potencial em termos de escalabilidade, é preciso tratar outros aspectos como automação, orquestração, descoberta de serviço, balanceamento de carga e *clustering* [109]. Fan & Ma [3], por sua vez, ressaltam que a questão de escalabilidade da arquitetura de microsserviço deve estar integrada a descoberta de serviço, API *Gateway* e *circuit Breaker*.

Em resumo, pode-se dizer que escalar microsserviços é mais fácil que monolíticos. Escalar sistemas monolíticos requer um grande investimento em termos de hardware e, muitas vezes, ajuste fino do código [2]. De acordo com Hasselbring & Steinacker [83], para escalabilidade de microsserviços, conteinerização tem sido a nova tendência [122]. Utilizando *containers*, por exemplo, através do Docker [123], é possível fazer o *deploy* de instâncias de serviço com custos operacionais mais baixos do que via virtualização.

Deployment. Microsserviços permitem o *deployment* independente para cada serviço [1], o que é altamente desejável pela indústria de software devido à sua elasticidade automática e a provisão de recursos *on demand* [19, 88]. Um típico exemplo é o *deployment* totalmente automatizado de *containers*, onde cada *container* executa um único microsserviço. Uma das qualidades desse tipo de *deployment* tem a ver com a redução do custo de execução [57], encurtando os ciclos de desenvolvimento e aumentando a capacidade de manutenção do microsserviço [19]. Contudo, o *deployment* pode ser demasiadamente complexo em múltiplas nuvens devido aos numerosos componentes sendo executados em ambientes heterogêneos (VM/container) e a intensa comunicação entre microsserviços [114]. Fazer o *deployment*, portanto, implica risco. Dentre as formas de reduzir o risco está mudar apenas o que precisa ser mudado, reduzindo o escopo do *deployment*, o que basicamente significa dizer que menores *releases* trazem menores riscos [16].

Coreografia e orquestração. Uma vez que os serviços tenham sido projetados e *deployed* corretamente, a questão de integrá-los precisa ser abordada [57]. Segundo Butzin et al. [33], o conceito mais importante quando se fala em manter vários serviços juntos é o da orquestração e coreografia. Na orquestração existe a figura de um controlador central que coordena os serviços a serem chamados. Na coreografia, por sua vez, cada participante faz sua parte por conta própria, isto é, os serviços interagem entre si, cada qual com suas responsabilidades, para alcançar o resultado desejado. Do ponto de vista da arquitetura de microsserviço, ambos são possíveis. Na prática, deve-se optar pela coreografia, a não ser que exista boas razões para usar a orquestração. Em uma orquestração para se introduzir um serviço adicional/novo, o controlador precisa ser alterado, mas isso só é possível se não se tratar de um produto de outro fornecedor. Utilizando a coreografia, quando determinado evento ocorre, cada serviço que escuta o evento é acionado para fazer sua parte, ou seja, criar um serviço é bem mais fácil. O novo serviço só precisa ouvir esse evento e também fazer sua parte caso o evento ocorra. O problema é que, neste

caso, não há como rastrear se a execução foi bem-sucedida a não ser que haja um serviço adicional que monitore os serviços que estão sendo executados. Atualmente, os serviços de IoT (do inglês *Internet of Things*) são frequentemente combinados usando um estilo de orquestração, porque é mais fácil de implementar e protocolos como HTTP não têm suporte nativo à comunicação baseada em eventos. Uma exceção é o protocolo MQTT (do inglês *Message Queue Telemetry Transport*), que impõe o uso da comunicação baseada em eventos [33]. De acordo com Mazzara et al. [109], na arquitetura de microsserviços, a orquestração é necessária para gerenciar *containers* e infraestrutura de serviço. Sistemas de orquestração *open-source* como *Google Kubernetes*, *Mesosphere Marathon* e o *Docker built-in Swarm Mode*, oferecem vários recursos necessários para obter escalabilidade, como descoberta de serviços, balanceamento de carga e gerenciamento de *cluster* [109].

Containers. Para Butzin et al. [33], uma importante propriedade associada aos microsserviços é o conceito de "autocontido", usado para descrever que os serviços devem conter tudo o que precisam para cumprir suas tarefas por conta própria. Isto inclui não apenas a lógica de negócios, mas também o *front-end* e o *back-end*, além das bibliotecas necessárias. Mantendo assim, os serviços podem ser escalados individualmente, tendo várias instâncias. Além disso, o baixo acoplamento permite os serviços evoluírem independentemente [33]. Existe um forte vínculo entre arquiteturas baseadas em serviços e a tecnologia de *containers* [57]. Serviços individuais podem ser hospedados como um único *container*. Esses *containers*, portanto, incluem o próprio microsserviço, suas bibliotecas e os dados necessários, em conformidade ao requisito de autocontenção [33]. A virtualização (IAAS, PAAS) geralmente é alcançada por duas tecnologias conhecidas: o *hipervisor* e o *container*. Máquinas virtuais baseadas em *hipervisor* são pesadas, com sistemas operacionais completos e com alto consumo de memória. A tecnologia de *containers*, por outro lado, ganhou enorme popularidade devido à natureza leve do *container* [59]. Por meio de *containers* é possível fazer o *deploy* de inúmeras instâncias de serviços a um custo mais baixo que a virtualização de *hypervisor* e com um melhor isolamento [1], além de prover escalabilidade e disponibilidade mais facilmente [59]. Dentre as tecnologias baseadas em *container*, *Docker* é a mais conhecida [1, 33, 57], mas existem outras como *openVZ*, *lxc/lxd* e *rkt* para Linux e "*Windows Server Containers*"[33]. Imagens do *Docker* produzem o mesmo comportamento em diferentes ambientes [1], porém, como todos os *containers* usam o mesmo *kernel* existe a limitação de que um *container* do Windows não pode ser executado em um host Linux ou vice-versa. Ainda assim, *containers* levam vantagens em relação a virtualização de *hypervisor*, provendo melhor desempenho, menor tempo de inicialização (segundos e não minutos) e espaço de armazenamento [33].

2.3.3 Decomposição em serviços

A modularização de sistemas foi primeiramente proposta em 1972 por Parnas [124] e recentemente o modo de decomposição vem assumindo outra dimensão graças aos sistemas nativos de nuvem, em especial os microsserviços [102]. Aplicações em microsserviços podem ser compostas de centenas ou até milhares de serviços [125] e cada serviço é uma parte do software que deve melhor representar o mundo real em que opera [16].

Granularidade. Segundo Cojocaru et al. [19], a granularidade dos componentes é o foco principal da arquitetura de microsserviços [19, 126]. Para Gouigoux & Tamzalit [57] a principal dificuldade na busca da granularidade correta em microsserviços é que não há estado da arte que aborde o tema. Há, entretanto, algumas formas de abordar a decomposição de sistemas e de determinar a granularidade do serviço [16]. Em geral são decomposições manuais (caixa branca) ou semi-automáticas baseadas principalmente na análise de domínio de negócio [40]. Na prática, elementos altamente acoplados devem ser agrupados, enquanto elementos não acoplados podem ser isolados e apresentados como microsserviços independentes [110]. É comum começar com um "serviço monolítico" e gradualmente refatorá-lo visando um sistema mais desacoplado, contudo, para uma decomposição ideal do sistema, é necessário um exame mais profundo das características do sistema e uma abstração mais alta [46]. A granularidade não deve ser subestimada. Diversos estudos [14, 127, 128] mostram que uma das tarefas mais difíceis em microsserviços é justamente definir o nível apropriado de granularidade ao decompor cada serviço. Definir incorretamente a granularidade pode implicar sérios problemas de acoplamento e desempenho da aplicação, a um alto custo de correção.

Refatoração. Embora alguns pesquisadores considerem que extrair microsserviços da atual base de código monolítica é mais fácil do que construir microsserviços a partir do zero, não há evidências de que as relações encontradas no monolítico irão corresponder às capacidade de negócios do serviço [32, 105]. Para Newman [16], mesmo se tendo acesso ao código-fonte do monolítico, nem sempre fica claro o que fazer. Se o monolítico estiver bem estruturado, uma possibilidade é extrair a parte do código que representa a funcionalidade a ser implementada em microsserviços sem, no entanto, remover esta funcionalidade do monolítico, pelo menos por um tempo. Um problema comum é que quase sempre o código-fonte do monolítico não está organizado em torno de domínio de negócios. Uma maneira de lidar com isto é utilizar o conceito proposto por Michael Feathers [129] em *"Working Effectively with Legacy Code"*. Basicamente se define uma "costura"(do inglês *seam*) em torno da peça de código a ser modificada, substituindo a nova implementação após a mudança estar concluída. Ao utilizar esta técnica é importante,

sempre que possível, aplicar os princípios de DDD. Em todo caso, o ideal é realizar o processo de desacoplamento do sistema monolítico iniciando pelo desenvolvimento de *features* não-críticas [2], isto é, priorize as funcionalidades que podem ser mais fáceis de extrair [16]. Newman [16] enfatiza que, decidindo por quebrar o monolítico existente, que seja feito um pouco de cada vez. Uma abordagem incremental ajuda a aprender sobre microsserviços e limita o impacto caso algo errado aconteça [16]. "Explodir" todo o sistema raramente termina bem e o custo de experimentar uma migração para microsserviços pode ser alto. Fazendo tudo de uma vez pode dificultar o *feedback* sobre o que está ou não dando certo. Uma estratégia eficaz para definir serviços é selecionar contextos (ou subdomínios) que não tenham ou tenham poucos relacionamentos. O número de *bugs* associados a um particular contexto também pode ser um fator de escolha [6]. A extração de microsserviços não é considerada pronta até que o código da aplicação esteja rodando em seu próprio serviço e os dados estejam logicamente isolados em seu próprio banco de dados [16].

Segundo Mayank et al. [5], vários fatores precisam ser levados em conta para identificar partes funcionais de um sistema a serem implementadas como microsserviços: (i) número de funções de negócios coesas presentes na aplicação; (ii) tamanho das equipes disponíveis para desenvolver e manter os microsserviços; (iii) interface de mensagem entre os componentes; (iv) acesso a dados evitando a dependência de vários microsserviços na mesma tabela de banco ou repositório de dados; e (v) componentes diferentes em termos de escalabilidade devem fazer parte de diferentes microsserviços. À medida que o processo de migração para microsserviços evolui, pesquisadores começam a abordar o problema de decomposição da arquitetura de uma maneira mais sistemática [19]. Richardson [25], por exemplo, propôs quatro estratégias de decomposição: (i) *"Decomposição por capacidade de negócios"* [130]: define serviços segundo a capacidade de negócios, isto é, algo que uma empresa faz para gerar valor; (ii) *"Decomposição por subdomínio"* [131]: define os serviços conforme os subdomínios DDD (*Domain-Driven Design*); cada subdomínio corresponde a uma parte diferente do negócio; (iii) *"Serviço autônomo"* [132]: define um serviço de modo que responda a uma solicitação síncrona sem aguardar a resposta de qualquer outro serviço; e (iv) *"Serviço por equipe"* [133]: define o serviço com base na responsabilidade que uma equipe tem para com determinado serviço. Muitos desses padrões são considerados subjetivos e precisam da intervenção do arquiteto para a tomada de decisão [19].

Outra proposta para obter uma granularidade apropriada é a decomposição vertical de sistemas autônomos (do inglês *Self-contained System* - SCS) [134]. Trata-se de uma abordagem que adota certas regras para dividir um sistema grande em vários sistemas independentes menores, evitando problemas comuns a grandes monolíticos, como o desordenado crescimento ao longo do tempo, onde eventualmente os sistemas se tornam insustentáveis do ponto de vista da manutenção [83]. Existem ainda várias outras pro-

postas de decomposição. Abdullah et al. [40] propõem um novo método baseado em uma abordagem "*black-box*" que usa os *logs* de acesso e um método de aprendizado de máquina (do inglês *machine-learning*) não supervisionado para decompor automaticamente aplicações monolíticas em microsserviços. Kazanavicius & Mazeika [43] faz um revisão de diferentes métodos e técnicas de migração, sob o aspecto da decomposição, analisando seus benefícios e desvantagens. Taibi & Systä [102] propõe um *framework* em seis etapas para reduzir a subjetividade do processo de decomposição, validando o método em um estudo de caso. Escolher métodos e técnicas mais adequados para uma organização é uma tarefa muito difícil [43] e a melhor escolha vai depender muito do contexto individual.

Para Jin et al. [105], a principal tarefa da extração de microsserviços é descobrir em um monolítico existente quais entidades de software (por exemplo, métodos, classes) devem ser agrupadas para formarem os microsserviços candidatos. Os métodos atuais extraem microsserviços analisando o código-fonte e seguindo a suposição de que "classes com fortes relações devem estar no mesmo serviço". O problema desta abordagem é que (i) muitos comportamentos da aplicação não podem ser explicitamente refletidos no código-fonte; e (ii) a relação a nível de código não é equivalente a mesma funcionalidade do serviço [105]. A divisão de microsserviços usando apenas as características estruturais estáticas da aplicação pode facilmente levar a partições inadequadas de microsserviços [11]. *Abdullah et al* enfatizam que mesmo as análises baseadas em domínio de negócio não consideram o impacto no desempenho da aplicação, pois não conseguem identificar as partes da aplicação que atraem altas demandas e correm o risco de se tornarem gargalos [40]. A decomposição de sistemas, na realidade, costuma ser delegada à experiência de arquitetos de software [81] que normalmente recebem ajuda apenas baseada na análise estática de dependências, utilizando ferramentas como a *Structure 101* [135]. Essas ferramentas, entretanto, não são capazes de capturar o comportamento dinâmico do sistema e as dependências em tempo de execução [102].

Desafios da decomposição. A modernização de sistemas monolíticos em microsserviços se torna difícil porque é preciso decidir como estabelecer as dependências entre serviços preservando a funcionalidade original [110]. Há muitas questões a serem abordadas em microsserviços, dentre as quais a mais desafiadora é a decomposição. O processo de decomposição geralmente é implementado manualmente e não existe uma metodologia de avaliação sólida [12], fazendo por vezes a própria definição de serviços confusa [81]. Diversos autores [12, 102] relatam a decomposição de microsserviços como uma tarefa extremamente complexa. Isto porque, se a divisão do sistema em serviços for inadequada, os serviços terão fortes dependências e os benefícios dos microsserviços serão perdidos [34]. Dividir um sistema em serviços muito pequenos gera intensa comunicação e consequente sobrecarga de rede (*overhead*) [46, 127, 136]. Por outro lado, se a divisão resultar em

serviços muito grandes, além de ser mais difícil a manutenibilidade, limita-se também a escalabilidade, isto é, perde-se a capacidade de apenas replicar parte da aplicação que está sob alta demanda. Na prática, identificar componentes da aplicação monolítica que possam ser transformados em serviços independentes e coesos pode ser um esforço manual e tedioso [84], bem como frequentemente um difícil processo de tentativa e erro [57]. Segundo Newman [16], a decomposição em microsserviços pode causar alguns desagradáveis problemas de performance. Espalhar as funcionalidades através de múltiplos e separados processos e *hosts* não garante robustez, pelo contrário, pode aumentar a superfície de falhas. Por isso, caso a granularidade do sistema seja o foco principal e não seja necessário manter o estado na aplicação, considere o uso de *Serverless* [126] ao vez de microsserviços [19]. Quanto mais o número de serviços aumenta, mais chamadas entre serviços acontecem, tornando a aplicação mais vulnerável a problemas de resiliência. Neste caso, é preciso inicialmente lidar com duas questões: qual chamada pode falhar? E se falhar, o que se deve fazer? A partir das respostas, o desafio é trabalhar uma possível solução, seja ela comunicação assíncrona, *circuit breakers* ou múltiplas cópias de serviços [16].

DDD. Apesar dos desafios da decomposição, o uso de *Domain-driven design* - DDD pode ser considerado a alternativa mais próxima ao estado da arte neste quesito [1]. Esta abordagem introduzida por Eric Evans [94] oferece um conjunto de conceitos e técnicas que suportam a modularização de sistemas [6], contribuindo para definir as fronteiras dos serviços, desenvolver um modelo de domínio e priorizá-los [16, 19]. Dentre os elementos do DDD está o *contexto delimitado*. Cada contexto delimitado representa uma potencial unidade de decomposição, e um ótimo ponto de começo para definir as fronteiras de microsserviços [16]. Os contextos delimitados representam domínios de negócios autônomos e são usados para organizar e identificar microsserviços [6, 74]. Na prática, as funcionalidades correlacionadas são combinadas em uma única capacidade de negócios que é então implementado como um serviço [109].

2.3.4 Reestruturação da base de dados

Reestruturar a base de dados de um sistema de software legado requer um planejamento cuidadoso [2, 6]. Idealmente microsserviços devem ter seus próprios e independentes banco de dados [6, 16] de modo que outros microsserviços só podem acessar esses dados pelas interfaces fornecidas (API) [11]. Ocultar um banco de dados por trás de uma bem-definida interface permite o serviço limitar o escopo do que é exposto [16], além de melhorar o desempenho e a escalabilidade [51]. Contudo, isto nem sempre é viável. Participantes em um estudo conduzido por Taibi et al. [2] relataram que majoritariamente adotaram a arquitetura de microsserviço mantendo a conexão com a base de dados legada, mesmo

admitindo que isso reduziria parcialmente os benefícios dos microsserviços. Um outro estudo proposto por S. da Silva et al. [6], a decisão foi migrar o banco de dados em etapas incrementais devido a sua complexidade. E de fato, a decomposição de banco de dados pode ser algo inicialmente adiado, mas não para sempre [16].

Compartilhamento de dados. Manter todos os dados em uma única base é contrário à ideia de descentralização inerente aos microsserviços [6]. Segundo Newman [16], microsserviços funcionam melhor quando encapsulam seus próprios dados. Para ele, se possível, novos serviços devem ter seus próprios e independentes *schemas*[3]. Compartilhar um único banco de dados entre múltiplos serviços pode levar a diversos problemas, a começar pela perda de gerência sobre o que deve ser compartilhado e o que deve ser ocultado, sendo difícil saber quais partes do *schema* podem ser modificadas com segurança. Além disto, torna-se confuso identificar quem controla os dados, isto é, qual lógica de negócio que manipula determinado dado [16]. Portanto, via de regra a recomendação é para que haja a divisão dos dados existentes e cada microsserviço acesse seu próprio banco de dados privado [2]. Não se deve compartilhar dados, a menos que seja realmente preciso. Ocultar o banco de dados atrás do serviço garante a redução do acoplamento [16].

Transações. De acordo com Hasselbring & Steinacker [83], em sistemas monolíticos é comum usar transações para garantir a consistência dos dados ao atualizar múltiplos registros, ainda que a custo de um alto acoplamento. Em microsserviços é preciso descentralizar a responsabilidade dos dados, contudo, as transações distribuídas são notoriamente difíceis de implementar e, como consequência, atualizações entre serviços são coordenadas sem transações, aceitando-se que a consistência pode ser apenas uma consistência eventual e que os problemas devem ser resolvidos com operações de compensação [83]. Para Newman [16], decompor e descentralizar um banco de dados envolve desafios como lidar com integridade referencial, aumento de latência e maior complexidade ao gerar relatórios. Além disto, perde-se a garantia das propriedades *ACID*[4], sobretudo, em relação a atomicidade. A atomicidade assegura que todas as operações realizadas dentro de uma transação ou são executadas por completo ou todas serão desfeitas. Para lidar com transações distribuídas e tentar minimizar eventuais inconsistências, alguns algoritmos têm sido propostos como *Two-Phase Commits* (2PC), *Saga* [137] e *GRIT* [138]. Contudo, estes algoritmos tendem a ser limitados, podendo aumentar a complexidade e introduzir *delays*. O ideal seria evitar transações distribuídas, mas havendo a necessidade de manter os dados atômicos e consistentes, a melhor opção é tentar não dividir os dados, mantendo-os em um único banco de dados sendo gerenciados por um único serviço. Em todo o caso,

[3]Tecnicamente um *schema* é um conjunto de tabelas logicamente separadas que contém dados [16]
[4]ACID é um acrônimo para Atomicidade, Consistência, Isolamento e Durabilidade

ainda que transações distribuídas sejam inevitáveis, recursos como compensações (SAGA) podem ser úteis em reverter operações efetivadas no banco de dados [16].

Relatórios. Newman [16] ressalta que em termos de relatórios, microsserviços podem requerer especial atenção. Sistemas monolíticos tipicamente possuem banco de dados monolíticos. Esses dados sendo quebrados em diferentes *schemas* para estar em conformidade com os microsserviços podem dificultar que *stakeholders* consultem diretamente no banco de dados as informações por meio de SQL (utilizando *joins*) e gerem relatórios personalizados. Se a ideia é causar o mínimo de impacto na forma com que os usuários trabalham, uma saída seria criar uma réplica somente-leitura do banco de dados monolítico para uso de ferramentas de relatórios, o que talvez nem seja necessário se a organização já utiliza fontes dedicadas de dados para relatórios, como *data warehouse* ou *data lake*. Para isto, é preciso prover uma forma dos microsserviços alimentarem esta base de dados de relatórios com dados requeridos pelos usuários que fazem as consultas. Outra técnica para resolver isto é projetar um *schema* para relatórios a partir de *views* exportadas por múltiplos bancos de dados dos microsserviços [16].

2.3.5 Componentes e patterns

Não existe uma representação precisa do estilo arquitetural de microsserviços [74], porém, diversos pesquisadores [3, 15, 16, 47, 48, 49, 136] vêm empreendendo esforços para identificar componentes e *patterns* que possam melhor representá-los. Para ter uma arquitetura de microsserviços completamente funcional e obter seus benefícios, alguns componentes precisam ser necessariamente utilizados [23]. De igual modo, é preciso avaliar os prós e contras de cada *pattern* e decidir se são aplicáveis ou não ao contexto [16]. Vários componentes e *patterns* têm sido catalogados para lidar com desafios das diferentes áreas relacionadas à arquitetura de microsserviços, alguns brevemente discutidos nesta Seção.

***Strangler Application* ou *Strangler Fig*.** Técnica frequentemente utilizada quando se está fazendo a reescrita de sistemas. Fowler [44] recentemente renomeou este *pattern* para *StranglerFigApplication* e sua inspiração veio de certos tipos de árvores cujas sementes em galhos mais altos descem em direção ao chão para criar raízes que vão gradualmente evolvendo o tronco da árvore original até que ela morra, permanecendo apenas a nova estrutura. No Brasil, há dezenas de espécies de *Strangler Figs*, fenômeno que pode ser observado, por exemplo, nas gameleiras. Em termos de sistemas, a ideia é que o velho e o novo coexistam, dando tempo para que o novo sistema cresça e ocupe o lugar do legado, conforme ilustra a Figura 2.6. Segundo Newman [16], este *pattern* permite uma migração incremental do novo sistema, sendo possível inclusive pausar e até parar a migração,

além de assegurar que cada pequeno passo seja reversível. Em termos práticos isto pode ser feito copiando o código do monolítico ou reimplementando a funcionalidade no novo microsserviço, considerando ainda a persistência de dados, se houver. Basicamente três etapas devem ser seguidas: (i) identificar as partes do atual sistema as quais se deseja migrar; (ii) implementar as funcionalidades na arquitetura microsserviços; e (iii) refazer as chamadas do monolítico para os novos microsserviços.

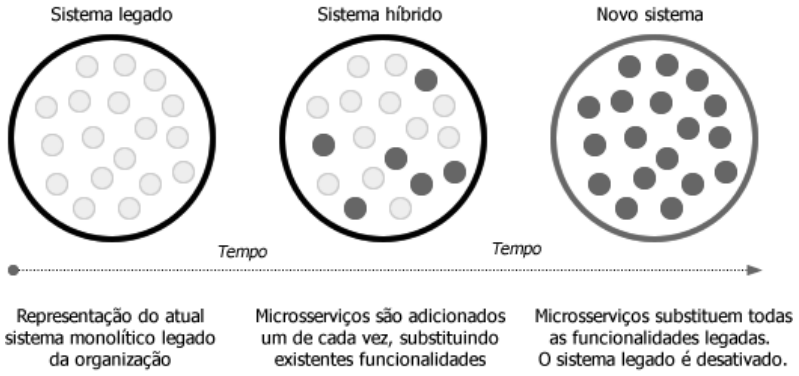


Figura 2.6: Strangler Application pattern.
(Fonte: baseado em CM [139])

Um estudo proposto por Yoder & Merson [45] aborda possíveis caminhos para lidar com *Strangler Application*. Segundo eles, "*Strangler*" é um processo evolutivo em que a migração deve começar com pequenas transformações (do inglês, *start small*) e progredir passo a passo (do inglês *baby steps*). A sequência para "estrangular" o monolítico pode envolver (i) preparação, (ii) escrever o primeiro microsserviço, e (iii) extrair a funcionalidade para microsserviços. Alguns princípios desse processo, são:

- *Wrapping the Monolith*: permite acessar o novo sistema de modo antigo;
- *Start Small*: comece com uma equipe pequena e evolua aos poucos e devagar;
- *Pave the Road*: prepare a infraestrutura e o ambiente técnico/operacional;
- *Microservices First*: escreva qualquer novo código como microsserviço;
- *Replace as Microservice*: reimplemente componentes do monólito como serviços;
- *Macro then Micro*: extraia partes maiores, só depois quebre em menores;
- *Extract Component and Add Façade*: disponibilize serviço sem afetar clientes;
- *Proxy Monolith Components to Microservices*: redirecione chamadas para novos serviços.

Decomposição pelo subdomínio. Define serviços correspondendo a subdomínios do DDD. Em DDD, domínio é o problema/negócio da aplicação e consiste em múltiplos subdomínios, sendo que cada subdomínio corresponde a diferentes partes do negócio (Figura 2.7).

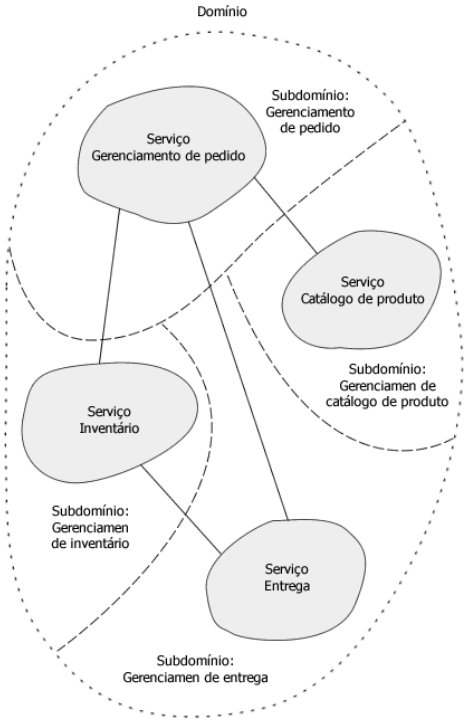


Figura 2.7: Exemplo de decomposição pelo subdomínio.
(Fonte: baseado em Richardson [131])

API *Gateway*. Também conhecido como "servidor de borda", pode ter várias aplicabilidades, como por exemplo, expor APIs ao público [23]. Para clientes acessarem um serviço, a *API Gateway* funciona como um ponto único de entrada para todas requisições, conforme mostra a Figura 2.8. A API *Gateway* pode centralizar atribuições de autenticação, autorização, transformações, segurança, *log*, entre outros. Newman [16] acrescenta que na condição de *proxies*, também pode ser utilizada para transformar protocolos. Por exemplo, expor uma interface SOAP/HTTP, sendo que o microsserviço irá suportar gRPC. O *proxy* então se encarregará de converter as requisições. Quando se trata da arquitetura de microsserviços, o mantra é *"Keep the pipes dumb, the endpoints smart"*, em uma tradução livre seria algo como "mantenha os *pipes* burros e os *endpoints* inteligentes". Uma sugestão para fazer a conversão de protocolo seria colocar o mapeamento dentro do próprio serviço, de modo que o serviço suporte ambos protocolos. Microsserviços devem ser vistos como uma coleção de funcionalidades que suportam diferentes consumidores, formatos e requisições. Também é comum que programadores tentem escrever seus próprios *proxies*,

porém, isto pode ser ineficiente, adicionando significativa lentidão ao sistema. Uma possibilidade para este caso seria utilizar um *proxy* dedicado [16], como Nginx, HAProxy ou Kong.

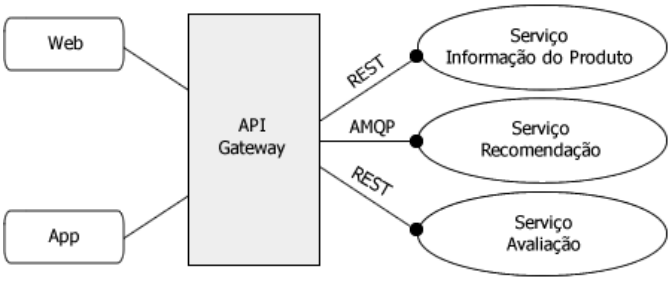


Figura 2.8: Exemplo de API Gateway.
(Fonte: baseado em Richardson [140])

Instância do serviço por *container*. Empacota o serviço como uma imagem de *container* (Docker) e faz o *deploy* de cada instância do serviço como um *container* [141]. Os benefícios desta abordagem, são: (i) é mais fácil escalar um serviço, ou seja, aumentar ou diminuir o número de instâncias do *container* conforme demanda; (ii) o *container* encapsula detalhes da tecnologia usada na construção do serviço de modo que todos os serviços são, por exemplo, iniciados e parados exatamente da mesma maneira; (iii) cada serviço é uma instância isolada para qual é possível impor limites à CPU e à memória consumida por instância de serviço; e (iv) os *containers* são extremamente rápidos em operações de *build* e inicialização.

Descoberta de serviço *server-side*. No *deployment* de um tradicional sistema distribuído, os serviços são executados em locais fixos e conhecidos (*host* e porta) e, portanto, podem facilmente chamar uns aos outros. No entanto, isto pode ser complicado em aplicações baseadas em microsserviços, geralmente rodando em *containers*, onde o número de instâncias do serviço e suas localizações mudam dinamicamente. Assim, para o cliente de um serviço descobrir a localização de uma instância do serviço, é preciso primeiro enviar a requisição a um *router* que está em uma já conhecida localização. O *router*, então, consulta um *serviço de registro* (do inglês *service registry*) e reencaminha a requisição para uma instância de serviço disponível, como mostra a Figura 2.9 [142]. Para Montesi & Weber [143], a descoberta de serviços é um componente essencial de qualquer microsserviço, pois nenhuma localização lhe é atribuída no estágio de *design*. Segundo Mazzara et al. [109] a descoberta de serviços fornece mais do que pesquisas simples de DNS (do inglês *Domain Name System*), também inclui mecanismos de verificação de integridade que garantem que os serviços para os quais resolve nomes estejam realmente ativos e dis-

poníveis. Em outras palavras, a descoberta de serviço é essencial para a arquitetura de microsserviços, pois os serviços não têm endereços IP estáticos e precisam ser mapeados a partir de um nome de *host*. A descoberta de serviço também pode fazer uso da localidade do solicitante, resolvendo nomes de *host* de uma instância de serviço que esteja mais próxima dele, alcançando a escalabilidade geográfica. A descoberta de serviço cria a ilusão de estar interagindo com um único serviço, contudo, uma sequência de requisições pode na realidade estar sendo tratada por várias réplicas de serviço [109].

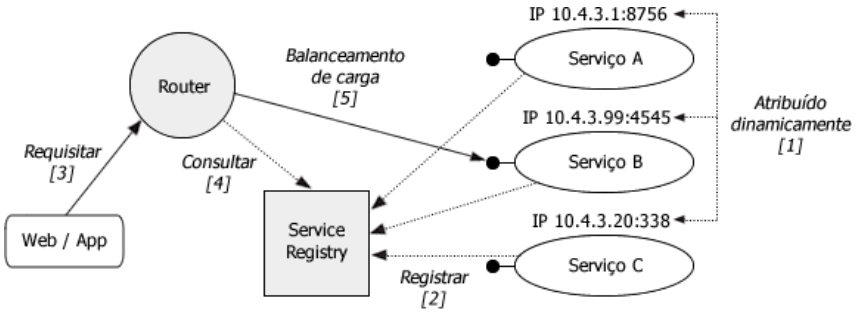


Figura 2.9: Exemplo de descoberta de serviço server-side.
(Fonte: baseado em Richardson [142])

Token de acesso. Uma forma de comunicar a identidade do requisitante para um serviço é por meio de *token*. A API Gateway pode autenticar a requisição ao serviço e passar o *token* de acesso a cada chamada ao serviço. Um serviço pode também incluir o *token* de acesso às chamadas que faz a outros serviços. Desta forma os serviços podem seguramente verificar se o solicitante está autorizado a realizar uma operação [144]. Um padrão aberto (RFC 7519) frequentemente utilizado para esta finalidade é o JWT [145] (JSON Web Token). Com JWT informações (objetos JSON) podem ser transmitidas com segurança, pois são assinadas usando um segredo com o algoritmo HMAC ou um par de chaves pública/privada usando RSA/ECDSA.

Servidor de configuração. Esse é um dos princípios da Integração Contínua (CI) para desacoplar o código-fonte de sua configuração, pois permite alterar a configuração da aplicação sem *redeployment* do código. Como uma arquitetura de microsserviços tem muitos serviços, um *redeployment* custaria caro. Neste caso, o melhor é ter um servidor de configuração para que os serviços possam buscar nele as respectivas configurações [23].

Balanceador de carga. Para ser escalável uma aplicação deve ser capaz de distribuir a carga direcionada a um serviço específico entre suas várias instâncias. Esse é o dever de um balanceador de carga que, em geral, verifica as instâncias disponíveis consultando o

componente de descoberta de serviço. O *balanceador de carga* (do inglês *Load balancing*) é tipicamente implementado como parte do serviço de descoberta e orquestração. Caso os serviços se integrem por meio de um sistema de mensageria, a distribuição de mensagens e eventos para diferentes réplicas pode igualmente ajudar a balancear a carga [109].

Circuit breaker. A tolerância a falhas deve ser uma característica inerente às aplicações nativas de nuvem, em especial na arquitetura de microsserviços onde grande parte dos serviços trabalham juntos. A falha em um desses serviços que tenham alguma dependência pode resultar em falha encadeada. Padrões como o *Circuit Breaker* ajudam a mitigar essas perdas a um nível mais baixo [23]. *Circuit breaker* surgiu da necessidade de lidar com o estado de saúde do serviço e também com falhas inesperadas [33, 146]. Basicamente ele funciona como um "disjuntor" que "liga" ou "desliga" o acesso a um serviço dependendo do seu status de funcionamento, isto é, se o número de chamadas malsucedidas atingirem determinado limite. No caso de um serviço remoto quebrado, quando o disjuntor é acionado, retorna um erro ao invés de encaminhar a chamada. Após um certo período de tempo, o disjuntor tenta acessar o serviço novamente para verificar se o serviço foi restabelecido e entra em um estado semiaberto se o teste for bem-sucedido. Serviços para os quais o disjuntor está semiaberto, o número de solicitações roteadas é reduzido, devido ao alto tráfego de entrada. Serviços quebrados, momentaneamente, não são utilizados. O *Circuit breaker* funciona perfeitamente em conjunto com o padrão "*Load Balancer*". Nesse caso, o disjuntor permite que o balanceador de carga trabalhe apenas em serviços que estão em bom estado de funcionamento [33].

Base de dados por contexto delimitado. Mesmo antes de implementar o código da aplicação como microsserviço é possível trabalhar a decomposição do banco de dados conforme os contextos delimitados identificados. Neste caso, cada contexto teria seu próprio e totalmente exclusivo banco de dados separado, o que posteriormente poderia facilitar uma futura implementação de microsserviços. Este *pattern* pode ser interessante tanto para sistemas novos quanto para monolíticos, ainda que nunca sejam quebrados em microsserviços, pois mantém uma clara separação dos *schemas*, invoca alguns benefícios do desacoplamento, reduz a complexidade e tende a ser útil principalmente em projetos onde existem muitas pessoas envolvidas [16]. A Figura 2.10 ilustra melhor este *pattern*.

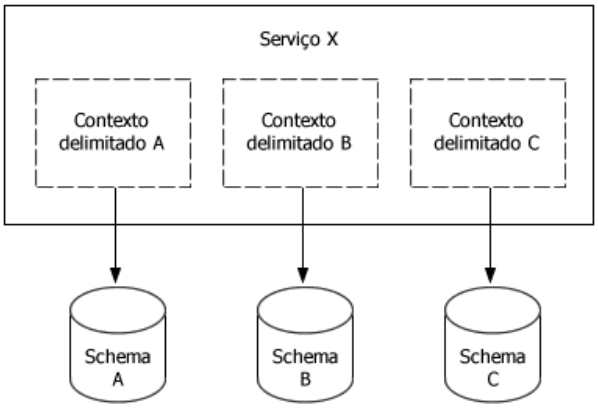


Figura 2.10: Exemplo de base de dados por contexto delimitado.
(Fonte: baseado em Newman [16])

Base de dados por serviço. Mantém privada a persistência de dados para cada microsserviço de modo que o serviço seja acessível somente pela sua API e as transações envolvam apenas seu banco de dados (Figura 2.11). Não é necessário prover um servidor de banco de dados exclusivo para cada serviço. Neste caso, o que pode ser feito é definir tabelas ou *schema* próprios para cada serviço. A fim de forçar esta modularidade desejável e evitar a tentativa do desenvolvedor burlar a API do serviço, uma boa prática é atribuir um usuário de banco de dados para cada serviço, dando as devidas permissões de acesso.

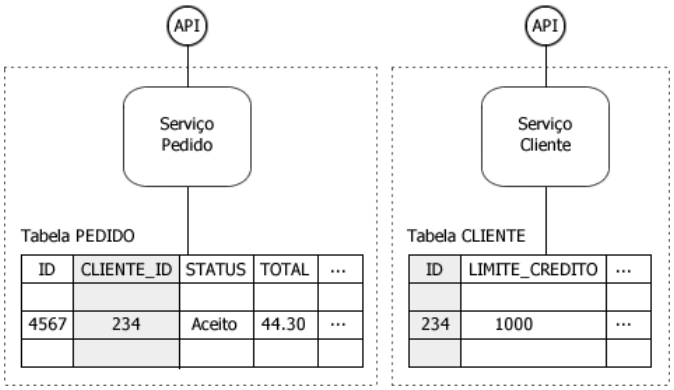


Figura 2.11: Exemplo de base de dados por serviço.
(Fonte: baseado em Richardson [147])

Base de dados *wrapping service*. Assim como nas *views*, este *pattern* (do inglês *database wrapping service*) permite controlar o que é compartilhado e o que é oculto. Em situações onde é difícil lidar com a separação do banco de dados, às vezes esconder a desordem pode fazer sentido. O *pattern Wrapping Service* oculta o banco de dados por trás de um serviço. Isso tira a dependência de um banco de dados central e coloca a

dependência em um serviço, encorajando os desenvolvedores a armazenarem seus dados localmente (junto ao serviço). Adotar este *pattern* requer que os consumidores dos dados façam mudanças em sua aplicações trocando o acesso direto ao banco de dados por chamadas API (Figura 2.12) [16].

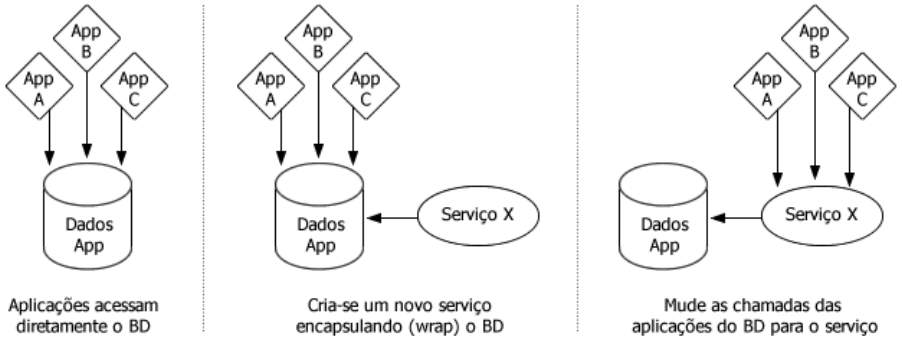


Figura 2.12: Exemplo de base de dados *Wrapping Service*.
(Fonte: baseado em Newman [16])

Change data ownership. Este *pattern* pode ser útil quando é preciso acessar dados que estão em posse de outra funcionalidade. Desmembrar certos dados de um banco de dados controlado por um monolítico pode ser complexo, principalmente devido ao impacto em chaves estrangeiras. Então, ao invés do novo serviço acessar os dados do monolítico, migra-se os dados para dentro do serviço. Neste caso, o monolítico passa a fazer chamadas diretamente ao serviço. Deste modo, os dados relativos a um serviço devem, sempre que possível, estar sob controle deste serviço e fazer isto não é um processo simples [16].

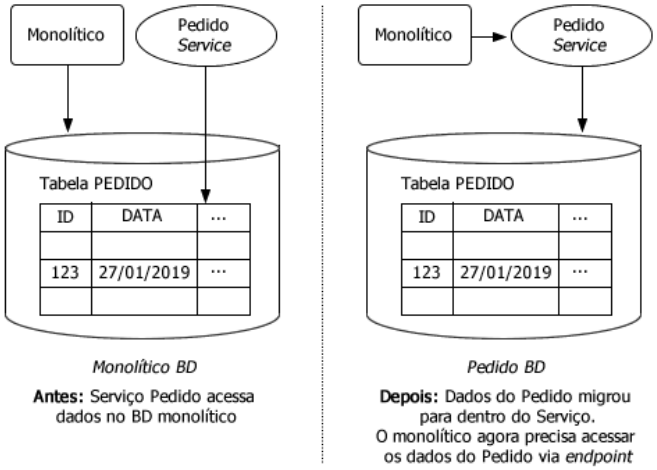


Figura 2.13: Exemplo de *Change data ownership pattern*.
(Fonte: baseado em Newman [16])

2.3.6 Migração para microsserviços

Um frequente questionamento das organizações é sobre como prolongar o ciclo de vida de um sistema legado. Uma possível resposta pode estar em migrá-lo para a arquitetura de microsserviço [11]. E de fato, diversas corporações têm recentemente migrado ou considerado migrar seus sistemas para microsserviços [28, 148], sobretudo, em razão de seus benefícios [1, 14, 15, 16]. Entretanto, microsserviços são um estilo arquitetural relativamente novo e ainda não há um modo amplamente aceito de fazer a migração. Isto tem levado os desenvolvedores a criarem e experimentarem diferentes *patterns* e técnicas de migração [5, 43].

***Gap* metodológico.** Apesar da popularidade em alta, muitas corporações ainda hesitam em migrar seu legado para microsserviços porque não conseguem avaliar claramente os prós e contras [2]. Outras, preferem evitá-los, pois consideram microsserviços um tipo de *marketing* ou simplesmente porque desconhecem o processo de migração, seus benefícios e problemas [2, 41]. Este quadro se agrava ainda mais pelo fato de não existir um método ou técnica adequada que ajude a migrar por completo uma aplicação para microsserviços [28], talvez porque cada sistema monolítico legado é único e a migração para microsserviços tende a criar diferentes desafios [43]. Ao que parece, a literatura ainda é incipiente em relação às diretrizes passo a passo para apoiar os profissionais a realizarem a migração de uma estrutura monolítica existente para uma arquitetura baseada em microsserviços [6].

Decisão racional. É preciso ponderar se realmente microsserviços são de fato uma solução adequada à organização. Newman [16] ressalta que adotar microsserviços não é uma meta, mas uma decisão racional. Não se deve pensar em migrar para microsserviços a fim de alcançar algo que é alcançável com sua atual arquitetura de sistemas. Segundo ele, é um risco assumir que "se microsserviços são bons para *Netflix*, eles serão bons para nós". Deve-se agir com cautela e buscar um claro entendimento do que se espera alcançar, até porque microsserviço requer significativo investimento [16]. A decisão de realizar uma migração precisa levar em conta o esforço extra que será necessário para lidar com problemas como *deployment* automatizado, monitoramento, falha, consistência eventual e outros fatores inerentes à arquitetura de microsserviço [6]. Por outro lado, a arquitetura de microsserviços pode ser perfeitamente apropriada a quem busca uma alternativa para superar as limitações e os problemas encontrados em estilos arquiteturais monolíticos [28]. Muitas organizações com sistemas legados, grandes e complexos encontraram nos microsserviços uma oportunidade de resgatar a alta manutenibilidade e escalabilidade de suas aplicações [33], inclusive criando soluções em menos tempo do que em tradicionais arquiteturas [6, 85].

Qualificação requerida. Uma das primeiras coisas a se considerar em uma migração é se a organização ou a equipe está apta ou não para mudar o atual monolítico [16]. Balalaie et al. [1] alerta que o desenvolvimento de sistemas distribuídos, como microsserviços, precisa de desenvolvedores qualificados. É preciso lidar com uma série de elementos e conceitos inerentes à arquitetura, como servidor de borda, descoberta de serviço e balanceador de carga que tomam muito tempo dos desenvolvedores, sobretudo, os principiantes que mesmo quando estão mais familiarizados com a arquitetura, ainda assim, em diversas situações acabam cometendo falhas [1].

Processo de migração. Apesar dos dilemas sobre a arquitetura de microsserviços, muitos pesquisadores estão imbuídos em mudar o *status quo* em prol da migração. Uma das mais importantes recomendações para guiar o processo de migração é: divida a grande jornada em vários passos menores. Aprenda com cada passo e, se algo der errado, retroceda, afinal, foi apenas um pequeno passo [16]. Em muitos casos, uma reescrita completa do sistema é inviável. O sensato é migrar o sistema legado gradualmente, substituindo partes específicas dele por novos módulos, a exemplo do que sugere o *Strangler Application pattern* [2]. Esta abordagem ajuda a minimizar os riscos durante a migração e distribui o esforço de desenvolvimento ao longo do tempo [6]. Portanto, a migração para microsserviços não deve ser feita rapidamente, principalmente em razão do alto custo associado à decomposição do monolítico e a constante necessidade de iterações [8, 43, 149]. Também não deve ocorrer de maneira *ad hoc* ou improvisada. Existem algumas invariantes que devem ser satisfeitas durante a transição de monolíticos para microsserviços [150], a fim de não se perder elementos indispensáveis ao processo.

É importante destacar que em grande parte das publicações sobre microsserviços, o processo de migração é quase que exclusivamente associado à etapa da decomposição em serviços. Por ser uma atividade crítica, muitas técnicas automatizadas de decomposição têm sido propostas [151] visando otimizar a granularidade do serviço sob aspecto da coesão e do acoplamento do software. Em geral, são técnicas de análise estática que se baseiam no código-fonte [152] e técnicas de análise dinâmica, como aquelas que se concentram em *logs* do sistema [153]. Alguns métodos também permitem automatizar a identificação de *serviços candidatos*. Um serviço candidato é um produto intermediário sujeito a refinamentos que é desenhado durante o processo de decomposição, tendo o potencial para ser implementado como um serviço (do inglês *Service Candidate Identification*) [154]. Isso basicamente é feito dividindo entidades de um sistema monolítico como métodos e classes em vários grupos de funções, cada qual vindo a ser um potencial candidato a serviço.

Métricas. Segundo Newman [16], é comum que erros sejam cometidos durante a migração. Por isso é importante definir algumas métricas que possam ajudar a rastrear possíveis problemas. Não se trata apenas de métricas quantitativas, mas também de métricas qualitativas a partir do *feedback* das pessoas envolvidas no processo. Basicamente é preciso definir pontos de checagem (*check point*) que irão permitir a equipe refletir se está indo no caminho certo. Essas métricas dependem dos objetivos traçados que se espera alcançar. Por exemplo, para melhorar o tempo de entregas (*time-to-market*) poderia se mensurar o tempo do ciclo de desenvolvimento, número de *deployment* e taxas de falhas. É preciso estar atento para que não ocorram manipulações de métricas, seja intencionalmente ou não. Ao primeiro problema avistado, não é preciso parar ou mudar todo o curso, contudo, ignorar as evidências que as métricas apontam pode ser um grande erro. O ideal é fazer pequenas mudanças, dando um passo de cada vez. Isto torna mais fácil evitar as armadilhas e reverter eventuais impactos negativos [16]. É importante destacar que, apesar da grande quantidade de métricas de software, a maioria delas não possui um *range* de valores esperados (ou valores de referências), além do que, estes indicadores podem variar conforme o tamanho, domínio ou tipo de aplicação [155].

Uma abordagem conhecida como *Goal Question Metric* - GQM pode ser útil na identificação de métricas. No GQM cada meta estabelecida gera uma série de questionamentos que devem ser respondidos por um conjunto de métricas. Este modelo de medição "*top-down*" é, portanto, baseado em três níveis [156, 157]: (i) *conceitual* (GOAL), que define uma meta sobre um produto, processo ou recurso a qual se deseja mensurar; (ii) *operacional* (QUESTION), que corresponde a um conjunto de questões usadas para avaliar/alcançar uma meta específica; e (iii) *quantitativo* (METRIC), que representam as possíveis métricas que respondem aos questionamentos de maneira mensurável. O modelo GQM pode ser visto como uma estrutura hierárquica, conforme ilustra a Figura 2.14.

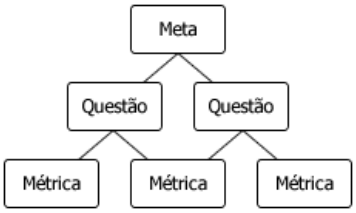


Figura 2.14: Modelo *Goal Question Metric* - GQM.
(Fonte: baseado em Basili et al. [156])

Capítulo 3

Revisão do Estado da Arte

Esse capítulo apresenta o estado da arte sobre os métodos vigentes de migração de sistemas monolíticos para a arquitetura de microsserviços, identificando aspectos de maior relevância que possam ser utilizados como referência para guiar o presente estudo. Para esse fim (i) inicialmente é feita a seleção dos artigos, reunindo as literaturas conforme critérios definidos; (ii) a partir disto, os artigos são analisados quanto aos métodos de migração propostos; e (iii) por fim, as informações são consolidadas e comparadas em uma matriz de similaridades, apresentando uma visão concisa do estado da arte.

3.1 Critérios de seleção da literatura

A exemplo de Francesco [14], a Figura 3.1 retrata o processo de seleção da literatura realizado nas bases de dados *Web of Science* (WoS), *Scopus* e *IEEE Xplore* e os sucessivos estágios de refinamento até se obter a literatura endereçada ao estudo. Os artigos são selecionados com base em diversos critérios pré-definidos.

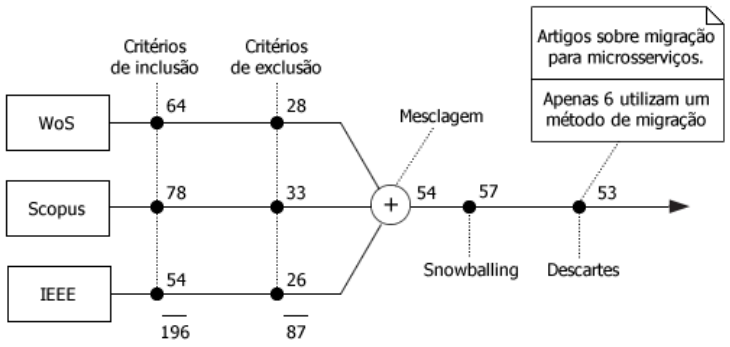


Figura 3.1: Processo de seleção da literatura.
(Fonte: própria)

Critérios de inclusão. Dentre os *critérios de inclusão*, deve-se considerar (i) publicações que tratam da migração ou decomposição de monolíticos em microsserviços. Em termos de *string* de busca isto pode ser expresso por (microservice$ AND monolithic$) AND (migrati* OR decompos* OR split* OR extract* OR transform*). Foram encontradas 196 publicações. Também são *critérios de inclusão* os (ii) estudos escritos em inglês, (iii) revisados por pares e que (iv) tenham sido publicados no período de 2015 a 2020.

Critérios de exclusão. O resultado da busca também trouxe artigos que não tinham significativas contribuições. São *critérios de exclusão* aqueles (i) estudos em que microsserviços são apenas utilizados como exemplo, (ii) estudos que não possuem o texto completo disponível e (iii) estudos secundários do tipo editoriais e revistas (*magazines*). Também foram excluídos os (iv) artigos que em seus títulos ou *abstracts* não tinham relação com a migração de microsserviços. Após aplicado o filtro de exclusão, restaram 87 artigos.

Mesclagem. Além disto, é possível que um mesmo artigo possa constar simultaneamente em diferentes bases de dados. A solução para isto foi criar um estágio para *mesclagem* (remoção de duplicatas), onde todos os artigos são colocados em um único repositório, eliminando-se as publicações redundantes. Isto reduziu o número de artigos para 54 publicações distintas.

Snowballing. Em seguida foi utilizada a técnica de *snowballing* elevando o número de publicações para 57. Segundo Wohlin [158], o *snowballing* basicamente significa usar a lista de referência para identificar novos *papers* (*backward*) ou identificar novos *papers* com base em artigos que citam o *paper* que está sendo examinado (*forward*). Neste caso, foram considerados somente os artigos relacionados à migração para microsserviços.

Descartes. Finalmente, à medida que a análise dos artigos avança, alguns descartes ainda são necessários para preservar o propósito da pesquisa. Nesta revisão, pode-se constatar que alguns artigos não oferecem a contribuição esperada, seja por terem aplicabilidade restrita a cenários muito específicos, como o uso de tecnologias e ferramentas proprietárias, ou por não abordarem com a mínima profundidade o processo de migração para microsserviços. Assim, aplicando-se este último filtro, tem-se o total de 53 artigos.

Dos 53 artigos relevantes à pesquisa, foi verificado que apenas 6 deles [1, 2, 3, 4, 5, 6] lidam com a migração para microsserviços adotando algum tipo de método ou processo. Estes 6 artigos, enfoque deste estudo, serão analisados em detalhes adiante e constam identificados no Apêndice A. Em consulta as três bases de dados supracitadas, é possível verificar que o primeiro artigo contendo a *string "microservice$ AND monolithic$"* foi publicado em 2015, embora em 2014 o termo *"microsserviço"* já tivesse sido cunhado

por James Lewis e Martin Fowler [8, 33]. Desde então tem crescido a popularidade dos microsserviços [2], tanto no meio acadêmico, quanto na indústria de software [4].

Segundo a *Web of Science*, dentre as revistas que mais publicaram sobre os termos desta pesquisa está a (i) *Journal of Systems And Software*, (ii) *IEEE International Conference on Software Architecture Workshops - ICSAW* e (iii) *Asia-Pacific Software Engineering Conference - APSEC*. Os autores mais citados são (i) *Taibi, Davide*, (ii) *Balalaie, Armin* e (iii) *Mazlami, Gene*. No que diz respeito a quantidade de publicações, o Brasil ocupa a terceira posição empatado com outros países, sendo a China e a Alemanha os primeiros no *ranking*. Os artigos mais citados foram (i) *Processes, Motivations, and Issues for Migrating to Microservices Architectures: an Empirical Investigation* [2], (ii) *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture* [1], e (iii) *Extraction of Microservices from Monolithic Software Architectures* [84].

3.2 Análise de métodos de migração

Embora os princípios de *design* em torno do estilo arquitetural de microsserviços tenham sido identificados, existem aspectos ainda não claros ou inexplorados [159], fazendo com que diferentes maneiras para produzir microsserviços venham sendo adotadas [127]. Seis dos artigos selecionados relataram algum tipo de método, processo ou *framework* para guiar a migração. O desafio desta seção é, portanto, mapear as tarefas e atividades do processo de migração na visão de cada um desses autores. Alguns ajustes de nomenclatura e de classificação foram necessários, preservando sempre a semântica original. Além disto, foram estipulados seis grupos com etapas comuns identificadas na migração para microsserviços, conforme Figura 3.2. Cada etapa dos métodos avaliados é enquadrada em um desses grupos, numerados de 1 a 6, de modo a possibilitar uma posterior comparação entre os métodos dos artigos selecionados.

Figura 3.2: Grupos: etapas comuns à migração para microsserviços .
(Fonte: própria)

3.2.1 Método de migração Balalaie et al. [1]

Neste estudo, Balalaie et al. [1] relatam suas experiências e lições aprendidas durante a migração e a refatoração de uma aplicação *mobile* para microsserviços. Os autores se concentram em mostrar como as práticas *DevOps* facilitam a migração e propõem um conjunto de *patterns* reutilizáveis em projetos de migração. A Tabela 3.1 resume em alto nível os passos idealizados no processo de migração de Balalaie et al. Segundo consta, (i) as equipes devem ser pequenas e interdisciplinares, estruturadas em torno dos serviços, isto é, cada equipe sendo responsável por seus próprios serviços; (ii) a partir deste contexto, a ideia é integrar os trabalhos dos desenvolvedores por meio de mecanismos de CI como primeiro passo em direção ao CD; (iii) é criado então um serviço (*DeveloperData*) para concentrar informações dos desenvolvedores que usam metadados do modelo de domínio; (iv) após isto, são implementadas boas práticas de CD como a separação do código-fonte e da configuração, de modo que seja possível mudar a configuração sem a necessidade de *redeploy* a aplicação. A adoção de imagens *Docker* ajuda a manter a aplicação com o mesmo comportamento em ambientes diferentes; (v) na sequência, é introduzido um servidor de borda (API *Gateway*) com intuito de minimizar o impacto às mudanças internas; (vi) outra medida para tornar o sistema mais resiliente, é a adoção de serviços de descoberta, balanceador de carga e *circuit breaker*; (vii) feito isto, a próxima etapa trata da implementação propriamente dita dos serviços e suas *features*; (viii) os serviços são então instalados em *clusters* ao invés de um servidor único. Neste momento, entram os testes independentes dos serviços usando *consumer-driven tests*; (ix) por fim, cada serviço é monitorado independentemente. Isto permite que qualquer anomalia detectada resulte em ações corretivas para manter o serviço online.

Tabela 3.1: Método de migração Balalaie et al. [1] .

#	Etapa	Atividade
1	Estrutura da equipe GRUPO: 1	- Adotar equipes multidisciplinares - Adotar uma equipe central (*core team*) - Manter cada equipe focada em um serviço
2	Integração Contínua - CI GRUPO: 5	- Criar servidor CI (Jenkins) - Criar repositório de código (GitLab) - Criar repositório de artefatos (Artifactory) - Adotar sistema de gerenciamento de configuração - Adotar *Docker Registry* - Adotar *consumer-driven contracts* [160]
3	Serviço de metadados GRUPO: 2	- Criar serviço que mantém informações do domínio - Adotar *Spring Boot* - Expor funcionalidades em *RESTful API*

4	Entrega Contínua - CD (*deployment* automatizado) GRUPO: 5	- Separar o código-fonte e configuração - Adotar *Spring Cloud Configuration Server* - Adodar *Spring Cloud Context* - Criar repositório de código-fonte para cada serviço - Criar *Dockerfile* para cada serviço - Criar um job CI por serviço - Adotar *Docker Compose*
5	Servidor de borda GRUPO: 5	- Criar um servidor de borda (API *Gateway*) - Adaptar as chamadas na aplicação
6	Serviços dinâmicos GRUPO: 5	- Criar *Service Discovery* - Criar *Load Balancer* - Criar *Circuit Breaker*
7	Implementação de serviços GRUPO: 4	- Criar serviços da aplicação
8	Clusterização GRUPO: 5	- Criar um *cluster* de *CoreOS* - Adotar agentes *Kubernetes* - Fazer *deployment* dos serviços (no *cluster*) - Testar serviços usando *consumer-driven tests*
9	Monitoramento GRUPO: 6	- Adotar *Kibana* para visualização - Adotar *Elasticsearch* para consolidar métricas - Adotar *Logstash*

3.2.2 Método de migração Taibi et al. [2]

O trabalho de Taibi et al. [2] caracterizado na Tabela 3.2 propõe um *framework* baseado no processo de migração de três diferentes companhias. Esse *framework* visa atender tanto ao desenvolvimento de sistemas a partir do zero quanto a criação de novas *features* implementadas como microsserviços, gradualmente suprimindo o legado existente. O processo de migração identificado começa com (i) a análise do sistema em termos de estrutura e códigos, tendo o objetivo de melhor compreender as dependências dos módulos. Essa primeira etapa do processo pode ser auxiliada por ferramentas como *Structure101* e *SchemaSpy* ou mesmo manualmente; (ii) em um segundo momento, é definida a arquitetura do sistema, o que inclui modelar a decomposição da aplicação e do banco de dados em pequenos microsserviços, concomitantemente estabelecendo uma série de diretrizes e princípios; (iii) também é feita a priorização de *features* e serviços a serem desenvolvidos. Essa priorização pode ser feita com base (a) no valor que o serviço tem para o cliente, (b) nos serviços com menores dependências e que precisam ser entregues a curto prazo, seja por *bugs* ou necessidade do cliente, ou (c) considerando que apenas novas *features* devam ser implementadas como microsserviços; (iv) feito isto, a codificação dos microsserviço é iniciada; (v) como última etapa, são realizados testes de unidade e integração. O micros-

serviço pode ser testado como caixa-preta (do inglês *black-box test*), isto é, o resultado de saída é simplesmente comparado ao resultado original previsto.

Tabela 3.2: Método de migração Taibi et al. [2] .

#	Etapa	Atividade
1	Análise do sistema GRUPO: 2	- Identificar dependências[1,2] - Realizar análise de *code smells*
2	Arquitetura do sistema GRUPO: 3	- Definir estrutura do sistema - Definir diretrizes e princípios arquiteturais - Particionar sistema em microsserviços - Definir ferramentas e *frameworks* - Definir protocolo de comunicação e APIs - Adotar plano de contingência arquitetural (plano B) - Realizar análise de risco
3	Priorização de funcionalidades GRUPO: 3	- Priorizar serviços a serem desenvolvidos
4	Codificação GRUPO: 4	- Implementar microsserviços
5	Testes GRUPO: 5	- Realizar testes de unidade e integração - Testar microsserviço como *black-box*

3.2.3 Método de migração Fan & Ma [3]

Em razão da escassez de métodos que facilitem a migração para microsserviços, Fan & Ma [3] propõem um método baseado no ciclo de vida de desenvolvimento de software (do inglês *software development life cycle* - SDLC), apresentando técnicas e ferramentas utilizadas durante o *design* e desenvolvimento. Os autores demonstram a eficácia do seu método de migração em um exemplo ilustrativo de uma aplicação *mobile* afetada por problemas de acoplamento e latência. Assim, utilizando-se de uma abordagem de migração gradual e iterativa (i) o primeiro passo é realizar uma análise interna do atual sistema por meio do DDD, de modo a extrair microsserviços candidatos; (ii) depois, cabe uma análise do banco de dados a fim de verificar a compatibilidade do *schema* com os microsserviços candidatos, filtrando aqueles inadequados; (iii) na terceira etapa, o código-fonte relacionado ao serviço é extraído e uma interface Java (Interface de programação OO - Orientada a Objetos) passa a ser tratada, temporariamente, como interface do serviço; (iv) na sequência, são definidos alguns padrões a serem adotados, entre eles o protocolo de comunicação (Ex: síncrono ou assíncrono), o formato de dados (Ex: JSON) e o *framework* do microsserviços; (v) feito isto, é procedida a codificação dos microsserviços. Nesta etapa, as interfaces Java são transformadas em interfaces de serviço reais, como REST ou MQTT, permitindo a comunicação entre os serviços; (vi) *Fan & Ma* abordam não apenas

o desenvolvimento, mas também um série de atividades ligadas à operação, sobretudo, a automatização de *commit* no controle de versão, testes e *deploy*; (vii) ao final do processo, é preciso manter o *feedback* e monitoramento dos serviços por meio de ferramentas de análise de *log* e métricas. O método de *Fan & Ma* é sumariamente descrito na Tabela 3.3.

Tabela 3.3: Método de migração Fan & Ma [3] .

#	Etapa	Atividade
1	Análise da atual arquitetura GRUPO: 3	- Extrair microsserviços candidatos com DDD
2	Análise do banco de dados (BD) GRUPO: 2	- Verificar se o BD atende ao microsserviço candidato
3	Extração de código de serviço GRUPO: 3	- Extrair do monolítico o código candidato ao serviço - Tratar a interface OO como interface do serviço
4	Definição de padrões GRUPO: 3	- Definir entre comunicação *async* e *sync* - Definir formato dos dados (JSON) - Definir o *framework* - Definir se o atual BD é apropriado
5	Codificação GRUPO: 4	- Transformar interfaces OO em interfaces de serviço - Desenvolver o microsserviço
6	Automatização GRUPO: 5	- Efetuar *commit* no controle de versão - Realizar teste de unidade - Realizar *build* - Realizar teste do serviço - Realizar teste *end-to-end* - Realizar teste de aceitação do usuário - Realizar *deploy*
7	Monitoramento GRUPO: 6	- Utilizar ferramentas de análise de *log* e métricas

3.2.4 Método de migração Ghani & Zakaria [4]

Segundo Ghani & Zakaria [4] os microsserviços vêm ganhado forças e têm sido visto por inúmeras companhias como uma forma de ajudá-las a conquistarem o sucesso alcançado pela Netflix. Por considerarem que não existe um método sistemático para se produzir microsserviços, *Ghani & Zakaria* propõem uma forma de desenhar microsserviços em quatro principais etapas, validadas em um estudo de caso intitulado "MyFlix". O proposto método representado na Tabela 3.4 sugere inicialmente (i) modelar a estrutura organizacional. Isto, não significa descrever uma estrutura hierárquica da organização, mas uma estrutura organizacional viável baseada nas definições dos sistemas e suas interconexões; (ii) após isto, é preciso modelar o processo de negócio, mapeando os principais processos e atividades da organização/sistema, por exemplo utilizando a "*promise theory*" [161], que

é uma nova abordagem empregada na construção de sistemas distribuídos e autônomos como microsserviços; (iii) nesta etapa os microsserviços candidatos já possuem um nome e uma API, podendo então ser implementados; (iv) os artefatos gerados nas fases anteriores (processo de negócio e criação de microsserviços) devem ser colocados em uma estrutura única e atualizada, servindo como documentação e referência a qualquer agente que deseje subscrever o microsserviço existente.

Tabela 3.4: Método de migração Ghani & Zakaria [4] .

#	**Etapa**	**Atividade**
1	Estrutura organizacional GRUPO: 2	- Modelar estrutura tecnológica - Modelar estrutura geográfica - Modelar estrutura cliente / fornecedor - Modelar a atual estrutura da organização - Identificar atividades da organização/sistema
2	Processo de negócio GRUPO: 2	- Mapear processo de negócios relevantes
3	Criação de microsserviços GRUPO: 3 e 4	- Identificar microsserviços candidatos - Definir APIs dos microsserviços - Modelar diagrama de dependência dos microsserviços - Implementar microsserviços
4	Estrutura de artefatos GRUPO: 3	- Mapear a estrutura de processos e microsserviços

3.2.5 Método de migração Mishra et al. [5]

O artigo de Mishra et al. [5] trás uma abordagem que promete uma transição suave do monolítico para microsserviços, minimizando a complexidade enfrentada por um arquiteto de soluções. Segundo eles, modernizar uma aplicação é um processo de refatoração do software legado para se alinhar às modernas necessidades de negócios. Isto basicamente pode ser alcançado quebrando uma tradicional aplicação monolítica em pequenas unidades gerenciáveis, chamadas microsserviços. Para tal, propõem um processo de rearquiteturação em quatro estágios: (i) primeiro, é preciso compreender os componentes subjacentes a qualquer arquitetura, agrupá-los pela coesão das funcionalidades e expressá-los em diagramas UML; (ii) a partir dessa compreensão, é realizado o processo de identificar funcionalidades que podem ser implementadas como microsserviços, atentando-se para fatores como coesão, interfaces de mensagens, dependências de banco de dados - BD e escalabilidade; (iii) uma vez que alguns microsserviços podem ser mais escaláveis que outros, o ideal é haver uma definição das regras de troca de mensagens que costuma ser de modo síncrono (Ex: REST) ou assíncrono (Ex: Pub-Sub) dependendo da natureza do serviço; (iv) o próximo passo é a reestruturação dos dados. Neste caso, espera-se que

cada microsserviço tenha seu próprio repositório de dados, se possível. Como não está explícito no artigo, é assumido que desenvolvimento do microsserviço ocorra nessa epata; (v) por fim, existe um preocupação para lidar com problemas ligados a replicação. Isso é pertinente, pois a medida que as instâncias de serviços se replicam, ocorrem complicações associadas a consistência de dados e a manutenção do estado (*statefulness*). A Tabela 3.5 resume as tarefas e atividades do método de Mishra et al.

Tabela 3.5: Método de migração Mishra et al. [5] .

#	Etapa	Atividade
1	Formalidade GRUPO: 2	- Entender os componentes subjacentes à arquitetura - Expressar aplicação como um diagrama de classe UML - Agrupar classes com funcionalidades coesas
2	Componentização GRUPO: 3	- Identificar funcionalidades candidatas a microsserviços - Identificar número de funções coesas presentes - Definir interface de mensagens entre componentes (API) - Evitar múltiplas dependências entre BD e serviços - Gerenciar a escalabilidade
3	Seleção de Interface GRUPO: 3	- Definir padrão de mensagens *sync* ou *async*
4	Reestruturação dos dados GRUPO: 3 e 4	- Reestruturar o repositório de dados - [Desenvolvimento]
5	Manipulação de problemas GRUPO: 6	- Lidar com inconsistências devido a replicação

3.2.6 Método de migração Silva et al. [6]

Por intermédio de dois estudos de casos voltados à migração de sistemas legados para microsserviços, Silva et al. [6] compartilham lições aprendidas e identificam três principais fases que guiam o processo de migração (Tabela 3.6). O *roadmap* (i) começa com a análise e identificação das funcionalidades chaves da aplicação e suas respectivas responsabilidades, de modo a obter uma clara visão do domínio. É importante destacar que isto é um processo iterativo e incremental; (ii) na sequência, o objetivo é destilar o monolítico existente usando conceitos de DDD na expectativa de otimizar a granularidade e a coesão dos serviços, sem haver, no entanto, a preocupação com detalhes de implementação. Essa decomposição é na verdade um esforço colaborativo envolvendo desenvolvedores, especialistas do domínio e *stakeholders*; (iii) havendo definição sobre os *contextos delimitados* (do inglês *Bounded Contexts*) e o mapeamento elaborado em etapas anteriores, finalmente é feita a migração para a arquitetura de microsserviços. Nesta fase, o foco está na implementação e testes. É importante enfatizar que os *contextos delimitados* desempenham um papel fundamental, tanto para identificar quanto organizar os microsserviços. Silva et al.

também mencionam que outras etapas podem surgir no processo de migração dependendo da especificidade de cada sistema.

Tabela 3.6: Método de migração Silva et al. [6] .

#	Etapa	Atividade
1	Análise do monolítico GRUPO: 2	- Analisar código-fonte do legado - Analisar banco de dados do legado - Identificar principais funcionalidades - Estabelecer responsabilidades e fronteiras - Migrar o sistema legado para uma versão modularizada
2	Identificação do domínio GRUPO: 3	- Identificar domínios e subdomínios - Criar um modelo de domínio - Identificar contexto delimitado - Construir um mapa de contexto - Aplicar DDD (*aggregate*, *value objects* e *domain*)
3	Arquitetura de software GRUPO: 3, 4 e 5	- Decompor cada contexto em candidato a microsserviço - Criar estrutura padrão de diretório para cada *contexto* - Rodar testes de unidade e integração - Migrar banco de dados legados incrementalmente - Migrar monolítico modularizado para microsserviço

3.3 Consolidação do Estado da Arte

Nesta seção é feita uma comparação sumária entre os métodos de migração. Após a breve síntese dos métodos de migração supracitados e a devida classificação empírica por equivalência a um dos grupos definidos na Seção 3.2, é possível estabelecer um perfil comparativo entre os métodos estudados, conforme mostra a Tabela 3.7. Mediante o exposto, infere-se que os métodos de migração para microsserviços estudados tendem a se concentrar mais em torno das etapas de *análise* e de *desenvolvimento* e menos em assuntos como *planejamento* e *monitoramento*. Embora tais "etapas" não sejam garantia de migração bem-sucedida, para Fan & Ma [3], elas auxiliam o processo de migração a fluir alinhada ao ciclo de vida do desenvolvimento de software e às estratégias *DevOps*.

Tabela 3.7: Comparação entre métodos de migração.

Método \ Etapa (Suporte)	Planejamento operacional	Análise do domínio	Decomposição em serviços	Desenvolvimento do microsserviço	Integração DevOps	Monitoramento e otimização
Balalaie et al. [1]	✓	✓		✓	✓	✓
Taibi et al. [2]		✓	✓	✓	✓	
Fan & Ma [3]		✓	✓	✓	✓	✓
Ghani & Zakaria [4]		✓	✓	✓		
Mishra et al. [5]		✓	✓			✓
Silva et al. [6]		✓	✓	✓	✓	

Em resumo, o estado da arte tem o desafio de mapear e discutir certa produção acadêmica. Representa o nível mais alto de um processo de desenvolvimento em determinado momento, servindo como ponto de partida para o estudo proposto [162]. Neste sentido, considera-se que o objetivo deste capítulo tenha sido alcançado, uma vez que foram identificados os artigos científicos de maior relevância sobre tema e que servirão para fundamentar o método *Microservice Full Cycle* - MFC apresentado no Capítulo 5.

Capítulo 4

Metodologia

Metodologia é um conjunto de princípios, práticas e procedimentos aplicados a um específico ramo do conhecimento [163]. Este capítulo apresenta a sistemática utilizada para alcançar os objetivos definidos na Seção 1.4. Em geral, trata-se de uma pesquisa de natureza aplicada, uma vez que é voltada a gerar conhecimentos de aplicação prática e dirigida à solução de problemas específicos [164]. Quanto aos procedimentos técnicos, é utilizado uma abordagem similar a *pesquisa-ação* [165] intitulada Metodologia de Pesquisa em *Design Science* (do inglês *design science research methodology* - DSRM), porém, melhor adaptada a Sistemas de Informação (SI) [163], detalhada na Seção 4.1.

4.1 Método de pesquisa

Método é o caminho pelo qual se chega a determinado resultado [166], isto é, um conjunto de atividades sistemáticas e racionais que permitem alcançar o objetivo de forma convincente [167, 168]. Para isso, tem se adotado o método de Pesquisa em *Design Science* - DSRM ilustrado na Figura 4.1.

Segundo Peffers et al. [163], o método de Pesquisa DSRM incorpora princípios, práticas e procedimentos requeridos para a realização de pesquisas, sobretudo, no design de pesquisa em engenharia e ciência da computação. Esse método de pesquisa tem sido utilizado em diversas pesquisas ligadas a SI/TI [4, 169, 170] e um dos objetivos finais de um processo DSRM é fornecer um modelo mental, isto é, um "*modelo da realidade em pequena escala*", o que pode se assemelhar aos modelos de arquitetos ou diagramas de físicos, pois sua estrutura é análoga à estrutura da situação real que eles representam [163]. A simulação proposta segue as seis atividades do DSRM, descritas a seguir:

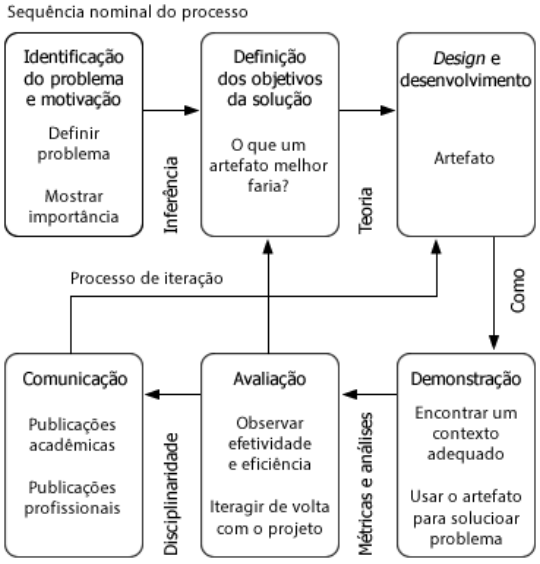


Figura 4.1: Modelo de Processo DSRM.
(Fonte: baseado em Peffers et al. [163])

Atividade 1 - Identificação do problema e motivação. Definir e explorar um problema de pesquisa ajuda a capturar a complexidade e a desenvolver um artefato que leve a uma solução [163]. Conforme Seção 1.1, o problema de pesquisa é descrito como a carência de um método que suporte a migração de sistemas monolíticos legados para arquitetura de microsserviços. Na prática, busca-se um modo sistemático para modernizar sistemas de software legados, a exemplo do SISP. Na realidade, esse é um problema endereçado a maioria das organizações que veem seus sistemas crescerem e se tornarem complexos, perdendo qualidade em manutenibilidade, desempenho, interoperabilidade, entre outros.

Atividade 2 - Definição dos objetivos da solução. Esta etapa descreve os objetivos de uma possível e viável solução a partir da definição do problema. Evidencia como uma solução seria melhor que as soluções existentes ou como se espera que um novo artefato suporte soluções para problemas ainda não abordados [163]. Por esta razão, os objetivos discutidos na seção 1.4 são elencados para prover o método MFC e validá-lo através de uma simulação, o que inclui a coleta de métricas e análise do resultado.

Atividade 3 - Projeto e desenvolvimento. Refere-se à criação do artefato proposto, podendo ser um modelo, método, novos recursos técnicos ou qualquer objeto projetado que contribua com a pesquisa. Essa atividade envolve arquitetar o artefato desejado para então criá-lo [163]. No caso deste trabalho, isso se faz pela elaboração do método MFC, cujo arcabouço é inspirado em experiências de outros pesquisadores, no ciclo de vida de desenvolvimento de software e em estratégias DevOps, conforme descrito no Capítulo 5.

Atividade 4 - Demonstração. Um modo de provar que a ideia funciona é utilizar o artefato proposto para resolver uma ou mais instâncias do problema, o que pode ser feito por meio da experimentação, simulação, estudo de caso, entre outros [163]. Nesta etapa descrita na Seção 6.1, o propósito é realizar uma simulação em pequena escala do processo de migração, onde parte do módulo SISP (monolítico legado) é submetido às etapas e atividades do método MFC.

Atividade 5 - Avaliação. Esta etapa lida com métricas e técnicas de análise, a fim de observar e mensurar o quão bem o artefato suporta a solução do problema. Para isto, pode-se comparar se os resultados reais alcançados estão de acordo com os objetivos da solução inicialmente definidos, utilizando-se, por exemplo, de medidas quantificáveis do desempenho do sistema, como latência (tempo de resposta) e *throughput* (taxa de transferência). Essa avaliação discutida na Seção 6.1, inclui qualquer evidência empírica ou prova lógica e, conforme resultados, o pesquisador pode decidir rever o artefato ou seguir adiante, deixando as melhorias para projetos futuros [163].

Atividade 6 - Comunicação. A comunicação é necessária para difundir o conhecimento sobre o aprendizado e os resultados alcançados. É importante comunicar aos pesquisadores e profissionais da área sobre o problema de estudo e sua relevância, o artefato e sua utilidade, o rigor do *design* e sua eficácia [163] a fim de incentivar outras abstrações e questionamentos que promovam o avanço científico. Neste caso, cumpre esse papel, esta dissertação e os possíveis artigos científicos dela originados, comunicando os desafios e achados da presente pesquisa. A Figura 4.2 resume o progresso do presente trabalho dentro do método de Pesquisa DSRM.

DSRM	Identificação do problema e motivação	Definição dos objetivos da solução	*Design* e desenvolvimento	Demonstração	Avaliação	Comunicação
Progresso	Seção 1.1 Necessidade de modernização do legado Falta de um método de migração	Seção 1.4 Elaborar o MFC Executar experimento de validação	Cap. 5 Elaboração do MFC (criar o artefato)	Seção 6.1 Experimento - SISP (demonstrar que o artefato funciona)	Seção 6.2 Métricas (mensurar quão bem o artefato suporta a solução do problema)	Defesa da dissertação (Banca) Publicação de dissertação e artigos

Figura 4.2: Resumo do DSRM vs Trabalho de pesquisa.
(Fonte: própria)

Capítulo 5

Microservice Full Cycle - MFC

Microservice Full Cycle - MFC é um método constituído por um conjunto de etapas e atividades que visam a gradual modernização de sistemas de software monolíticos legados, utilizando-se da arquitetura de microsserviços. Conforme ilustra a Figura 5.1, o MFC é inspirado em experiências vivenciadas na literatura científica, no tradicional ciclo de vida de desenvolvimento de software e em estratégias *DevOps*, ressaltando aspectos da arquitetura de microsserviços. Não é pretensão do método MFC esgotar o tema em torno da arquitetura de microsserviços, podendo, entretanto, contribuir em elucidar o processo de migração a partir de sistemas legados em organizações que considerem a adoção de microsserviços tecnicamente viável a sua realidade.

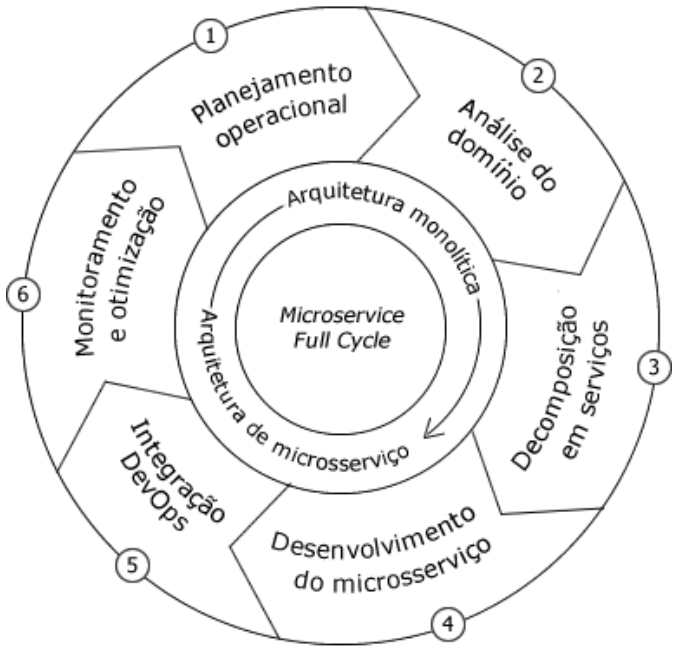


Figura 5.1: Método *Microservice Full Cycle* - MFC.
(Fonte: própria)

Segundo Ghani & Shanudin [4], um método não deve aumentar a complexidade de construção do sistema, mas reduzi-la a um grau que possa ser controlada pelos desenvolvedores [4]. Neste sentido, uma abordagem de fácil compreensão é pensar o *Microservice Full Cycle* - MFC a partir de uma visão em que suas atividades são baseada em *"pacotes de trabalhos"*, a exemplo da *Work breakdown structure* (WBS), descrito pelo PMBOK®[1] e ilustrado na Figura 5.2. As atividades recomendam *"o que fazer"* e deixam o *"como fazer"* a critério do utilizador do método, que deve buscar as soluções mais adequadas, quebrando-as em tarefas menores executáveis quando necessário.

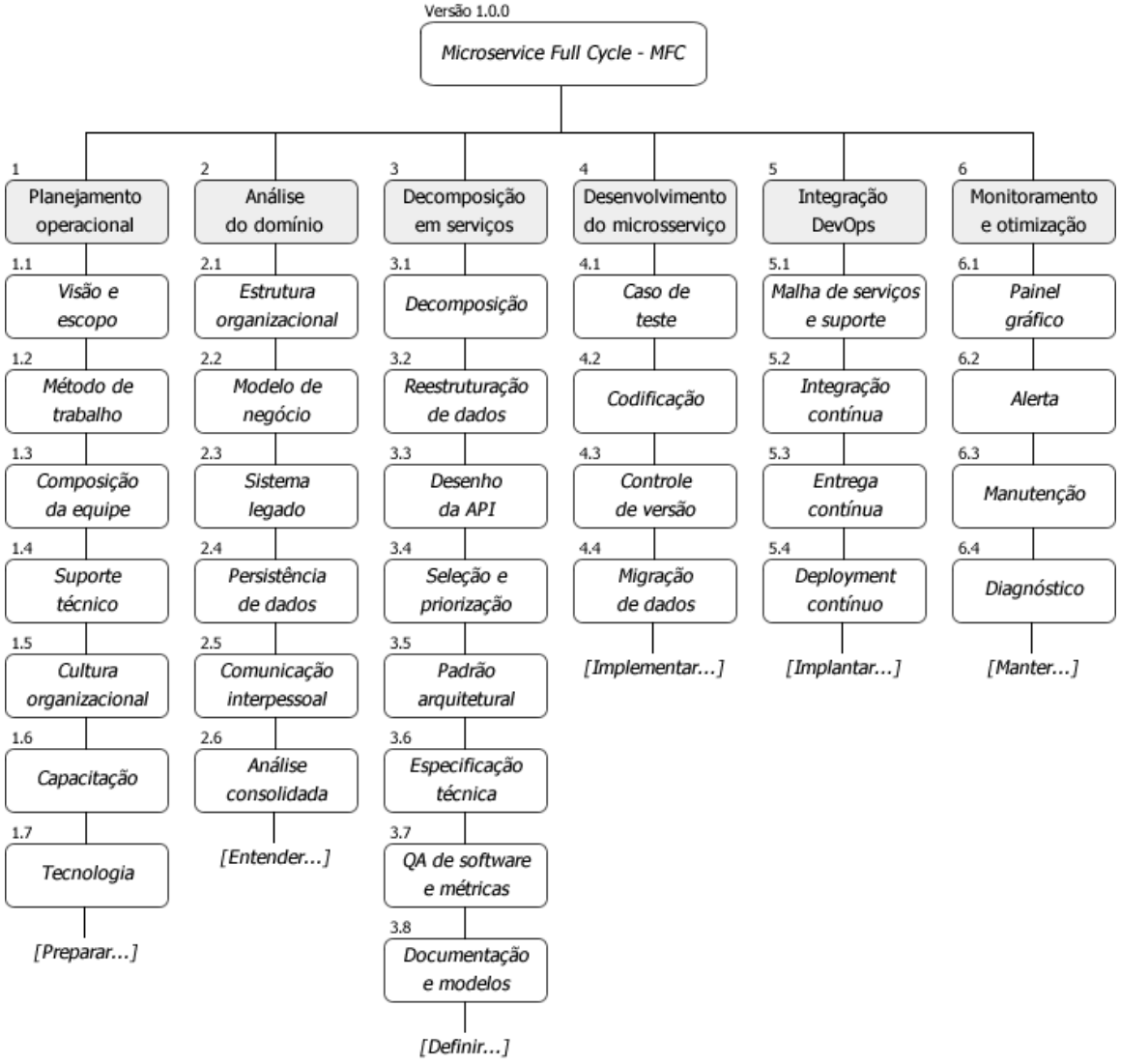


Figura 5.2: WBS - Microservice Full Cycle - MFC.
(Fonte: própria)

[1] *Project Management Body of Knowledge* (PMBOK)

Para os casos em que a organização já possui soluções baseadas em microsserviços, aplicar o MFC ao monolítico tende a ser mais simples. Do contrário, irá requerer uma longa e difícil jornada, sendo recomendado o uso de *patterns* como "*Pave the Road*" e "*Baby Steps*" [45]. Além disto, como MFC emprega infraestrutura tecnológica (hardware e software) em praticamente todas as suas etapas, é comum que desenvolvedores sejam tentados a criar já no início do projeto uma estrutura minimamente funcional, seja instalando ferramentas ou preparando o ambiente operacional. Embora isto seja possível, tende a aumentar a complexidade da primeira etapa, exigindo maior conhecimento sobre recursos que só serão utilizados em etapas futuras. Uma alternativa é fazer o trabalho progressivamente à medida que surgem as demandas. Por exemplo, pode-se esboçar a intenção de uso de *containers* na etapa de *"Planejamento operacional"*, mas prorrogar a escolha e instalação do software apropriado para as etapas seguintes, quando então o software efetivamente começará a ser utilizado. A seguir, as etapas e atividades estão propositalmente descritas de modo sucinto, pois considera-se implícito o arcabouço prático e teórico inerentes à arquitetura de microsserviços.

Etapa 1 - Planejamento operacional. Lidar com microsserviços requer previamente algumas discussões e preparativos, sobretudo, ligados às pessoas e operações. Embora a etapa de planejamento possa ter similidades com a gestão de projetos (escopo, recursos, *stakeholders*, risco, viabilidade, orçamento, cronograma, etc.), o mérito deste trabalho fica restrito às atividades necessárias para iniciar e estruturar o processo de migração com microsserviços. É o momento de pensar sobre o escopo do problema e a viabilidade de soluções, isto é, discutir os objetivos e buscar meios de alcançá-los. Esta é, portanto, uma etapa de aprendizado, preparativos e aperfeiçoamento contínuo, cujo *feedback* leva a melhores resultados e menos retrabalho. Dentre os desafios desta etapa, estão:

VISÃO E ESCOPO
Descreve a visão geral, problemas e escopo funcional (porção) do sistema monolítico legado, alvo do corrente ciclo de migração para arquitetura de microsserviços.

MÉTODO DE TRABALHO
Estipula os métodos a serem utilizados para guiar a migração, bem como qualquer abordagem que contribua na sistematização racional do processo.

COMPOSIÇÃO DA EQUIPE
Trata da composição da equipe de desenvolvimento que irá lidar com os microsserviços, devendo ser pequena, multidisciplinar, independente, autônoma e alinhada às práticas *DevOps*.

SUPORTE TÉCNICO
Busca suporte e supervisão de especialista(s) técnico com reconhecida competência profissional e de especialista(s) de negócio com elevada experiência organizacional.

CULTURA ORGANIZACIONAL
Fomenta a aceitação das mudanças decorrentes da adoção de microsserviços e busca formas de institucionalizá-la como cultura organizacional.

CAPACITAÇÃO
Identifica habilidades e aprendizados necessários para lidar com a arquitetura de microsserviço, provendo a devida capacitação técnica aos envolvidos.

TECNOLOGIA
Esboça a infraestrutura tecnológica (hardware/software), equipamentos, ferramentas ou qualquer mecanismos e serviços idealizados à implementação dos microsserviços.

Etapa 2 - Análise do domínio. Nesta etapa o objetivo está concentrado em conhecer o domínio da aplicação. Sistemas de software tendem a incorporar conceitos e elementos da relação com o domínio, afinal, o propósito do software na vida real é resolver problemas de negócios [171]. Na análise do domínio o que se espera, portanto, é tão somente colher informações suficientes para subsidiar decisões que mostrem por onde começar a decomposição [16]. O *"business core"* da organização pode ser obtido por meio de artefatos e da comunicação interpessoal, sempre mantendo a perspectiva de identificar dependências, responsabilidades, fronteiras e agrupar as funcionalidades comuns, sem tirar o foco dos problemas sistêmicos que realmente precisam ser tratados [16]. Algumas atividades, são:

ESTRUTURA ORGANIZACIONAL
Analisa a estrutura organizacional em artefato administrativo (documentos e organogramas), contexto tecnológico e geográfico e relação pessoal/departamental, ressaltando valores do domínio.

MODELO DE NEGÓCIO
Analisa o mapeamento dos processos de negócios da organização, buscando investigar e aprender sobre as nuances do domínio (*"business core"*)

SISTEMA LEGADO
Analisa a documentação e o código-fonte do sistema monolítico legado, observando rotinas e relações significativas à compreensão do domínio.

PERSISTÊNCIA DE DADOS
Analisa o banco de dados ou outras formas de persistência do legado, observando entidades e atributos que exercem influência sobre o domínio da aplicação.

COMUNICAÇÃO INTERPESSOAL
Coleta informações baseadas em *feedback* de pessoas dentro e fora da organização, assimilando expectativas e frustrações que mais impactam o domínio.

ANÁLISE CONSOLIDADA
Compilado do conteúdo essencial analisado, dando uma visão geral de como as partes do domínio se apresentam, se comportam e se relacionam.

Etapa 3 - Decomposição em serviços. Nesta etapa predominam os aspectos da arquitetura de software, com ênfase na decomposição do sistema legado em microsserviços. Um serviço deve ser desenhado segundo os aspectos organizacionais e de negócio que provê aos usuários [32, 87]. O grande desafio, entretanto, é definir a granularidade dos serviços (tamanho adequado) [83], dividindo o sistema em pequenas unidades de modo que tenham baixo acoplamento e alta coesão [12, 172]. Embora não exista uma maneira aprovada de decompor serviços [3], o *Domain-driven design* - DDD está entre as práticas mais comuns para guiar a transformação de uma arquitetura monolítica em microsserviços [23, 32, 36, 173]. Cada microsserviço tende a ser responsável por um *contexto delimitado*, isto é, o microsserviço deve fornecer apenas um conjunto de funcionalidades coesas e com escopo bem definido, atendendo uma específica capacidade de negócio [82]. Embora haja também algumas técnicas (semi)automatizadas que podem ajudar na decomposição em serviços, a principal recomendação é para que inicialmente se priorize as funcionalidades que tenham baixo impacto/risco [6], elegendo, ao final, microsserviços candidatos com maior valor agregado e menor número de dependências eferentes [2]. É preciso ainda lidar com uma série de outros elementos arquiteturais indispensáveis, como sugeridos a seguir:

DECOMPOSIÇÃO
Provê o *design* da decomposição do monolítico em microsserviços por meio de abordagens como DDD: linguagem ubíqua, subdomínio, objeto de valor, agregados, contextos delimitados, etc.

REESTRUTURAÇÃO DE DADOS
Arquiteta a reestruturação de dados a partir da base de dados do monolítico, utilizando-se de um ou mais *patterns* que idealmente tornem os dados dedicados ao microsserviço.

DESENHO DA API
Expressa funcionalidades do microsserviço na forma de API, passando a responder exclusivamente por estas interfaces publicadas como um contrato do serviço.

SELEÇÃO E PRIORIZAÇÃO
Identifica os serviços candidatos com maior potencial de serem desenvolvidos, priorizando-os em ordem de importância conforme critérios previamente estabelecidos.

PADRÃO ARQUITETURAL
Estabelece padrões arquiteturais, boas práticas, desenhos de conectividade e componentes de software que favoreçam o processo de desenvolvimento e otimização.

ESPECIFICAÇÃO TÉCNICA
Define especificações técnicas do software como serviço: linguagem de programação, *framework*, estrutura de diretórios, configurações, protocolos de comunicação, formato dos dados, etc.

QA DE SOFTWARE E MÉTRICAS
Define os atributos de qualidade de software (QA) desejáveis e métricas associadas para mensurar o alcance de objetivos propostos.

DOCUMENTAÇÃO E MODELOS
Elabora eventual documentação, modelos e diagramas arquiteturais julgados essenciais pela equipe de desenvolvimento.

Etapa 4 - Desenvolvimento do microsserviço. Uma vez que haja a compreensão do escopo do problema e o prévio desenho da solução, é hora de executar o planejado, isto é, criar o microsserviço. Esta é a etapa de codificação, podendo cada contexto delimitado do domínio ser um microsserviço [32]. Em termos de implementação, existem duas opções para migrar um sistema monolítico para uma arquitetura baseada em serviços: reescrever do zero ou extrair serviços (funcionalidades) do código-fonte legado. Se o código legado tiver alto valor, este último é o recomendado [154]. O ideal é que a versão antiga do monolítico deixe de ser usada após a implementação do microsserviço, do contrário, haverá um problema de manutenção com a mesma funcionalidade coexistindo em dois lugares. Em caso de incompatibilidade de chamadores antigos do monolítico com a nova versão do microsserviço, *patterns* como *Replace as Microservice*, *Extract Components and Add Façade* e *Proxy Monolith Components to Microservices* podem ajudar neste sentido [45]. Comece com um serviço por equipe e considere que um serviço deve resolver apenas um problema [174]. Ao final desta etapa a proposta é que o microsserviço esteja funcionalmente pronto para ser homologado (*stage step*). As atividades mais comuns são:

CASO DE TESTE
Testa determinada funcionalidade do sistema a partir da elaboração de casos de teste, a exemplo de abordagens "*test-first*" como TDD e BDD.

CODIFICAÇÃO
Implementa a estrutura e o microsserviço conforme padrões e arquiteturas pré-definidas, além de prover a integração com o monolítico legado e consumidores.

CONTROLE DE VERSÃO
Estipula versionamento para contrato de serviço (API) e controle de versão de código-fonte, mantendo preferencialmente repositórios individuais para cada microsserviço.

MIGRAÇÃO DE DADOS
Cria a base de dados do microsserviço, importando e sincronizando os dados do monolítico se necessário, assumindo a possibilidade de consistência eventual.

Etapa 5 - Integração *DevOps*. As práticas *DevOps* visam diminuir o tempo entre uma mudança efetivada na aplicação e sua transferência para o ambiente de produção [1]. Embora aqui representado em uma única etapa, *DevOps* costuma abranger e integrar todo o processo de criação do software. Neste contexto, entretanto, a ênfase é infraestrutura, operações e automação, princípios *DevOps* fundamentais que promovem uma abordagem ágil [99], indispensáveis quando o número de microsserviços começa a crescer e já não

é fácil geri-los. A proposta *DevOps* visa facilitar a comunicação e colaboração entre as equipes de *desenvolvimento* (Dev) e *operações* (Ops) de modo a garantir um fluxo contínuo de entrega de software. Em geral, isto é alcançado por meio de conceitos como *integração contínua* (CI) e *entrega contínua* (CD). Ao final desta etapa, espera-se, portando, que os microsserviços estejam "prontos" para serem automaticamente publicados no ambiente de homologação, potencialmente elegíveis à produção. Algumas atividades, são:

MALHA DE SERVIÇOS E SUPORTE
Adota mecanismos como containerização e orquestração, *service mesh*, servidor de mensageria, API *Gateway*, além de recursos como serviço de descoberta, balanceador de carga, etc.

INTEGRAÇÃO CONTÍNUA
Automatiza o processo de *build* e testes após o desenvolvedor realizar o *commit/push* do novo código em uma *branch* principal.

ENTREGA CONTÍNUA
Automatiza o processo de entrega de software para um ambiente similar à produção (*stage*, *release* ou *test*) sem efetivar o *deploy* em produção (manual).

DEPLOYMENT CONTÍNUO
Automatiza o *pipeline* completo (CI + CD), incluindo o *deploy* das mudanças do software em ambiente de produção sem a intervenção humana.

Etapa 6 - Monitoramento e otimização. Um bom processo é baseado na sua capacidade de obter *feedback* do sistema [74], neste caso, por meio do monitoramento do microsserviço, rapidamente intervindo com ações preventivas, corretivas ou de melhorias, quando preciso for. Havendo sido implementado o registro sistemático de *logs* e métricas que favoreçam a observabilidade dos microsserviços, é possível monitorar os eventos em tempo real em *dashboard* e extrair informações relevantes sobre o estado atual da aplicação, inclusive propondo mudanças mensuráveis no sistema [74]. Nesta etapa, a ideia é, portanto, fornecer dados internos para aumentar as medições externas, utilizando-se de eventos de telemetria e métricas para avaliar os atributos de qualidade de software, detectar instabilidades, degradação de serviços e tratativas. Basicamente as atividades desta etapa são:

PAINEL GRÁFICO
Apresenta por meio de painéis (*dashboards*) e gráficos de monitoramento, o contínuo *feedback* sobre métricas, eventos e saúde dos microsserviços.

ALERTA
Dispara mensagens de avisos (alarmes) quando o sistema entra em um estado crítico ou de atenção, permitindo a detecção e mitigação de problemas antes que os serviços experimentem interrupções.

MANUTENÇÃO
Coordena e mantêm ações preventivas, corretivas e de melhorias a partir dos dados de monitoramento, tratando anomalias e otimizando a qualidade do serviço à medida que *feedbacks* chegam.

DIAGNÓSTICO
Faz leitura e diagnóstico, manual ou automático, da observabilidade e monitoramento do serviços sobre aspectos como segurança, conformidade, atividades incomuns e problemas operacionais.

Concluindo, cabe ressaltar que as etapas e atividades do MFC podem ser seletivamente adotadas, sem qualquer obrigatoriedade ou sequência ideal. Novas atividades também podem ser acrescidas a critério do utilizador do método, preservando a semântica idealizada. Também é importante observar que o MFC incorpora diversos conceitos introduzidos pelos métodos avaliados no Capítulo 3, tendo, sobretudo, evidenciado características como (i) suporte a atividades em todas etapas da migração, (ii) dissociação tecnológica, e (iii) aplicabilidade seletiva (*on demand*), dentre outras.

Capítulo 6

Resultados e Análises

Este capítulo apresenta os resultados e as análises do presente trabalho, isto é, as atividades de *"demonstração"* e *"avaliação"* do artefato previstas pelo método de Pesquisa *Design Science* - DSRM (Seção 4.1). O objetivo é verificar a aplicabilidade do método *Microservice Full Cycle* - MFC em um contexto real, observando por meio de métricas a capacidade de evolução do sistema monolítico SISP quando migrado para a arquitetura de microsserviços. Para isto, busca-se produzir um "*modelo da realidade em pequena escala*" [60, 163], desenvolvido na simulação a seguir.

6.1 Demonstração

De acordo com o DSRM, a *"atividade de demonstração"* deve utilizar o artefato proposto para resolver uma ou mais instâncias do problema. Assim, nesta seção o método MFC é utilizado como estratégia para resolver anomalias de parte do *módulo cursos* do SISP (monolítico legado), um genérico Sistema de Seleção de Pessoal.

***Framework* de desenvolvimento.** A regra de ouro é que a migração ocorra gradualmente. Isso permite que eventuais falhas possam ser revertidas mais facilmente [16]. Para melhor organizar e gerenciar o processo, é proposto um *framework* aos moldes do *Scrum* [175], sem observar a totalidade dos preceitos, conforme mostra a Figura 6.1. Cabe destacar que *Scrum* não é pré-requisito para utilização do método MFC, embora oportuno e prático. O *framework* é iniciado com o *backlog* recebendo como entrada, "itens" a serem desenvolvidos. Esses itens tanto podem vir de *"histórias do usuário"* (do inglês *user story*) [176], quanto de uma *work breakdown structure* (WBS) [177]. A WBS/MFC em questão encontra-se definida no Capítulo 5. Uma vez que as atividades tenham sido selecionadas e priorizadas no *backlog*, têm início as *sprints*. Uma *sprint* funciona como um ciclo de trabalho realizado em determinado espaço de tempo (*time-box*). Segundo Sutherland &

Schwaber [175], ao final de uma *sprint* imediatamente começa uma nova *sprint* e assim, consecutivamente, o produto/software vai sendo entregue de modo iterativo e incremental. Isto reduz os riscos devido a maior oportunidade de *feedback*. Os demais trâmites do *framework* estão baseados no *Scrum Guide*[1], não carecendo maiores aprofundamentos.

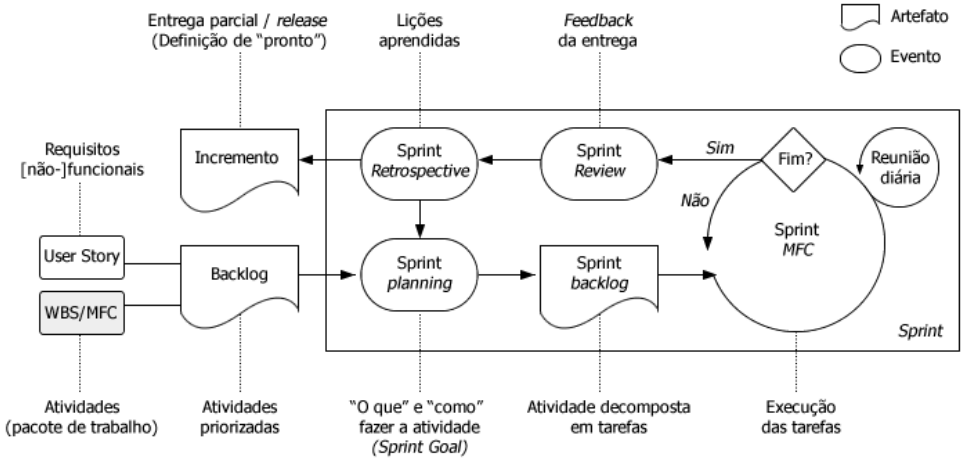


Figura 6.1: *Framework* de desenvolvimento *like-Scrum* - MFC.
(Fonte: própria)

Backlog. Seguindo a proposta do *framework*, inicialmente o *backlog* é populado com as atividades da WBS/MFC. A partir de então, existem pelo menos duas possibilidades. A primeira é tratar o *backlog* de modo tradicional, isto é, descrever o pacote de trabalho que vem da WBS/MFC como uma necessidade de alto nível ou como um requisito de usuário/software, procedendo a priorização e estimativa das atividades por meio de técnicas como *story points*, *task points*, *ideal days*, pontos de função, *planning poker*, dentre outras. Neste caso, o pacote de trabalho passa a representar um agrupamento ou visão do produto, podendo ser quebrado em tarefas durante o planejamento da *sprint*.

A segunda, e propositadamente menos informal, é simplesmente manter o pacote de trabalho da WBS/MFC no *backlog*, sem qualquer tratativa. À medida que cada planejamento de *sprint* ocorrer, adiciona-se à *sprint* somente os pacotes de trabalhos necessários para cumprir a meta da *sprint*, quebrando-os em tarefas estimadas e menores. A ideia é que isto funcione semelhante a uma lista de compras de supermercado. Na maioria das vezes, a lista de supermercado é só uma descrição simples (*backlog*). Quantidade, validade ou marca, por exemplo, não é uma preocupação até que o item seja de fato adicionado ao carrinho de compras (*sprint backlog*). Na prática, basta ver o que é necessário para atingir a meta da *sprint* atual e criar uma ou mais instâncias das atividades a serem executadas dentro da *sprint*. Por exemplo, se a meta é entender sobre o domínio da aplicação,

[1]https://www.scrum.org/resources/scrum-guide

adicione à *sprint* o pacote *"Modelo de negócio"*; se é preciso dispor de algum tempo com aprendizado, adicione *"Capacitação"*; se é preciso instalar alguma ferramenta, adicione *"Tecnologia"*, e assim por diante. Se a atividade não existir na WBS/MFC, basta criar sua própria. As tarefas originadas dos pacotes de trabalho que precisem ser atribuídas a mais de um desenvolvedor (Ex: Capacitação), podem ser replicadas e então atribuídas. A Figura 6.2 retrata uma possível projeção do *backlog* organizado em potenciais *sprints*.

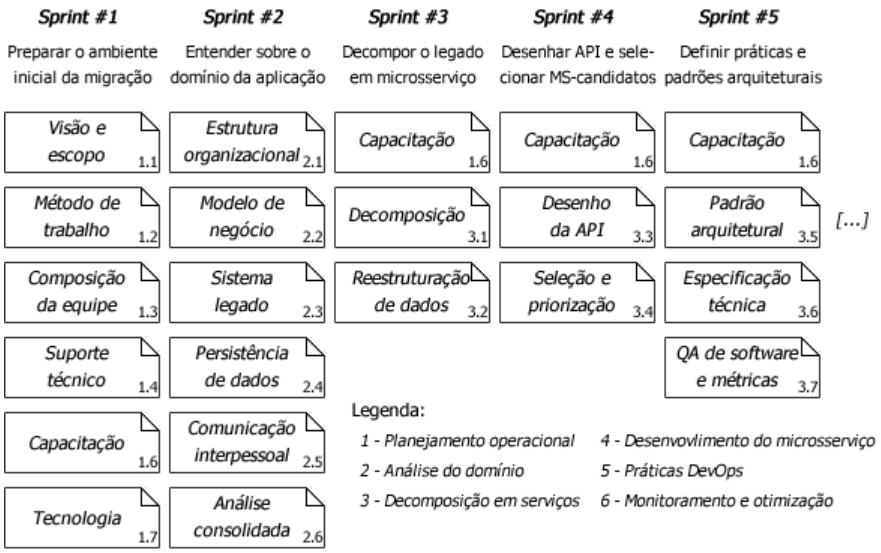


Figura 6.2: Exemplo de projeção do *backlog* em *sprints* - MFC.
(Fonte: própria)

Sprint. Ao iniciar uma *sprint*, extrai-se do *backlog* somente os itens que serão desenvolvidos durante a *sprint* e desde que exequíveis no *time-box* da *sprint*. É desejável que os desenvolvedores tenham prévio conhecimento técnico do trabalho que será realizado dentro das *sprints*, caso contrário, será preciso alocar tempo para capacitação. Todo trabalho a ser realizado deve estar contido na *sprint*, incluindo o aprendizado, se necessário. A regra é *"tudo acontece dentro da sprint, nada acontece fora da sprint"* [178]. Isso reforça a ideia de que não existe vácuos entre *sprints* e que, após uma *sprint*, imediatamente vem outra *sprint* [175]. A condução de cada *sprint* deve ser pautada em metas (*Spring Goal*) estabelecidas sobre o que deve ser construído e alcançado [175]. Dimensionar o trabalho em múltiplas *sprints* contribui para que as tarefas sejam diluídas em pequenas e constantes entregas, evitando que o desenvolvedor incorra na prática BDUF (*Big Design Up Front*). Isso é interessante, pois muitas vezes o sistema monolítico alvo não é totalmente conhecido e não se consegue estimar todos esforços para completar a migração. Utilizando-se de *sprints*, portanto, é possível realizar as atividades de modo gradual e incremental "rodando" tantas *sprints* quantas forem necessárias.

A dinâmica da simulação realizada pode ser melhor compreendida pela Figura 6.3 que mostra a relação entre DSRM, MFC, Scrum e o processo de migração. Basicamente, dentro da etapa de "*demonstração*" prevista no DSRM, os pacotes de trabalho da WBS/MFC alimentam o *backlog* do Scrum. À medida que as *sprints* ocorrem ao longo do tempo, o monolítico legado vai sendo suprimido pelos microsserviços que aumentam em número, e assim a migração é processada.

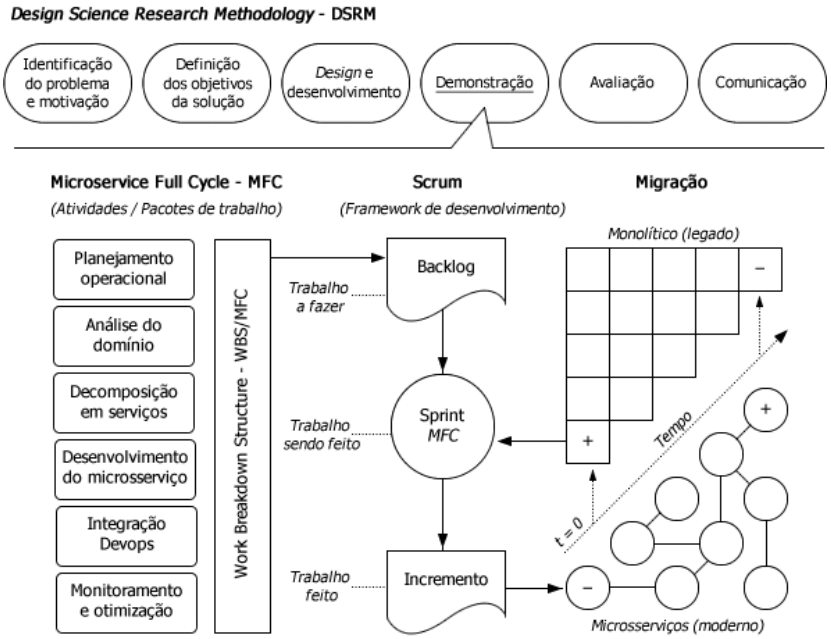


Figura 6.3: Relação entre DSRM, MFC, *Scrum* e migração.
(Fonte: própria)

Seguindo essa sistemática, foram necessárias 10 *sprints* para executar por completo a migração inicialmente prevista, sendo o *time-box* de cada *sprint* de 2 semanas e carga-horária de 6 horas/dia. Assim, *sprint* após *sprint*, as atividades do MFC foram consumidas gradualmente atendendo as metas das *sprints* até que o processo de migração fosse consolidado. **Detalhes técnicos de implementação estão descritos no Apêndice B** e mostraram que, de fato, o MFC conduziu o processo de migração, gerando um novo produto baseado em microsserviços. Resta saber, de que forma os resultados contribuem para a solução do problema do SISP, o que é feito em uma avaliação na próxima seção.

6.2 Avaliação

Segundo o DSRM utilizado neste estudo, a Avaliação é a etapa destinada a mensurar quanto o artefato atende à solução do problema. Entende-se por "artefato", o método *Microservice Full Cycle* - MFC e por "problema", os desafios descritos na Seção 1.1. Para

responder a questão de pesquisa e determinar se a transformação de sistemas monolíticos em microsserviços por meio do método MFC é capaz de mitigar possíveis problemas encontrados em sistemas legados, é preciso analisar as evidências que as métricas sustentam. Portanto, parte-se da premissa de que se as métricas coletadas forem favoráveis à arquitetura de microsserviços, o método MFC terá alcançado seu objetivo. A avaliação é feita estritamente em função da simulação demonstrada no Apêndice B e compara aspectos entre o monolítico (ML) e os microsserviços (MS), especialmente quanto à manutenibilidade, desempenho e interoperabilidade. Cabe ressaltar que os resultados e constatações obtidos na avaliação podem não refletir outras realidades dependendo do contexto.

6.2.1 Quanto à manutenibilidade

Comparar arquiteturas diferentes não é uma tarefa simples, a começar pela necessidade de isolar a *feature* que está sendo avaliada. No monolítico (ML) isso significa extrair de um todo, parte do código-fonte que realiza determinada funcionalidade. Em microsserviços (MS), ao contrário, o código de todos os serviços envolvidos nessa mesma funcionalidade devem estar agrupados. Feito isto, foi utilizado o SonarQube para individualmente proceder a análise estática dos referidos códigos (ML *vs* MS), possibilitando algumas inferências, ainda que em pequena escala. Por padrão o SonarQube exclui da análise arquivos que pertencem ao framework (*/vendor/**). Os resultados estão computados na Tabela 6.1.

Tabela 6.1: Análise estática - SonarQube.

#	MÉTRICA	ML	MS
1	Linhas de código	3047	2558
2	Quantidade de classes	14	87
3	Quantidade de funções	174	206
4	Taxa de comentários	26.3%	30.2%
5	Complexidade ciclomática	559	298
6	Complexidade cognitiva	756	131
7	Code smells	378	54
8	Taxa de dívida técnica	2.9%	1.1%
9	Número de Vulnerabilidades	0	0
10	Problemas de segurança	1	0
11	Número de bugs	1	0
12	Duplicidade de código	2.2%	41.4%
ML=Monolítico / MS=Microsserviço			

Os indicadores mostram que os microsserviços, embora apresentem menos linhas de código (*linha #1*), tiveram mais classes (*linha #2*) que o monolítico para realizar a mesma *feature*. Isso pode evidenciar uma melhor divisão de responsabilidade (coesão) ou um excesso de granularidade fina (*fine-grained*), ou seja, serviços com poucas operações. Em todo o caso, menos código concentrados em classes especializadas pode significar maior

legibilidade e facilidade de manutenção, o que também faz com que os microsserviços tenham apresentado menor complexidade de código (*linhas #5 e #6*). Essa constatação confirma o que outros autores [69] defendem, inclusive, reportando a manutenibilidade como o benefício mais importante alcançado pela adoção de microsserviços [2]. Os números mostram, ainda, que os microsserviços avaliados tendem a ser mais confiáveis, com menores índices de *code smells*, dívida técnica e *bugs* (*linhas #7 a #11*). Por outro lado, como nítida desvantagem, microsserviços apresentam excessiva duplicidade de código (*linha #12*). Isso, porém, pode ser justificado ao passo que cada serviço tem sua própria base de código, a fim de evitar acoplamento e garantir sua independência compartilhando o mínimo de código possível.

6.2.2 Quanto ao desempenho

O desempenho de uma aplicação pode ser verificado sob vários aspectos, como *throughput*, latência, *missed events*, dentre outros. Porém, uma importante limitação deste trabalho está na impossibilidade de se realizar testes de desempenho no monolítico SISP, por questões internas da Organização (EB). De qualquer modo, os microsserviços puderam ser testados sem depender do monolítico, o que foi feito por meio de suas APIs. Para isto, foi utilizado o Apache jMeter[2] (Figura 6.4) para simular as requisições de múltiplos usuários perfazendo o caso de uso UC001 descrito na simulação do Apêndice B.

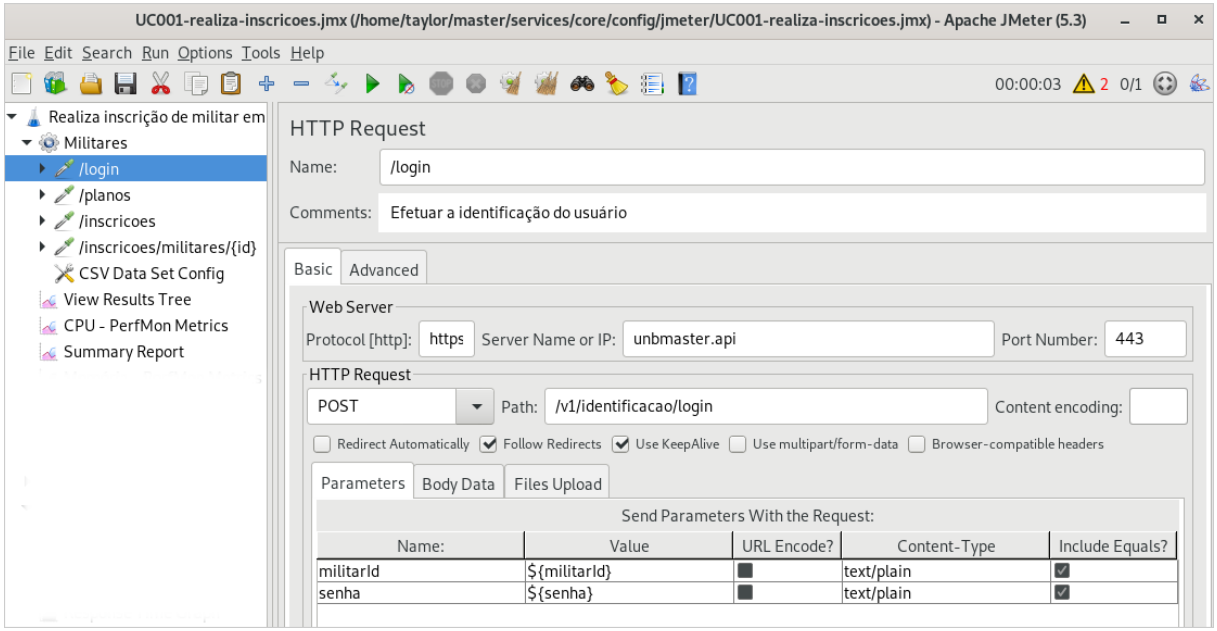


Figura 6.4: Configuração jMeter.
(Fonte: própria)

[2]https://jmeter.apache.org/

O teste permitiu simular usuários concorrentes realizando consecutivas operações de (i) efetuar login, (ii) visualizar os planos (cursos/estágios), (iii) realizar inscrição em um curso selecionado; e (iv) consultar as inscrições realizadas, além de (v) receber e-mail de inscrição realizada. Para todos efeitos, o resultado é considerado satisfatório se os microsserviços suportarem ao menos 5 mil usuários únicos e simultâneos do público alvo do SISP. O teste é realizado em 5 fases, incrementando-se amostras de 1000 até chegar aos 5 mil pretendidos. Sempre que o número de erros (*missed events*) for superior a 0.1%, o teste seguinte não deve ser feito sem antes escalar o microsserviço sobrecarregado (coluna "Instâncias" da tabela de Teste). Cada teste é realizado no tempo mínimo (*ramp-up*) de 60 segundos. A operação simulada (UC001) possui 4 chamadas diretas a microsserviços e ao menos outras 4 internas, não computadas pelo jMeter como acesso ao Redis, PostgreSQL e RabbitMQ. Para melhor acurácia os testes são repetidos 3 vezes subsequentes e a média dos resultados computada nas Tabelas 6.2, 6.3, 6.4, 6.5 e 6.6. Também são verificados em cada teste, em tempo real, o consumo de memória e processador por *host*.

Tabela 6.2: Teste 1 - Análise de desempenho - jMeter.

Teste #1	Tempo 60s	Amostra 1000	Chamadas 4(+4)	Repetições 3	Total 12000
Recurso	Instâncias	Bytes	Latência	Throughput	Erros %
/login	1	966.7	77ms	10.4/s	0%
/planos	1	1952.7	16ms	10.4/s	0%
/inscricoes	1	679.6	187ms	10.4/s	0%
/inscricoes/militares/{id}	1	2150.2	133ms	10.4/s	0%
Total/Média		1437.3	103ms	41.6/s	0%

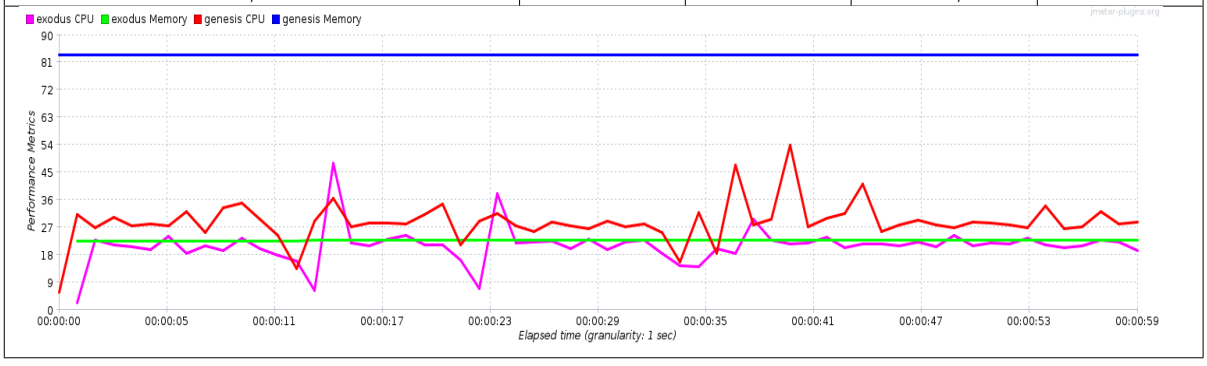


Tabela 6.3: Teste 2 - Análise de desempenho - jMeter.

Teste #2	Tempo 60s	Amostra 2000	Chamadas 4(+4)	Repetições 3	Total 24000
Recurso	Instâncias	Bytes	Latência	Throughput	Erros %
/login	1	966.1	107ms	19.5/s	0%
/planos	1	1952.4	20ms	19.5/s	0%
/inscricoes	1	679.3	455ms	19.5/s	0%
/inscricoes/militares/{id}	1	2098.7	340ms	19.5/s	0%
Total/Média		1424.1	231ms	77.8/s	0%

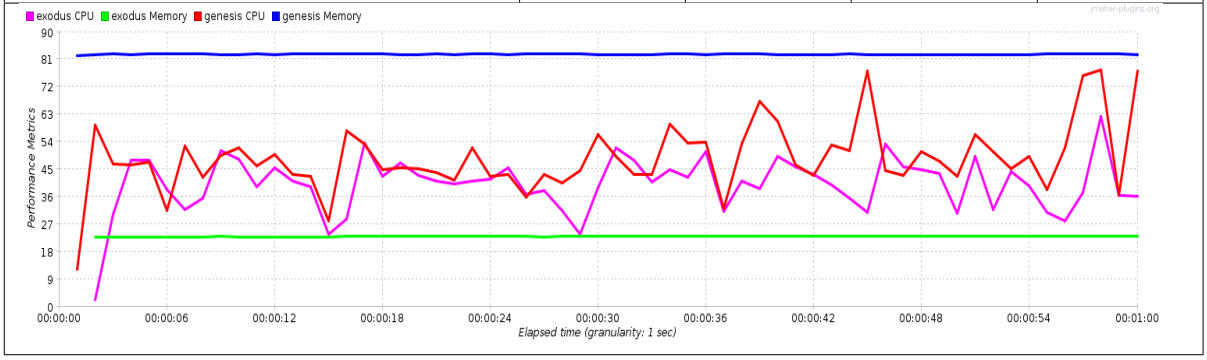


Tabela 6.4: Teste 3 - Análise de desempenho - jMeter.

Teste #3	Tempo 60s	Amostra 3000	Chamadas 4(+4)	Repetições 3	Total 36000
Recurso	Instâncias	Bytes	Latência	Throughput	Erros %
/login	1	965.9	72ms	23.2/s	0%
/planos	1	1952.3	17ms	23.2/s	0%
/inscricoes	1	689.2	1019ms	23.2/s	0.44%
/inscricoes/militares/{id}	1	1076.3	726ms	23.2/s	0.27%
Total/Média		1170.9	459ms	92.7/s	0.18%

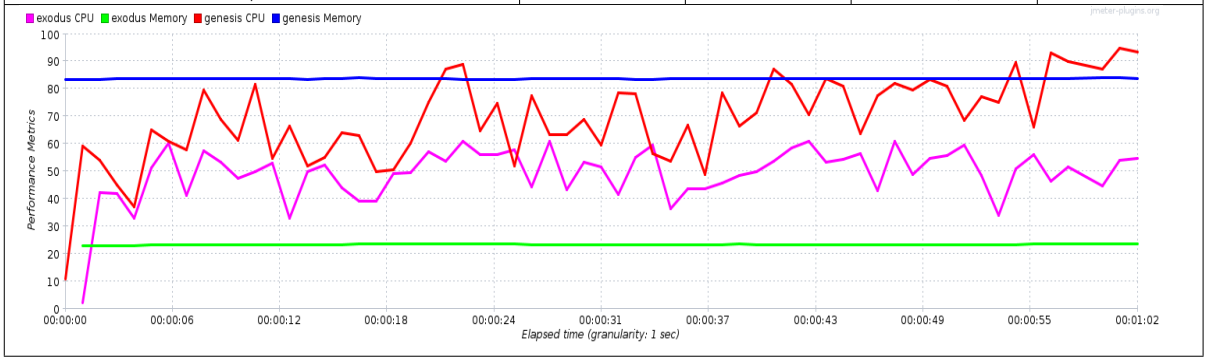
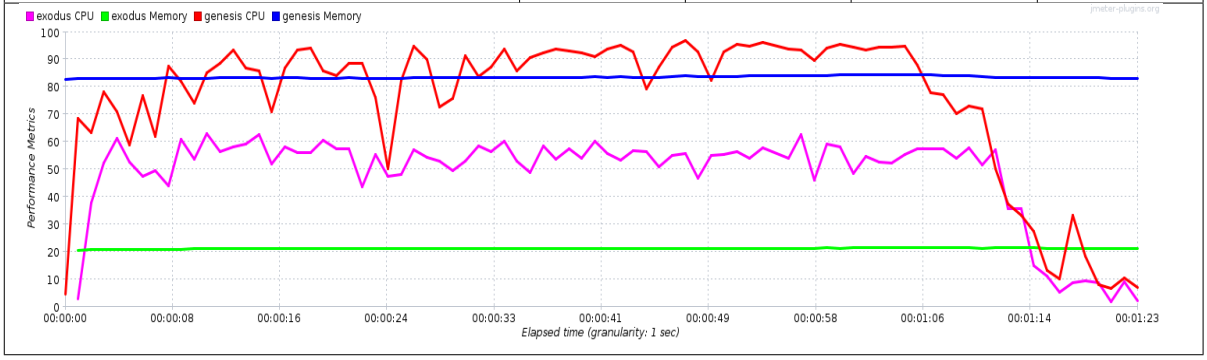


Tabela 6.5: Teste 4 - Análise de desempenho - jMeter.

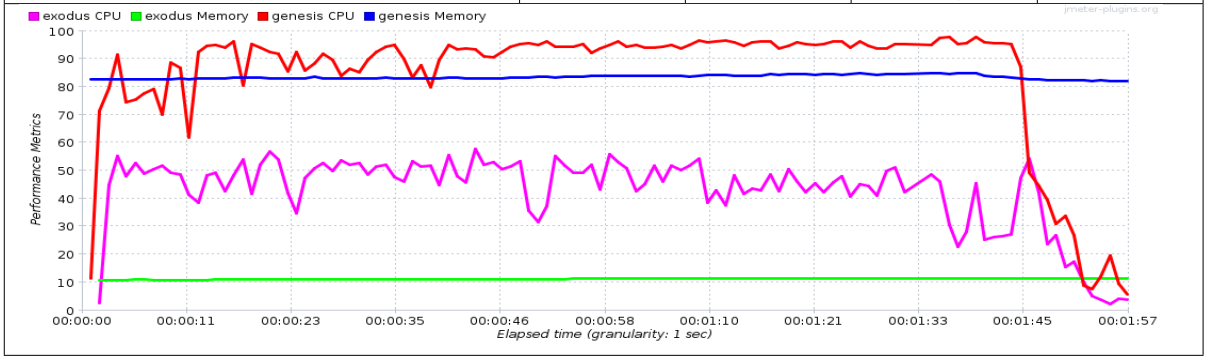
Teste #4	Tempo 60s	Amostra 4000	Chamadas 4(+4)	Repetições 3	Total 48000
Recurso	Instâncias	Bytes	Latência	Throughput	Erros %
/login	1	966	128ms	18.5/s	0%
/planos	1	1952.4	28ms	18.5/s	0%
/inscricoes	2	674.4	3033ms	17.7/s	8.21%
/inscricoes/militares/{id}	2	2144.9	2338ms	17.7/s	5.96%
Total/Média		1434.4	1382ms	70.9/s	3.54%



[*] Métricas[3] de CPU e memória foram obtidas durante a primeira execução de cada teste.

Tabela 6.6: Teste 5 - Análise de desempenho - jMeter.

Teste #5	Tempo 60s	Amostra 5000	Chamadas 4(+4)	Repetições 3	Total 60000
Recurso	Instâncias	Bytes	Latência	Throughput	Erros %
/login	1	966.2	169ms	11.5/s	0%
/planos	1	1952.4	24ms	11.5/s	0%
/inscricoes	4	685.3	194ms	11.5/s	0%
/inscricoes/militares/{id}	4	1579.3	187ms	11.5/s	0%
Total/Média		1295.8	144ms	46.0/s	0%



[3]https://jmeter-plugins.org/wiki/PerfMon/

Também se constatou durante os testes uma propriedade comum aos microsserviços em que, por meio da escalabilidade, demandas colapsadas pela sobrecarga são restabelecidas. Isso é demonstrado na Figura 6.5, a partir de dados coletados do jMeter (linha Total/Média). Contudo, comparar dados com diferentes unidades de medidas requer uma prévia normalização. Nesse sentido, foi utilizado a fórmula a seguir para manter todos os valores dentro do intervalo de 0 e 1 (Eixo Y), onde "x" é o valor a ser normalizado:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

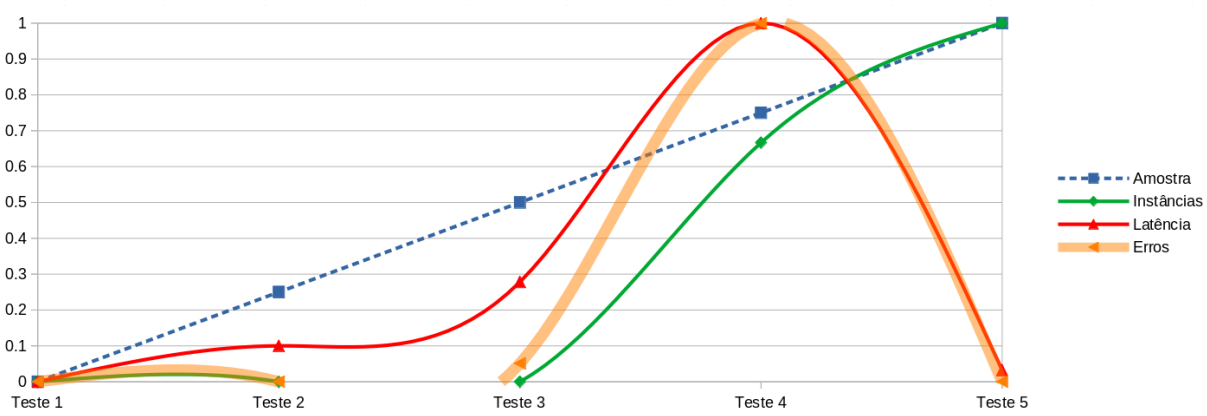


Figura 6.5: Teste de carga em microsserviços.
(Fonte: própria)

Conforme mostra a Figura 6.5, à medida que as requisições aumentam [linha azul], crescem também o tempo de resposta (latência) [linha vermelha] e a taxa de erros (*missed events*) [linha laranja]. Em geral, esses erros estão associados ao *status code* "504 *gateway timeout*" ou "502 *bad gateway*" quando servidor fica sobrecarregado. Muito provavelmente um monolítico, dado sua dificuldade de escalar, ficaria fora do ar nesse momento (Teste 4). Nota-se, porém, que ao escalar o microsserviço que está sob alta demanda em múltiplas instâncias [linha verde], há uma brusca redução da latência, cessando as falhas. Monolíticos também podem ser escaláveis, mas não em específicos gargalos como feito nesta simulação ao se criar 4 instâncias de Inscrição *Service*, utilizando o simples comando `docker service scale inscricao-prd=4`. Além disto, a inicialização de microsserviços embarcados em *containers* Docker é extremamente rápida, ocorre em poucos segundos. Em um último teste, foram feitas chamadas diretas a um microsserviço individualmente, apresentando resultados ainda melhores, como mostra a Tabela 6.7.

Tabela 6.7: Teste de desempenho individual - jMeter.

Recurso	Amostra	Bytes	Latência	Throughput	Erros %	Duração
POST /login	5000	966.2	60ms	160.3/s	0%	32s
GET /planos	5000	1962.0	54ms	168.6/s	0%	30s
POST /inscricoes	5000	665.9	185ms	120.9/s	0%	41s

Em termos de desempenho, os resultados foram satisfatórios (Tabela 6.8). Testes com 5 mil usuários simultâneos se mostraram estáveis, isto é, zero erros e latência média inferior a 100 milissegundos. O objetivo, portanto, foi alcançado e bem-sucedido, o que faz da arquitetura de microsserviços uma possível estratégia para aumentar o desempenho do monolítico SISP, atualmente enfrentando recorrentes episódios de lentidão e inacessibilidade quando sob alta demanda. Entretanto, a despeito das vantagens que microsserviços possam ter, cabe ressaltar que o desempenho em ambientes heterogêneos pode variar consideravelmente devido a interferências [114] e *overhead* de comunicação via rede [40], enquanto monolíticos se beneficiam de chamadas locais [16].

Tabela 6.8: Desempenho médio com 5 mil requisições.

#	MÉTRICA	ML	MS
1	*Throughput*	-	149/s
2	Latência	-	99ms
3	Erros (*missed events*)	-	0%
4	Consumo de memória	-	50%
5	Consumo de CPU	-	70%
ML=Monolítico / MS=Microsserviço			

[*] Teste de desempenho não realizado no monolítico (ML) devido às restrições da organização

6.2.3 Quanto a interoperabilidade

Interoperabilidade pode ter diferentes conotações. No que tange ao estudo, interoperabilidade é a capacidade de sistemas distintos se comunicarem utilizando uma linguagem comum. Por isto a importância de adotar padrões abertos. Havendo interoperabilidade, a organização está apta a integrar sistemas com tecnologias heterogêneas através de protocolos compartilhados. O SISP não oferece qualquer suporte à comunicação com outros sistemas, o que fez crescer o número de aplicações satélites e trâmites burocráticos. Por outro lado, os microsserviços operam por meio de interfaces (API) que independem de linguagem de programação ou tecnologias, favorecendo a interoperabilidade e a integração com distintos sistemas de software, inclusive de forma distribuída. A Tabela 6.9 faz uma breve comparação entre o o monolítico SISP (ML) e os microsserviços (MS) desenvolvidos.

Uma vantagem observada é que, à medida que os microsserviços entram em operação, o monolítico ganha capacidades interoperáveis, passando a interagir com outras aplicações.

Tabela 6.9: Interoperabilidade.

#	MÉTRICA	ML	MS	OBSERVAÇÃO
1	Descoberta e consulta do serviço	Não	Sim	Metadados da API
2	Interfaces expostas (API)	0	4	-
3	Padrão de formato de dados aberto	Não	Sim	JSON
4	Padrão de codificação de caracteres	Sim	Sim	ISO-8859-1 e UTF-8
ML=Monolítico / MS=Microsserviço				

Em resumo, o estudo mostrou que o método *Microservice Full Cycle* - MFC, por meio de suas etapas e atividades sistemáticas, pode contribuir para que monolíticos legados experimentem benefícios da arquitetura de microsserviços. Dentre as vantagens estão códigos mais coesos e desacoplados, pequenas equipes, independência tecnológica, agilidade em *build*, teste e *deploy* (automação), escalabilidade sob alta demanda, maior interoperabilidade e integração, além de monitoramento e *feedback* em tempo real. Contudo, a recomendação é para que a organização avalie a necessidade de migração do monolítico legado, tendo a motivação correta para lidar com a complexidade dos microsserviços.

Capítulo 7

Considerações Finais

Este último capítulo é destinado à síntese, conclusão e recomendações para futuros trabalhos. A escolha do estilo arquitetural é uma decisão importante e tem um impacto em vários atributos de qualidade de software [179]. Na realidade, não existe melhor ou pior arquitetura, mas sim a mais adequada a determinados contextos [43]. A arquitetura monolítica, por exemplo, é a mais utilizada e costuma ser aquela em que todo código-fonte fica em único servidor [68, 69], sendo melhor adaptável a simples aplicações. Microsserviços, ao contrário, tende ser a melhor escolha para aplicações complexas e em evolução [43, 149]. O problema é que muitos sistemas monolíticos legados ficaram complexos e estão em processo de deterioração, precisando de uma alternativa para se manterem operando. Uma possível solução pode estar na arquitetura de microsserviços. Porém, migrar entre arquiteturas não é uma tarefa trivial. O mérito do presente trabalho está em prover um método que possa orientar este processo de transformação.

Monolítico. Monolítico não necessariamente é sinônimo de legado, mas uma escolha arquitetural que pode ser boa dependendo do cenário [16]. Monolíticos se caracterizam por ser um bloco único de *deploy* que engloba funcionalidades pertencentes a diferentes subdomínios [45]. Muitas vezes, são sistemas críticos e de grande valor, na qual a organização é altamente dependente. Os transtornos com monolíticos começam à medida que as funcionalidades aumentam em tamanho e complexidade [154], muito em razão da incapacidade de lidar com as mudanças sistêmicas ao longo dos anos [38]. Com os crescentes problemas de manutenibilidade e escalabilidade [154], surge a necessidade de modernização do monolítico. Contudo, reescrever por completo o monolítico envolve custos e prazos que tornam a reescrita inviável [45]. Como alternativa, microsserviços podem, aos poucos, emancipar partes do legado com mínimo *downtime* e transtorno ao usuário final.

Microsserviços. A arquitetura de microsserviços oferece muitas vantagens [43] e é recomendada por vários pesquisadores para superar as limitações e problemas encontrados no

monolítico [28]. Microsserviços podem ser descritos como uma composição de pequenas unidades funcionais autônomas, executando seu próprio processo e comunicando-se através da troca de mensagens [79, 82]. Por ser ainda uma arquitetura relativamente recente, existe um *gap* de pesquisa e a necessidade de um estudo que faça o mapeamento sistemático do progresso de migração [88]. Para minimizar esse *gap*, o presente estudo tem sido idealizado com a proposta de elaborar um método intitulado *Microservice Full Cycle* - MFC, provendo um conjunto de etapas e atividades que oriente a pretendida migração.

MFC. O método *Microservice Full Cycle* - MFC, apresentado no Capítulo 5, é constituído de seis etapas alinhadas ao ciclo de vida do desenvolvimento de software e a estratégias DevOps. O método foi elaborado a partir de um arcabouço conceitual provido pelos fundamentos da arquitetura de microsserviços (Capítulo 2) e da experiência de diversos autores que passaram pelo processo de migração (Capítulo 3). Como resultado, o MFC tem coberto áreas do processo migração que vão desde o planejamento até o monitoramento do microsserviço.

Metodologia. O trabalho desenvolvido foi conduzido em conformidade às fases do Método de Pesquisa em *Design Science* - DSRM, cujo propósito é fornecer um "*modelo da realidade em pequena escala*" [163]. Na 1ª fase do DSRM, foi identificado a motivação e o problema de pesquisa (Seção 1.1), que é dado pela necessidade de modernização do legado e a falta de um método de migração. Na 2ª fase, foram definidos os objetivos da solução (Seção 1.4), ou seja, a perspectiva de elaboração do método MFC e a realização de uma simulação para validá-lo. A 3ª fase diz respeito ao *design* e desenvolvimento do artefato (Capítulo 5), momento em que o MFC, de fato, é concebido. Na 4ª fase é feita uma demonstração para atestar que o artefato funciona (Seção 6.1). Para isto, foi realizado a simulação baseada no sistema legado SISP colocando o MFC à prova. Essa etapa demonstrou na prática o emprego do MFC. Na 5ª fase do DSRM foi feita uma avaliação para mensurar quão bem o artefato suporta a solução do problema (Seção 6.2). Neste caso, foram utilizadas métricas para avaliar a migração, comparando o monolítico legado aos novos microsserviços. Por fim, a 6ª fase trata da comunicação dos resultados, que é feito por meio da publicação desta dissertação e artigos científicos resultantes.

Resultados. Na prática, o estudo foi motivado pela necessidade de modernização do SISP, um sistema de software legado real que apresenta difícil manutenção, constante lentidão, inacessibilidade mediante alta demanda e falta de integração com outros sistemas. Como possível solução, o método MFC foi aplicado ao legado SISP. Mediante ao que foi executado e mensurado, pode-se dizer que a arquitetura de microsserviços trouxe, em geral, resultados satisfatórios em lidar com os problemas do legado, sob três aspectos:

a) Quanto à manutenibilidade. Os novos microsserviços levaram vantagens em relação ao monolítico SISP. Indicadores do *SonarQube* mostraram que o código ficou melhor segmentado dentro de uma responsabilidade única (coesão). A análise estática também revelou que os microsserviços diminuíram a complexidade e se tornaram mais confiáveis, com menores índices de *code smells*, dívida técnica e *bugs*, o que facilita a legibilidade e entendimento do código por parte do desenvolvedor e favorece a manutenção. O fator negativo é que para manter a independência do serviço (desacoplamento), códigos comuns não foram compartilhados, mas replicados em cada instância do serviço, aumentando em muito o percentual de duplicidade de código. Isto porém pode ser contornado transformando tais códigos compartilhados em novos serviços, neste caso, implicando aumento do número de chamadas entre serviços e *overhead* de comunicação via rede. É, portanto, uma questão de *tradeoff*.

b) Quanto ao desempenho. Microsserviços se mostraram uma boa estratégia para resolver problemas de lentidão e inacessibilidade do sistemas legado quando estão sob alta demanda. Embora testes de carga não tenham sido autorizados no monolítico SISP (limitação da pesquisa), constatou-se que os microsserviços responderam bem a grandes número de requisições. Testes com 5 mil usuários simultâneos se mostraram estáveis, isto é, chegando a zero erros e latência média inferior a 100 milissegundos. Mesmo a 40 mil requisições (ou mais), esses índices se mantiveram estáveis graças ao recurso da escalabilidade, tornando possível tratar gargalos específicos ao criar novas instâncias do serviço demandado e não de toda a aplicação como seria preciso em um monolítico. Práticas *DevOps* também deram agilidade com a automação de *build*, teste e *deploy*, além do monitoramento e *feedback* em tempo real, elementos importantíssimos inexistentes no SISP. Por haver poucos microsserviços distribuídos em rede local, não foi perceptível problemas de interferências [114] e *overhead* de comunicação via rede [40], o que pode vir a ocorrer em cenários com o crescente números de microsserviços trocando mensagens.

c) Quanto a interoperabilidade. O SISP não "conversa" com nenhum outro sistema, quer seja dentro ou fora da organização. Portanto, era esperado que neste quesito microsserviços teriam vantagens. À medida que os microsserviços eram postos em produção, não apenas o monolítico SISP ganhava a capacidade de consumir esses serviços, mas qualquer outra aplicação. Microsserviços representarem nichos de negócios (domínios) e essa interoperabilidade provida facilita o processo de integração e expansão contínua dos sistemas de software organizacionais que agora podem ser pensados e planejados à nível corporativo e não setorialmente, evitando que cada departamento seja detentor exclusivo do dado e burocratize o fluxo de informações que deveriam permear os processos organizacionais. Microsserviços também podem operar de modo geo-distribuído e com maior resiliência,

aspectos interessantes para o Exército Brasileiro que precisa manter os sistemas operando de modo confiável em todo o território nacional.

Conclusão. De acordo com o exposto, é possível afirmar que os objetivos de pesquisa foram alcançados. Os resultados mostraram uma resposta positiva à questão de pesquisa, isto é, o método *Microservice Full Cycle* - MFC contribuiu a contento para orientar o processo de migração, demonstrando que o legado SISP pode encontrar na arquitetura de microsserviços perspectivas de modernização, especialmente quanto à manutenibilidade, desempenho e interoperabilidade. Também foram observados potenciais ganhos em outros atributos de qualidade de software, como em confiabilidade (tolerância a falhas e recuperabilidade), embora não mensurados.

O estudo pode ser plenamente reproduzido (reprodutibilidade) em relação aos métodos empregados, porém, os resultados estão passíveis de variações a depender de equipamentos utilizados, redes de computadores, implementações de código e processos simultâneos executando em memória. Além disto, por ser um relato de migração em estágio inicial, existe a possibilidade de que complicações não observadas ocorram em cenários de maior complexidade. Cada organização deve, portanto, construir seu próprio "*modelo da realidade em pequena escala*", experimentar e decidir sobre a adoção de microsserviços.

É fato, porém, que nem toda corporação está preparada ou disposta a aceitar o aumento de complexidade inerente aos microsserviços [43, 154]. Existem organizações que entram nessa jornada pela motivação errada e acabam frustradas. Por isto, recomenda-se não adotar microsserviços (i) pela sua popularidade [43]; (ii) porque deu certo com a Netflix [16]; (iii) se o monolítico atende [73]; (iv) se não vai resolver o seu problema [16]; (v) sem uma equipe capacitada; e (vi) sem convencimento e aval da alta direção. Antes de empregar o MFC e optar pela arquitetura de microsserviços, a organização deve ponderar e capturar o "por quê" da migração, pois nem todo monolítico requer mudanças. Neste sentido, o "*Design Rationale*" pode contribuir em explicitar as razões e justificativas por trás de uma decisão, conjecturando inclusive outras alternativas e *tradeoffs*. O fato é que microsserviços não resolvem todos os problemas, mas em muitos casos, podem ser uma solução viável.

Por fim, cabe lembrar que a arquitetura de microsserviços está em plena expansão. Novos conceitos, técnicas e ferramentas mudam ou surgem a todo momento. Microsserviço é um jeito diferente de desenvolver software [31], onde a codificação passa a ser apenas um dos aspectos. É a partir desse novo *mindset* que o presente trabalho tem impactado a organização (DISP). Por meio da classificação em etapas e atividades, o MFC tem clarificado e inspirado a ideia da migração em sistemas legados, de modo que já se começa a discutir e implementar na prática microsserviços de suporte à aplicação principal (SISP), tais como serviços de relatórios (*iReport*), notificação por e-email/SMS e autenticação com

token JWT, o que tem sido uma boa maneira de introduzir, aprender e ganhar confiança, antes de lidar com a decomposição de monolíticos. Além disto, existe em andamento um estudo avaliando o uso de *containers* em detrimento às inúmeras máquinas virtuais (VM), o que irá facilitar o processo de automatização da integração e entrega contínua (CI/CD) e a otimização de recursos computacionais. O MFC trouxe à organização, perspectivas para que o SISP e outros legados possam evoluir metodicamente, de forma gradual, alcançando a médio e longo prazo a modernização almejada. O grande desafio, porém, ainda são pessoas, não tecnologias. O receio do novo, a resistência às mudanças e a inexperiência de desenvolvedores devem ser enfrentados com informação e capacitação, pois em termos estatísticos, microsserviços já provaram seu valor quando em um contexto adequado.

7.1 Limitações e trabalhos futuros

- Microsserviços abrangem diferentes aspectos conceituais e técnicos, portanto, uma série de abordagens que podem potencializar suas vantagens, mas que não foram cobertos em profundidade neste trabalho, como padrões CQRS, *Event Sourcing* e princípios da Arquitetura Hexagonal, entre outros.

- O trabalho transcorreu em ambiente *on-premises* (local). Seria recomendável experimentar os resultados em nuvens públicas, tais como AWS[1], Google Cloud[2] e Azure[3]. Não usufruir da solução deste *pool* de recursos virtuais implica perdas de investimentos com infraestrutura e das vantagens que a nuvem oferece. Por outro lado, algumas organizações não podem rodar seus sistemas em nuvem pública (questões legais); usar *clusters* locais significa gastar menos com provedor de nuvem.

- Todo o estudo ficou restrito ao uso de API dos microsserviços e não explorou a manipulação de *views* e *templates*. A sugestão é que isto, porém, seja uma interessante abordagem a ser pormenorizada em trabalhos futuros.

- Para orquestrar os *containers*, o estudo utilizou o Docker no modo *Swarm*. Embora o *Swarm* seja mais simples de entender e rápido em levantar novos *containers*, para *clusters* maiores seria oportuno fazer uma análise com *Kubernetes*, sobretudo, por dispor de recursos de escalonamento automático.

[1]https://aws.amazon.com/
[2]https://cloud.google.com/
[3]https://azure.microsoft.com/

Referências

[1] Balalaie, Armin, Abbas Heydarnoori e Pooyan Jamshidi: *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture*. IEEE Software, 33(3):42–52, maio 2016, ISSN 0740-7459, 1937-4194. `https://ieeexplore.ieee.org/document/7436659/`, acesso em 2019-05-24. viii, xiii, 1, 3, 4, 6, 13, 15, 16, 20, 22, 23, 24, 25, 26, 27, 28, 32, 42, 43, 46, 47, 48, 55, 64, 103, 118

[2] Taibi, Davide, Valentina Lenarduzzi e Claus Pahl: *Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation*. IEEE Cloud Computing, 4(5):22–32, setembro 2017, ISSN 2325-6095. `http://ieeexplore.ieee.org/document/8125558/`, acesso em 2019-02-25. viii, xiii, 1, 2, 4, 6, 12, 14, 15, 16, 17, 18, 27, 30, 32, 33, 42, 43, 46, 47, 49, 50, 55, 63, 72, 103, 114

[3] Fan, Chen Yuan e Shang Pin Ma: *Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report*. Em *2017 IEEE International Conference on AI & Mobile Services (AIMS)*, páginas 109–112, Honolulu, HI, USA, junho 2017. IEEE, ISBN 978-1-5386-1999-5. `http://ieeexplore.ieee.org/document/8027278/`, acesso em 2019-02-25. viii, xiii, 1, 2, 3, 4, 6, 7, 12, 16, 21, 27, 34, 46, 50, 51, 54, 55, 63, 103, 118, 122

[4] Ghani, Ahmad Tarmizi Abdul e Mohamad Shanudin: *Method for Designing Scalable Microservice-based Application Systematically: A Case Study*. International Journal of Advanced Computer Science and Applications, 9(8), 2018, ISSN 21565570, 2158107X. `http://thesai.org/Publications/ViewPaper?Volume=9&Issue=8&Code=ijacsa&SerialNo=17`, acesso em 2019-02-25. viii, xiii, 4, 6, 7, 13, 17, 46, 47, 51, 52, 55, 56, 60, 103

[5] Mishra, Mayank, Shruti Kunde e Manoj Nambiar: *Cracking the monolith: challenges in data transitioning to cloud native architectures*. Em *Proceedings of the 12th European Conference on Software Architecture Companion Proceedings - ECSA '18*, páginas 1–4, Madrid, Spain, 2018. ACM Press, ISBN 978-1-4503-6483-6. `http://dl.acm.org/citation.cfm?doid=3241403.3241440`, acesso em 2020-02-06. viii, xiii, 4, 30, 42, 46, 52, 53, 55, 103

[6] Silva, Hugo S. da, Glauco Carneiro e Miguel Monteiro: *Towards a Roadmap for the Migration of Legacy Software Systems to a Microservice based Architecture:*. Em *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, páginas 37–47, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications, ISBN 978-989-758-365-0. `http://www.scitepress.org/`

DigitalLibrary/Link.aspx?doi=10.5220/0007618400370047, acesso em 2019-09-09. viii, xiii, 4, 12, 19, 20, 26, 30, 32, 33, 42, 43, 46, 53, 54, 55, 63, 103, 118

[7] Richardson, Chris: *Microservices Pattern: Database per service*, 2018. http://microservices.io/patterns/data/database-per-service.html, acesso em 2019-05-13. xii, 118, 146, 148

[8] Lewis, James e Martin Fowler: *Microservices - a definition of this new architectural term*, 2014. https://martinfowler.com/articles/microservices.html, acesso em 2019-05-13. 1, 2, 13, 26, 43, 47

[9] Villamizar, Mario, Oscar Garces, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas e Santiago Gil: *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.* Em *2015 10th Computing Colombian Conference (10CCC)*, páginas 583–590, Bogota, Colombia, setembro 2015. IEEE, ISBN 978-1-4673-9464-2. http://ieeexplore.ieee.org/document/7333476/, acesso em 2019-01-20. 1

[10] Ihde, Steven e Karan Parikh: *From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture*, 2014. https://www.infoq.com/presentations/linkedin-microservices-urn/, acesso em 2019-06-06. 1

[11] Ren, Zhongshan, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei e Tao Huang: *Migrating Web Applications from Monolithic Structure to Microservices Architecture.* Em *Proceedings of the Tenth Asia-Pacific Symposium on Internetware - Internetware '18*, páginas 1–10, Beijing, China, 2018. ACM Press, ISBN 978-1-4503-6590-1. http://dl.acm.org/citation.cfm?doid=3275219.3275230, acesso em 2019-05-29. 1, 12, 25, 31, 32, 42

[12] Chen, Rui, Shanshan Li e Zheng Li: *From Monolith to Microservices: A Dataflow-Driven Approach.* Em *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, páginas 466–475, Nanjing, dezembro 2017. IEEE, ISBN 978-1-5386-3681-7. http://ieeexplore.ieee.org/document/8305969/, acesso em 2019-02-25. 1, 6, 7, 13, 31, 63

[13] Eski, Sinan e Feza Buzluca: *An automatic extraction approach: transition to microservices architecture from monolithic application.* Em *Proceedings of the 19th International Conference on Agile Software Development Companion - XP '18*, páginas 1–6, Porto, Portugal, 2018. ACM Press, ISBN 978-1-4503-6422-5. http://dl.acm.org/citation.cfm?doid=3234152.3234195, acesso em 2019-05-07. 1, 2, 18

[14] Di Francesco, Paolo, Patricia Lago e Ivano Malavolta: *Architecting with microservices: A systematic mapping study.* Journal of Systems and Software, 150:77–97, abril 2019, ISSN 01641212. https://linkinghub.elsevier.com/retrieve/pii/S0164121219300019, acesso em 2019-05-16. 1, 2, 29, 42, 45

[15] Alshuqayran, Nuha, Nour Ali e Roger Evans: *Towards Micro Service Architecture Recovery: An Empirical Study*. Em *2018 IEEE International Conference on Software Architecture (ICSA)*, páginas 47–4709, Seattle, WA, abril 2018. IEEE, ISBN 978-1-5386-6398-1. https://ieeexplore.ieee.org/document/8417116/, acesso em 2019-06-08. 1, 13, 34, 42

[16] Newman, S.: *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019, ISBN 978-1-4920-4784-1. https://books.google.com.br/books?id=iul3wQEACAAJ. 1, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 30, 32, 33, 34, 36, 37, 39, 40, 41, 42, 43, 44, 62, 67, 77, 79, 82, 114, 117, 118, 120, 123

[17] Merson, Paulo: *Defining Microservices - Paulo Merson*, 2015. https://insights.sei.cmu.edu/sei_blog/2015/11/defining-microservices.html, acesso em 2019-02-16. 1

[18] Kazman, R., S.G. Woods e S.J. Carriere: *Requirements for integrating software architecture and reengineering models: CORUM II*. Em *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, páginas 154–163, Honolulu, HI, USA, 1998. IEEE Comput. Soc, ISBN 978-0-8186-8967-3. http://ieeexplore.ieee.org/document/723185/, acesso em 2019-11-13. 1

[19] Cojocaru, Michel Daniel, Ana Oprescu e Alexandru Uta: *Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications*. Em *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, páginas 84–93, Amsterdam, Netherlands, junho 2019. IEEE, ISBN 978-1-72813-801-5. https://ieeexplore.ieee.org/document/8790889/, acesso em 2019-09-09. 1, 2, 13, 16, 19, 20, 21, 22, 27, 29, 30, 32

[20] Bass, Len, Paul Clements e Rick Kazman: *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edição, 2012, ISBN 0-321-81573-4 978-0-321-81573-6. 1, 3, 5, 9, 119, 120

[21] Bonér, Jonas: *Reactive Microservices Architecture*. página 54, 2015. 1, 2, 18, 19

[22] Singh, Vindeep e Sateesh K Peddoju: *Container-based microservice architecture for cloud applications*. Em *2017 International Conference on Computing, Communication and Automation (ICCCA)*, páginas 847–852, Greater Noida, maio 2017. IEEE, ISBN 978-1-5090-6471-7. http://ieeexplore.ieee.org/document/8229914/, acesso em 2019-02-25. 1, 2, 17, 117

[23] Balalaie, Armin, Abbas Heydarnoori e Pooyan Jamshidi: *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*. Em Celesti, Antonio e Philipp Leitner (editores): *Advances in Service-Oriented and Cloud Computing*, volume 567, páginas 201–215. Springer International Publishing, Cham, 2016, ISBN 978-3-319-33312-0 978-3-319-33313-7. http://link.springer.com/10.1007/978-3-319-33313-7_15, acesso em 2019-05-29. 1, 6, 15, 17, 25, 26, 34, 36, 38, 39, 63, 117

[24] Richardson, Chris: *Microservices From Design to Deployment.* página 80, 2016. 1, 15

[25] Richardson, Chris: *What are microservices?*, 2019. `http://microservices.io/index.html`, acesso em 2019-06-08. 1, 30

[26] Di Francesco, Paolo, Patricia Lago e Ivano Malavolta: *Migrating Towards Microservice Architectures: An Industrial Survey.* Em *2018 IEEE International Conference on Software Architecture (ICSA)*, páginas 29–2909, Seattle, WA, abril 2018. IEEE, ISBN 978-1-5386-6398-1. `https://ieeexplore.ieee.org/document/8417114/`, acesso em 2019-02-25. 1, 2

[27] Thönes, Johannes: *SOFTWARE ENGINEERING.* SOFTWARE ENGINEERING, página 4, 2015. 1, 2

[28] Ahmed, Shahbaz, Abdul Razzaq, Saeed Ullah e Salman Ahmed: *Matrix Clustering based Migration of System Application to Microservices Architecture.* International Journal of Advanced Computer Science and Applications, 9(1), 2018, ISSN 21565570, 2158107X. `http://thesai.org/Publications/ViewPaper?Volume=9&Issue=1&Code=ijacsa&SerialNo=39`, acesso em 2019-05-29. 1, 2, 19, 42, 80

[29] Merson, Paulo: *Defining microservices.* `https://insights.sei.cmu.edu/sei_blog/2015/11/defining-microservices.html`, acesso em 2019-02-16. 2, 13

[30] Asik, Tugrul e Yunus Emre Selcuk: *Policy enforcement upon software based on microservice architecture.* Em *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, páginas 283–287, London, United Kingdom, junho 2017. IEEE, ISBN 978-1-5090-5756-6. `http://ieeexplore.ieee.org/document/7965739/`, acesso em 2019-02-25. 2, 13

[31] Christoforou, Andreas, Lambros Odysseos e Andreas Andreou: *Migration of Software Components to Microservices: Matching and Synthesis:.* Em *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, páginas 134–146, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications, ISBN 978-989-758-375-9. `http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0007732101340146`, acesso em 2019-09-09. 2, 12, 13, 16, 82

[32] Newman, Sam: *Building microservices: designing fine-grained systems.* O'Reilly Media, Beijing Sebastopol, CA, first edition edição, 2015, ISBN 978-1-4919-5035-7. OCLC: ocn881657228. 2, 16, 17, 18, 19, 20, 21, 29, 63, 64, 117

[33] Butzin, Bjorn, Frank Golatowski e Dirk Timmermann: *Microservices approach for the internet of things.* Em *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, páginas 1–6, Berlin, Germany, setembro 2016. IEEE, ISBN 978-1-5090-1314-2. `http://ieeexplore.ieee.org/document/7733707/`, acesso em 2019-05-29. 2, 13, 20, 22, 25, 27, 28, 39, 42, 47, 118

[34] Shimoda, Atsushi e Tsubasa Sunada: *Priority Order Determination Method for Extracting Services Stepwise from Monolithic System*. Em *2018 7th International Congress on Advanced Applied Informatics (IIAI-AAI)*, páginas 805–810, Yonago, Japan, julho 2018. IEEE, ISBN 978-1-5386-7447-5. `https://ieeexplore.ieee.org/document/8693253/`, acesso em 2019-05-29. 2, 10, 31

[35] Tabbaa, Bishr: *Anti-Patterns of Microservices*, junho 2019. `https://itnext.io/anti-patterns-of-microservices-6e802553bd46`, acesso em 2018-08-31. 2

[36] Vernon, Vaughn: *Domain-driven design distilled*. Addison-Wesley, Boston, 2016, ISBN 978-0-13-443442-1. OCLC: ocn935986578. 2, 63, 145

[37] Foote, Brian e Joseph Yoder: *Big Ball of Mud*, 1999. `http://www.laputan.org/mud/`, acesso em 2019-06-11. 2

[38] Pautasso, Cesare, Olaf Zimmermann, Mike Amundsen, James Lewis e Nicolai Josuttis: *Microservices in Practice, Part 2: Service Integration and Sustainability*. IEEE Software, 34(2):97–104, março 2017, ISSN 0740-7459. `http://ieeexplore.ieee.org/document/7888407/`, acesso em 2019-06-09. 2, 79

[39] Kecskemeti, Gabor, Attila Kertesz e Attila Csaba Marosi: *Towards a Methodology to Form Microservices from Monolithic Ones*. Springer Berlin Heidelberg, New York, NY, 2017, ISBN 978-3-319-58942-8. 2

[40] Abdullah, Muhammad, Waheed Iqbal e Abdelkarim Erradi: *Unsupervised learning approach for web application auto-decomposition into microservices*. Journal of Systems and Software, 151:243–257, maio 2019, ISSN 01641212. `https://linkinghub.elsevier.com/retrieve/pii/S0164121219300408`, acesso em 2019-05-29. 2, 29, 31, 77, 81, 118

[41] Luz, Welder, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto e Rodrigo Bonifácio: *An experience report on the adoption of microservices in three Brazilian government institutions*. Em *Proceedings of the XXXII Brazilian Symposium on Software Engineering - SBES '18*, páginas 32–41, Sao Carlos, Brazil, 2018. ACM Press, ISBN 978-1-4503-6503-1. `http://dl.acm.org/citation.cfm?doid=3266237.3266262`, acesso em 2019-02-25. 2, 3, 6, 9, 42

[42] Stevenson, Chris e Andy Pols: *An Agile Approach to a Legacy System*. Em Kanade, Takeo, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Jutta Eckstein e Hubert Baumeister (editores): *Extreme Programming and Agile Processes in Software Engineering*, volume 3092, páginas 123–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, ISBN 978-3-540-22137-1 978-3-540-24853-8. `http://link.springer.com/10.1007/978-3-540-24853-8_14`, acesso em 2019-01-19. 2

[43] Kazanavicius, Justas e Dalius Mazeika: *Migrating Legacy Software to Microservices Architecture*. Em *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, páginas 1–5, Vilnius, Lithuania, abril 2019. IEEE,

ISBN 978-1-72812-499-5. https://ieeexplore.ieee.org/document/8732170/, acesso em 2019-09-09. 2, 11, 23, 24, 31, 42, 43, 79, 82

[44] Fowler, Martin: *Strangler Application*, 2004. https://martinfowler.com/bliki/StranglerApplication.html, acesso em 2019-01-20. 2, 8, 34

[45] Yoder, Joseph W. e Paulo Merson: *Strangler Patterns*. 27th Conference on Pattern Languages of Programs (PLoP). PLoP'20, October 12-16, 2020, 2020. 2, 35, 61, 64, 79, 105

[46] Ahmadvand, Mohsen e Amjad Ibrahim: *Requirements Reconciliation for Scalable and Secure Microservice (De)composition*. Em *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, páginas 68–73, Beijing, China, setembro 2016. IEEE, ISBN 978-1-5090-3694-3. http://ieeexplore.ieee.org/document/7815609/, acesso em 2019-02-25. 3, 6, 7, 10, 15, 23, 26, 29, 31

[47] Richardson, Chris: *Microservices Pattern: Microservice Architecture pattern*, 2018. http://microservices.io/patterns/microservices.html, acesso em 2019-02-02. 3, 20, 25, 34

[48] Gartner, Research e Kevin Matheny: *Gartner - Solution Path for Using Microservices Architecture Principles to Deliver Applications*, setembro 2018. https://www.gartner.com/en/documents/3867868. 3, 16, 17, 34

[49] Balalaie, Armin, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri e Theo Lynn: *Microservices migration patterns: Microservices migration patterns*. Software: Practice and Experience, 2015, ISSN 00380644. http://doi.wiley.com/10.1002/spe.2608, acesso em 2019-05-29. 3, 9, 16, 34

[50] Levcovitz, Alessandra, Ricardo Terra e Marco Tulio Valente: *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems*. página 8, 2016. 3

[51] Furda, Andrei, Colin Fidge, Olaf Zimmermann, Wayne Kelly e Alistair Barros: *Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency*. IEEE Software, 35(3):63–72, maio 2018, ISSN 0740-7459. https://ieeexplore.ieee.org/document/8186442/, acesso em 2019-05-29. 3, 32

[52] EBCloud: *EBCloud | Home*, 2019. https://ebcloud.eb.mil.br/, acesso em 2019-09-02. 3

[53] BE: *BE nº 46, de 17 NOV 2017 (Pág. 40). Diretriz de Racionalização de TIC do QuartelGeneral do Exército.*, 2017. https://goo.gl/NRcPg7, acesso em 2019-06-06. 3

[54] Gehlert, Andreas e Daniel Pfeiffer: *Utilizing Theories to Reduce the Subjectivity of Method Engineering Processes*. página 14, 2007. 5

[55] Burns, Timothy e Fadi Deek: *A Methodology Tailoring Model for Practitioner Based Information Systems Development Informed by the Principles of General Systems Theory*. 4, 2011. 5

[56] Li, Shanshan: *Understanding Quality Attributes in Microservice Architecture*. Em *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, páginas 9–10, Nanjing, dezembro 2017. IEEE, ISBN 978-1-5386-2649-8. `http://ieeexplore.ieee.org/document/8312516/`, acesso em 2019-02-25. 6

[57] Gouigoux, Jean Philippe e Dalila Tamzalit: *From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture*. Em *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, páginas 62–65, Gothenburg, Sweden, abril 2017. IEEE, ISBN 978-1-5090-4793-2. `http://ieeexplore.ieee.org/document/7958457/`, acesso em 2019-02-25. 6, 13, 20, 21, 27, 28, 29, 32

[58] Kecskemeti, Gabor, Attila Csaba Marosi e Attila Kertesz: *The ENTICE approach to decompose monolithic services into microservices*. Em *2016 International Conference on High Performance Computing & Simulation (HPCS)*, páginas 591–596, Innsbruck, Austria, julho 2016. IEEE, ISBN 978-1-5090-2088-1. `http://ieeexplore.ieee.org/document/7568389/`, acesso em 2019-02-25. 6

[59] Sarkar, Santonu, Gloria Vashi e Pp Abdulla: *Towards Transforming an Industrial Automation System from Monolithic to Microservices*. Em *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, páginas 1256–1259, Turin, setembro 2018. IEEE, ISBN 978-1-5386-7108-5. `https://ieeexplore.ieee.org/document/8502567/`, acesso em 2019-05-29. 6, 28

[60] Wohlin, C., P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell e A. Wesslén: *Experimentation in Software Engineering*. Computer Science. Springer Berlin Heidelberg, 2012, ISBN 9783642290442. `https://books.google.com.br/books?id=QPVsM1_U8nkC`. 7, 67

[61] Yanaga, Edson: *Migrating to Microservice Databases*. página 72, 2017. 9

[62] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord e J. Stafford: *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Pearson Education, 2010, ISBN 9780132488594. `https://books.google.com.br/books?id=UTZbsrA4qAsC`. 9

[63] Kazman, Rick, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson e Jeromy Carriere: *The Architecture Tradeoff Analysis Method*. página 11, 1998. 9

[64] Bachmann, Felix, Len Bass e Mark Klein: *Illuminating the fundamental contributors to software architecture quality*. Relatório Técnico CMU/SEI-2002-TR-025, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002. `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6193`. 9

[65] Akinnuwesi, Boluwaji A., Faith Michael E. Uzoka, Stephen O. Olabiyisi e Elijah O. Omidiora: *A framework for user-centric model for evaluating the performance of distributed software system architecture*. Expert Systems with Applications, 39(10):9323–9339, agosto 2012, ISSN 09574174. https://linkinghub.elsevier.com/retrieve/pii/S0957417412003119, acesso em 2019-05-03. 9

[66] Salah, Tasneem, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri e Yousof Al-Hammadi: *The evolution of distributed systems towards microservices architecture*. Em *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, páginas 318–325, Barcelona, Spain, dezembro 2016. IEEE, ISBN 978-1-908320-73-5. http://ieeexplore.ieee.org/document/7856721/, acesso em 2019-02-25. 9

[67] Lamersdorf, Winfried: *Paradigms of Distributed Software Systems: Services, Processes and Self-organization*. Em Obaidat, Mohammad S., José L. Sevillano e Joaquim Filipe (editores): *E-Business and Telecommunications*, volume 314, páginas 33–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ISBN 978-3-642-35754-1 978-3-642-35755-8. http://link.springer.com/10.1007/978-3-642-35755-8_3, acesso em 2019-05-26. 9

[68] Bakshi, Kapil: *Microservices-based software architecture and approaches*. Em *2017 IEEE Aerospace Conference*, páginas 1–8, Big Sky, MT, USA, março 2017. IEEE, ISBN 978-1-5090-1613-6. http://ieeexplore.ieee.org/document/7943959/, acesso em 2019-02-25. 9, 10, 12, 79

[69] Daya, Shahir, Nguyen Van Duy, Kameswara Eati, Carlos M Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, Valerie Lampkin, Marcelo Martins, Shishir Narain e Ramratan Vennam: *Microservices: From Theory to Practice*. página 170, 2015. 9, 10, 12, 13, 72, 79

[70] Namiot, Dmitry e Manfred Sneps-Sneppe: *On Micro-services Architecture*. 2(9):4, 2014. 9

[71] Villamizar, Mario, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano e Mery Lang: *Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures*. Em *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, páginas 179–182, Cartagena, Colombia, maio 2016. IEEE, ISBN 978-1-5090-2453-7. http://ieeexplore.ieee.org/document/7515686/, acesso em 2019-05-25. 10

[72] Fowler, Martin: *Monolith First*, 2015. https://martinfowler.com/bliki/MonolithFirst.html, acesso em 2019-05-24. 10

[73] Fowler, Martin: *Microservice Premium*, 2015. https://martinfowler.com/bliki/MicroservicePremium.html, acesso em 2019-06-12. 10, 11, 82

[74] Nadareishvili, Irakli, Ronnie Mitra, Matt McLarty e Mike Amundsen: *Microservice Architecture: Aligning Principles, Practices, and Culture*. página 146, 2016. 10, 18, 20, 32, 34, 65

[75] Linthicum, David: *From containers to microservices: How to modernize legacy applications*, 2018. `https://techbeacon.com/enterprise-it/containers-microservices-modernizing-legacy-applications`, acesso em 2019-11-11. 10, 11

[76] Brown, Simon: *Distributed big balls of mud - Coding the Architecture*, 2014. `http://www.codingthearchitecture.com/2014/07/06/distributed_big_balls_of_mud.html`, acesso em 2020-01-04. 10

[77] Tilkov, Stefan: *Don't start with a monolith*, 2015. `https://martinfowler.com/articles/dont-start-monolith.html`, acesso em 2020-01-04. 10

[78] Kamimura, Manabu, Keisuke Yano, Tomomi Hatano e Akihiko Matsuo: *Extracting Candidates of Microservices from Monolithic Application Code*. Em *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, páginas 571–580, Nara, Japan, dezembro 2018. IEEE, ISBN 978-1-72811-970-0. `https://ieeexplore.ieee.org/document/8719439/`, acesso em 2019-05-28. 12

[79] Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin e Larisa Safina: *Microservices: Yesterday, Today, and Tomorrow*. Em Mazzara, Manuel e Bertrand Meyer (editores): *Present and Ulterior Software Engineering*, páginas 195–216. Springer International Publishing, Cham, 2017, ISBN 978-3-319-67424-7 978-3-319-67425-4. `http://link.springer.com/10.1007/978-3-319-67425-4_12`, acesso em 2019-09-16. 12, 13, 23, 80

[80] Richardson, Chris: *Microservices: Decomposing Applications for Deployability and Scalability*, 2014. `https://www.infoq.com/articles/microservices-intro`, acesso em 2019-02-18. 12

[81] Sayara, Anika, Md. Shamim Towhid e Md. Shahriar Hossain: *A probabilistic approach for obtaining an optimized number of services using weighted matrix and multidimensional scaling*. Em *2017 20th International Conference of Computer and Information Technology (ICCIT)*, páginas 1–6, Dhaka, dezembro 2017. IEEE, ISBN 978-1-5386-1150-0. `https://ieeexplore.ieee.org/document/8281804/`, acesso em 2019-05-29. 12, 20, 31

[82] Engel, Thomas, Melanie Langermeier, Bernhard Bauer e Alexander Hofmann: *Evaluation of Microservice Architectures: A Metric and Tool-Based Approach*. Springer Berlin Heidelberg, New York, NY, 2018, ISBN 978-3-319-92900-2. 12, 23, 63, 80

[83] Hasselbring, Wilhelm e Guido Steinacker: *Microservice Architectures for Scalability, Agility and Reliability in E-Commerce*. Em *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, páginas 243–246, Gothenburg, Sweden, abril 2017. IEEE, ISBN 978-1-5090-4793-2. `http://ieeexplore.ieee.org/document/7958496/`, acesso em 2019-02-25. 12, 13, 15, 18, 20, 21, 22, 23, 24, 25, 27, 30, 33, 63

[84] Mazlami, Genc, Jurgen Cito e Philipp Leitner: *Extraction of Microservices from Monolithic Software Architectures*. Em *2017 IEEE International Conference on Web Services (ICWS)*, páginas 524–531, Honolulu, HI, USA, junho 2017. IEEE, ISBN 978-1-5386-0752-7. http://ieeexplore.ieee.org/document/8029803/, acesso em 2019-02-25. 12, 25, 32, 47

[85] Singleton, Andy: *The Economics of Microservices*. IEEE Cloud Computing, 3(5):16–20, setembro 2016, ISSN 2325-6095. http://ieeexplore.ieee.org/document/7742218/, acesso em 2019-11-11. 12, 42

[86] Tserpes, Konstantinos: *stream-MSA: A microservices' methodology for the creation of short, fast-paced, stream processing pipelines*. ICT Express, página S2405959519301092, abril 2019, ISSN 24059595. https://linkinghub.elsevier.com/retrieve/pii/S2405959519301092, acesso em 2019-05-29. 13

[87] Friedrichsen, Uwe: *Resilient Functional Service Design*, 2017. https://www.infoq.com/presentations/resilience-functional-service-design/, acesso em 2019-11-09. 13, 63

[88] Alshuqayran, Nuha, Nour Ali e Roger Evans: *A Systematic Mapping Study in Microservice Architecture*. Em *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, páginas 44–51, Macau, China, novembro 2016. IEEE, ISBN 978-1-5090-4781-9. http://ieeexplore.ieee.org/document/7796008/, acesso em 2019-09-16. 13, 27, 80

[89] EUR-Lex, Europa: *General Data Protection Regulation*, 2016. https://eur-lex.europa.eu/eli/reg/2016/679/oj, acesso em 2019-11-13. 15

[90] Baresi, Luciano: *Microservices Are Not Just Tiny Web Services*. Springer Berlin Heidelberg, New York, NY, 2018, ISBN 978-3-319-74780-4. 15

[91] Mueller, Ernest: *What Is DevOps?*, agosto 2010. https://theagileadmin.com/what-is-devops/, acesso em 2019-11-09. 15

[92] Waller, Jan, Nils C. Ehmke e Wilhelm Hasselbring: *Including Performance Benchmarks into Continuous Integration to Enable DevOps*. ACM SIGSOFT Software Engineering Notes, 40(2):1–4, abril 2015, ISSN 01635948. http://dl.acm.org/citation.cfm?doid=2735399.2735416, acesso em 2019-11-02. 16, 129

[93] Santis, Sandro De, Luis Florez, Duy V Nguyen e Eduardo Rosa: *Evolve the Monolith to Microservices with Java and Node*. página 132, 2016. 16

[94] Evans, Eric: *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003, ISBN 978-0-321-12521-7. https://books.google.com.br/books?id=7dlaMs0SECsC. 16, 32, 108, 145

[95] Cichini, Rafael: *DevOps - Diferença entre Continuous Integration, Delivery e Deployment*, novembro 2014. http://blog.justdigital.com.br/devops-qual-a-diferencas-entre-continuous-delivery-continuous-integration-e-cont acesso em 2020-03-26, Library Catalog: blog.justdigital.com.br Section: Business Agility e Agile. 16, 17

[96] Shahin, Mojtaba, Muhammad Ali Babar e Liming Zhu: *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*. IEEE Access, 5:3909–3943, 2017, ISSN 2169-3536. http://ieeexplore.ieee.org/document/7884954/, acesso em 2020-03-25. 16, 17

[97] Puppet: *2015 State of DevOps Report | Puppet.com*, 2015. https://puppet.com/resources/report/2015-state-devops-report/, acesso em 2020-03-25. 16

[98] Weber, Ingo, Surya Nepal e Liming Zhu: *Developing Dependable and Secure Cloud Applications*. IEEE Internet Computing, 20(3):74–79, maio 2016, ISSN 1089-7801. http://ieeexplore.ieee.org/document/7465693/, acesso em 2020-03-26. 16, 17

[99] Humble, Jez e David Farley: *Continuous Delivery*. 2010. 17, 64

[100] Carvalho, Luiz, Alessandro Garcia, Wesley K. G. Assunção, Rodrigo Bonifácio, Leonardo P. Tizzei e Thelma Elita Colanzi: *Extraction of configurable and reusable microservices from legacy systems: an exploratory study [industry]*. Em *Proceedings of the 23rd International Systems and Software Product Line Conference - volume A - SPLC '19*, páginas 1–6, Paris, France, 2019. ACM Press, ISBN 978-1-4503-7138-4. http://dl.acm.org/citation.cfm?doid=3336294.3336319, acesso em 2020-09-17. 18

[101] C Martin's, Robert: *The Single Responsibility Principle*, 2014. https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html, acesso em 2019-06-09. 18, 20

[102] Taibi, Davide e Kari Systä: *From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining:*. Em *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, páginas 153–164, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications, ISBN 978-989-758-365-0. http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0007755901530164, acesso em 2019-09-09. 18, 21, 29, 31

[103] Killalea, Tom: *The hidden dividends of microservices*. Communications of the ACM, 59(8):42–45, julho 2016, ISSN 00010782. http://dl.acm.org/citation.cfm?doid=2975594.2948985, acesso em 2019-05-29. 18

[104] Dijkstra, Edsger: *On the role of scientific thought*. janeiro 1974. 18

[105] Jin, Wuxia, Ting Liu, Qinghua Zheng, Di Cui e Yuanfang Cai: *Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering*. Em *2018 IEEE International Conference on Web Services (ICWS)*, páginas 211–218, San Francisco, CA, USA, julho 2018. IEEE, ISBN 978-1-5386-7247-1. https://ieeexplore.ieee.org/document/8456351/, acesso em 2019-05-29. 19, 29, 31, 118

[106] Fowler, Martin: *bliki: TolerantReader*, 2011. https://martinfowler.com/bliki/TolerantReader.html, acesso em 2019-09-28. 19, 20, 25

[107] Vernon, Vaughn: *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1st edição, 2013, ISBN 0-321-83457-7 978-0-321-83457-7. 19, 119

[108] Villaça, Luís H. N., Leonardo G. Azevedo e Fernanda Baião: *Query strategies on polyglot persistence in microservices*. Em *Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18*, páginas 1725–1732, Pau, France, 2018. ACM Press, ISBN 978-1-4503-5191-1. `http://dl.acm.org/citation.cfm?doid=3167132.3167316`, acesso em 2019-04-07. 19

[109] Mazzara, Manuel, Nicola Dragoni, Antonio Bucchiarone, Alberto Giaretta, Stephan T. Larsen e Schahram Dustdar: *Microservices: Migration of a Mission Critical System*. IEEE Transactions on Services Computing, páginas 1–1, 2018, ISSN 1939-1374, 2372-0204. `https://ieeexplore.ieee.org/document/8585089/`, acesso em 2019-09-09. 19, 20, 21, 23, 26, 27, 28, 32, 37, 38, 39, 118

[110] Escobar, Daniel, Diana Cardenas, Rolando Amarillo, Eddie Castro, Kelly Garces, Carlos Parra e Rubby Casallas: *Towards the understanding and evolution of monolithic applications as microservices*. Em *2016 XLII Latin American Computing Conference (CLEI)*, páginas 1–11, Valparaíso, Chile, outubro 2016. IEEE, ISBN 978-1-5090-1633-4. `http://ieeexplore.ieee.org/document/7833410/`, acesso em 2019-02-25. 20, 29, 31, 106

[111] Klock, Sander, Jan Martijn E. M. Van Der Werf, Jan Pieter Guelen e Slinger Jansen: *Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures*. Em *2017 IEEE International Conference on Software Architecture (ICSA)*, páginas 11–20, Gothenburg, Sweden, abril 2017. IEEE, ISBN 978-1-5090-5729-0. `http://ieeexplore.ieee.org/document/7930194/`, acesso em 2019-06-09. 21

[112] Hamzehloui, Mohammad Sadegh, Shamsul Sahibuddin e Ardavan Ashabi: *A Study on the Most Prominent Areas of Research in Microservices*. International Journal of Machine Learning and Computing, 9(2):7, 2019. 21

[113] *ELK Stack: Elasticsearch, Logstash e Kibana | Elastic*. `https://www.elastic.co/pt/what-is/elk-stack`, acesso em 2020-02-02. 21

[114] Noor, Ayman, Devki Nandan Jha, Karan Mitra, Prem Prakash Jayaraman, Arthur Souza, Rajiv Ranjan e Schahram Dustdar: *A Framework for Monitoring Microservice-Oriented Cloud Applications in Heterogeneous Virtualization Environments*. Em *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, páginas 156–163, Milan, Italy, julho 2019. IEEE, ISBN 978-1-72812-705-7. `https://ieeexplore.ieee.org/document/8814569/`, acesso em 2019-09-14. 22, 27, 77, 81

[115] Google, Kubernetes: *What Is Kubernetes*, 2020. `https://cloud.google.com/learn/what-is-kubernetes`, acesso em 2020-02-03. 23

[116] AWS: *Amazon Web Services (AWS) - Cloud Computing Services*, 2020. `https://aws.amazon.com/`, acesso em 2020-02-03. 23

[117] Azure: *Cloud Computing Services | Microsoft Azure*, 2020. https://azure.microsoft.com/en-us/, acesso em 2020-02-03. 23

[118] Mayer, Benjamin e Rainer Weinreich: *An Approach to Extract the Architecture of Microservice-Based Software Systems*. Em *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, páginas 21–30, Bamberg, março 2018. IEEE, ISBN 978-1-5386-5207-7. https://ieeexplore.ieee.org/document/8359145/, acesso em 2019-11-11. 23, 24

[119] Brooks, F.P.: *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Pearson Education, 1995, ISBN 978-0-13-211916-0. https://books.google.com.br/books?id=Yq35BY5Fk3gC. 25

[120] Robinson, Ian: *Consumer-Driven Contracts: A Service Evolution Pattern*, 2006. https://martinfowler.com/articles/consumerDrivenContracts.html, acesso em 2020-01-24. 25

[121] O'Brien, Liam, Liam, Bass, Len, Paulo Merson e Paulo: *Quality Attributes and Service-Oriented Architectures*. 2005. 25

[122] Marmol, Victor, Rohit Jnagal e Tim Hockin: *Networking in Containers and Container Clusters*. página 4, 2015. 27

[123] *Enterprise Application Container Platform | Docker*, 2019. https://www.docker.com/, acesso em 2019-05-25. 27

[124] Parnas, D L: *On the Criteria To Be Used in Decomposing Systems into Modules*. 15(12):6, 1972. 29

[125] Granchelli, Giona, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino e Amleto Di Salle: *Towards Recovering the Software Architecture of Microservice-Based Systems*. Em *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, páginas 46–53, Gothenburg, Sweden, abril 2017. IEEE, ISBN 978-1-5090-4793-2. http://ieeexplore.ieee.org/document/7958455/, acesso em 2019-11-04. 29

[126] Eyk, Erwin van, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uta e Alexandru Iosup: *Serverless is More: From PaaS to Present Cloud Computing*. IEEE Internet Computing, 22(5):8–17, setembro 2018, ISSN 1089-7801, 1941-0131. https://ieeexplore.ieee.org/document/8481652/, acesso em 2019-11-13. 29, 32

[127] Hassan, Sara e Rami Bahsoon: *Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap*. Em *2016 IEEE International Conference on Services Computing (SCC)*, páginas 813–818, San Francisco, CA, USA, junho 2016. IEEE, ISBN 978-1-5090-2628-9. http://ieeexplore.ieee.org/document/7557535/, acesso em 2019-05-16. 29, 31, 47

[128] Fritzsch, Jonas, Justus Bogner, Alfred Zimmermann e Stefan Wagner: *From Monolith to Microservices: A Classification of Refactoring Approaches*. Em Bruel, Jean Michel, Manuel Mazzara e Bertrand Meyer (editores): *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, volume 11350, páginas 128–141. Springer International Publishing, Cham, 2019, ISBN 978-3-030-06018-3 978-3-030-06019-0. http://link.springer.com/10.1007/978-3-030-06019-0_10, acesso em 2019-02-26. 29

[129] Feathers, M.C.: *Working Effectively with Legacy Code*. Martin, Robert C. Prentice Hall PTR, 2004, ISBN 978-0-13-117705-5. https://books.google.com.br/books?id=vlo_nWophSYC. 29

[130] Richardson, Chris: *Microservices Pattern: Decompose by business capability*, 2018. http://microservices.io/patterns/decomposition/decompose-by-business-capability.html, acesso em 2019-05-13. 30

[131] Richardson, Chris: *Microservices Pattern: Decompose by subdomain*, 2019. http://microservices.io/patterns/decomposition/decompose-by-subdomain.html, acesso em 2019-06-14. 30, 36, 118

[132] Richardson, Chris: *Microservices Pattern: Self-contained service*, 2019. http://microservices.io/patterns/decomposition/self-contained-service.html, acesso em 2020-01-18. 30

[133] Richardson, Chris: *Microservices Pattern: Service per team*, 2019. http://microservices.io/patterns/decomposition/service-per-team.html, acesso em 2020-01-18. 30

[134] SCS: *SCS: Self-Contained Systems*, 2020. https://scs-architecture.org/, acesso em 2020-02-03. 30

[135] Structure101: *Structure101 Home » Structure101*, 2020. https://structure101.com/, acesso em 2020-02-03. 31

[136] Garriga, Martin: *Towards a Taxonomy of Microservices Architectures*. Em Cerone, Antonio e Marco Roveri (editores): *Software Engineering and Formal Methods*, volume 10729, páginas 203–218. Springer International Publishing, Cham, 2018, ISBN 978-3-319-74780-4 978-3-319-74781-1. http://link.springer.com/10.1007/978-3-319-74781-1_15, acesso em 2019-05-16. 31, 34

[137] Richardson, Chris: *Microservices Pattern: Sagas*, 2018. http://microservices.io/patterns/data/saga.html, acesso em 2019-12-28. 33

[138] Zhang, Guogen, Kun Ren, Jung Sang Ahn e Sami Ben-Romdhane: *GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases*. Em *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, páginas 2024–2027, Macao, Macao, abril 2019. IEEE, ISBN 978-1-5386-7474-1. https://ieeexplore.ieee.org/document/8731442/, acesso em 2019-12-28. 33

[139] CM, First: *Modernize with the Strangler Application Pattern and Microservices - CM First Group*, 2019. https://cmfirstgroup.com/modernize-with-the-strangler-application-pattern-and-microservices/, acesso em 2020-01-26. 35

[140] Richardson, Chris: *Microservices Pattern: API gateway pattern*, 2018. http://microservices.io/patterns/apigateway.html, acesso em 2019-05-13. 37

[141] Richardson, Chris: *Microservices Pattern: Service instance per container pattern*, 2019. http://microservices.io/patterns/deployment/service-per-container.html, acesso em 2019-06-14. 37

[142] Richardson, Chris: *Microservices Pattern: Server-side service discovery pattern*, 2019. http://microservices.io/patterns/server-side-discovery.html, acesso em 2019-06-17. 37, 38

[143] Montesi, Fabrizio e Janine Weber: *Circuit Breakers, Discovery, and API Gateways in Microservices*. arXiv:1609.05830 [cs], setembro 2016. http://arxiv.org/abs/1609.05830, acesso em 2019-05-25, arXiv: 1609.05830. 37

[144] Richardson, Chris: *Microservices Pattern: Access token*, 2019. http://microservices.io/patterns/security/access-token.html, acesso em 2019-06-14. 38

[145] JWT: *JWT.IO*, 2020. http://jwt.io/, acesso em 2021-02-03. 38

[146] Microsoft: *Circuit Breaker Pattern*, 2015. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn589784, acesso em 2020-01-27. 39

[147] Richardson, Chris: *Microservices Pattern: Database per service*, 2019. http://microservices.io/patterns/data/database-per-service.html, acesso em 2019-06-14. 40

[148] ThoughtWorks: *Zhamak Dehghani – Real World Microservices - Lessons from the Frontline*. https://www.youtube.com/watch?v=hsoovFbpAoE&feature=youtu.be, acesso em 2019-05-16. 42

[149] Dehghani, Zhamak: *How to break a Monolith into Microservices*, 2018. https://martinfowler.com/articles/break-monolith-into-microservices.html, acesso em 2019-11-11. 43, 79

[150] Ahmad, Aakash, Pooyan Jamshidi e Claus Pahl: *Classification and comparison of architecture evolution reuse knowledge-a systematic review: CLASSIFICATION AND COMPARISON OF AERK*. Journal of Software: Evolution and Process, 26(7):654–691, julho 2014, ISSN 20477473. http://doi.wiley.com/10.1002/smr.1643, acesso em 2019-09-28. 43

[151] Anquetil, Nicolas e Jannik Laval: *Legacy Software Restructuring: Analyzing a Concrete Case*. Em *2011 15th European Conference on Software Maintenance and Reengineering*, páginas 279–286, Oldenburg, Germany, março 2011. IEEE,

ISBN 978-1-61284-259-2. http://ieeexplore.ieee.org/document/5741272/, acesso em 2019-11-09. 43

[152] Candela, Ivan, Gabriele Bavota, Barbara Russo e Rocco Oliveto: *Using Cohesion and Coupling for Software Remodularization: Is It Enough?* ACM Transactions on Software Engineering and Methodology, 25(3):1–28, junho 2016, ISSN 1049331X. http://dl.acm.org/citation.cfm?doid=2943790.2928268, acesso em 2019-11-09. 43

[153] Shatnawi, Anas, Abdelhak Djamel Seriai, Houari Sahraoui e Zakarea Alshara: *Reverse engineering reusable software components from object-oriented APIs.* Journal of Systems and Software, 131:442–460, setembro 2017, ISSN 01641212. https://linkinghub.elsevier.com/retrieve/pii/S016412121630098X, acesso em 2019-11-09. 43

[154] Jin, Wuxia, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo e Qinghua Zheng: *Service Candidate Identification from Monolithic Systems based on Execution Traces.* IEEE Transactions on Software Engineering, páginas 1–1, 2019, ISSN 0098-5589, 1939-3520, 2326-3881. https://ieeexplore.ieee.org/document/8686152/, acesso em 2019-09-14. 43, 64, 79, 82

[155] Ferreira, Kecia A.M., Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes e Heitor C. Almeida: *Identifying thresholds for object-oriented software metrics.* Journal of Systems and Software, 85(2):244–257, fevereiro 2012, ISSN 01641212. https://linkinghub.elsevier.com/retrieve/pii/S0164121211001385, acesso em 2020-03-17. 44, 124

[156] Basili, Victor R, Gianluigi Caldiera e H Dieter Rombach: *THE GOAL QUESTION METRIC APPROACH.* página 10, 1994. 44

[157] Kassou, Meryem e Laila Kjiri: *A Goal Question Metric Approach for Evaluating Security in a Service Oriented Architecture Context.* 9(4):13, 2012. 44

[158] Wohlin, Claes: *Guidelines for snowballing in systematic literature studies and a replication in software engineering.* Em *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, páginas 1–10, London, England, United Kingdom, 2014. ACM Press, ISBN 978-1-4503-2476-2. http://dl.acm.org/citation.cfm?doid=2601248.2601268, acesso em 2019-05-18. 46

[159] Francesco, Paolo Di, Ivano Malavolta e Patricia Lago: *Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption.* Em *2017 IEEE International Conference on Software Architecture (ICSA)*, páginas 21–30, Gothenburg, Sweden, abril 2017. IEEE, ISBN 978-1-5090-5729-0. http://ieeexplore.ieee.org/document/7930195/, acesso em 2019-05-16. 47

[160] Daigneau, R.: *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley signature series Service design patterns. Addison-Wesley, 2012, ISBN 978-0-321-54420-9. https://books.google.com.br/books?id=tK3_vB304bEC. 48

[161] Bergstra, J.A. e M. Burgess: *Promise Theory: Principles and Applications*. Promise Theory. Createspace Independent Pub, 2014, ISBN 978-1-4954-3777-9. https://books.google.com.br/books?id=ydGmngEACAAJ. 51

[162] Ferreira, Norma Sandra de Almeida: *As pesquisas denominadas "estado da arte"*. EducaÃ§Ã Sociedade, 23:257 – 272, agosto 2002, ISSN 0101-7330. http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-73302002000300013&nrm=iso. 55

[163] Peffers, Ken, Tuure Tuunanen, Marcus A. Rothenberger e Samir Chatterjee: *A Design Science Research Methodology for Information Systems Research*. Journal of Management Information Systems, 24(3):45–77, dezembro 2007, ISSN 0742-1222, 1557-928X. https://www.tandfonline.com/doi/full/10.2753/MIS0742-1222240302, acesso em 2020-02-12. 56, 57, 58, 67, 80

[164] Silva, Edna Lúcia da e Estera Muszkat Menezes: *Metodologia da Pesquisa e Elaboração de Dissertação*, 2005. 56

[165] Chatterjee, Samir, Bengisu Tulu, Tarun Abhichandani e Haiqing Li: *Sip-based enterprise converged networks for voice/video-over-ip: Implementation and evaluation of components*. Selected Areas in Communications, IEEE Journal on, 23:1921 – 1933, novembro 2005. 56

[166] Hegenberg, L.: *Etapas da investigação científica*. Editora Pedagógica e Universitária, 1976. https://books.google.com.br/books?id=yQAbPQAACAAJ. 56

[167] Wazlawick, Raul Sidnei: *Metodologia de Pesquisa para Ciência da Computação*. página 124, 2009. 56

[168] Lakatos, E.M. e M. de Andrade Marconi: *Fundamentos de metodologia científica*. Atlas, 2005, ISBN 978-85-224-4015-3. https://books.google.com.br/books?id=r0ruAAAACAAJ. 56

[169] Fleig, Christian, Dominik Augenstein e Alexander Maedche: *Tell me what's my business - development of a business model mining software*. páginas 105–113, junho 2018, ISBN 978-3-319-92900-2. 56

[170] Bogner, Justus, Stefan Wagner e Alfred Zimmermann: *Towards a practical maintainability quality model for service-and microservice-based systems*. Em *Proceedings of the 11th European Conference on Software Architecture Companion Proceedings - ECSA '17*, páginas 195–198, Canterbury, United Kingdom, 2017. ACM Press, ISBN 978-1-4503-5217-8. http://dl.acm.org/citation.cfm?doid=3129790.3129816, acesso em 2019-04-06. 56

[171] Avram, Abel, Floyd Marinescu, Dan Bergh Johnsson, Vladimir Gitlevich e Eric Evans: *Domain-driven design quickly: [a summary of Eric Evans' Domain-driven design*. C4Media, LaVergne, TN, 2006, ISBN 978-1-4116-0925-9. OCLC: 540271783. 62

[172] Gysel, Michael, Lukas Kölbener, Wolfgang Giersche e Olaf Zimmermann: *Service Cutter: A Systematic Approach to Service Decomposition*. Em Aiello, Marco, Einar Broch Johnsen, Schahram Dustdar e Ilche Georgievski (editores): *Service-Oriented and Cloud Computing*, volume 9846, páginas 185–200. Springer International Publishing, Cham, 2016, ISBN 978-3-319-44481-9 978-3-319-44482-6. http://link.springer.com/10.1007/978-3-319-44482-6_12, acesso em 2019-02-25. 63

[173] Merson, Paulo e Joseph Yoder: *Saturn 2019 talk: Modeling microservices with ddd*. https://saturn2019.sched.com/event/LY5d/modeling-microservices-with-ddd, acesso em 2019. 63

[174] Richardson, Chris: *Jfokus 2020 - Cubes, Hexagons, Triangles, and More - Understanding Microservices*, 2020. https://chrisrichardson.net/microservices/2020/02/04/jfokus-geometry-of-microservices.html, acesso em 2020-02-19. 64

[175] Sutherland, Jeff e Ken Schwaber: *Scrum Guide | Scrum Guides*, 2019. https://www.scrumguides.org/scrum-guide.html, acesso em 2020-03-27. 67, 68, 69

[176] Vidal, A., V. Macedo e L. Stok: *Agile Think Canvas*. BRASPORT, 2017, ISBN 978-85-7452-795-6. https://books.google.com.br/books?id=4NMeDgAAQBAJ. 67

[177] Pries, K.H. e J.M. Quigley: *Scrum Project Management*. CRC Press, 2010, ISBN 978-1-4398-2517-4. https://books.google.com.br/books?id=Of6JC-1DHloC. 67

[178] Agrawal, Ankit: *How should PO & DT collaborate for mutual understanding of PBL?*, 2017. https://www.scrum.org/index.php/forum/scrum-forum/12175/how-should-po-dt-collaborate-mutual-understanding-pbl, acesso em 2020-03-28, Library Catalog: www.scrum.org. 69

[179] Shahmohammadi, Gholamreza, Saeed Jalili e Seyed Mohammad Hossein Hasheminejad: *Identification of System Software Components Using Clustering Approach.* The Journal of Object Technology, 9(6):77, 2010, ISSN 1660-1769. http://www.jot.fm/contents/issue_2010_11/article4.html, acesso em 2019-11-04. 79

[180] Netflix, Technology: *Full Cycle Developers at Netflix — Operate What You Build*, maio 2018. https://medium.com/netflix-techblog/full-cycle-developers-at-netflix-a08c31f83249, acesso em 2019-09-08. 105

[181] Microsoft, Docs: *Projetando um microsserviço orientado a DDD*, 2018. https://docs.microsoft.com/pt-br/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice, acesso em 2020-02-01. 108

[182] Microsoft, Docs: *Domain analysis for microservices*, 2019. https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis, acesso em 2019-03-19. 108

[183] Conway, Melvin E: *Conway's Law*, 2019. http://www.melconway.com/Home/Conways_Law.html, acesso em 2019-07-12. 110

[184] Mogosanu, Mike: *Sapiensworks | DDD - The Bounded Context Explained*, 2012. https://blog.sapiensworks.com/post/2012/04/17/DDD-The-Bounded-Context-Explained.aspx, acesso em 2020-04-25. 113

[185] Microsoft: *Anti-Corruption Layer pattern - Cloud Design Patterns*, 2017. https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer, acesso em 2019-05-13. 118

[186] Docker, Docs: *Use swarm mode routing mesh*, abril 2020. https://docs.docker.com/engine/swarm/ingress/, acesso em 2020-04-23, Library Catalog: docs.docker.com. 121

[187] Red, Hat: *O que é middleware?*, 2020. https://www.redhat.com/pt-br/topics/middleware/what-is-middleware, acesso em 2020-08-25. 127

[188] Guimarães, Everton T, Renato B Lavor e Roberta S Coelho: *Test First vs Test Last: Uma Análise Comparativa de Abordagens de Teste na Construção de Sistemas*. página 11. 129

Apêndice A

Artigos de pesquisa

Tabela A.1: Artigos relacionados a método de migração

(Total de citações baseado no Google Scholar).

#	**Artigo**	**Autor**	**Ano**	**Citações**
1	Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture	Balalaie, A. et al [1]	2016	530
2	Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation	Taibi, Davide et al [2]	2017	127
3	Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report	Fan, Chen-Yuan; Ma, Shang-Pin [3]	2017	26
4	Method for Designing Scalable Microservice-based Application Systematically: A Case Study	Ghani, A.T.A.; Zakaria, M.S. [4]	2018	3
5	Cracking the Monolith : Challenges in Data Transitioning to Cloud Native Architectures	Mishra, Mayank et al [5]	2018	5
6	Towards a roadmap for the migration of legacy software systems to a microservice based architecture	Da Silva, H.H.O.S. et al [6]	2019	1

Apêndice B

Implementação - MFC

Esta seção é parte integrante Capítulo 6 e descreve o uso do *Microservice Full Cycle* - MFC aplicado ao monolítico legado SISP. Nesse processo optou-se por utilizar o Scrum. Basicamente o MFC "diz o que fazer", enquanto o Scrum "dita um ritmo" de execução dos trabalhos por meio de suas *sprints*, conforme ilustra a Figura B.1. De acordo com a meta estipulada em cada sprint, são adicionados ao *backlog* os pacotes de trabalho MFC necessários para cumprir a tarefa. Assim, à medida que as *sprints* rodam, os pacotes de trabalho vão sendo consumidos e entregues, e a cada novo incremento, gradualmente, o monolítico é substituído pelos microsserviços. Por convenção, as atividades MFC são identificadas pelo padrão T{*nº da Sprint*}.{*nº MFC*}.{*nº da tarefa*}, o que permite a completa rastreabilidade e retrata a informação do geral para o mais específico.

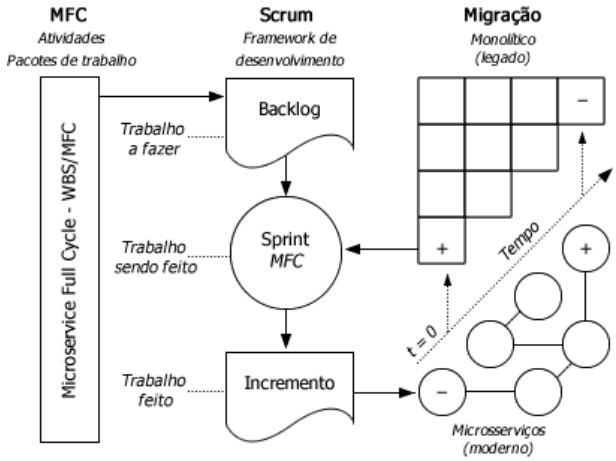


Figura B.1: Processo de desenvolvimento e migração.
(Fonte: própria)

B.1 Sprint 1

Tabela B.1: Sprint 1.

Sprint 1	**Meta:** preparar o ambiente inicial da migração			
Duração:	2 semanas	**Trabalho:** 6h/dia	**Início:** –/–/2020	**Fim:** –/–/2020
#	**Tarefa**			**Estimativa**
T1.1.6.1	Identificar habilidades e capacidades iniciais requeridas			30h
T1.1.1.2	Descrever o monolítico legado alvo e o escopo da migração			12h
T1.1.2.3	Definir método(s) para guiar o processo de migração			6h
T1.1.4.4	Definir especialista técnico e de negócio para apoiar a migração			12h

T1.1.6.1 - Identificar habilidades e capacidades iniciais requeridas. Para enfrentar o primeiro projeto de microsserviço, é preciso se certificar que a equipe de desenvolvimento tem infraestrutura e ambiente adequados (técnico e organizacional). É o que Yoder & Merson [45] chamam de "pavimentar a estrada" (do inglês *Pave the Road*). Isto requer conhecer as habilidades e conceitos necessários para lidar com a arquitetura de microsserviço em estágio inicial, amadurecendo determinadas teorias e práticas de desenvolvimento de software e operações, sobretudo, ligadas à infraestrutura e automação. É desejável que o desenvolvedor saiba *"operar o que constrói"* (do inglês *Operating What You Build*), um dos princípios do movimento *DevOps* [180]. Embora os membros de uma equipe tenham diferentes níveis de conhecimento técnico, é preciso buscar um grau mínimo de nivelamento. Assim, para cumprir esta etapa, o ideal é desenvolver noções sobre certos conceitos e *patterns* associados aos microsserviços, conforme mostra a Figura B.2.

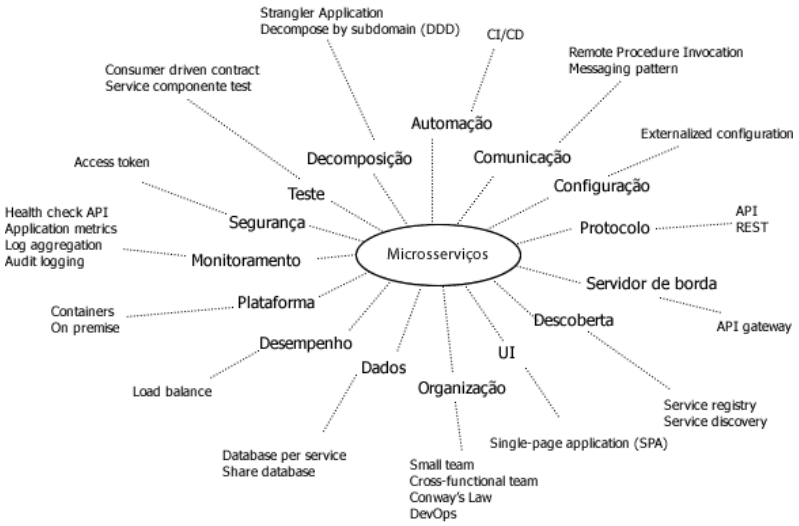


Figura B.2: Áreas de conhecimentos requeridas para a simulação.
(Fonte: própria)

T1.1.1.2 - Descrever o monolítico legado alvo e o escopo da migração. Segundo o MFC, cabe a essa atividade descrever a funcionalidade corrente do sistema monolítico legado que será migrada para arquitetura de microsserviços. Entretanto, como se trata do primeiro ciclo MFC, será apresentado uma visão geral do monolítico como um todo e, posteriormente, os detalhes funcionais a serem implementados. O *Sistema de Seleção de Pessoal* - SISP é o principal ativo de informações da *Diretoria de Seleção de Pessoal* - DISP. Além disto, tem inestimável importância para o *Exército Brasileiro* - EB, sobretudo, quando se trata de seleção de pessoal de carreira para cursos e estágios.

Como muitos dos sistemas monolíticos legados, o SISP vem se tornando grande e complexo, com recorrentes problemas de manutenibilidade, desempenho e interoperabilidade. Sabe-se que a degradação do software é um processo natural, tanto pelo acúmulo de endividamento técnico, quanto pela depreciação das tecnologias. Busca-se por meio da arquitetura de microsserviços, uma forma de restruturação e modernização do SISP. A escolha dessa arquitetura se dá pela possibilidade de "quebrar" gradualmente a aplicação em pequenas partes independentes e autônomas, tendo o mínimo *downtime* e impacto ao usuário final. Dentre os benefícios, microsserviços oferecem a perspectiva de trabalhar em pequenas equipes, tornando mais fácil a manutenção e escalabilidade do software.

A modernização de software pretendida passa pelo (i) entendimento do atual sistema e (ii) transformação do estado "*as-is*" para "*to-be*" [110]. A Figura B.3 destaca a visão geral em termos de arquitetura sobre como o SISP é (AS-IS: monolítico) e como está se propondo a ser (TO-BE: microsserviços). Por se tratar de um projeto-piloto, a simulação incidirá inicialmente apenas sobre uma fração do *módulo cursos* SISP (escopo), ficando a cargo da equipe de desenvolvimento futuros avanços, se for o caso.

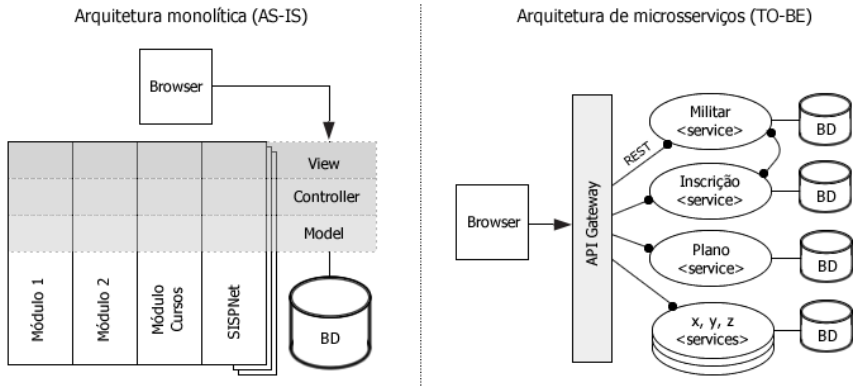


Figura B.3: Visão *As-is* e *To-be* do SISP.
(Fonte: própria)

T1.1.2.3 - Definir método(s) para guiar o processo de migração. Para conduzir o processo de migração, é utilizado o método *Microservice Full Cycle* - MFC, descrito pela Figura B.4. O método MFC é um compilado baseado em diversas literaturas e experiências que de algum modo contribuíram para um processo de migração bem-sucedido utilizando a arquitetura de microsserviços. Tecnicamente, o método é composto por seis etapas alinhadas ao ciclo de desenvolvimento de software e a estratégias *DevOps*. Cabe ressaltar que as atividades previstas não tem caráter sequencial ou obrigatório. Além disto, optou-se por adotar o processo de desenvolvimento baseado no *framework Scrum* (Figura 6.1). Com o *Scrum* ditando um ritmo incremental e o MFC mostrando "o que" fazer, espera-se conduzir a migração do legado rumo à arquitetura de microsserviços.

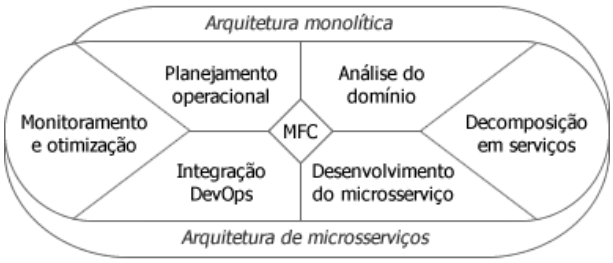


Figura B.4: Método *Microservice Full Cycle* - MFC.
(Fonte: própria)

T1.1.4.4 - Definir especialista técnico e de negócio para apoiar a migração. Essa atividade visa definir o apoio de (i) especialista técnico com reconhecida competência profissional; e (ii) especialista de negócio com elevada experiência organizacional. Ter suporte e supervisão de um especialista é uma das mais importantes partes do processo de migração. Em termos técnicos, este trabalho conta com o apoio do Prof. P.F.M., mestre em Engenharia de Software pela *Carnegie Mellon University*. P.F.M. atua com desenvolvimento de software há mais de 30 anos. É palestrante internacional e consultor com larga experiência em arquitetura de software, sobretudo, em microsserviços. P.F.M. também é cientista visitante do Instituto de Engenharia de Software (do inglês *Software Engineering Institute* - SEI) e membro do programa de mestrado do departamento de Ciência da Computação da Universidade de Brasília - UnB.

Além do apoio técnico, há o apoio da área de negócios. Especialistas em domínios (do inglês *domain experts*) são imprescindíveis para construir uma linguagem ubíqua, de modo que o os microsserviços possam ser alinhados ao modelo de negócio. Colabora neste sentido o Capitão J.D.R.S.J., militar com mais de 12 anos de serviço, destes, 4 anos lidando com os processos do SISP/DISP. Como profundo conhecedor da organização e grande experiência em Tecnologia da Informação, o Capitão J.D.R.S.J. presta assessoria ao Comando em

muitas tomadas de decisão e está comprometido em contribuir com os questionamentos de negócio do presente trabalho. Por ser um projeto-piloto de baixa complexidade, considera-se suficiente o apoio do Capitão J.D.R.S.J, embora outras participações sejam bem-vindas.

B.2 Sprint 2

Tabela B.2: Sprint 2.

Sprint 2	**Meta:** entender sobre o domínio da aplicação					
Duração:	2 semanas	**Trabalho:**	6h/dia	**Início:**	–/–/2020	**Fim:** –/–/2020
#	**Tarefa**					**Estimativa**
T2.1.6.5	Preparar estudo preliminar sobre DDD					30h
T2.2.1.6	Analisar a estrutura organizacional					6h
T2.2.2.7	Analisar o modelo de negócio					6h
T2.2.3.8	Analisar o sistema monolítico legado					6h
T2.2.4.9	Analisar a persistência de dados					6h
T2.2.6.10	Consolidar a análise do domínio					6h

T2.1.6.5 - Preparar estudo preliminar sobre DDD. Antes de começar a análise do domínio, é interessante entender o negócio da organização sob a perspectiva do *Domain-Driven Design* - DDD, ao menos os princípios básicos. Segundo Evans [94], o software precisa incorporar conceitos e elementos *"core"* do domínio, expressando a real relação entre eles. Isso é feito pela abstração do domínio, ou seja, por um modelo. Um modelo do domínio não é um diagrama, mas a ideia que o diagrama pretende transmitir. Assim, a informação é então sistematizada, dividida em partes menores e agrupadas de modo lógico, buscando sempre extrair o essencial e a generalizar o domínio [94]. DDD é, portanto, sobre delimitar as fronteiras do sistema. O processo constante de refinamento do modelo permite melhor compreender a complexidade do domínio e a esboçar contextos delimitados que, posteriormente, tendem a se tornar potenciais candidatos a microsserviços [181], conforme mostra o ciclo básico de análise do domínio com DDD da Figura B.5

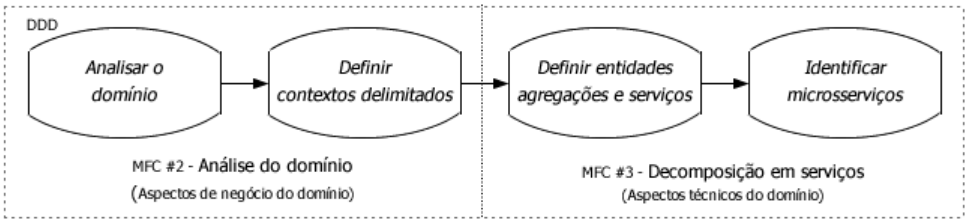


Figura B.5: Ciclo básico de análise do domínio com DDD.
(Fonte: Baseado em *Microsoft* [182])

Em termos práticos, o importante para o cenário deste estudo é observar e traçar os limites do domínio, identificando entidades e relações que melhor expressem o negócio da organização. Por se tratar de uma assunto relativamente complexo, a recomendação é que seja utilizado como referência para o estudo, títulos como *"Domain-Driven Design: Tackling Complexity in the Heart of Software"*, de *Eric Evans* e *"Implementing Domain-Driven Design"*, de *Vaughn Vernon.*

T2.2.1.6 - Analisar a estrutura organizacional. Cabe a esta etapa verificar documentos, planilhas, organogramas, relações interpessoais e departamentais associadas ao monolítico legado alvo da migração, representado pelo *módulo cursos* do SISP. A Figura B.6[1] mostra uma abordagem *top-down* em que o objeto de interesse deste estudo é realçado ao centro, sendo refinado sucessivamente, começando pelo domínio do Exército Brasileiro[2] até chegar à DISP e ao *módulo cursos* do SISP.

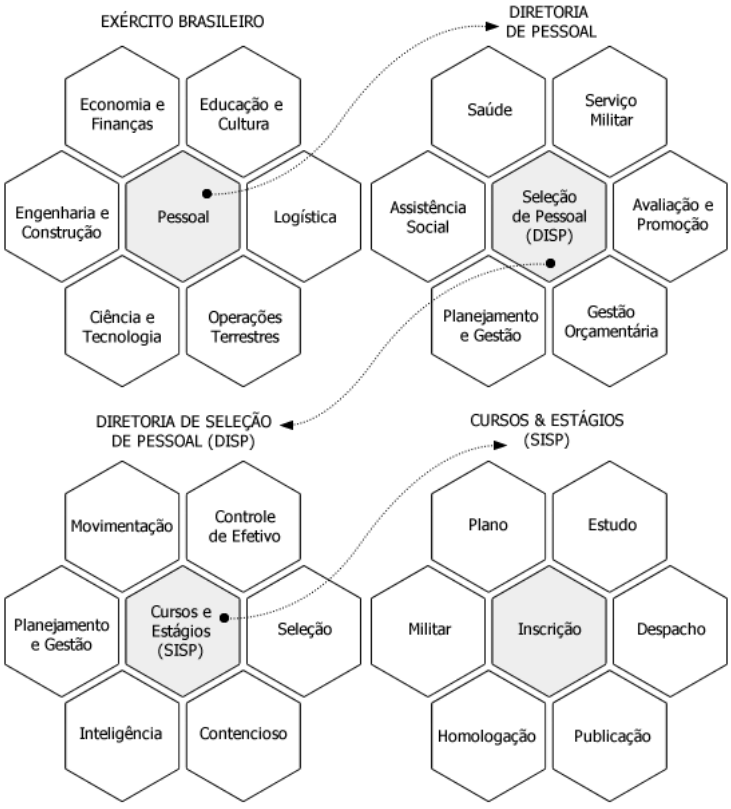


Figura B.6: Visão macro do contexto organizacional SISP/DISP.
(Fonte: própria)

[1]Algumas informações são fictícias visando resguardar a organização.
[2]http://www.eb.mil.br/estrutura-organizacional

Essa visão macro não esboça simplesmente uma hierarquia, mas capacidades de negócios presentes no domínio. Segundo a Lei de *Conway* [183], qualquer organização que projete um sistema produzirá um *design* cuja estrutura é uma cópia da sua estrutura de comunicação. Isto pode ser observado na Figura B.7. A "missão institucional" da DISP em selecionar pessoal militar para cursos e estágio está alinhada a sua estrutura do organograma que, por sua vez, assemelha-se à composição dos módulos do sistema legado (SISP). Dentre os pilares do negócio organizacional, apenas o processo de "Inscrição" do módulo *Cursos e Estágios* (em tom cinza) será alvo da simulação proposta.

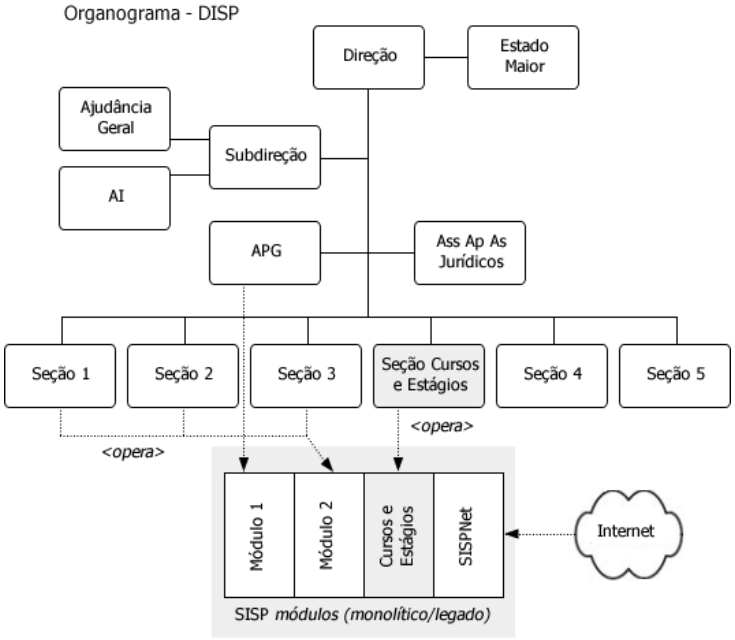


Figura B.7: Organograma DISP *vs* módulos do SISP.
(Fonte: própria)

T2.2.2.7 - Analisar o modelo de negócio. Com a investigação restrita ao módulo *cursos* do SISP (monolítico-alvo), observa-se que o principal problema com que a *Seção de Cursos e Estágios* precisa lidar é a *"seleção de candidato militar para realização de curso/estágio"*. Conforme mostra a Figura B.8, trata-se de um processo similar a qualquer processo seletivo, em que o candidato se inscreve para determinado curso, são aplicados diversos critérios e filtros, é feita uma ordenação ou priorização (meritocracia) e, caso o estudo seja aprovado pela chefia, finalmente o resultado é publicado em aditamento para conhecimento e devidas providências. Essa visão traz maiores evidências sobre o *"core"* (a razão de existir) do trabalho/negócio realizado pela organização. Além disto, contribui para identificar possíveis atores, entidades, eventos e suas relações, subsídios importantes a posterior modelagem de *contextos delimitados*.

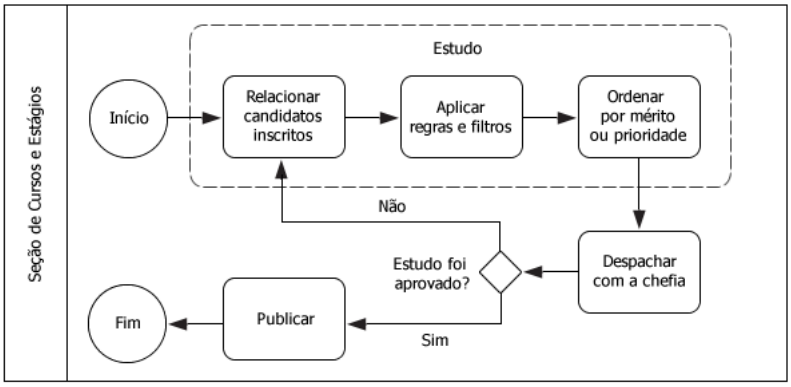


Figura B.8: Processo de seleção de candidato militar para curso/estágio.
(Fonte: própria)

T2.2.3.8 - Analisar o sistema monolítico legado. À medida que a análise avança, novos elementos sobre o monolítico surgem, expondo partes importantes do domínio. Nesta etapa, a proposta é conhecer um pouco mais sobre o módulo *cursos*. Por meio das interfaces do usuário (telas), buscou-se descrever o fluxo de navegação primário, conforme ilustra a Figura B.9.

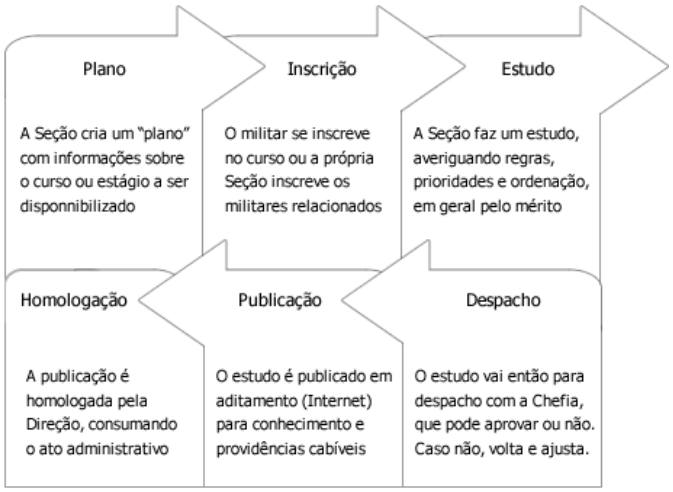


Figura B.9: Fluxo de navegação do *módulo cursos* do SISP.
(Fonte: própria)

O processo começa com a criação de um "plano" pela *Seção de Cursos e Estágios*. O termo "plano" é propositalmente genérico e se refere a um curso ou estágio a ser cadastrado e ofertado. Dentre as informações do plano (curso/estágio) estão as regras/filtros e o local de realização. Uma vez que o curso esteja ativo, tem início o período de inscrição. Dependendo do plano, a inscrição pode ser realizada *online* pelos militares que se enquadrem naquele universo ou internamente pela própria *Seção de Cursos e Estágios*.

Ao término das inscrições, é feito um "estudo" para analisar os candidatos e priorizá-los conforme critérios pré-estabelecidos, geralmente baseado na meritocracia (Valorização do Mérito - VM). O estudo após concluído segue então para apreciação da chefia, etapa conhecida como "despacho". Quando aprovado em despacho, o estudo é publicado em aditamento/boletim pela Internet para que os interessados tomem conhecimento e as devidas providências. O processo se encerra com a homologação da publicação, consumando o ato administrativo.

Do ponto de vista da análise estática do código-fonte do módulo *cursos* do SISP, buscou-se também reconstituir por engenharia reversa o diagrama de classes, esboçado na Figura B.10. Apesar da baixa fidelidade e a ênfase no relacionamento entre classes, identificar as principais entidades e dependências que participam do domínio da aplicação é um passo importante na concepção de um modelo realístico.

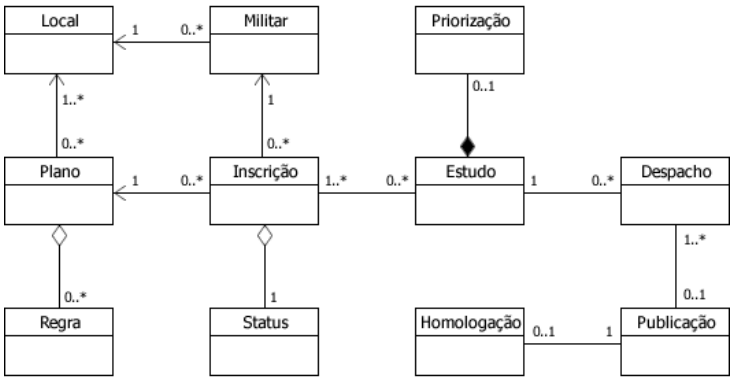


Figura B.10: Diagrama de classes do *módulo cursos* do SISP.
(Fonte: própria)

T2.2.4.9 - Analisar a persistência de dados. Os dados do módulo *cursos* do SISP integram uma imensa base de dados corporativa, o que na maioria dos casos significa pouca ou nenhuma autonomia em realizar as operações DML[3] ou DDL[4]. Isto pode ser um complicador em constituir microsserviços independentes e provavelmente irá requerer o uso de *patterns* como *"database view"*, *"wrapping service"* ou *"change data ownership"* discutidos anteriormente na Seção 2.3.5. Para resguardar a Instituição e o sigilo de dados, nenhum modelo de entidade e relacionamento - MER (do inglês *entity–relationship model*) será apresentado, sendo suficiente saber que, assim como na maioria dos sistemas legados, o SISP faz uso de uma base de dados relacional e monolítica, hospedada e gerenciada externamente à DISP. As tabelas e registros observados basicamente revelam entidades, relacionamentos e dados estruturados em conformidade à análise feita até o momento.

[3]DML - *Data Manipulation Language*. Ex: INSERT, DELETE e UPDATE
[4]DDL - *Data Definition Language*. Ex: CREATE, ALTER e DROP

T2.2.6.10 - Consolidar a análise do domínio. Mediante ao que tem sido analisado sobre o domínio *cursos e estágios* do SISP é possível inferir os principais subdomínios, *contextos delimitados* e suas relações, conforme ilustra o modelo da Figura B.11. A ideia é que o modelo seja uma representação abstrata do domínio e busque capturar aspectos relevantes. A partir deste esboço, pode-se também construir um *mapa de contexto* (do inglês *context map*) que expresse espaços de abrangência da linguagem ubíqua, divisão de responsabilidades e, sobretudo, que permitam posteriormente derivar microsserviços independentes e escaláveis [184], a exemplo da Figura B.12.

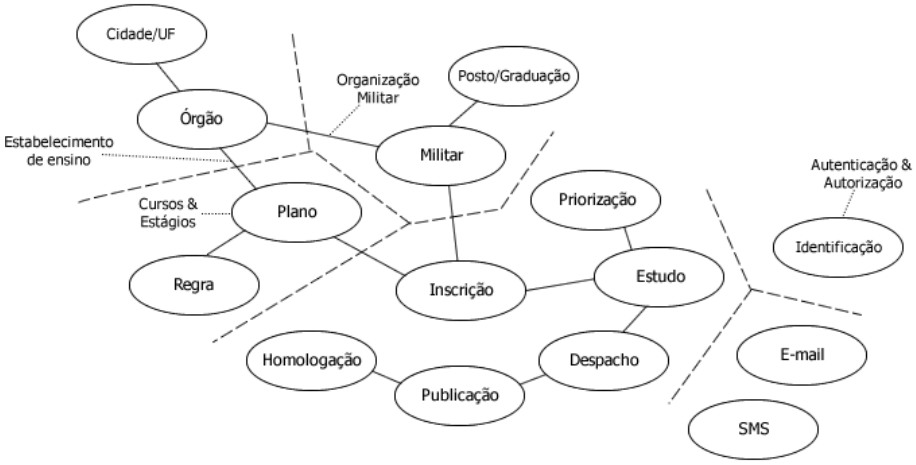


Figura B.11: Modelo do domínio *cursos e estágios* do SISP.
(Fonte: própria)

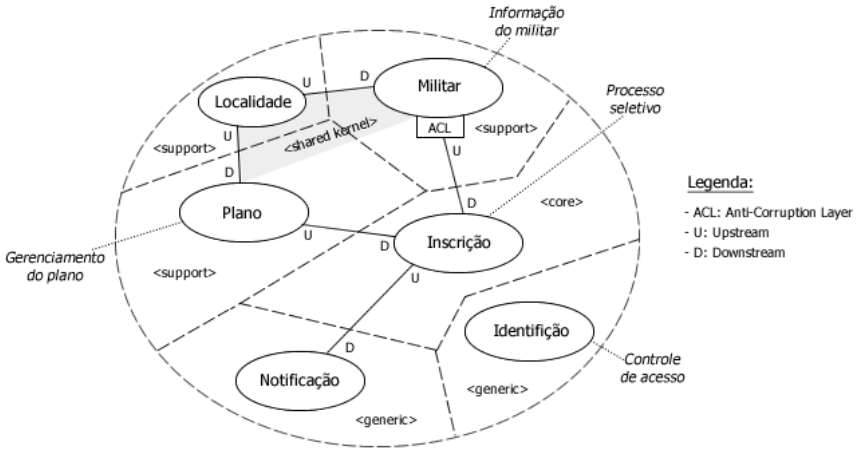


Figura B.12: Mapa de contexto de *cursos e estágios* do SISP.
(Fonte: própria)

B.3 Sprint 3

Tabela B.3: Sprint 3.

Sprint 3	**Meta:** decompor o legado em microsserviço			
Duração:	2 semanas	**Trabalho:** 6h/dia	**Início:** –/–/2020	**Fim:** –/–/2020
#	**Tarefa**			**Estimativa**
T3.3.4.11	Identificar os serviços candidatos			12h
T3.3.8.12	Especificar modelos que auxiliem o entendimento			12h
T3.3.3.13	Esboçar o desenho da API			12h
T3.3.1.14	Decompor monolítico baseado no DDD			24h

T3.3.4.11 - Identificar os serviços candidatos. Uma vez que se tenha uma ampla visão do monolítico e a definição da *feature* a ser migrada, é hora de selecionar e priorizar os serviços candidatos a serem desenvolvidos. Conforme recomenda a literatura, deve-se começar a experiência com microsserviços de modo gradual e incremental, preferencialmente escolhendo contextos ou subdomínios que não tenham ou tenham poucos relacionamentos (desacoplados) [2, 16]. A *feature* que vai servir como ponto de partida para a simulação é a "*inscrição militar em curso/estágio*"que é uma das muitas operações do SISP.

Conforme descreve o Capítulo 5, o MFC não exige que as atividades (pacotes de trabalho) sejam sequenciais, podendo inclusive ser concomitantes. Para conhecer melhor a fatia de negócio que será extraída do monolítico, alguns diagramas serão elaborados no decorrer da simulação, servindo de insumo para a atividade de "*Documentação e modelo*" (MFC/3.8). Essa espécie de engenharia reversa em que requisitos de usuário são inferidos a partir do monolítico, permitiu identificar alguns serviços de negócios primordiais, candidatos a serem desenvolvidos, descritos na Tabela B.4. Outros serviços de suporte como *redis*, *rabbitmq*, *haproxy* e *elastic stack* serão agregados a posteriori para prover novas capacidades à aplicação baseada em microsserviços.

Tabela B.4: Serviços de negócio a serem desenvolvidos.

Serviço	Descrição
identificação	Responsável pelo controle de acesso aos microsserviços
plano	Responsável pelo gerenciamento do plano (cursos/estágios)
inscrição	Responsável pelo processo seletivo de militares
notificação	Responsável pela notificação de eventos (email/SMS)

T3.3.8.12 - Especificar modelos que auxiliem o entendimento. O entendimento sobre "o que" e "como" desenvolver o microsserviço ocorre à medida que o monolítico vai se tornando conhecido. Criar modelos e diagramas pode ajudar nesse sentido, desde que não seja demasiadamente oneroso à equipe. No estudo em questão, verificou-se que para se candidatar ao processo seletivo de um curso/estágio do Exército, o fluxo principal (caminho feliz) do sistema, representado pelo caso de uso UC001 da Figura B.13, é aquele em que o militar deve:

1. Efetuar o login (Figura B.14);
2. Visualizar os cursos/estágios disponíveis (Figura B.15);
3. Selecionar e realizar a inscrição no curso de interesse (Figura B.16); e
4. Consultar inscrições já realizadas (Figura B.17).

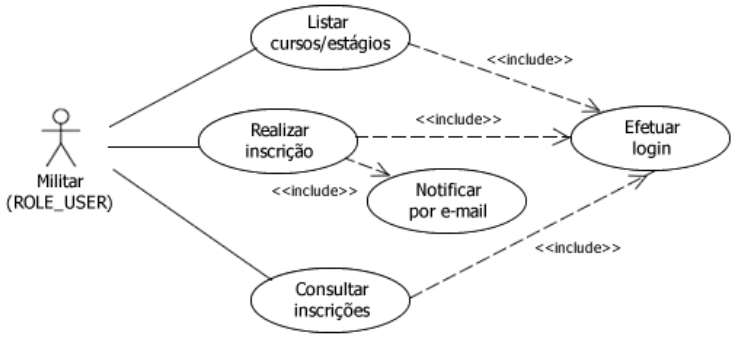


Figura B.13: Caso de uso UC001 - Inscrição militar em curso/estágio.
(Fonte: própria)

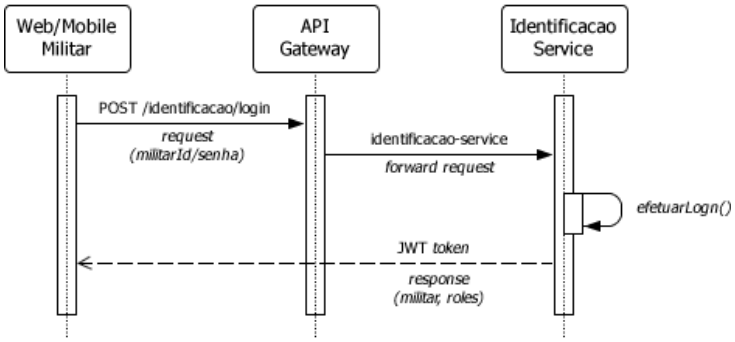


Figura B.14: Diagrama de sequência - Efetua login.
(Fonte: própria)

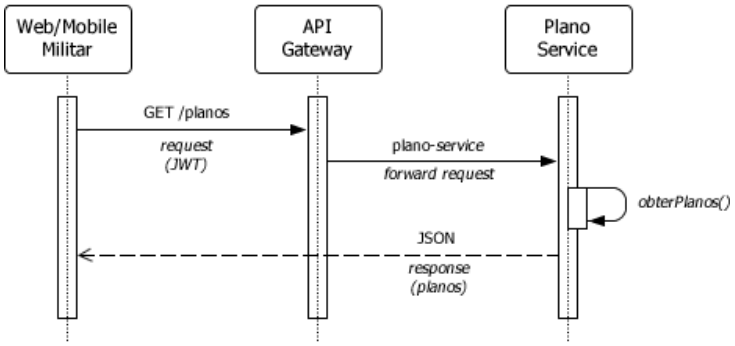


Figura B.15: Diagrama de sequência - Visualiza Planos (cursos/estágios). (Fonte: própria)

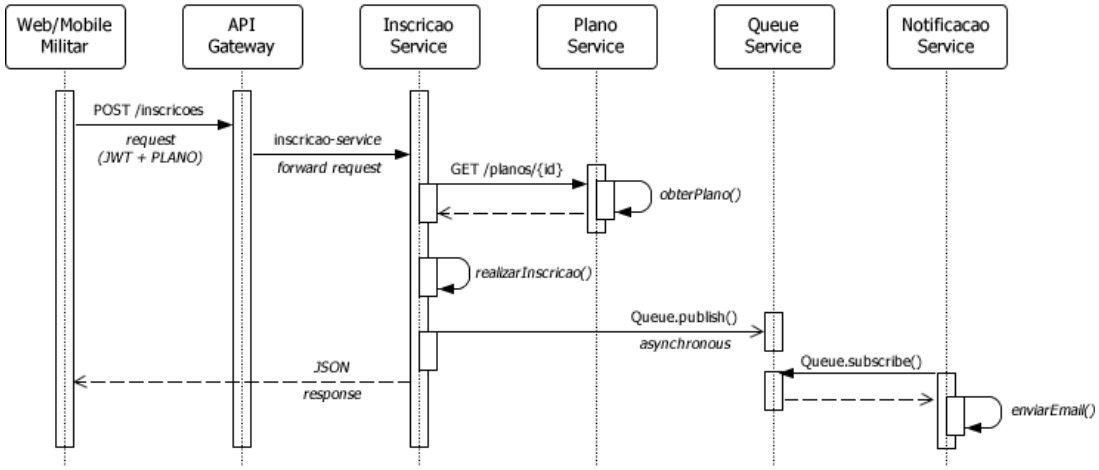


Figura B.16: Diagrama de sequência - Realiza inscrição. (Fonte: própria)

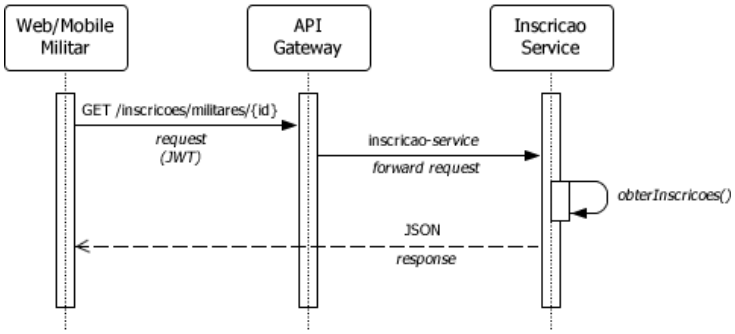


Figura B.17: Diagrama de sequência - Consulta inscrições. (Fonte: própria)

T3.3.3.13 - Esboçar o desenho da API. A interface do serviço deve ser pensada sob o ponto de vista dos consumidores do serviço e explicitamente expostas como um contrato [16]. Para manter a simplicidade da simulação, optou-se pelo desenho da API no estilo REST [22, 23, 32], cuja semântica e sintaxe são amplamente difundidas. Ferramentas como *Swagger*[5] podem ajudar na organização de APIs. A Figura B.18 mostra de modo simplificado as APIs dos serviços selecionados. Já a Figura B.19, exemplifica uma API em maiores detalhes quanto ao "*request/response*". Prevenindo futuras mudanças da API, a versão "v1" acrescida ao *path* do serviço, evitando eventual quebra de contrato.

Serviço: *identificação*

POST	/v1/identificao/login	Efetua o login do usuário

Serviço: *plano*

GET	/v1/planos	Retorna todos os planos disponíveis
GET	/v1/planos/militares/{id}	Retorna todos os planos do militar pelo id

Serviço: *inscrição*

POST	/v1/inscricoes	Cria uma inscrição
GET	/v1/inscricoes/militares/{id}	Retorna todas as inscrições do militar pelo id

Serviço: *notificação*

POST	/v1/notificacao/email	Envia um e-mail

Figura B.18: Desenho da API - Simplificado.
(Fonte: própria)

Serviço: *identificação*

Método	Caminho	Descrição
POST	/v1/identificao/login	Efetua o login do usuário

Parâmetro [requisição]	Tipo	Obrigatório	Descrição
militarId	Int(10)	Sim	Identidade do militar
senha	String(8)	Sim	Senha do militar

Parâmetro [resposta]	Tipo	Status HTTP [Sucesso]	Descrição
token	JSON	201	Token de acesso (JWT)

Figura B.19: Desenho da API Identificação *Service* - Detalhado.
(Fonte: própria)

[5]https://swagger.io/

T3.3.1.14 - Decompor monolítico baseado no DDD. No cenário proposto, a decomposição ocorre manualmente, analisando-se o domínio de negócio [40]. Nesse caso, são combinadas funcionalidades correlacionadas em uma única capacidade de negócio [109]. O ideal é que não se deixe "contaminar" a concepção do serviço com vícios do monolítico, o que costuma ser frequente já que o código-fonte legado nem sempre reflete a mesma funcionalidade do serviço [105]. Para isso, o *pattern* de Camada Anticorrupção [185] (do inglês *Anti-Corruption Layer Pattern*) pode ser necessário. Sabe-se também que o DDD auxilia o processo de decomposição para microsserviços [1, 3, 6, 16, 33, 131], sendo a linguagem ubíqua fundamental para originar os contextos delimitados, candidatos a microsserviços [16]. A Figura B.20 exemplifica a decomposição do domínio (Plano *Service*) a partir de princípios de DDD; e a Figura B.21 ilustra como os dados do domínio são tratados no processo de decomposição utilizando o *pattern* "*Database per service*" [7].

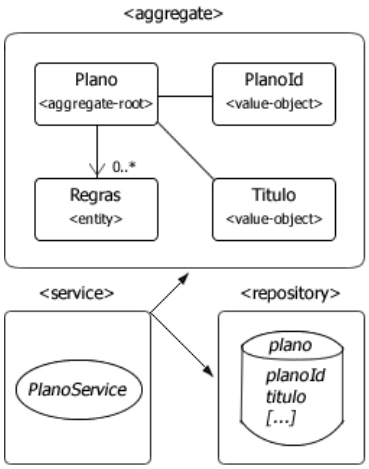


Figura B.20: Decomposição do domínio com princípios DDD (Exemplo).
(Fonte: própria)

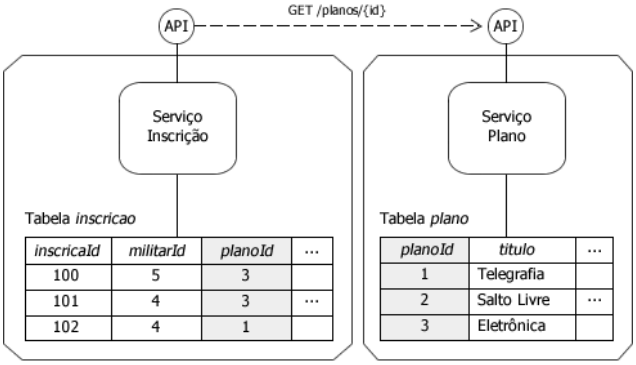


Figura B.21: Representação dos dados do domínio no processo de decomposição.
(Fonte: própria)

B.4 Sprint 4

Tabela B.5: Sprint 4.

Sprint 4	**Meta:** definir arquitetura, especificações e métricas de software			
Duração: 2 semanas	**Trabalho:** 6h/dia	**Início:** –/–/2020	**Fim:** –/–/2020	
#	**Tarefa**			**Estimativa**
T4.3.5.15	Definir práticas e padrões arquiteturais			36h
T4.3.7.16	Definir métricas de software			24h

T4.3.5.15 - Definir práticas e padrões arquiteturais. A arquitetura de um sistema pode ter diferentes perspectivas ou *design* [20]. Com relação à simulação, destaca-se:

a) Quanto à estrutura e camadas. Em termos de "*layers*" e estratégias DDD, separar uma aplicação em camadas ajuda a isolar o domínio de outras preocupações [107]. É importante preservar a responsabilidade de cada camada, ainda que não haja consenso sobre as estruturas de diretórios utilizadas para representá-las, ou nem mesmo da organização do código-fonte. Existe, por exemplo, uma grande discussão se o *Controller* deveria estar em UI ou Application. Neste estudo, a decisão de colocá-lo em UI está baseada na Figura 4.1 (Pág. 72) do livro *Domain-driven Design*, de Eric Evans. A Figura B.22 ilustra a tradicional arquitetura em camadas aplicável ao DDD.

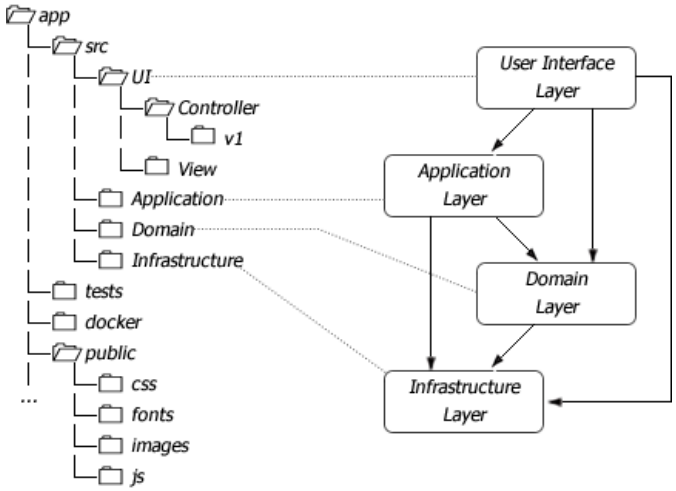


Figura B.22: Arquitetura em camadas tradicional comum ao DDD.
(Fonte: Baseado em Vernon [107])

Interface do usuário (*user interface layer*): trata das requisições e visualização do usuário. Não contém lógica de negócio, nem validações do modelo do domínio.

Aplicação (*application layer*): coordena operações (*workflow*) inerentes aos objetos de domínio, expressando casos de uso ou histórias de usuário.

Domínio (*domain layer*): possui toda a lógica de negócio. Isto inclui agregações, entidades, objetos de valor, eventos de domínio, interface de repositório, etc.

Infraestrutura (*infrastructure layer*): mantém mecanismos de persistência, mensagens e componentes técnicos que forneçam serviços de baixo nível para a aplicação.

b) Quanto à padronização do código-fonte. Embora possível construir microsserviços com diferentes linguagens de programação, a recomendação é que a organização estabeleça um limite [16]. No caso deste estudo, é mantida a linguagem de programação PHP, nativa do SISP. Isto diminui a curva de aprendizagem e facilita o aproveitamento de código legado. O *Framework Interoperability Group*[6], composto por membros de projetos como *Doctrine*, *Symfony* e *Laravel*, tem recomendado uma série de padrões por meio de PSR (do inglês *PHP Standard Recommendation*), incluindo estilos de codificação para garantir o alto nível de interoperabilidade (PSR-1 e PSR-12). Outras iniciativas como "*PHP The Right Way*"[7], SOLID[8] e IDEALS[9] também contribuem com a padronização de aspectos da modelagem do sistema e legibilidade do código-fonte.

c) Quanto à interação de componentes. Para construir uma aplicação distribuída e altamente escalável, esta seção esboça alguns modelos arquiteturais, sobretudo, ligados à orquestração de *containers*, automatização de processos (CI/CD) e monitoramento/alertas. Não existe um modelo correto, nem melhor [20]. Tudo depende da complexidade e aplicabilidade. A Figura B.23 mostra uma típica arquitetura utilizada em microsserviços.

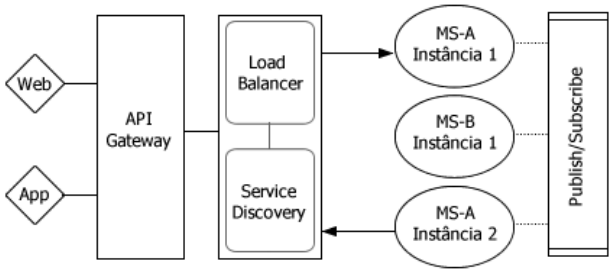


Figura B.23: Típica arquitetura de microsserviços.
(Fonte: própria)

[6]https://www.php-fig.org/
[7]https://phptherightway.com/
[8]http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
[9]https://www.infoq.com/articles/microservices-design-ideals/

Neste tipo de arquitetura, quando um microsserviço fica *online*, sua localização de natureza dinâmica (IP/porta) é registrada no "*service discovery*". As requisições que chegam são então centralizadas na "API *Gateway*" (para diferentes propósitos) e encaminhadas ao "*load balance*", que consulta o *service discovery* e faz o endereçamento aos serviços requisitados. Também é utilizado um barramento de mensageria para que os microsserviços possam trocar mensagens de modo assíncrono, favorecendo o desacoplamento.

Para simular um ambiente minimamente distribuído, foram utilizados na simulação dois *hosts* (computadores reais), descritos na Tabela B.6.

Tabela B.6: Computadores utilizados na simulação.

Característica	**Host 1**: *Genesis*	**Host 2**: *Exodus*
Produto	Dell Inc. Inspiron 7348	Dell Inc. OptiPlex 3040
SO	Debian 10 GNU/Linux 64 bits	Debian 10 GNU/Linux 64 bits
CPU	Intel(R) Core i7-5500U 2.40GHz	Intel(R) Core i5-6500T 2.50GHz
Memória	8Gb DDR3 1600 MHz Samsung	16Gb DDR3 1600 MHz Kingston
Disco	SSD 240GB Kingston	SSD 480GB Kingston

Cada host possui o *Docker* instalado e estão interconectados por uma rede *overlay*[10], formando um cluster *swarm*[11], conforme Figura B.24(a).

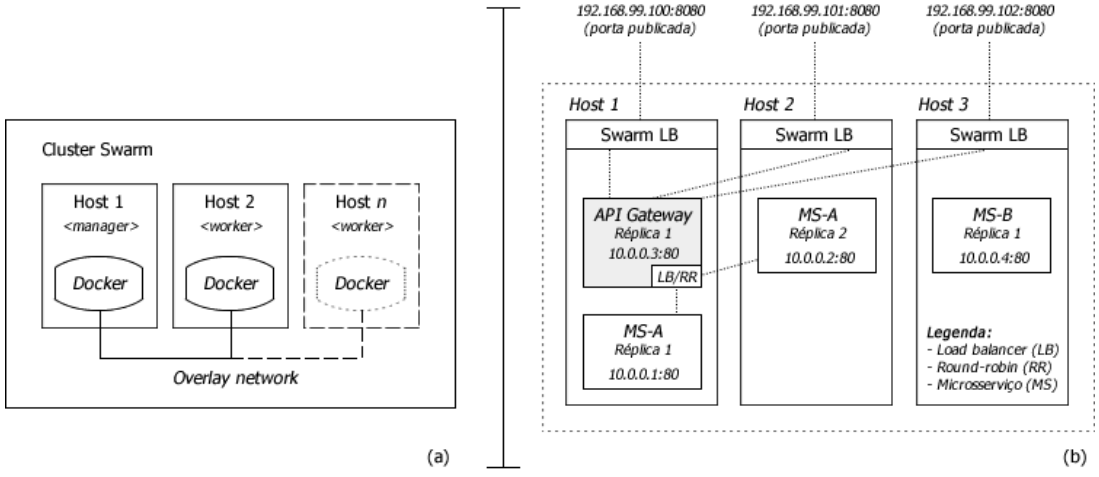


Figura B.24: Cluster swarm *(a)* e Routing mesh *(b)*.
(Fonte: baseado em *docs.docker.com* [186])

Neste cenário, todos os *hosts* participam de uma malha de roteamento (do inglês *ingress routing mesh*[12]) que aceita conexões em portas publicadas para qualquer serviço em execução no *swarm*, mesmo se não houver nenhuma tarefa em execução no *host* [186]. Na

[10]https://docs.docker.com/network/overlay/
[11]https://docs.docker.com/engine/swarm/
[12]https://docs.docker.com/engine/swarm/ingress/

prática, cada microsserviço é encapsulado em um *container* e não importa em qual *host* o *container* esteja rodando, o *swarm* é capaz de localizar o serviço requisitado. No exemplo da Figura B.24(b), apenas a *API gateway* tem uma porta publicada (8080) intermediando todas requisições que chegam. Os demais serviços rodam em uma rede privada, inacessível externamente. Outro fator importante é que o Docker tem um DNS embarcado, permitindo que os *containers* (serviços) se referenciem pelo nome, e não apenas IP. Essa infraestrutura pode facilmente se expandir, adicionando outros *hosts* ao cluster *swarm*.

d) Quanto à integração e entrega contínua (CI/CD). Embora haja poucos microsserviços inicialmente envolvidos neste estudo, é de grande valia automatizar o processo de entrega de software. Até porque, à medida que os microsserviços crescem em quantidade, torna-se impraticável gerenciá-los individualmente. Neste sentido, técnicas *DevOps* como integração e entrega contínua (CI/CD) são comumente usadas para automatizar o processo de desenvolvimento e operação [3]. No modelo da Figura B.25, é construído um *pipeline Jenkins*[13], isto é, um fluxo de atividades suportadas por diferentes *plug-ins* para colocar o software em produção de modo mais confiável e automático.

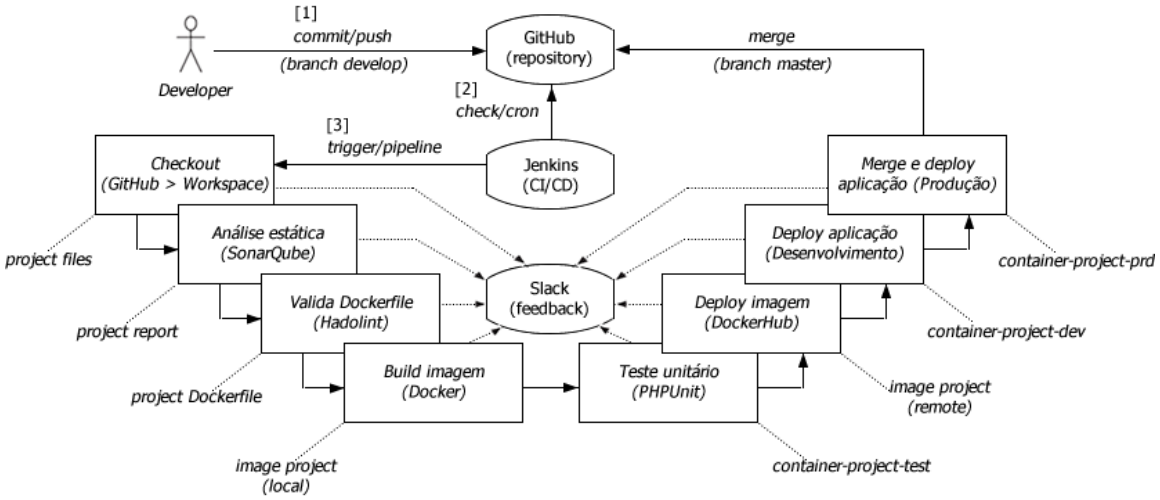


Figura B.25: Integração e entrega contínua (CI/CD).
(Fonte: própria)

Observa-se na figura que tudo começa com o *commit/push* do desenvolvedor no repositório de código-fonte, neste caso o *GitHub*[14]. O *Jenkins* de tempo em tempo (*scheduler*) verifica se houve mudanças no repositório. Quando algo muda, o *Jenkins* dispara o *pipeline* idealizado, executando automaticamente tarefas como análise estática, teste de unidade e *build/deploy* da aplicação em ambiente de desenvolvimento e produção, se for o caso.

[13]https://www.jenkins.io/
[14]https://github.com/

Havendo falha em qualquer etapa, o *pipeline* é imediatamente interrompido e comunicado à equipe de desenvolvimento, evitando que o software com defeito vá para produção. É importante notar que todas as etapas enviam *feedback* para uma central de mensagens, representada pelo *Slack*[15].

e) Quanto ao monitoramento e alertas. Em estágios avançados, tudo tende a ser mais complexo, sendo indispensável monitorar os microsserviços para prevenir eventuais falhas [16]. Isso pode ser feito por meio de coleta de dados em *logs* ou eventos, gerando *dashboards* e alertas quando necessário. Para essa finalidade, o presente estudo adota o *Elastic Stack*[16], uma solução *open-source* baseada em um conjunto de ferramentas para coletar, analisar e visualizar dados do sistema em tempo real. No modelo proposto pela Figura B.26, o *Filebeat* coleta métricas baseadas em *logs*, geralmente em "*/var/log/*.log*" e envia para o *Logstash*, responsável por fazer tratamento, filtro e transformação desses dados antes de repassar ao *Elastisearch* para armazená-los. A partir dos dados persistidos e indexados, o *Kibana* fornece uma série de visualizações em *dashboards*, inclusive com disparos de alertas encaminhados ao *Slack* sob condições pré-definidas. Todas essas ferramentas estão sendo utilizadas por meio de imagens/*containers* Docker, exceto o *Metricbeat*, instalado em cada *host* com o objetivo de obter métricas de processador, memória, rede e processos, entre outras.

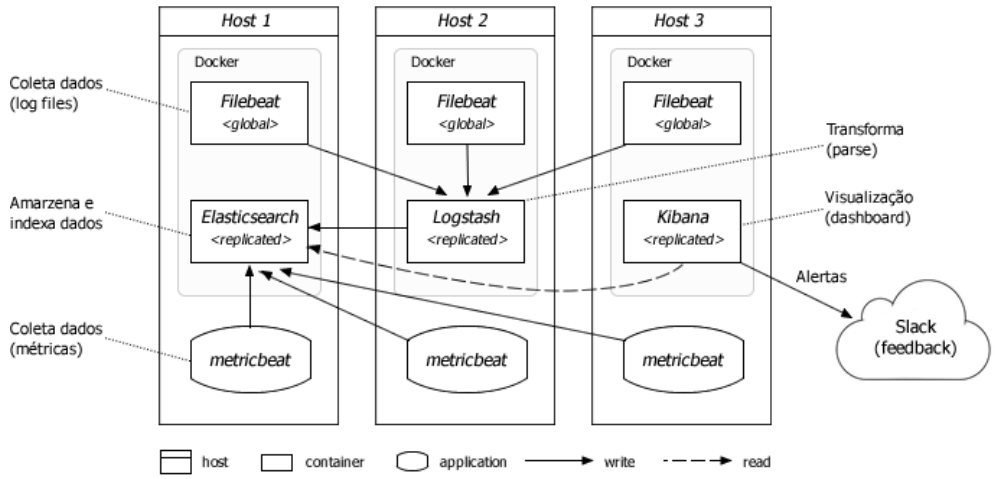


Figura B.26: Monitoramento e alertas.
(Fonte: própria)

[15]https://slack.com/
[16]https://www.elastic.co/

T4.3.7.16 - Definir os atributos de qualidade de software e métricas. A proposta de modernização do SISP por meio da arquitetura de microsserviços visa superar muitas das limitações monolíticas, principalmente relacionadas à *manutenibilidade* (facilitar a manutenção do sistema), *desempenho* (reduzir a lentidão sob alta demanda) e *interoperabilidade* (permitir a comunicação entre sistemas). Nesse sentido, pode ser útil mensurar o ganho pretendido em relação aos atributos de qualidade de software, formulando indicadores que permitam avaliar o progresso da migração. Infelizmente, não existe um valor de referência/padrão para a maioria das métricas, que podem variar conforme tamanho, domínio e tipos de sistemas [155]. Em todo o caso, o modelo *Goal Question Metric* - GQM descrito na Seção 2.3.6 tem contribuído para elucidar a identificação de métricas utilizadas nessa simulação, conforme mostram as Tabelas B.7, B.8 e B.9.

Tabela B.7: Métricas de manutenibilidade - GQM & QA.

META Melhorar os indicadores de manutenibilidade **PERGUNTAS** - Qual a taxa de complexidade do sistema? - Qual o grau de acoplamento e coesão? - Quanto o sistema é coberto por testes? - Quanto o código-fonte é documentado/comentado?	**QA** Manutenibilidade 
MÉTRICA	**FONTE**
Linhas de código	SonarQube
Quantidade de classes	SonarQube
Quantidade de funções	SonarQube
Taxa de comentários	SonarQube
Complexidade ciclomática	SonarQube
Complexidade cognitiva	SonarQube
Code smells	SonarQube
Taxa de dívida técnica	SonarQube
Número de vulnerabilidades	SonarQube
Problemas de segurança	SonarQube
Número de bugs	SonarQube
Duplicidade de código	SonarQube
Cobertura de testes	SonarQube

Tabela B.8: Métricas de desempenho - GQM & QA.

META Melhorar os indicadores de desempenho **PERGUNTAS** - Quantas requisições por segundos o sistema suporta? - Qual o tempo entre requisição e resposta? - Qual o número de requisições que falham? - Qual o consumo de recursos computacionais?	**QA** Desempenho 
MÉTRICA	**FONTE**
Requisições atendidas por segundo (*throughput*)	jMeter
Tempo entre *input* e *output* (*latency*)	jMeter
Eventos não processados no prazo (*missed events*)	jMeter
Percentual de consumo de memória RAM	jMeter
Percentual de consumo de CPU	jMeter

Tabela B.9: Métricas de interoperabilidade - GQM & QA.

META Melhorar os indicadores de interoperabilidade **PERGUNTAS** - Existe mecanismo de descoberta/consulta do serviço - Qual a quantidade de interfaces expostas (API)? - Existe padrão de formato de dados aberto? (XML) - Existe padrão de codificação de caracteres? (UTF-8)	**QA** Interoperabilidade 
MÉTRICA	**FONTE**
Mecanismo de descoberta/consulta do serviço	Inspeção
Quantidade de interfaces expostas (API)	Inspeção
Padrão de formato de dados aberto	Inspeção
Padrão de codificação de caracteres	Inspeção

B.5 Sprint 5

Tabela B.10: Sprint 5.

Sprint 5	**Meta:** desenvolver microsserviço "identificação"					
Duração:	2 semanas	**Trabalho:**	6h/dia	**Início:**	–/–/2020	**Fim:** –/–/2020
#	**Tarefa**					**Estimativa**
T5.5.1.17	Criar ambiente operacional mínimo de suporte					12h
T5.4.2.18	Criar estrutura e codificação base					12h
T5.4.1.19	Criar teste de unidade para o serviço identificação					6h
T5.4.2.20	Criar o microsserviço identificação					30h

T5.5.1.17 - Criar ambiente operacional mínimo de suporte. Antes da codificação propriamente dita, é necessário ter um ambiente operacional minimamente funcionando, de modo a suportar o microsserviço que está sendo construído. Para o estudo em questão, isto inclui, por exemplo, a instalação do Docker[17], utilizado para criar a imagem e *containers* e do Redis[18] (v6.0.5), que será utilizado como banco de dados na porta 6379 para atender o serviço Identificação. Redis é um *open-source* no estilo "*key-value*" que armazena estrutura de dados em memória com possibilidade de persistência em disco. É importante lembrar que o método MFC é constituído de ciclos e que, portanto, esse ambiente inicialmente estabelecido (Figura B.27) será gradualmente aperfeiçoado.

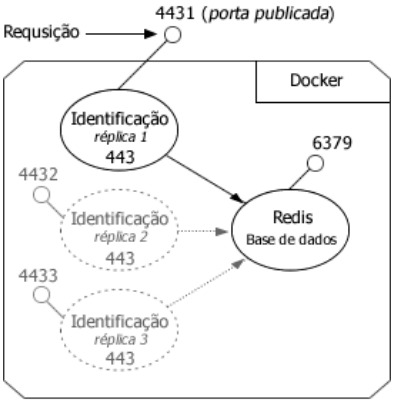


Figura B.27: Ambiente inicial de *containers*/serviços.
(Fonte: própria)

T5.4.2.18 - Criar estrutura e codificação base. Nesta etapa inicia a implementação de fato. Porém, é normal primeiro criar uma estrutura mínima de suporte e codificação, servindo de ponto de partida para outros serviços. Não existe um jeito totalmente certo ou errado de se fazer isto. Por exemplo, muitos desenvolvedores segmentam o diretório "Domain" em *Entities*, *Interfaces*, *Services*, etc. Embora cenários complexos possam requerer estruturas complexas, este trabalho optou por manter as coisas simples, com eventuais variações, seguindo o fluxo de chamadas internas ao serviço conforme ilustra a Figura B.28. É oportuno lembrar que, por uma questão de sigilo da organização, os códigos-fontes apresentados podem ser parciais ou terem partes omitidas. Também cabe registrar que a simulação não irá cobrir a manipulação de *templates* ou *views*.

[17]https://docs.docker.com/get-docker/
[18]https://redis.io/

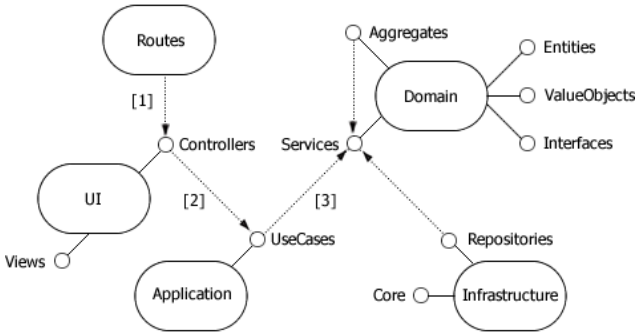


Figura B.28: Fluxo de chamadas internas ao serviço.
(Fonte: própria)

a) Core. Para este estudo especificamente, foi criado um diretório "*Core*", também conhecido como "*Shared*", em que reside uma série de implementações comuns a outros serviços. Em geral, são *libraries* ou *helpers* que não implementam regras de domínio, mas geralmente dão suporte ao serviço (infraestrutura). Não se trata de recursos compartilhados entre serviços gerando dependência e acoplamento. Esses códigos são injetados no serviço no momento de *build*, de modo que cada serviço detém uma cópia *readonly* exclusiva do *Core*. Em teoria, isto poderia ferir o princípio DRY (do inglês *Don't Repeat YourSelf*) devido a duplicidade de código. Entretanto, as chances de que mudanças em parte do sistema (*Core*) afetem outras partes (réplicas do *Core*) são raríssimas, dada a natureza estável do código de suporte. A ideia é que só exista um único *Core* central passível de mudanças (*write*), cujas réplicas sempre são sobrescritas, evitando perda de manutenibilidade. De qualquer modo, esta é uma daquelas escolhas do tipo *tradeoff*, em que se pesa vantagens e desvantagens de uma ação; afinal o que é pior, ter um único código compartilhado, gerando acoplamento entre os serviços que o consomem ou manter códigos ambíguos em múltiplas instâncias do serviço, dificultando a manutenção?

b) Correlation-Id. Um tipo de software que fornece serviços e recursos comuns a aplicações é o *middleware* [187]. Dentre os *middlewares* utilizados neste trabalho, está o *CorrelationIdMiddleware*. Sua tarefa é propagar um UUID (do inglês *Universally Unique Identifier* - UUID) em consecutivas chamadas entre microsserviços, facilitando uma posterior rastreabilidade. Conforme mostra o Código B.1, a cada requisição o *middleware* tenta extrair do HEADER o parâmetro "*X-Correlation-Id*" contendo o provável UUID. Se nada existir, um novo UUID é gerado. Do contrário, o corrente UUID é retransmitido via cabeçalho HTTP, algo como "*X-correlation-id: 3c520cf9-31a9-47b6-a8e7-f87be3946299*".

Código B.1: *Middleware Correlation-Id*

```php
<?php
class CorrelationIdMiddleware
{
    public function __invoke(Request $request, RequestHandler $handler)
    {
        $response = $handler->handle($request);
        $correlationId = $request->getHeaderLine('X-Correlation-Id');
        if (empty($correlationId)) {
            $correlationId = Random::UUID();
        }
        return $response->withHeader('X-Correlation-Id', $correlationId);
    }
}
?>
```

c) Token (JWT). Outro *middleware* desenvolvido é o *AuthMiddleware*, responsável pela autenticação e autorização, através de *tokens* JWT[19] (do inglês *JSON Web Token*). JWT é um padrão aberto (RFC 7519) para transmitir informações com segurança entre duas partes. Essas informações podem ser verificadas quanto à integridade, pois são assinadas digitalmente. De acordo com a Figura B.29, o *token* é obtido após o primeiro login bem-sucedido com o par "*usuário/senha*". A partir de então, todas as requisições subsequentes transmitem o *token* via cabeçalho HTTP *Authorization*, ficando a cargo do *AuthMiddleware* validar o *token* e garantir acesso ao recurso do microsserviço.

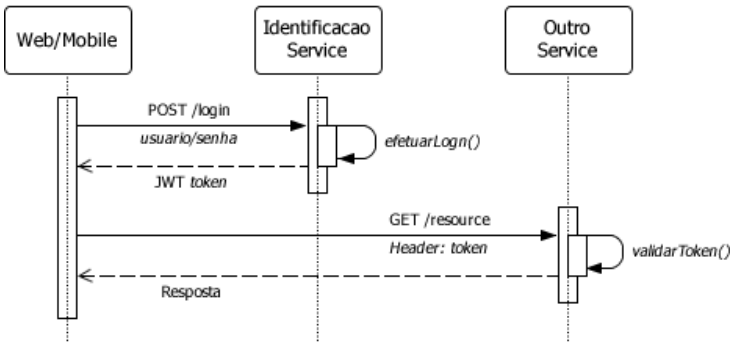


Figura B.29: Autenticação com JWT.
(Fonte: própria)

[19]https://jwt.io/

A Figura B.30 mostra que o JWT consiste em três partes separadas por "." (ponto): (i) *header*, trata do algorítimo de assinatura empregado (Ex: HS256); (ii) *payload*, contém os dados a serem transmitidos; e (iii) *signature*, combina *header*, *payload* e uma "*secret key*", assinados pelo algoritmo especificado no *header*. Os microsserviços conhecedores da "*secret key*" conseguem validar a assinatura, tendo a garantia de que a mensagem não foi alterada ao longo do caminho. Um *expiration time* (*exp*) também pode ser definido para limitar a vida útil do *token* e ampliar a segurança.

Figura B.30: Composição do JWT.
(Fonte: baseado em jwt.io)

T5.4.1.19 - Criar teste de unidade para o serviço identificação. Em geral, testes são negligenciados e erros só aparecem tardiamente. Porém, escrever testes antes mesmo de implementar a aplicação (*test-first*) traz benefícios como melhor compreensão do sistema, redução de defeitos e processo de modificação mais seguro [188]. Além disso, a execução automatizada de testes é uma das chaves para o sucesso *DevOps* [92]. Na prática, pode-se começar com o ciclo TDD (do inglês *Test Driven Development*): (i) escrever um teste que falhe; (ii) fazer o código funcionar; e (iii) eliminar redundâncias (refatoração).

Utilizando esse princípio e uma série de padrões já discutidos, a seguir são apresentados três códigos em relação ao serviço "Identificação" a ser criado. Primeiro, o Código B.3

exemplifica um simples teste de unidade presumindo construir uma classe que realize o caso de uso na qual o *usuário efetua login*. Segundo, é feita a implementação de fato da classe idealizada no teste em questão, mostrada parcialmente no Código B.3. E por último, o Código B.4 representa a saída do teste executado em terminal. Se os valores de retorno coincidem com os resultados esperados, diz-se que o teste passou, do contrário falhou. Existem abordagens mais sofisticadas como BDD (do inglês *Behavior Driven Development*) que, a partir de histórias de usuário (*features*), utilizam linguagem nativa para descrever cenários de testes, porém, não tratado neste trabalho.

Código B.2: Teste de unidade *UsuarioEfetuaLogin*

```php
<?php
use Application\EfetuaLogin;
use PHPUnit\Framework\TestCase;

class EfetuaLoginTest extends TestCase
{
    public function testUsuarioEfetuaLogin()
    {
        $login = new EfetuaLogin();
        $data  = $login('1122334455', '12345');
        self::assertArrayHasKey('token', $data);
        self::assertNotEmpty($data['token']);
    }
}
?>
```

Código B.3: Implementação *EfetuaLogin*

```php
<?php
namespace Application;

use Domain\{Usuario, IdentificacaoService};

class EfetuaLogin
{
    public function __invoke($militarId, $senha)
    {

        # Identificar: o usuário existe?
        $service = new IdentificacaoService();
        $usuario = $service->identificarUsuario($militarId);
```

```
14
15          # Autenticar: é quem diz ser?
16          $autenticacao = false;
17          if ($usuario) {
18              $autenticacao = $service->autenticarUsuario($usuario, $senha);
19          }
20
21          # Autorizar: o que pode fazer?
22          $autorizacao = false;
23          if ($autenticacao) {
24              $autorizacao = $service->autorizarUsuario($usuario);
25          }
26
27          # Gerar token JWT: assinatura para acesso aos serviços
28          $token = null;
29          if ($autenticacao && $autorizacao) {
30              $token = $service->gerarToken($usuario);
31          }
32
33          # Dados retornados
34          return [
35              'usuario' => $usuario,
36              'token'   => $token
37          ];
38      }
39  }
40  ?>
```

Código B.4: Execução de testes - *Identificação Service*

```
$ ./vendor/bin/phpunit tests/
PHPUnit 9.2.3 by Sebastian Bergmann and contributors.
1 / 1 (100%)
Time: 00:00.056, Memory: 4.00 MB
OK (1 test, 2 assertions)
```

T5.4.2.20 - Criar o microsserviço identificação. Dentro da proposta de realizar a "*inscrição militar em curso/estágio*" (UC001), o papel do serviço de Identificação se limita a autenticar e autorizar o usuário fornecendo um *token* de acesso (JWT), caso o login seja bem-sucedido. Isto é feito conforme API estabelecida: POST */v1/identificacao/login*.

a) Dados. Em termos de dados para o Identificação *Service*, optou-se por utilizar o Redis como um serviço (*container*). O banco foi populado com uma amostra simulada de 20 mil registros contendo informações básicas de militares em JSON, a exemplo do Código B.5. A proposta é que seja mantido um *sync* com a base dados corporativa, tornando o processo de *login* extremamente mais rápido, já que os registros são exclusivos de usuários do sistema (SISP) e o Redis roda em memória, com eventual persistência em disco.

Código B.5: Tupla login do *Redis* (Exemplo)

```
{
  "militarId": "1122334455",
  "name": "Fulano",
  "rank": "2º Ten",
  "email": "fulano@mail.com",
  "password": "1e223bb05616fc5c1074e7b3e...",
  "createdAt": "2020-01-10",
  "status": "1",
  "roles": ["ROLE_USER", "ROLE_ADMIN"]
}
```

b) Implementação. A requisição de login chega ao serviço na porta externa 4431 (443 *Nginx*), onde é feito um *match* da URN com as rotas do sistema, conforme mostrado na *linha 4* do Código B.6. Vale destacar na *linha 3* a chamada ao *middleware* responsável por criar ou propagar o *Correlation-Id*, discutido anteriormente. Basicamente o que o serviço faz é comparar os dados de entrada do usuário (login) com os registros Redis. O *Controller* então se encarrega de receber, tratar e encaminhar a requisição para processamento do login, retornando um *token* associado ao *status code* "*201 Created*" em caso de sucesso, ou "*401 Unauthorized*", em caso de acesso não autorizado.

Código B.6: Mapeamento de rotas - *Identificação Service*

```
<?php
$app = AppFactory::create();
$app->add(new Core\CorrelationIdMiddleware);
$app->post('/v1/identificacao/login',
 ↪  'Controller\IdentificacaoController:login');
$app->run();
?>
```

c) Containerização. Na raiz do projeto de cada microsserviço existe um arquivo chamado Dockerfile que descreve passo a passo como a *imagem* Docker da aplicação deve ser construída. O Código B.7 mostra o Dockerfile do serviço Identificação, que é similar (não igual) para os demais serviços. De modo sucinto, o conjunto de comandos diz que a partir de um sistema operacional Debian (*linha 1*), será (i) instalado alguns pacotes, como Nginx e PHP (*linhas 3-8*), (ii) copiado arquivos customizados (*linhas 10-15*) e (iii) inicializado *daemons* (linha 17). Destaca-se a *linha 15*, responsável por copiar todos os arquivos do projeto local para o diretório de publicação do *webservice* Nginx dentro do *container*. A partir do comando `docker build -t unbmaster/identificacao:1.0 .`, a imagem é criada segundo o Dockerfile. Uma vez que esta imagem é executada, tem-se um *container* com "identificação Service" operando plenamente. Um *container* é, portanto, uma instância de uma imagem em execução, e pode ser gerado pelo comando `docker run -d -p 4431:443 unbmaster/identificacao:1.0`. As imagens e *containers* Docker desta etapa da migração estão listadas nos Códigos B.8 e B.9, respectivamente. Para melhor visualização, algumas colunas foram suprimidas. O resultado é um *container* com Debian, Ngnix, PHP e o código-fonte do microsserviço, tudo em 180MB, o que pode ainda ser otimizado.

Código B.7: Dockerfile - Identificação *Service*

```
1  FROM debian:stable-slim
2
3  RUN apt-get update -y \
4      && apt-get install --no-install-recommends -y \
5      nginx=1.14.2-2+deb10u1 php-fpm=2:7.3+69 \
6      php-xml=2:7.3+69 \
7      && apt-get clean \
8      && rm -rf /var/lib/apt/lists/*
9
10 COPY ./docker/nginx/custom-default  /etc/nginx/sites-available/default
11 COPY ./docker/nginx/certificate.crt /etc/ssl/certificate.crt
12 COPY ./docker/nginx/private.key     /etc/ssl/private.key
13 COPY ./docker/php/custom-www.conf   /etc/php/7.3/fpm/pool.d/www.conf
14 COPY ./docker/php/custom-php.ini    /etc/php/7.3/cli/php.ini
15 COPY ./                             /var/www
16
17 CMD /etc/init.d/php7.3-fpm start && nginx -g "daemon off;"
```

Código B.8: Docker image ls [progresso#1]

```
$ docker image ls
IMAGE ID            REPOSITORY                      TAG                 SIZE
0606b4a873a7        unbmaster/identificacao         1.0                 180MB
5b9cab93cc2f        unbmaster/redis                 1.0                 127MB
6e1ff7cb748e        debian                          stable-slim         69.2MB
```

Código B.9: Docker container ls [progresso#1]

```
$ docker container ls
CONTAINER ID    IMAGE                           NAMES               PORTS
b9ff1b1bdc45    unbmaster/identificacao:1.0     identificacao       4431->443/tcp
c6f983c47f74    unbmaster/redis:1.0             redis               6379/tcp
```

d) Integração. Uma vez que o serviço esteja implementado, é preciso avaliar uma forma de integrá-lo com o sistema legado. Para isso, o microsserviço precisa estar disponível *online*. Em relação à *feature login*, não há como suprimir a autenticação do legado pela autenticação utilizada pelos microsserviços, pois além de incompatíveis, o legado precisa continuar funcionando para atender o sistema. Neste caso, optou-se por manter o método de *login* original do legado, acrescentando um novo fragmento de código visto na *linha 14*. Com isto, é obtido o *token* remotamente e armazenado na *session* da aplicação para posterior invocação aos microsserviços, conforme Código B.10.

Código B.10: Login do legado (SISP) adaptado para obter o *token*

```
1  <?php
2  class C_Login extends CI_Controller {
3
4      public function login() {
5
6          // Autenticação original do legado
7          $idt = $this->input->post('idt');
8          $pwd = $this->input->post('pwd');
9          $sql = "EB.LOGIN(?,?) AS SID FROM DUAL";
10         $sid = $this->db->query($sql, [$idt, $pwd])->result_array();
11
12
```

```
13          // Adaptação para obter token de acesso aos microsserviços
14          $login = API::post(
15              'https://<url>/v1/identificacao/login',
16              ['militarId' = $idt, 'senha' = $pwd]
17          );
18          $this->session->set_userdata('token', $login['token']);
19          ...
20      }
21  }
22  ?>
```

B.6 Sprint 6

Tabela B.11: Sprint 6.

Sprint 6	**Meta:** prover malha de serviços e suporte					
Duração:	2 semanas	**Trabalho:**	6h/dia	**Início:**	–/–/2020	**Fim:** –/–/2020
#	**Tarefa**					**Estimativa**
T6.5.1.21	Implantar *cluster swarm* e API *gateway*					30h
T6.5.3.22	Implantar integração e entrega contínua					30h

T6.5.1.21 - Implantar cluster swarm e API gateway. Neste ponto da migração em que já se tem um ambiente operacional mínimo e um microsserviço *online*, a equipe de desenvolvimento pode decidir por produzir outros microsserviços ou prosseguir no ciclo do método MFC, implantando práticas *DevOps* como integração e monitoramento contínuos. Para maior fluidez do estudo, optou-se pela segunda alternativa, o que significa expandir um pouco mais a arquitetura inicialmente estabelecida (Figura B.23). Portanto, nesta etapa é criado um *cluster swarm* para interconectar outros *hosts*, bem como é adicionado uma API *Gateway* para centralizar as requisições aos microsserviço. Embora centralizador, a API *Gateway* não pode ser considerado um SPoF (do inglês *Single Point of Failure*), pois tem a capacidade de escalar, ou seja, de haver múltiplas instâncias, mitigando possíveis falhas.

Antes de tratar do *swarm*, é convencional criar uma rede com driver *overlay* com o comando `docker network create -d overlay -attachable app-net`, onde *app-net* é o nome atribuído à rede, possibilitando interligar múltiplos *hosts* que rodam *daemons* Docker. Feito isto, o cluster *swarm* pode então ser facilmente inicializado pelo comando `docker swarm init -advertise-addr=192.168.0.138`, onde 192.168.0.138 é o IP do

host referenciado como *manager*, a qual outros *hosts* participantes do *swarm* se conectarão. No modo *swarm*, para *deploy* a imagem de uma aplicação, deve-se criar um serviço (do inglês *service*[20]). Serviço é, portanto, qualquer programa executável rodando de modo distribuído, sendo escalonado entre os *hosts* do *swarm* na forma de uma ou mais tarefas replicadas (do inglês *tasks*). Cada tarefa invoca exatamente um *container*. A Figura B.31 ajuda a entender essa dinâmica.

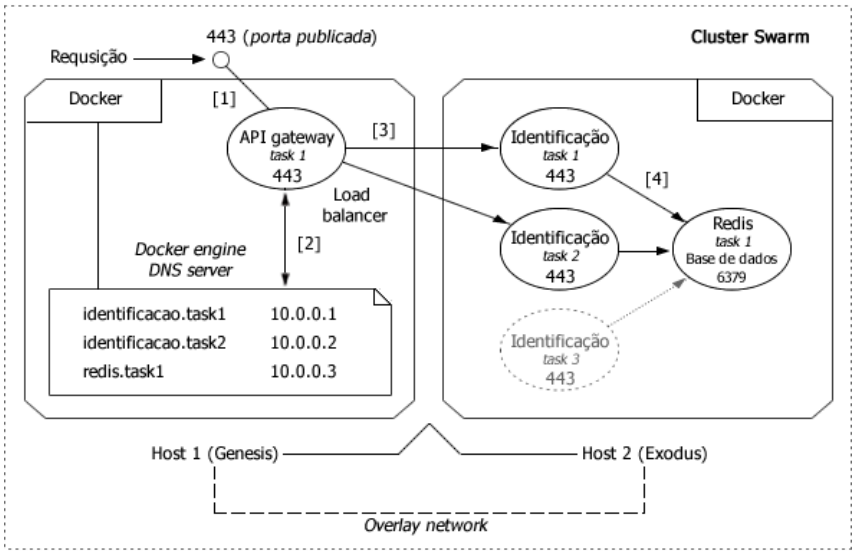


Figura B.31: Evolução do ambiente inicial: cluster swarm e API gateway.
(Fonte: própria)

Neste novo ambiente é utilizado uma imagem Docker do HAProxy[21] (2.0.15) na porta 443, para fazer o papel de API *Gateway* e *load balance*. A proposta é que a API *Gateway* rode como um serviço, sendo o único a prover uma porta publicada (443), conforme mostra a *linha 6* do Código B.11, utilizado para criar o servidor de borda. Os demais (micros)serviços são *deployed* sem porta publicada e inicializados em modo DNSRR (DNS *Round Robin*[22]), que é uma forma de *bypass* o *routing mesh* do Docker *swarm*, como definido no Código B.12. Em consequência, os serviços se mantêm isolados em uma rede interna acessível apenas pelo HAProxy que passa a fazer o *load balance*, reencaminhando as requisições baseadas no nome do serviço fornecido pelo DNS do Docker. O Código B.13 detalha como isto é feito. Basicamente, todas as requisições chegam pelo *frontend* na porta 443. Se a URL coincidir com o nome do serviço (*identificacao*), então entra em cena o *backend identificacao_prd*. Este *backend* é responsável por despachar a requisição para o serviço *identificacao-prd:443* conforme endereçamento do "*resolvers* docker" (DNS).

[20]https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/
[21]https://www.haproxy.com/
[22]https://docs.docker.com/network/overlay/

Código B.11: Deploy HAProxy *Service* / API gateway + load balance

```
docker service create \
  --mode replicated \
  --replicas 1 \
  --name haproxy \
  --network app-net \
  --publish published=443,target=443,protocol=tcp,mode=ingress \
  --mount type=bind,src=./,dst=/usr/local/etc/haproxy/,ro=true \
  --dns=127.0.0.11 \
  haproxytech/haproxy-debian:2.0
```

Código B.12: Deploy Identificação *Service*

```
docker service create \
  --name identificacao \
  --mode replicated \
  --replicas 1 \
  --network app-net \
  --endpoint-mode dnsrr \
  unbmaster/identificacao:1.0
```

Código B.13: Parcial *haproxy.cfg* que faz o redirecionamento ao serviço

```
...
resolvers docker
    nameserver dns1 127.0.0.11:53

frontend www-https
    bind *:443 ssl crt /usr/local/etc/haproxy/unbmaster.pem
    use_backend identificacao_prd if { path_reg \/v[0-9]+\/identificacao }

backend identificacao_prd
    balance roundrobin
    server-template identificacao- 5 identificacao-prd:443 check resolvers docker
...
```

T6.5.2.22 - Implantar integração e entrega contínua. Presumindo que os microsserviços irão aumentar em número, é preciso avançar nas práticas DevOps em termos

de automação. Neste sentido, tem-se utilizado Jenkins[23] com a finalidade de automatizar a construção (*building*), teste e implantação (*deploying*) do software. Para executar o modelo de integração e entrega contínua (CI/CD) anteriormente proposto e descrito pela Figura B.25, este trabalho adota para cada projeto de microsserviço um repositório próprio no GitHub, versionados em duas *branches*: *master* e *develop*, isto é, produção e desenvolvimento, respectivamente. Dependendo das tarefas que se pretende automatizar, é preciso a instalação de *plug-ins* ou sistemas de software *on-premises*. Alguns desses *plug-ins* Jenkins instalados, foram o Docker, SonarQube Scanner e Slack Notification. Também foram utilizados containers Docker do SonarQube[24] (porta 9001) para análise estática do código-fonte, Hadolint[25] (*bash*) para validação de Dockerfile e PHPUnit[26] para testes de unidade em PHP.

O Jenkins possibilita a criação de *jobs* do tipo *pipeline*, que funciona como um encadeamento de estágios (do inglês *stage*) e passos (do inglês *steps*) escritos na linguagem Groovy[27]. O que torna isto ainda mais poderoso é a possibilidade de executar *scripts shell* de linha de comando. Por exemplo, o Código B.14 mostra o *pipeline* em que é feita a validação do Dockerfile com Hadolint e, na sequência, o *build* da imagem Docker do microsserviço desenvolvido, tudo automaticamente. Além disto, é possível disparar alertas para o Slack, mantendo a equipe de desenvolvimento informada em tempo real sobre o processo de *build*, teste e *deploy*, apta a fazer as intervenções necessárias.

Código B.14: Jenkins *pipeline*: valida Dockerfile e *build* imagem Docker

```
1  stage('Valida Dockerfile (Hadolint)') {
2      steps {
3          script {
4              try {
5                  sh 'docker run --rm -i hadolint/hadolint < Dockerfile'
6                  sh 'exit_code=$?; if [ $exit_code -ne 0 ]; then exit 1; fi'
7              } catch (Exception e) {
8                  slackSend color: "danger", message: "Build ${BUILD_NUMBER}:
                    ↪  Dockerfile com problemas de sintaxe"
9                  sh "echo $e; exit 1"
10             }
11         }
12     }
13 }
```

[23]https://www.jenkins.io/
[24]https://www.sonarqube.org/
[25]https://github.com/hadolint/hadolint
[26]https://phpunit.de/
[27]http://www.groovy-lang.org/

```
14
15  stage('Build imagem (Docker)') {
16      steps{
17          script {
18              try {
19                  dockerImage = docker.build registry + ":${BUILD_NUMBER}"
20              } catch (Exception e) {
21                  slackSend color: "danger", message: "Build ${BUILD_NUMBER}: build
                     ↪  da imagem Docker falhou"
22                  sh "echo $e; exit 1"
23              }
24          }
25      }
26  }
```

Uma vez concluída a configuração CI/CD, sempre que o desenvolvedor fizer um *commit/push* na branch *develop*, o Jenkins dispara o pipeline executando todos os estágios e passos definidos. Isso significa dizer que o código-fonte do GitHub é baixado (*checkout*) para um repositório local do Jenkins (*workspace*) e processado pelo SonarQube em busca de *bugs*, vulnerabilidades e *code smells*. Não havendo problemas, o pipeline segue com o Hadolint validando o Dockerfile, arquivo a qual o Docker utiliza na sequência para criar a imagem do microsserviço desenvolvido. Cabe destacar que o número de *build* fica atrelado a *tag* da imagem, tornando possível distingui-la de outras. Com a imagem pronta, um *container* temporário é criado para executar o teste de unidade com PHPUnit, algo como `docker exec -i container-test ./var/www/vendor/bin/phpunit /var/www/tests`. Se o teste passar, a imagem é enviada para o repositório DockerHub, ficando disponível para os desenvolvedores. Por fim, é feito o *deploy* da aplicação, isto é, a aplicação é colocada no ar. Neste pipeline em especial, o *deploy* em ambiente de desenvolvimento é automático, porém, em produção é requerida intervenção humana. Para isto, abre-se uma caixa de diálogo com a pergunta: "*Deploy* para produção?". Caso positivo, é feito um *merge* com a *branch master* e criado o *container* de produção, tornando a aplicação *online*. Nesse cenário, o que diferencia a aplicação de desenvolvimento e de produção são os sufixos "*-dev*" e "*-prd*", respectivamente adicionados aos nomes dos *containers*. Cada equipe deve personalizar seu *pipeline* para atender suas demandas de projeto.

B.7 Sprint 7

Tabela B.12: Sprint 7.

Sprint 7	**Meta:** prover monitoramento e alertas						
Duração:	2 semanas	**Trabalho:**	6h/dia	**Início:**	–/–/2020	**Fim:**	–/–/2020
#	**Tarefa**						**Estimativa**
T7.6.1.23	Manter a coleta e visualização de dados						48
T7.6.2.24	Manter envio de alertas baseado em eventos						12

T7.6.1.23 - Manter a coleta e visualização de dados. Para suportar esta etapa foi utilizado o *Elastic Stack*[28], em conformidade com o modelo definido na Figura B.26. Em termos práticos, optou-se por adotar o projeto *docker-elk* disponível no GitHub que combina as últimas versões do Elasticsearch, Logstash e Kibana, possibilitando a criação de uma *stack* a partir do Docker Compose. Com a mínima configuração dos arquivos **.yml* providos, basta executar o comando `docker stack deploy -c docker-stack.yml elastic` e já é possível tratar, armazenar e visualizar os dados, faltando somente mecanismos de coleta. Isto é suprido pela família *Beats*, que possui agentes com capacidade de enviar dados de centenas ou milhares de computadores e sistemas para o Logstash ou Elasticsearch. Neste sentido, optou-se por utilizar uma imagem do Filebeat para obter e enviar dados de logs do Nginx para o Logstash tratar, antes encaminhar para o Elasticsearch, conforme mostra o *screenshot* Kibana da Figura B.32. O Filebeat foi definido em modo "*global*", forçando cada *host* ter uma réplica em operação.

Figura B.32: Dados *elasticsearch* enviados pelo *filebeat*.
(Fonte: própria)

[28]https://www.elastic.co/pt/elastic-stack

Por fim, com o objetivo de colher métricas relacionadas a processador, memória, rede e processos, também foi instalado em cada *host* o Metricbeat, que passou a submeter os dados diretamente para o Elasticsearch. A partir dos dados coletados, esta solução permitiu construir visualizações e *dashboards* para monitoramento dos microsserviços e *hosts* em tempo real, conforme exemplo parcial da Figura B.33.

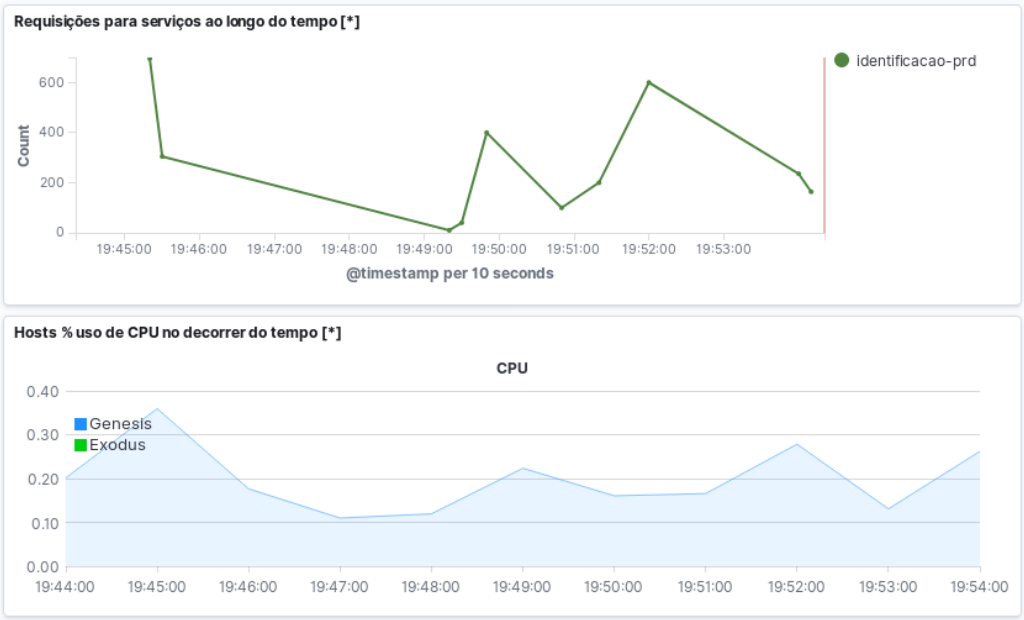


Figura B.33: Número de requisições e consumo de CPU.
(Fonte: própria)

T7.6.2.24 - Manter envio de alertas baseado em eventos. O Kibana tem uma seção "*Alerts and Actions*" onde é possível criar alertas e ações quando ocorre determinada condição. Por exemplo, foi definido um alerta para verificar a cada 1 minuto a quantidade de requisições e notificar via Slack (Figura B.34) em intervalos de 5 minutos caso o total de requisições nos últimos 3 minutos ultrapasse a 1000. Nos mesmos moldes, outros alertas foram criados, como alertas para monitorar o consumo de memória e processador por *host*, conforme mostra a configuração de alerta da Figura B.35.

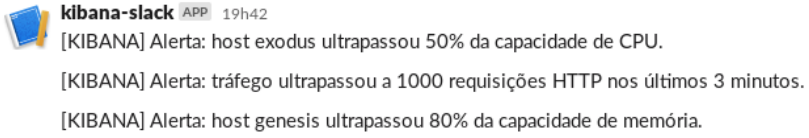


Figura B.34: Alertas de monitoramento chegando no Slack.
(Fonte: própria)

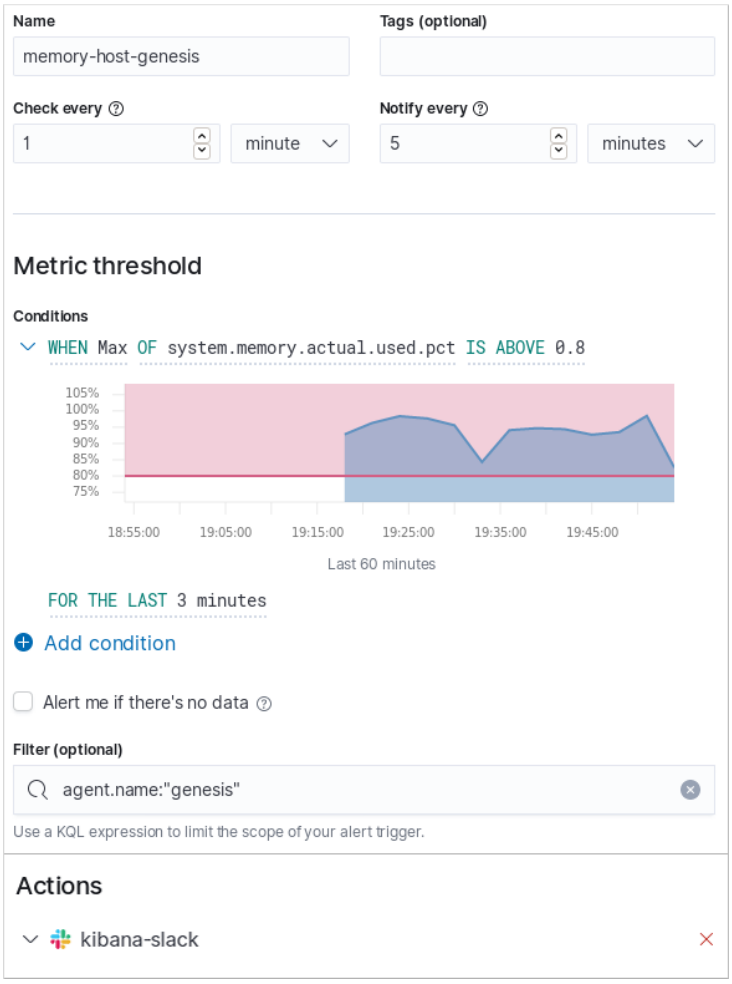


Figura B.35: Configuração de alerta de consumo de memória do *host genesis*. (Fonte: própria)

Após mais esse progresso, as imagens e *containers* Docker dessa etapa da migração aparecem listadas conforme mostram os Códigos B.15 e B.16, respectivamente.

Código B.15: Docker image ls [progresso#2]

```
$ docker image ls
IMAGE ID            REPOSITORY                      TAG                 SIZE
96e7852c06fc        unbmaster/haproxy               1.0                 118MB
9f51fd128bb0        unbmaster/filebeat              1.0                 553MB
e1ed619be9f7        sonarqube                       lts                 484MB
02830ef1af8b        hadolint/hadolint               latest              8.76MB
121454ddad72        elastic_elasticsearch           latest              810MB
01979bbd06c9        elastic_logstash                latest              789MB
df0a0da46dd1        elastic_kibana                  latest              1.29GB
f6411ebd974c        dockersamples/visualizer        latest              166MB
```

```
0606b4a873a7        unbmaster/identificacao         68                  180MB
5b9cab93cc2f        unbmaster/redis                 1.0                 127MB
6e1ff7cb748e        debian                          stable-slim         69.2MB
```

Código B.16: Docker container ls [progresso#2]

```
$ docker container ls
CONTAINER ID    IMAGE                          NAMES               PORTS
e24eec3d11f1    unbmaster/identificacao:68     identificacao-prd*
4150266bea03    unbmaster/identificacao:68     identificacao-dev*
5700b0116a4c    unbmaster/filebeat:1.0         filebeat_filebeat*
27d6090d95c4    elastic_kibana:latest          elastic_kibana.1e*  5601/tcp
7985e6f3f26f    elastic_logstash:latest        elastic_logstash.*  5044/tcp,9600/tcp
d5dff35e7be1    elastic_elasticsearch:lat*     elastic_elasticse*  9200/tcp,9300/tcp
153e398d86c8    dockersamples/visualizer:*     visualizer.1.s87i*  5555/tcp
67326cbee815    unbmaster/haproxy:1.0          haproxy.1.0pv53sk*
79aa182be310    sonarqube:lts                  sonarqube.1.lxio4*  9000/tcp
c6f983c47f74    unbmaster/redis:1.0            redis.1.whhf13usd*  6379/tcp
```

[*] As imagens e *containers* novos estão com a linha em destaque azul.

B.8 Sprints finais

Recapitulando o trabalho realizado até o momento, as *sprints* estiveram concentradas em (i) preparar o ambiente inicial de desenvolvimento, (ii) conhecer sobre a organização e o sistema legado, (iii) decompor parte do monolítico em serviços, (iv) desenvolver o primeiro microsserviço, (v) automatizar o processo de entregas (CI/CD), e (vi) prover monitoramento da aplicação em tempo real. Isto equivale a um ciclo completo do método *Microservice Full Cycle* - MFC já percorrido. Falta ainda desenvolver os microsserviços Plano, Inscrição e Notificação, o que requer novos ciclos para incrementar e aperfeiçoar o processo de migração. Entretanto, considerando que os moldes de desenvolvimento se assemelham aos descritos em *sprints* anteriores, para evitar redundâncias, as *sprints* subsequentes nº 8, 9 e 10, dedicadas a implementar os microsserviços restantes, não serão tratadas formalmente. Em todo o caso, para que o estudo não tenha perdas, este tópico está reservado a discutir temas ainda não cobertos e julgados oportunos.

Sprint 8 - Plano *Service*

No contexto do caso de uso UC001, esse microsserviço tem a finalidade de obter os planos (cursos/estágios) disponíveis para que o militar possa escolher aqueles de interesse.

a) Autenticação. Diferentemente de Identificação *Service* que requer o login do usuário, no Plano *Service* a autenticação e autorização é feita via *token*. Portanto, toda requisição a "GET */v1/planos*" é atrelada ao *AuthMiddleware*, responsável por validar e propagar o *token* presente no HEADER, conforme mostra o Código B.17. De acordo com a implementação do Código B.18, para acessar o serviço, basta que o usuário tenha o papel ROLE_USER, obtido no login inicial e embutido no *payload* JWT, do contrário, a resposta é o *status code* "*401 Unauthorized*".

Código B.17: Autenticação atrelada ao serviço

```php
<?php
use Slim\Factory\AppFactory;
$app = AppFactory::create();
$auth = new Core\AuthMiddleware;
$app->add(new Core\CorrelationIdMiddleware);
$app->get('/v1/planos', 'Controller\v1\PlanoController:planos')->add($auth);
$app->run();
?>
```

Código B.18: Implementação *AuthMiddleware*

```php
<?php
namespace Core;

class AuthMiddleware
{
    public function __invoke(Request $request, RequestHandler $handler)
    {
        $response = $handler->handle($request);
        $token = JWT::getTokenFromHeader($request);
        if (JWT::isValidToken($token) && JWT::isUserRole($token)) {
            $response = $response->withHeader('Authorization',  "Bearer $token");
        }
        else {
            $response = new Response();
            return $response->withStatus(401); # 401 Unauthorized
        }
        return $response;
    }
}
?>
```

b) Integração. Para integrar o microsserviço Plano ao legado (SISP), é feito uma substituição direta da *feature*, conforme mostra o Código B.19. Contudo, frequentemente isso não é simples e pode requerer uma camada Anticorrupção (do inglês *Anti Corruption layer - ACL*), evitando que o domínio seja corrompido. Segundo Vernon [36], a camada Anticorrupção *downstream* converte as representações em objetos de domínio de seu contexto local. Evans [94] compara a ACL como um tipo de classe cuja função é fazer uma tradução entre o legado e o novo, sendo muito comum o uso dos *patterns Adapter* e *Facade*. No caso do legado SISP, o método original era majoritariamente compatível com a chamada ao microsserviço, sendo preciso apenas a conversão de tipos (*json* para *array*).

Código B.19: Integração com legado - Plano *Service*

```php
<?php
class C_Plano extends CI_Controller
{
    public function listagemPlano($codigo)
    {
        ...
        // Código original legado
        # $lista = $this->m_plano->selecionarPlano(
        #    array('PLANO_CODIGO' => $codigo)
        # );

        // Código adaptado para microsserviços
        $token = this->session->userdata('token');
        $lista = API::get('https://<url>/v1/planos',
            ['planoId' = $codigo, 'token' = $token]
        );
        $lista = json_decode($lista, true); # Converte json para array
        ...
    }
}
?>
```

Sprint 9 - Inscrição *Service*

Por meio de sua API, o microsserviço Inscrição permite que o militar se inscreva em um curso ou estágio, submetendo um "POST */v1/inscricoes*". O serviço também possibilita listar essas inscrições realizadas, consultando "GET */v1/inscricoes/militares/{id}*", conforme mostram as rotas definidas no Código B.20.

Código B.20: Rotas Inscrição *Service*

```php
<?php
$auth = new Core\AuthMiddleware;
$app->add(new Core\CorrelationIdMiddleware);
$app->post('/v1/inscricoes',
 ↪  'Controller\InscricaoController:inscrever')->add($auth);
$app->get('/v1/inscricoes/militares/{id}',
 ↪  'Controller\InscricaoController:listar')->add($auth);
$app->run();
?>
```

a) Dados. O SISP acessa uma base de dados corporativa, e há pouca ingerência sobre mudanças DDL. Para manter a autonomia e escalabilidade do microsserviço, duas estratégias são consideradas. A primeira, cria um serviço de banco de dados com *schemas* individualizados para cada microsserviço, conforme Figura B.36(a). Outra alternativa, é mapear um volume Docker em cada serviço apontando para um *filesystem* compartilhado onde ficará o repositório de dados, acessível não apenas por múltiplos *containers*, mas também *hosts*, a exemplo da Figura B.36(b). Em ambos casos, os dados corporativos originais podem ser refletidos por meio de *views* materializadas ou algum mecanismo de sincronização, assumindo a possibilidade de uma consistência eventual. Neste estudo, as duas opções foram experimentadas, mas a solução (a) foi propensa a menos falhas. Plano*Services* operou com um base local SQLite dentro do *container* e com volume compartilhado, enquanto Inscrição *Services* usou o PostgreSQL como um serviço à parte.

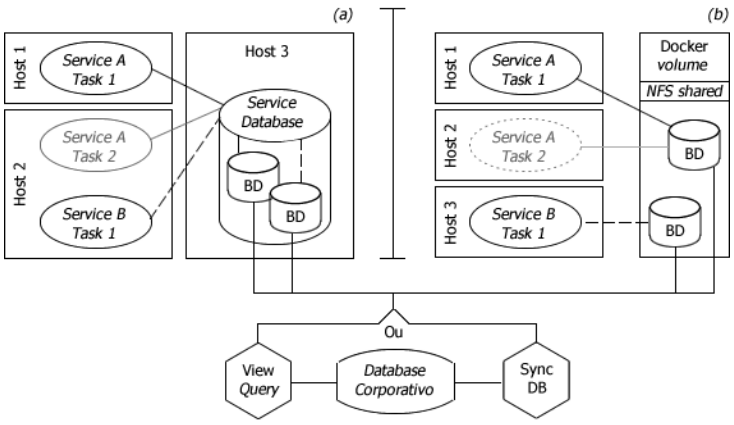


Figura B.36: Base de dados por serviço [7].
(Fonte: própria)

b) Integração. No contexto deste trabalho, integração é o processo de refatorar o sistema legado, adaptando-o para que as novas *features* implementadas em microsserviços vigorem. Neste caso em especial, isto acontece tornando sem efeito uma fração do código original legado e adicionando a nova implementação, conforme mostra o Código B.21. O monolítico continua recebendo a requisição, mas o processamento da inscrição é realizado pelo microsserviço Inscrição *Service*. E assim, em conformidade com o pattern *Strangler Application* (Seção 2.3.5), a arquitetura monolítica e de microsserviços vão coexistindo até que o legado seja, em parte ou no todo, substituído.

Código B.21: Integração com legado - Inscrição *Service*

```php
<?php
class C_Inscricao extends CI_Controller {

    public function inscrever() {
        ...
        // Código original legado
        #  $inscrever = $this->m_inscricao->inscreveMilitar($idt, $plano);

        // Código adaptado para microsserviços
        $token = this->session->userdata('token');
        $inscrever = API::get('https://<url>/v1/inscricoes',
            ['planoId' = $plano, 'token' = $token]
        );
        $inscrever = json_decode($inscrever, true); # Converte json para array
        ...
    }
}
?>
```

Sprint 10 - Notificação *Service*

Notificação é um serviço simples, cuja finalidade é enviar um e-mail ao militar sempre que uma inscrição em um curso ou estágio for realizada. O interessante é que isto é feito de modo assíncrono, totalmente desacoplado de outros serviços, conforme já descrito pela Figura B.16. O que possibilita essa independência é a troca de mensagens entre serviços provida por um intermediador, conhecido por "*message broker*". Para este fim, é adotado o RabbitMQ[29] e o padrão "*publish/subscribe*". No contexto deste trabalho, quando uma

[29]https://www.rabbitmq.com/

inscrição acontece, Inscrição *Service* publica (do inglês *publish*) uma mensagem *payload* numa fila de e-mail, por intermédio de um elemento chamado "*exchange*". *Exchange* tem por função apenas receber as mensagens e transmiti-las para todas as filas ligadas (do inglês *binding*) a ele. Por outro lado, Notificação *Service* subscreve (do inglês *subscribe*) a esta fila (do inglês *queue*) dedicada a receber as mensagens de e-mail. Sua tarefa é, portanto, ficar constantemente "escutando" (do inglês *listening*) essa fila. Para cada nova mensagem que chegar, Notificação *Service* então processa e dispara o e-mail ao destinatário, baseando-se na mensagem *payload*.

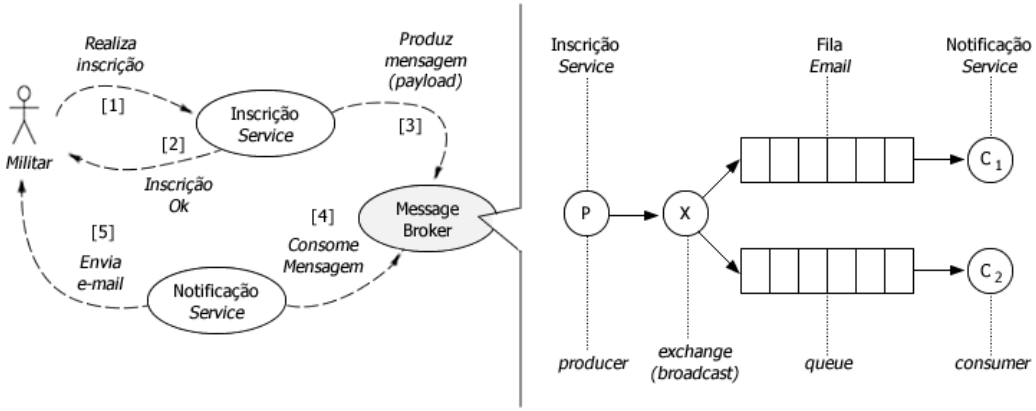


Figura B.37: Message Broker [7].
(Fonte: baseado em rabbitmq.com)

Nota-se na Figura B.37 que, do ponto de vista do militar que fez a inscrição, todo processo termina no passo [2] (Inscrição Ok). Inscrição *Service* se abstrai completamente da tarefa de enviar e-mail, que é feita por Notificação *Service* de modo transparente. Se por qualquer razão Notificação *Service* estiver offline, as mensagens não se perdem, ficam enfileiradas até que o serviço volte a estar disponível. A Figura B.38 mostra uma visão geral do processamento da fila de e-mail no RabbitMQ.

Por fim, concluindo as *sprints* previstas da simulação, cabe ressaltar que todos microsserviços recém desenvolvidos foram adicionados ao *pipeline* CI/CD do Jenkins, a exemplo da Figura B.39 e aos painéis de monitoramento Kibana, conforme Figura B.40, consolidando assim a etapa de *Demostração* do método DSRM. As imagens e *containers* Docker dessa última fase da migração aparecem listadas nos Códigos B.22 e B.23, respectivamente. O próximo desafio é proceder a *Avaliação*, buscando mensurar quão bem o MFC é proficiente na migração do legado.

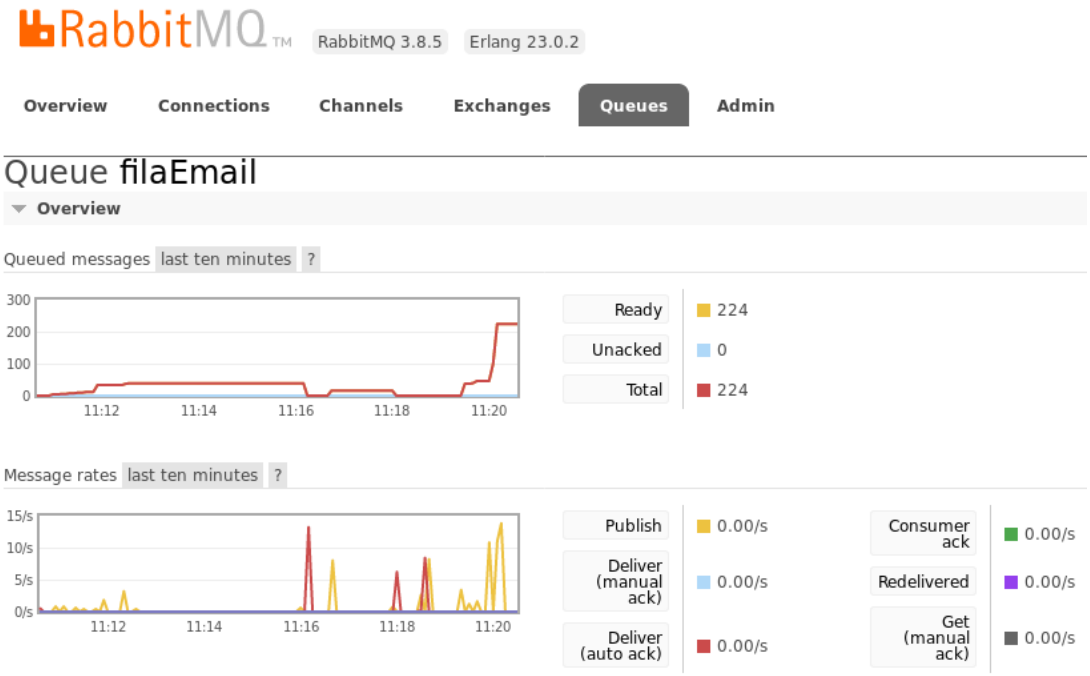


Figura B.38: Fila de e-mail - RabbitMQ.
(Fonte: própria)

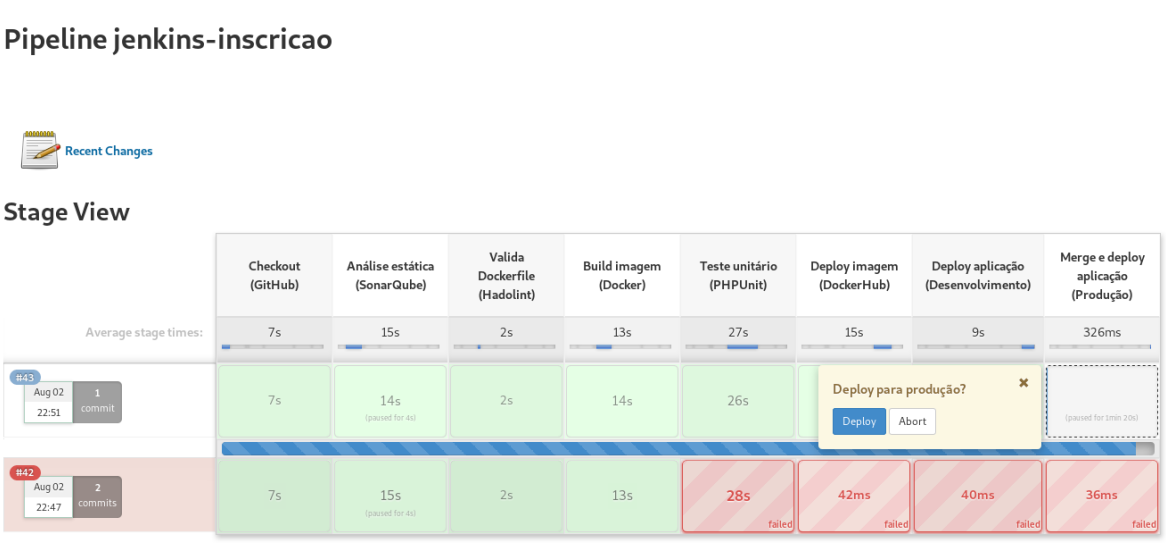


Figura B.39: Pipeline Inscrição *Service* - Jenkins.
(Fonte: própria)

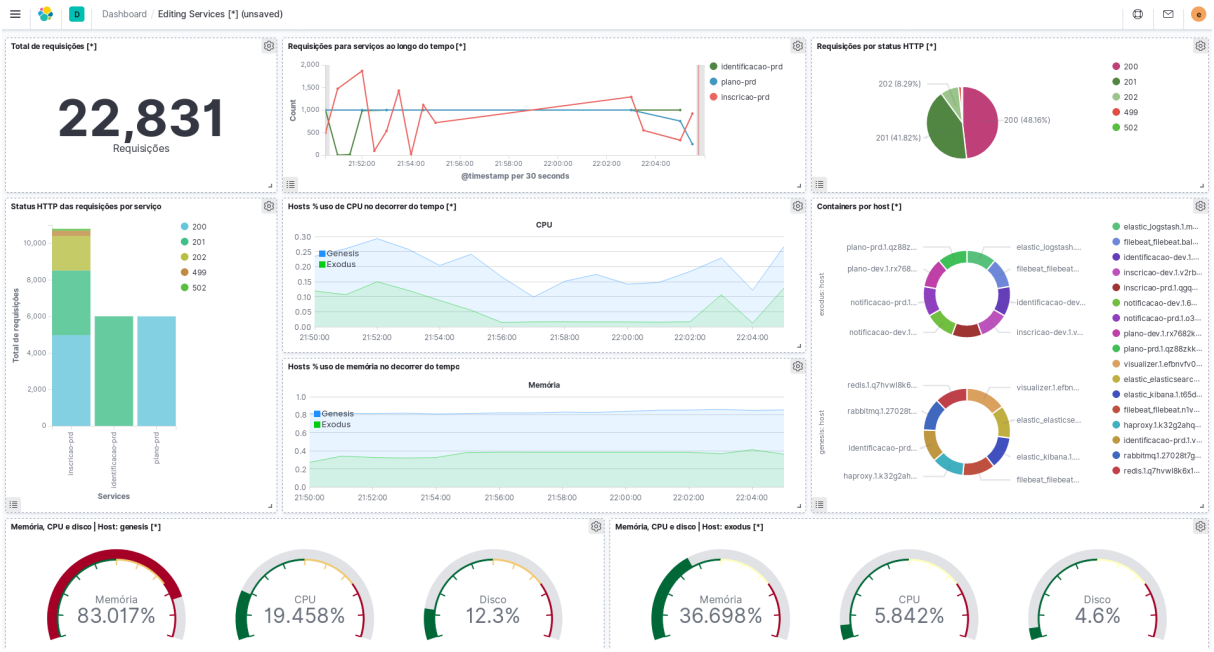


Figura B.40: Dashboard da simulação - Kibana.
(Fonte: própria)

Código B.22: Docker image ls [progresso#3]

```
$ docker image ls
IMAGE ID            REPOSITORY                      TAG                 SIZE
1e0ba3f22bfa        unbmaster/notificacao           19                  184MB
f80297960925        unbmaster/inscricao             38                  183MB
cea8e2bcc414        unbmaster/plano                 168                 183MB
f65f7c36d41b        rabbitmq                        latest              198MB
96e7852c06fc        unbmaster/haproxy               1.0                 118MB
9f51fd128bb0        unbmaster/filebeat              1.0                 553MB
e1ed619be9f7        sonarqube                       lts                 484MB
02830ef1af8b        hadolint/hadolint               latest              8.76MB
121454ddad72        elastic_elasticsearch           latest              810MB
01979bbd06c9        elastic_logstash                latest              789MB
df0a0da46dd1        elastic_kibana                  latest              1.29GB
0606b4a873a7        unbmaster/identificacao         68                  180MB
5b9cab93cc2f        unbmaster/redis                 1.0                 127MB
6e1ff7cb748e        debian                          stable-slim         69.2MB
f6411ebd974c        dockersamples/visualizer        latest              166MB
```

Código B.23: Docker container ls [progresso#3]

```
$ docker container ls
CONTAINER ID    IMAGE                          NAMES               PORTS
lu6vw22rgq7s    unbmaster/notificacao:19       identificacao-prd*
6561fd0826c8    unbmaster/notificacao:19       identificacao-dev*
96f97258f005    unbmaster/inscricao:68         inscricao-prd.1.r*
rixd8xvmho3r    unbmaster/inscricao:68         inscricao-dev.1.c*
5547676e3e0d    unbmaster/plano:68             plano-prd.1.pl56m*
fdf9b20e06ed    unbmaster/plano:68             plano-dev.1.s2zcg*
d6872822b952    rabbitmq:3-management          rabbitmq.1.u037uv*  4369/tcp,
                                                                   5671-5672/tcp,
                                                                   15671-15672/tcp,
                                                                   25672/tcp
e24eec3d11f1    unbmaster/identificacao:68     identificacao-prd*
4150266bea03    unbmaster/identificacao:68     identificacao-dev*
c6f983c47f74    unbmaster/redis:1.0            redis.1.whhf13usd*  6379/tcp
5700b0116a4c    unbmaster/filebeat:1.0         filebeat_filebeat*
27d6090d95c4    elastic_kibana:latest          elastic_kibana.1e*  5601/tcp
7985e6f3f26f    elastic_logstash:latest        elastic_logstash.*  5044/tcp,9600/tcp
d5dff35e7be1    elastic_elasticsearch:lat*     elastic_elasticse*  9200/tcp,9300/tcp
153e398d86c8    dockersamples/visualizer:*     visualizer.1.s87i*  5555/tcp
67326cbee815    unbmaster/haproxy:1.0          haproxy.1.0pv53sk*
79aa182be310    sonarqube:lts                  sonarqube.1.lxio4*  9000/tcp
```

[*] As imagens e *containers* novos estão com a linha em destaque azul.