Programa de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Diego Pereira da Rocha

Monólise: Uma Técnica para Decomposição de Aplicações Monolíticas em Microsserviços

São Leopoldo
2018

Diego Pereira da Rocha

MONÓLISE: UMA TÉCNICA PARA DECOMPOSIÇÃO DE APLICAÇÕES MONOLÍTICAS EM MICROSSERVIÇOS

Dissertação apresentada como requisito para a obtenção do título de Mestre pelo Programa de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos — UNISINOS

Orientador:
Prof. Dr. Kleinner Silva Farias de Oliveira

São Leopoldo
2018

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)

Rocha, Diego Pereira da

Monólise: Uma Técnica para Decomposição de Aplicações Monolíticas em Microsserviços / Diego Pereira da Rocha — 2018.

171 f.: il.; 30 cm.

Dissertação (mestrado) — Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, São Leopoldo, 2018.

"Orientador: Prof. Dr. Kleinner Silva Farias de Oliveira, Unidade Acadêmica de Pesquisa e Pós-Graduação".

1. Microsserviços. 2. Aplicação Monolítica. 3. Decomposição Aplicação. 4. Granularidade de Microsserviços. I. Título.

CDU 004.41

(Bibliotecária responsável: Magale da Silva — CRB 10/1739)

Diego Pereira da Rocha

Monólise: Uma Técnica para Decomposição de Aplicações Monolíticas em Microsserviços

Dissertação apresentada à Universidade do Vale do Rio dos Sinos – Unisinos, como requisito parcial para obtenção do título de Mestre em Computação Aplicada.

Aprovado em 17 de Setembro de 2018.

BANCA EXAMINADORA

Prof. Dr. Kleinner Silva Farias de Oliveira – PPGCA/UNISINOS

Prof. Dr. Jorge Luis Victória Barbosa – PPGCA/UNISINOS

Prof. Dr. Bruno Carreiro da Silva – California Polytechnic State University

Prof. Dr. Kleinner Silva Farias de Oliveira (Orientador)

Visto e permitida a impressão
São Leopoldo,

Prof. Dr. Rodrigo da Rosa Righi
Coordenador PPG em Computação Aplicada

AGRADECIMENTOS

Primeiramente agradeço a Deus, por ter conseguido cumprir mais esse objetivo em minha vida. Sem Ele, nada seria possível.

Agradeço aos meus pais, Antônio e Maria Regina (in memoriam) por terem me ensinado o valor dos estudos e a importância da educação, assim como pela dedicação em nossa criação.

Agradeço aos meus familiares (sobrinho, sobrinhas e cunhadas), em especial, aos meus irmãos pelo afeto e pelo apoio que tiveram em todos os momentos durante estes dois anos de mestrado.

Agradeço ao Programa de Pós-Graduação em Computação Aplicada da Unisinos, em especial, aos Professores, pela oportunidade de aprimoramento do conhecimento.

Ao meu orientador Prof Kleinner, por acreditar em meu trabalho, pela cuidadosa orientação nesse percurso e também por ter paciência às inúmeras vezes em que eu mudava a proposta da minha pesquisa, tentando achar a técnica perfeita. Obrigado, Professor, por ter me orientado neste trabalho.

Aos amigos e às pessoas que torcem pela minha trajetória pessoal e profissional.

E, por fim, mas não menos importante, agradeço especialmente a minha noiva Vanessa, foi ela quem me incentivou a fazer este mestrado. Obrigado por estar ao meu lado durante estes dois anos e por compreender os momentos em que precisei estar ausente. Obrigado também por me ajudar a revisar cuidadosamente o texto deste trabalho. Obrigado, amor, te amo.

"*Nós somos aquilo que fazemos repetidamente. Excelência, então, não é um modo de agir, mas um hábito*".
(Aristóteles)

RESUMO

A recorrente necessidade de as empresas entregarem seus softwares em curto espaço de tempo e de forma contínua, combinada ao alto nível de exigência dos usuários, está fazendo a indústria, de um modo geral, repensar como devem ser desenvolvidas as aplicações para o mercado atual. Nesse cenário, microsserviços é o estilo arquitetural utilizado para modernizar as aplicações monolíticas. No entanto, o processo para decompor uma aplicação monolítica em microsserviços é ainda um desafio que precisa ser investigado, já que, na indústria, atualmente, não há uma estrutura padronizada para fazer a decomposição das aplicações. Encontrar uma técnica que permita definir o grau de granularidade de um microsserviço também é um tema que desperta discussão na área de Engenharia de Software. Partindo dessas considerações, este trabalho propôs a Monólise, uma técnica que utiliza um algoritmo chamado Monobreak, que possibilita decompor uma aplicação monolítica a partir de funcionalidades e também definir o grau de granularidade dos microsserviços a serem gerados. Nesta pesquisa, a Monólise foi avaliada através de um estudo de caso. Tal avaliação consistiu na comparação da decomposição realizada pela Monólise com a decomposição executada por um especialista na aplicação-alvo utilizada no estudo de caso. Essa comparação permitiu avaliar a efetividade da Monólise através de oito cenários realísticos de decomposição. O resultado dessa avaliação permitiu verificar as semelhanças e diferenças ao decompor uma aplicação monolítica em microsserviços de forma manual e a partir de uma técnica semiautomática. O desenvolvimento deste trabalho demonstrou que a técnica de Monólise apresenta-se com uma grande potencialidade na área de Engenharia de Software referente à decomposição de aplicações. Além disso, as considerações do estudo evidenciaram que essa técnica poderá ser um motivador para encorajar desenvolvedores e arquitetos na jornada de modernização de suas aplicações monolíticas em microsserviços bem como diminuir possíveis erros cometidos nessa atividade por profissionais com pouca experiência em decomposição de aplicações.

Palavras-chave: Microsserviços. Aplicação Monolítica. Decomposição Aplicação. Granularidade de Microsserviços.

ABSTRACT

The recurring need for companies to deliver their software in a short time and on a continuous basis combined with the high level of demand of users is making the industry in general rethink how to develop the applications for the current market. In this scenario microservice is the architectural style used to modernize monolithic applications. However the process of decomposing a monolithic application into microservices is still a challenge that needs to be investigated since in industry there is currently no standardized framework for decomposing applications. Finding a technique that allows defining the degree of granularity of a microservice is also a topic that arouses discussion in the area of Software Engineering. Based on these considerations this work proposed the Monolise a technique that uses an algorithm called MonoBreak that allows to decompose a monolithic application from functionalities and also to define the degree of granularity of the microservices to be generated. In this research the Monolise was evaluated through a case study. Such evaluation consisted of comparing the decomposition performed by the Monolise with the decomposition performed by a specialist in the target application used in the case study. This comparison allowed to evaluate the effectiveness of the Monolise through eight realistic scenarios of decomposition. The result of this evaluation allowed to verify the similarities and differences in the decomposition of a monolithic application in microservices manually and from a semiautomatic technique. The development of this work demonstrated that the Monolise technique presents with great potentiality in the area of Software Engineering regarding the decomposition of applications. In addition the study's considerations showed that this technique could be a motivator to encourage developers and architects in the modernization of their monolithic applications in microservices as well as to reduce possible mistakes made in this activity by professionals with little experience in decomposing applications.

Keywords: Microservices. Monolithic Application. Application Decomposition. Microservice Granularity.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

API	Application Programming Interface
AWS	Amazon Web Services
CBSE	Component-based Software Engineering
CORBA	Common Object Request Broker Architecture
CRUD	Create Read Update Delete
DAO	Data Access Object
DDD	Domain Driven Design
DNS	Domain Name System
DTO	Data Transfer Object
EC2	Elastic Compute Cloud
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MVC	Model View Controller
MXML	Magic Extensible Markup Language
NIST	National Institute of Standards and Tecnology
PaaS	Platform as a Service
REST	Representational State Transfer
RMI	Remote Method Invocation
SaaS	Software as a Service
SMC	Sistema de Mapeamento de Competências
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service Oriented Computing
SWF	Shockwave Flash
TCC	Trabalho de Conclusão de Curso
TO	Transfer Object
VO	Value Object
WAR	Web application ARchive

SUMÁRIO

1 INTRODUÇÃO

A arquitetura de software é um assunto que desperta muito interesse de pesquisa tanto na indústria quanto na academia. No contexto acadêmico, esse tema surgiu no ano de 1980 com o advento e a divulgação da orientação a objeto (DRAGONI et al., 2017), no entanto, foi somente no ano de 1992, por Perry e Wolf (1992), que o conceito de arquitetura de software foi legitimado. A partir do trabalho desses autores, a arquitetura de software foi separada da definição de design de software, o que permitiu tanto aos pesquisadores quanto aos profissionais da indústria voltarem-se à aplicabilidade do conceito.

Tal interesse gerou um aumento no número de padrões de arquitetura de software também conhecidos como *estilos*. Devido à geração de diversos estilos arquiteturais, foi necessário criar uma forma de classificação para eles. Uma das obras mais notáveis na literatura com relação a esse assunto denomina-se *Software Architecture: Perspective on a Emerging Discipline* (SHAW; GARLAN, 1996).

Com a criação e disseminação da orientação a objetos, no ano de 1990, surgiu também uma significativa contribuição para a área de arquitetura de software, através da publicação de uma obra clássica da literatura, intitulada *Design Patterns: Elements of Reusable Object-oriented Software*, de Gamma et al. (1995). Nessa obra, os autores propuseram um catálogo de padrões de projeto que poderia ser utilizado para resolver problemas recorrentes no desenvolvimento de software. Antes mesmo das definições de padrões de projeto (GAMMA et al., 1995), já era utilizado um padrão de projeto arquitetural orientado a objeto, o qual é usado até os dias de hoje, o MVC (FOWLER, 2002).

Após a consolidação da orientação a objetos, surgiu o conceito Component-based Software Engineering (CBSE) (SZYPERSKI, 2002). Essa definição possibilitou melhor controle sobre o design, a implementação e a evolução do sistema de software. A partir dos anos 2000, surgiu um paradigma na área da computação distribuída chamado *Service-Oriented Computing* (SOC), o qual tem por objetivo atacar a complexidade dos sistemas distribuídos e integrar diferentes aplicações de software (MACKENZIE et al., 2006). No modelo do SOC, um programa é chamado de serviço e cada serviço oferece funcionalidades que se comunicam por meio do envio de mensagens. As principais características do SOC são: dinamismo, modularidade, reuso, desenvolvimento distribuído e integração heterogênea de sistemas.

Foi também no início dos anos 2000, segundo Dragoni et al. (2017), que surgiu a primeira geração de serviços baseada no paradigma de SOC, a Arquitetura Orientada a Serviços (SOA). Essa arquitetura utilizou os mesmos conceitos de SOC, porém adicionou a definição de capacidade de negócio, o que permitiu integrar diferentes serviços de negócios.

Segundo Xiao, Wijegunaratne e Qiang (2016), a SOA teve seu ápice entre os anos de 2002 a 2010. A partir de 2010, ela começou a perder espaço no mundo da indústria, devido à falta de uma definição clara de seus requisitos como, por exemplo, a descoberta e os contratos de serviços. Além disso, a utilização de SOA estava completamente acoplada às soluções de cada

fabricante e a protocolos específicos, como SOAP.

No ano de 2011, surgiu um termo chamado *microservices*, primeiramente introduzido em um workshop sobre arquitetura de software como uma forma de descrever ideias comuns em padrões de arquitetura de software (FOWLER; LEWIS, 2014). No universo acadêmico, o surgimento desse tema é recente, pois, segundo Pahl e Jamshidi (2016), ele foi encontrado em pesquisas somente a partir do ano de 2014, antes desse período, não havia um consenso sobre o que era, de fato, microsserviços. No entanto, antes mesmo dessa abordagem ter tal definição, empresas como Netflix já implementavam uma arquitetura similar, porém com outro nome, *Fine Grained SOA* (WANG; TONSE, 2013). A literatura da área permite afirmar que há diversas definições para microsserviços, mas a mais utilizada é:

> Microsserviços é uma abordagem que desenvolve um aplicativo único, como uma suíte de pequenos serviços, cada um executando seu próprio processo e se comunicando através de mecanismos leves, muitas vezes em uma API com recursos (FOWLER; LEWIS, 2014).

A crescente necessidade das empresas de entregar seus softwares em curto espaço de tempo e de forma contínua combinada ao alto nível de exigência dos usuários estão fazendo a indústria de um modo geral repensar como devem ser desenvolvidas as aplicações para o mercado atual. Nesse contexto, microsserviços é o estilo arquitetural que está despertando cada vez mais o interesse da área da indústria, pois trata-se de uma arquitetura nativa para nuvem. Tendo isso em vista, empresas como Google, Amazon e Netflix começaram a modernizar suas arquiteturas de software, decompondo suas aplicações monolíticas para uma arquitetura baseada em microsserviços.

No entanto, conforme Pahl e Jamshidi (2016), há poucas ferramentas e técnicas para auxiliar os profissionais na jornada de modernização de suas aplicações monolíticas em microsserviços. Segundo Chen, Li e Li (2017), realizar a decomposição de aplicações monolíticas em microsserviços é ainda uma tarefa complexa, com muitos desafios.

As considerações precedentes mostram a importância do conceito de microsserviços para o desenvolvimento de softwares, no entanto, é preciso descobrir de que modo é possível realizar a decomposição de aplicações monolíticas em microsserviços. Buscando responder essa questão, estão organizadas as seções seguintes deste trabalho.

1.1 Problemática

Conforme já mencionado, no mundo da indústria, o tema microsserviços surgiu através de experimentos realizados em grandes empresas de software, como Amazon, Google e Netflix. O uso de microsserviços por essas empresas deve-se ao fato delas buscarem maior agilidade no desenvolvimento e na entrega de seus produtos para o mercado, tendo em vista que as aplicações

monolíticas apresentam diversas limitações quando comparadas com arquiteturas nativas para nuvem, como microsserviços. Segundo Dragoni et al. (2017), aplicações monolíticas apresentam diversos problemas, dentre eles: alto acoplamento, limitação de escalabilidade, dificuldade de manutenção e homogeneidade tecnológica, dentre outros.

A limitação de escalabilidade refere-se à impossibilidade de escalar funcionalidades da aplicação de forma independente. Em aplicações monolíticas, é necessário escalar a aplicação por completo, o que inviabiliza, por exemplo, o uso desses tipos de aplicações em ambientes de computação em nuvem, nos quais se paga pelo consumo de recursos utilizados. Ao escalar toda a aplicação, acaba-se gerando um desperdício de recursos computacionais e consequentemente um maior custo financeiro a ser pago pelas empresas que utilizam um provedor de nuvem público.

A homogeneidade tecnológica é uma outra limitação que aplicações monolíticas apresentam. Elas não permitem a liberdade de escolha de uma melhor tecnologia para resolver uma necessidade de negócio. Aplicações monolíticas utilizam tecnologias que foram concebidas pela primeira vez que o software foi definido e não é mais possível alterá-lo, isso ocorre devido ao alto acoplamento entre as funcionalidades e também porque todas elas executam dentro de um mesmo artefato de software executável.

Aplicações monolíticas apresentam também problemas relacionados à dificuldade de manutenção. Esse tipo de aplicação contém uma alta complexidade tanto de implementação quanto de teste, o que acontece devido ao alto acoplamento e à baixa coesão das funcionalidades que estão disponibilizadas dentro de um único artefato de software. Tal complexidade acarreta em um maior controle na implementação, no teste e na entrega do software, fazendo com que as equipes de desenvolvimento tornem-se menos encorajadas na liberação de uma nova versão, o que resulta em uma entrega de software mais burocrática, ou seja, uma funcionalidade que deveria ser entregue em dias é, muitas vezes, entregue em meses ou até mesmo em anos.

Diante dos problemas enfrentados com aplicações monolíticas, pesquisas como a de Mustafa et al. (2018), Balalaie, Heydarnoori e Jamshidi (2016), Levcovitz, Terra e Valente (2016) e Knoche e Hasselbring (2018) propuseram técnicas manuais para realizar a decomposição de aplicações monolíticas em microsserviços. Técnicas semiautomáticas também foram propostas por Mazlami, Cito e Leitner (2017) e Chen, Li e Li (2017) e Escobar et al. (2016) para tentar diminuir o esforço e a complexidade no momento da decomposição das aplicações. Uma ferramenta para ajudar na decomposição de aplicações também foi proposta por Gysel et al. (2016) a fim de tentar estruturar o processo de decomposição; além disso, até arquiteturas de referência de microsserviços foram propostas por Bucchiarone et al. (2018) e Yugopuspito, Panduwinata e Sutrisno (2017).

No entanto, apesar de terem sido desenvolvidas técnicas e arquiteturas de referências, os trabalhos ainda apresentavam algumas lacunas a serem preenchidas por novas investigações, tal como será problematizado nesta dissertação. Os trabalhos encontrados, por exemplo, não recomendam microsserviços a nível de funcionalidade e também não mostram quais as classes

e os métodos que precisam ser migrados ao decompor uma funcionalidade para microsserviço.

Conforme destacado no trabalho de Mustafa et al. (2018), o grande desafio no processo de decomposição é encontrar o melhor grau de granularidade de um microsserviço. Assim, os trabalhos desenvolvidos até o momento não possibilitam ao usuário da técnica configurar o grau de decomposição ou a granularidade dos microsserviços a serem gerados.

Partindo dessa problemática, esta pesquisa então propõe o que denominou de Monólise, uma técnica semiautomática que utiliza um algoritmo sensível à arquitetura da aplicação e que possibilita ao usuário configurar qual o grau de granularidade em que os microsserviços devem ser gerados. Além disso, a Monólise permite demonstrar a nível do código quais as classes e os métodos das funcionalidade que terão de ser migrados para cada microsserviço recomendado.

1.2 Questões de Pesquisa

Conforme discutido neste trabalho, existem ainda muitas dúvidas referentes à temática sobre microsserviços, tendo em vista que o tema ainda é muito novo na área de Engenharia de Software e poucas pesquisas foram realizadas sobre o assunto.

Além disso, a partir dos estudos realizados, identificou-se que ainda é preciso investigar a respeito da decomposição de aplicações monolíticas em microsserviços. Diante dessa necessidade, este trabalho investigará as seguintes questões de pesquisa (**QP**):

- **QP1:** Como decompor aplicações monolíticas em microsservicos?
- **QP2:** Como definir um algoritmo que permita definir o grau de granularidade dos microsserviços e identificar as funcionalidades, as classes e os métodos que deverão ser migrados para cada microsserviço recomendado?
- **QP3:** Quais são as semelhanças e diferenças entre decompor uma aplicação monolítica manualmente e decompô-la utilizando a técnica proposta?

O desenvolvimento desta pesquisa buscará responder essas interrogações e, para auxiliar neste estudo, foram definidos alguns objetivos, os quais estão apresentados na próxima seção.

1.3 Objetivos

O objetivo principal deste trabalho é **propor uma técnica que auxilie os desenvolvedores e arquitetos de software na decomposição de suas aplicações monolíticas em microsserviços**. Para alcançar o objetivo geral, são definidos alguns objetivos específicos:

- **Implementar um algoritmo que recomende microsserviços, permitindo a definição do grau de granularidade e a identificação das funcionalidades, das classes e dos métodos a serem migrados**

Esse objetivo visa propor um algoritmo que permita identificar os microsserviços e configurar o seu grau de granularidade.

- **Avaliar a técnica proposta**

  Esse objetivo busca realizar a avaliação da técnica, por meio de uma comparação entre a decomposição gerada pela técnica e a decomposição realizada manualmente em uma aplicação desenvolvida a fim de analisar as semelhanças e diferenças resultantes desse processo.

1.4 Metodologia

Este trabalho ancora-se nos seguintes passos metodológicos: primeiro foi realizada uma pesquisa sobre técnicas que possibilitam a decomposição de aplicações monolíticas em microsserviços. Isso teve por finalidade encontrar e analisar quais os tipos de técnicas disponíveis para quebra de aplicações monolíticas em microsserviços. Após tal pesquisa, o segundo passo foi agrupar os trabalhos com base nos tipos de técnicas encontrados. A partir dessa categorização, foi realizada uma comparação entre os trabalhos relacionados para identificar as oportunidades de pesquisas. O terceiro passo metodológico consistiu em desenvolver uma técnica semiautomática que permite a decomposição de uma aplicação monolítica em microsserviços. Nesse passo são apresentados os aspectos de implementação da técnica e as suas principais características-chave.

Por fim, o último passo da metodologia consistiu em avaliar a precisão da técnica de decomposição proposta quando comparada com uma decomposição realizada manualmente, ou seja, verificou-se se os usuários poderiam deixar de realizar a decomposição manualmente para enfim utilizar uma técnica semiautomática.

1.5 Organização do Trabalho

O presente trabalho está organizado em seis capítulos, conforme descritos a seguir. Após este primeiro capítulo de introdução, está o Capítulo 2, que descreve os conceitos fundamentais para o entendimento da pesquisa; o Capítulo 3 apresenta uma análise comparativa do estado da arte da técnica de decomposição de aplicações monolíticas em microsserviços; o Capítulo 4 discorre sobre a implementação da técnica de Monólise; já o Capítulo 5 descreve a avaliação da técnica e, por fim, o Capítulo 6 apresenta a conclusão deste trabalho, como também as contribuições, as limitações e sugestões de trabalhos futuros desta pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos das teorias utilizadas neste trabalho. A fundamentação teórica desta pesquisa está organizada em seis seções: a seção 2.1 define o que é uma arquitetura de software e descreve quais as principais estruturas arquiteturais e os atributos de qualidade de software; na seção 2.2 são apresentados os principais conceitos de computação em nuvem; na seção 2.3 são apresentados os principais conceitos da computação orientada a serviço; na seção 2.4 é apresentada a arquitetura monolítica e seus principais problemas; na seção 2.5 são apresentadas as principais características de uma arquitetura orientada a serviço e, por fim, na seção 2.6 são descritas as principais características e benefícios de microsserviços

2.1 Arquitetura de Software

Sistemas de software são criados para alcançar os objetivos de negócio de uma organização e a arquitetura do software é o meio que permite alcançar os objetivos de negócio, os quais normalmente são abstratos até o resultado de um sistema concreto. Conforme Bass, Clements e Kazman (2012), a arquitetura de software pode ser definida como "a estrutura de um programa que compreende os seus componentes, as propriedades visíveis externamente, e os relacionamentos entre tais componentes".

2.1.1 Estruturas Arquiteturais

Para Bass, Clements e Kazman (2012) uma arquitetura possui três estruturas arquiteturais: módulos; componentes e conectores; e alocação, conforme Figura 1.

Figura 1 – Estruturas arquiteturais

	Software Structure	Element Types	Relations	Useful For	Quality Attributes Affected
Module Structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Engineering subsets, engineering extensions	"Subsetability," extensibility
	Layers	Layer	Requires the correct presence of, uses the services of, provides abstraction to	Incremental development, implementing systems on top of "virtual machines"	Portability
	Class	Class, object	Is an instance of, shares access methods of	In object-oriented design systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
	Data model	Data entity	{one, many}-to-{one, many}, generalizes, specializes	Engineering global data structures for consistency and performance	Modifiability, performance
C&C Structures	Service	Service, ESB, registry, others	Runs concurrently with, may run concurrently with, excludes, precedes, etc.	Scheduling analysis, performance analysis	Interoperability, modifiability
	Concurrency	Processes, threads	Can run in parallel	Identifying locations where resource contention exists, or where threads may fork, join, be created, or be killed	Performance, availability
Allocation Structures	Deployment	Components, hardware elements	Allocated to, migrates to	Performance, availability, security analysis	Performance, security, availability
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities	Development efficiency
	Work assignment	Modules, organizational units	Assigned to	Project management, best use of expertise and available resources, management of commonality	Development efficiency

Fonte: Bass, Clements e Kazman (2012)

- **Módulos**

  Módulos são estruturas estáticas que se concentram no modo como a funcionalidade do sistema é dividida. Os módulos permitem realizar questionamentos sobre o impacto no sistema quando as responsabilidades atribuídas a cada módulo mudam. A visão completa dos módulos de um sistema permite avaliar a modificabilidade do sistema e também ajuda a responder questionamentos como:

  - Qual a principal responsabilidade funcional atribuída a cada módulo?
  - Qual outros softwares usam ou dependem de um determinado módulo?
  - Quais os módulos estão relacionados a outros módulos por meio de relações de generalização ou especialização?

- **Componente e Conector**

  Componentes e conectores são estruturas dinâmicas, que se concentram no modo como os elementos interagem uns com os outros em tempo de execução para realizar as funções do sistema. Componentes representam comportamentos em tempo de execução no sistema e podem ser caracterizados como unidades computacionais, como serviços, servidores e clientes.

  Já os conectores descrevem a comunicação entre os componentes em um sistema, podendo ser representados como retornos de chamadas e operações de sincronização de processos. Componentes e conectores ajudam a responder questões sobre propriedades de execução de um sistema, como desempenho, segurança e disponibilidade. Abaixo seguem alguns exemplos de questões que podem ser respondidas com esse tipo de estrutura:

  - Quais os principais componentes em execução e como eles interagem em tempo de execução?
  - Quais partes do sistema podem rodar em paralelo?
  - Quais partes do sistema são replicadas?

- **Estrutura de Alocação**

  Alocação é um tipo de estrutura que descreve como o sistema se relaciona com estruturas não-software, como CPU, sistema de arquivos, redes e equipes de desenvolvimento. Esse tipo de estrutura mostra o relacionamento entre os elementos de um software e os ambientes externos em que o software é criado e executado. Essa estrutura permite responder a questões como:

  - Em qual processador cada elemento de software é executado?
  - Quais os diretórios ou arquivos que cada elemento armazena durante o desenvolvimento, o teste e a criação do sistema?
  - Qual é a atribuição de cada elemento de software para equipes de desenvolvimento?

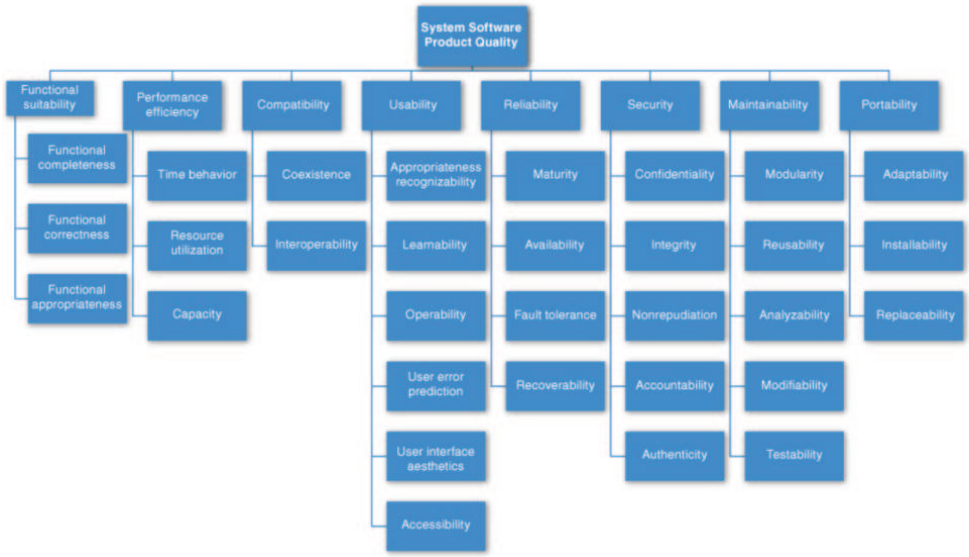
2.1.2 Atributos de Qualidade

Sistemas são normalmente redesenhados com frequência, não porque são funcionalmente deficientes e sim porque são difíceis de manter, portar, escalar ou pela falta de desempenho ou porque foram comprometidos por hackers. Nesse sentido, atributos de qualidade de software são os fatores que acarretam a mudança em um software.

Segundo Bass, Clements e Kazman (2012), atributos de qualidade de software é uma propriedade mensurável ou testável de um sistema que é utilizado para indicar o quão bem o sistema satisfaz às necessidades de seus *stakeholders*.

Neste trabalho serão descritos brevemente os atributos de qualidade especificados na norma ISO/IEC 25010, conforme Figura 2.

Figura 2 – Atributos de qualidade de software ISO/IEC 25010



Fonte: Bass, Clements e Kazman (2012)

- **Adequação Funcional**

  Esta característica representa o grau ao qual um produto ou sistema fornece funções que correspondam às necessidades explícitas e implícitas quando utilizado sob condições especificadas. Esta característica é composta das seguintes subcaracterísticas:

  – *Completude funcional*

    Grau em que o conjunto de funções abrange todas as tarefas especificadas e os objetivos do usuário.

  – *Correção funcional*

    Grau em que um produto ou sistema fornece os resultados corretos com o grau necessário de precisão.

  - *Adequação funcional*

    Grau em que as funções facilitam a realização de tarefas e objetivos específicos.

- **Eficiência de Desempenho**

  Esta característica representa o desempenho em relação à quantidade dos recursos utilizados sob condições estabelecidas. Ela é composta pelas seguintes subcaracterísticas:

  - *Comportamento do tempo*

    Grau em que os tempos de resposta, de processamento e as taxas de transferência de um produto ou sistema, ao efetuar as suas funções, atendem aos requisitos.

  - *Utilização dos recursos*

    Grau em que as quantidades e os tipos de recursos utilizados por um produto ou sistema, no desempenho das suas funções, atendem aos requisitos.

  - *Capacidade*

    Grau em que o limite máximo de um produto ou parâmetro do sistema atendem aos requisitos.

- **Compatibilidade**

  Grau em que um produto, sistema ou componente pode trocar informações com outros produtos, sistemas ou componentes ou, ainda, realizar suas funções necessárias, ao compartilhar o mesmo ambiente de hardware ou software. Esta característica é composta das seguintes subcaracterísticas:

  - *Coexistência*

    Grau em que um produto pode desempenhar as suas funções necessárias de forma eficiente ao compartilhar um ambiente com recursos comuns aos outros produtos, sem impacto negativo em qualquer outro produto.

  - *Interoperabilidade*

    Grau em que dois ou mais sistemas, produtos ou componentes podem trocar informações e usar as informações que tenham sido trocadas.

- **Usabilidade**

  Grau em que um produto ou sistema pode ser utilizado por usuários específicos para alcançar objetivos definidos com eficácia, eficiência e satisfação em um contexto de uso específico. Esta característica é composta das seguintes subcaracterísticas:

  - *Adequação Reconhecível*

    Grau em que os usuários podem reconhecer se um produto ou sistema é apropriado para suas necessidades.

- *Capacidade de Aprendizado*

  Grau em que um produto ou sistema pode ser utilizado por usuários específicos para alcançar determinados objetivos de aprendizagem. Esse grau permite usar o produto ou o sistema com eficácia, eficiência, liberdade de riscos e satisfação em um contexto de uso.

- *Operabilidade*

  Grau em que um produto ou sistema tem atributos que o tornam fácil de operar e controlar.

- *Proteção contra erro do usuário*

  Grau em que um sistema protege os usuários de cometer erros.

- *Estética da interface do usuário*

  Grau em que uma interface de usuário permite a interação agradável e satisfatória.

- *Acessibilidade*

  Grau em que um produto ou sistema pode ser usado por pessoas com maior variedade de características e recursos para alcançar um objetivo especificado em um contexto específico de uso.

- **Confiabilidade**

  Grau em que um sistema, o produto ou componente desempenha funções sob condições especificadas durante um período de tempo determinado. Esta característica é composta pelas seguintes subcaracterísticas:

  - *Maturidade*

    Grau em que um sistema, produto ou componente satisfaz às necessidades de confiabilidade em operação normal.

  - *Disponibilidade*

    Grau em que um sistema, produto ou componente está operacional e acessível quando necessário para utilização.

  - *Tolerância à falha*

    Grau em que um sistema, produto ou componente funciona como pretendido, apesar da presença de falhas de hardware ou software.

  - *Recuperabilidade*

    Grau em que, em caso de uma interrupção ou uma falha, um produto ou sistema pode recuperar os dados diretamente afetados e restabelecer o estado desejado do sistema.

- **Segurança**

  Grau em que um produto ou sistema protege informações e dados de modo que as pessoas ou outros produtos ou sistemas tenham o grau de acesso a dados adequados aos seus tipos e níveis de autorização. Esta característica é composta pelas seguintes subcaracterísticas:

  - *Confidencialidade*

    Grau em que um produto ou sistema garante que os dados são acessíveis apenas às pessoas autorizadas.

  - *Integridade*

    Grau em que um sistema, produto ou componente impede o acesso não autorizado ou alteração de programas de computador ou dados.

  - *Não-repúdio*

    Grau em que ações ou eventos podem ser provados de modo que os eventos ou as ações não possam ser repudiadas mais tarde.

  - *Responsabilidade*

    Grau em que as ações de uma entidade podem ser atribuídas exclusivamente à entidade.

  - *Autenticidade*

    Grau em que a identidade de um sujeito ou recurso podem ser comprovados ao serem reivindicados.

- **Manutenibilidade**

  Esta característica representa o grau de eficácia e eficiência com que um produto ou sistema pode ser modificado para melhorá-lo, corrigi-lo ou adaptá-lo às mudanças definidas no ambiente e nos requisitos. Esta característica é composta das seguintes subcaracterísticas:

  - *Modularidade*

    Grau em que um programa de sistema ou computador é composto de componentes discretos tal que uma mudança em um componente tem um impacto mínimo sobre outros componentes.

  - *Reusabilidade*

    Grau em que um ativo pode ser usado em mais de um sistema ou na construção de outros ativos.

  - *Analisabilidade*

    Grau de eficácia e eficiência com a qual é possível avaliar o impacto sobre um produto ou sistema de uma mudança destinada a uma ou mais de suas partes. Esse grau

permite também diagnosticar um produto para deficiências ou causas de falhas ou ainda identificar peças a serem modificadas.

– *Modificabilidade*

Grau em que um produto ou sistema pode ser modificado de forma eficaz e eficiente, sem a introdução de defeitos ou degradação da qualidade do produto existente.

– *Testabilidade*

Grau de eficácia e eficiência com que os critérios de teste podem ser estabelecidos para um sistema, produto ou componente. Nesse grau os testes podem ser realizados para determinar se esses critérios foram satisfeitos.

- **Portabilidade**

Grau de eficácia e eficiência com que um sistema, produto ou componente pode ser transferido de um hardware ou software para outro ambiente operacional. Esta característica é composta pelas seguintes subcaracterísticas:

– *Adaptabilidade*

Grau em que um produto ou sistema pode ser eficientemente adaptado para um hardware, software ou para outros ambientes operacionais, conforme a sua utilização. Esse grau é também conhecido como escalabilidade.

– *Instalabilidade*

Grau de eficácia e eficiência com que um produto ou sistema pode ser instalado ou desinstalado em um ambiente específico com sucesso.

– *Substituível*

Grau em que um produto pode substituir um outro produto de software especificado para o mesmo fim no mesmo ambiente.

2.2 Computação em Nuvem

Segundo Erl, Puttini e Mahmood (2013), o termo *computação em nuvem* remete às origens da computação de utilidade, um conceito que o cientista da computação John McCarthy propôs publicamente em 1961 através da seguinte reflexão:

> Se os computadores do tipo que eu defendo tornarem-se os computadores do futuro, a computação talvez algum dia será organizada como um serviço público, assim como o sistema telefônico é uma utilidade pública. A utilidade do computador poderia se tornar a base de uma indústria nova e importante.

Pode-se dizer que McCarthy estava correto em suas afirmações já que a computação em nuvem é utilizada hoje em dia como se fosse um serviço telefônico, ao qual o usuário paga em

conformidade com o seu consumo, o que mudou completamente a compreensão de computação que existia até então. O surgimento do conceito de computação em nuvem foi observado em prática em meados de 1990, quando empresas como Google e Yahoo começaram a fornecer serviços na internet. A partir de seus motores de busca, eles iniciaram a validação dos conceitos básicos da computação em nuvem.

Ainda nos anos de 1990 a Salesforce.com foi a primeira empresa a implantar serviços provisionados remotamente dentro das corporações e, em 2002, a Amazon lançava a plataforma Amazon Web Services (AWS), um conjunto de serviços orientados a empresas que fornece serviços de armazenamento, recursos de computação e funcionalidades empresariais.

Apesar de já existirem empresas utilizando os conceitos de nuvem foi somente no ano de 2006 que o termo computação em nuvem de fato surgiu na área comercial. Nesse ano a Amazon lançava o serviço de Elastic Compute Cloud (EC2). Esse serviço permitiu às empresas alugarem capacidades de computação e poder de processamento para executar suas aplicações corporativas.

2.2.1 O que é Computação em Nuvem?

Existem muitas definições relacionadas ao termo *Cloud Computing*, porém a mais aceita foi definida pela NIST:

> A computação em nuvem é um modelo para permitir acesso de rede onipresente, conveniente e sob demanda a um pool compartilhado de recursos de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem ser fornecidos e liberados rapidamente com pouco esforço de gerenciamento ou interação do provedor de serviços.

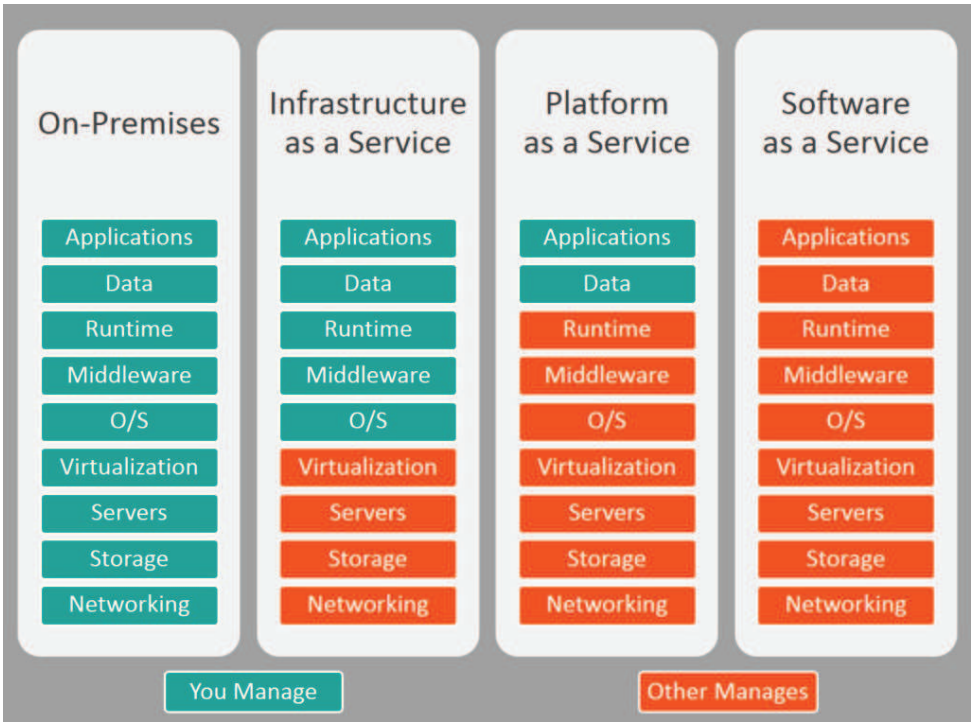
Conforme a NIST a computação em nuvem possui algumas características essenciais: autoatendimento sob demanda, acesso à rede ubíqua, agrupamento de recursos, independência de localização, elasticidade rápida e serviço de medição e Multi-tenancy. Segundo Erl, Puttini e Mahmood (2013), a computação em nuvem apresenta diversos benefícios, como: redução de investimentos e custos proporcionais, permite maior escalabilidade, disponibilidade e confiabilidade. Apesar desses benefícios, a nuvem também apresenta alguns riscos, como: maior vulnerabilidades de segurança e redução do controle de governança operacional. Além disso, a nuvem conta com alguns desafios, tais como: portabilidade limitada entre fornecedores de nuvem, conformidade multi-regional e questões legais entre diferentes estados e países.

2.2.2 Modelos de Serviços na Nuvem

Existem três principais modelos de serviços na nuvem: Software como um Serviço (SaaS), Plataforma como um Serviço (PaaS) e Infraestrutura como um Serviço (IaaS) (WITTIG; WIT-

TIG, 2015), conforme pode ser visto na Figura 3.

Figura 3 – Modelos de serviços da computação em nuvem



Fonte: http://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose

- **Software como um Serviço (SaaS)**

  O consumidor neste caso é um usuário final. Ele usa aplicativos que estão sendo executados em uma nuvem. As aplicações podem ser variadas, como e-mail, calendários, transmissão de vídeo e colaboração em tempo real.

  O consumidor não gerencia ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais, armazenamento ou mesmo recursos de aplicativos individuais. Pode-se citar como exemplos de SaaS o Amazon WorkSpaces, o Microsoft Office 365 e o Gmail.

- **Plataforma como um Serviço (PaaS)**

  O consumidor neste caso é um desenvolvedor ou administrador do sistema. A plataforma fornece uma variedade de serviços que o consumidor pode escolher. Esses serviços podem incluir várias opções, como banco de dados, balanceamento de carga, disponibilidade e ambientes de desenvolvimento. O consumidor implementa aplicações na infraestrutura da nuvem usando linguagens de programação e ferramentas suportadas pelo provedor.

  O consumidor não gerencia ou controla a infra-estrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais ou armazenamento, mas tem controle sobre as

aplicações implantadas e, possivelmente, configurações de ambiente de hospedagem de aplicativos. Alguns níveis de atributos de qualidade (por exemplo, tempo de atividade, tempo de resposta, segurança, tempo de correção de falhas) podem ser especificados por acordos de nível de serviço (SLAs). Pode-se citar como exemplo de PaaS o AWS Elastic Beanstalk, o Google App Engine e o Pivotal Cloud Foundry.

- **Infraestrutura como um Serviço (IaaS)**

  O consumidor neste caso é um desenvolvedor ou administrador do sistema. A capacidade fornecida ao consumidor é providenciar processamento, armazenamento, redes e outros recursos de computação, através dos quais o consumidor pode implantar e executar software arbitrário, incluindo sistemas operacionais e aplicativos. O consumidor pode, por exemplo, escolher criar uma instância de um computador virtual e fornecê-lo com alguma versão específica do Linux.

  O consumidor não gerencia ou controla a infraestrutura da nuvem subjacente, mas tem controle sobre sistemas operacionais, armazenamento, aplicativos implantados e possivelmente controle limitado de componentes de rede selecionados (por exemplo, firewalls de host). Novamente, os SLAs costumam ser usados para especificar atributos de qualidade chave. Pode-se citar como exemplo de IaaS o Amazon EC2, o Google Engine Compute e o Openstack.

2.2.3 Modelos de Implantação na Nuvem

Basicamente, existem dois modelos de implantação em nuvem, pública e privada, e duas variações desses modelos que são a nuvem híbrida e a nuvem da comunidade.

- **Nuvem Pública**

  A infraestrutura da nuvem é disponibilizada ao público em geral ou a um grande grupo industrial e é de propriedade de uma organização que comercializa serviços na nuvem.

- **Nuvem Privada**

  A infraestrutura da nuvem é propriedade exclusiva de uma única organização e operada exclusivamente para aplicativos de propriedade dessa organização. O objetivo principal da organização não é a venda de serviços na nuvem.

- **Nuvem Híbrida**

  A infraestrutura da nuvem é uma composição de duas ou mais nuvens (privadas, comunitárias ou públicas) que permanecem entidades únicas. O consumidor irá implantar aplicativos em alguma combinação da nuvem constituinte. Um exemplo disso é uma organização que utiliza uma nuvem privada, exceto por períodos em que os picos de carga

levam a atender alguns pedidos de uma nuvem pública. Essa técnica é chamada de "explosão de nuvem".

- **Nuvem da comunidade**

  A infraestrutura da nuvem é compartilhada por várias organizações e suporta uma comunidade específica que compartilhou preocupações (por exemplo, missão, requisitos de segurança, políticas e considerações de conformidade).

2.3 Computação Orientada a Serviço

Nesta seção serão apresentados os fundamentos de design de serviços, a definição da computação orientada a serviço, os modelos de serviços e os benefícios da computação orientada a serviço.

2.3.1 Fundamentos de Design

Segundo Erl (2007), para melhor compreender Computação Orientada a Serviço (SOC), é necessário definir algumas terminologias relacionadas ao design. Os principais fundamentos de design são: característica, princípio, paradigma, modelo, linguagem de modelo, padrão e boa prática.

- **Característica de Design**

  É uma qualidade ou um atributo específico de um corpo da lógica que se documenta em uma especificação e em um plano de design a serem entendidos no desenvolvimento.

- **Princípio de Design**

  É uma orientação altamente recomendável para dar forma à lógica de solução de maneira certa e com os objetivos certos em mente. Esses objetivos normalmente associam-se ao estabelecimento de uma ou várias características específicas de design.

- **Paradigma de Desing**

  É uma abordagem que rege o design da lógica, ou seja, é um conjunto de regras ou princípios complementares que define coletivamente a abordagem ampla representada pelo paradigma.

- **Modelo de Design**

  Descreve um problema comum e fornece uma solução correspondente. Essencialmente ele documenta a solução no formato de um modelo genérico a fim de que possa ser aplicada repetidamente.

- **Linguagem de Modelo de Design**

  É um cadeia de modelos de design relacionados que estabelecem uma sequência configurável à qual os modelos podem ser aplicados.

- **Padrão de Design**

  São convenções de design personalizadas, cuja função é permitir que se predeterminem, de modo consistente, as características do design da solução no suporte aos objetivos organizacionais otimizados para ambientes corporativos específicos.

- **Boa Prática**

  É uma técnica ou abordagem para resolver ou evitar certos problemas. A boa prática é uma prática que tem o reconhecimento do mercado e que surgiu da experiência.

2.3.2 Definição da Computação Orientada a Serviço

Segundo Erl (2007), computação orientada a serviço representa uma nova geração da plataforma da computação distribuída. Como tal, ela abrange muitas coisas, incluindo seu próprio paradigma de design, princípios de design, catálogos de modelos de design, padrões de linguagens, um modelo arquitetônico distinto, conceitos, tecnologias e frameworks relacionados.

A computação orientada a serviços possui seis elementos principais conforme Figura 4. São eles: arquitetura orientada a serviços, orientação a serviços, lógica orientada a serviços, serviços, composição de serviços e inventário de serviços.

- **Arquitetura Orientada a Serviços**

  Arquitetura orientada a serviço (SOA) é um modelo arquitetônico que visa a aprimorar a eficiência, a agilidade e a produtividade de uma empresa. Para tanto, utiliza os serviços como os principais meios para que a solução lógica seja representada no suporte à realização dos objetivos estratégicos associados à computação orientada a serviços.

- **Orientação a Serviços**

  É um paradigma de design que abrange um conjunto específico de princípios de design.

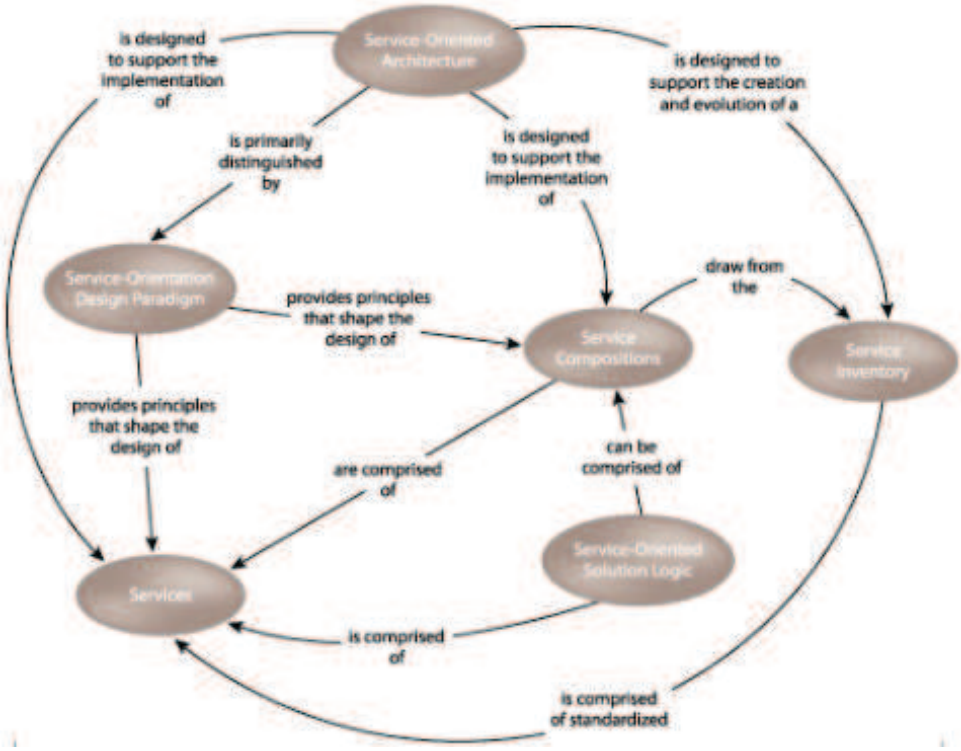
- **Lógica Orientada a Serviço**

  É o resultado da aplicação dos princípios de design ao design da lógica. A unidade mais fundamental da lógica orientada a serviços é o serviço.

- **Serviços**

  São programas de softwares fisicamente independentes, com características de design distintas que dão suporte à obtenção dos objetivos estratégicos associados à computação orientada a serviços. Cada serviço possui seu contexto funcional distinto e um conjunto de capacidades relacionadas a esse contexto.

Figura 4 – Service-Oriented Computing (SOC)



Fonte: Erl (2007)

- **Composição de Serviços**

  É um agregado coordenado de serviços. A agregação de serviços tem por objetivo fornecer funcionalidades requeridas para automatizar uma tarefa ou um processo específico de negócio.

- **Inventário de Serviços**

  É uma coleção padronizada e governada de maneira independente dos serviços que se complementam dentro de um limite que representa uma empresa ou um segmento significativo de uma empresa.

2.3.3 Modelos de Serviços

Segundo Erl (2007), existem três classificações comuns de serviços, que representam os modelos de serviços. São eles: serviço de entidade, serviço tarefa e serviço utilitário. Abaixo serão descritos resumidamente cada modelo.

- **Serviço de entidade**

  É um serviço centralizado no negócio que fundamenta o contexto e o limite funcional em uma ou mais entidades de negócios relacionadas. Esse modelo de serviço é considerado altamente reusável, porque ele é agnóstico à maioria dos processos da empresa. Nesse

caso, um único serviço pode ser reusado para automatizar uma série de processos de um negócio da empresa. Esse modelo de serviço é conhecido também como serviço de negócios centrados ou serviço de entidade de negócios.

- **Serviço tarefa**

  É um serviço de negócio com limite funcional diretamente associado a uma tarefa ou a um processo específico de uma empresa. Esse tipo de serviço tende a ser menos reutilizado, pois normalmente ele é posicionado como um controlador de uma composição de diversos serviços agnósticos. Esse modelo de serviço é conhecido também como serviço de processo, serviço de processo de negócio ou serviço de orquestração.

- **Serviço utilitário**

  É um serviço dedicado a fornecer funcionalidades reusáveis de serviço utilitário, como registro de eventos de logs, notificação e tratamento de exceções. Esse tipo de serviço é agnóstico a aplicativos, pois consiste em uma série de capacidades obtidas por diversos sistemas e recursos. Esse modelo de serviço é conhecido também como serviço de aplicativos, serviço de infraestrutura ou serviço de tecnologia.

2.3.4 Benefícios da Computação Orientada a Serviços

A computação orientada a serviço promete trazer grandes contribuições para organizações que desejam melhorar a eficácia no setor da sua área de TI. Um conjunto de benefícios comuns surgiu para que as empresas adotem com sucesso esse tipo de paradigma baseado em serviços.

Os principais benefícios estratégicos são: maior interoperabilidade intrínseca, maior federação, mais opções de diversificação de fornecedores, maior alinhamento do domínio do negócio e de tecnologia, maior retorno sobre o investimento, maior agilidade organizacional e menor carga de trabalho da TI.

2.4 Arquitetura Monolítica

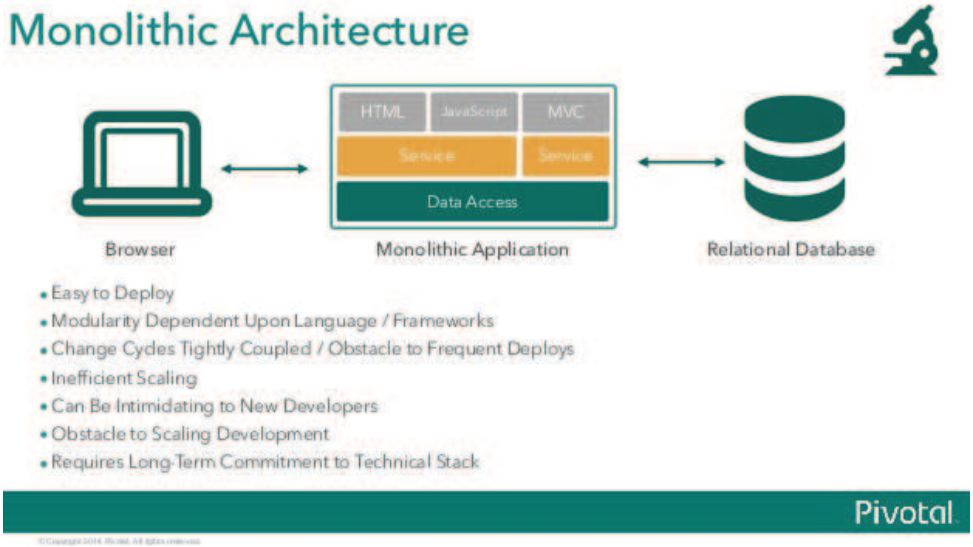
Linguagens de programação como Java, C/C++ e Python provêm abstrações para quebrar a complexidade dos programas em módulos. No entanto, essas linguagens são projetadas para criar um único artefato executável, o qual é chamado de monolítico.

Conforme Dragoni et al. (2017), "monolítico é uma aplicação de software cujo os módulos não podem ser executados independentemente". Assim, a arquitetura de um software monolítico tem a característica de compartilhar os recursos de uma mesma máquina, como memória, banco de dados e arquivos, ou seja, não existe um isolamento de uso de recurso por um determinado módulo da aplicação, todos utilizam o mesmo recurso.

Os componentes de uma arquitetura monolítica são separados em camadas lógicas dentro de uma aplicação. Na Figura 5 é possível identificar que os módulos e as camadas estão dentro de

uma única aplicação e todos os módulos da aplicação compartilham o mesmo banco de dados.

Figura 5 – Arquitetura monolítica



Fonte: https://www.slideshare.net/mstine/microservices-cf-summit

Segundo Dragoni et al. (2017), apesar de ter sido criado frameworks como RMI e CORBA para permitir o uso de programação distribuída em software com arquitetura monolítica, uma aplicação monolítica ainda apresenta alguns problemas, como: dificuldade de manutenção, alto acoplamento, dificuldade de implantação, falta de escalabilidade e homogeneidade tecnológica. Abaixo serão discutidos cada um dos problemas relatados.

- **Dificuldade de Manutenção**

  Aplicações monolíticas apresentam uma grande quantidade de linhas de código, o que acarreta o aumento da complexidade do software, maior dificuldade na sua compreensão e grandes chances de problemas no sistema. Consequentemente, isso acaba trazendo uma maior dificuldade na manutenção do software.

- **Alto acoplamento**

  Apesar de aplicações monolíticas apresentarem algum grau de modularidade, o modo de comunicação entre os componentes acaba gerando um alto grau de acoplamento, pois todos eles estão dentro da mesma aplicação.

- **Dificuldade de implantação**

  O alto grau de dependência entres os componentes de uma aplicação monolítica acaba afetando a implantação do software. A alteração de uma simples funcionalidade do sistema requer que o software seja completamente reiniciado, o que dificulta a implantação e gera um maior tempo de indisponibilidade do software devido ao seu grande tamanho.

- **Falta de escalabilidade**

  Aplicações monolíticas não permitem escalar uma determinada funcionalidade devido ao fato de os componentes estarem em um único executável. Para realizar a escalabilidade é necessário, então, escalar toda a aplicação, o que gera desperdício de recurso computacional, como processamento e memória. A alocação de recursos sob demanda, como observado em computação em nuvem, não pode ser implementada nesse estilo de aplicação, pois todos os módulos estão dentro de um mesmo sistema de software, o que inviabiliza a utilização de recursos como a elasticidade.

- **Homogeneidade tecnológica**

  Aplicações monolíticas ficam presas às tecnologias utilizadas no momento da concepção do software. Assim, não é possível trocar uma tecnologia de um determinado módulo do sistema, pois todo os módulos estão dentro da mesma aplicação, o que impõe aos desenvolvedores a utilização da linguagem de programação da aplicação.

2.5 Arquitetura Orientada a Serviço

Segundo Dragoni et al. (2017), a Arquitetura Orientada a Serviços (SOA) é a primeira geração de serviços baseada no paradigma de computação orientada a serviço. Segundo Xiao, Wijegunaratne e Qiang (2016), SOA surgiu no início dos anos 2000 e atingiu o seu maior nível de maturidade nos final da primeira década do milênio.

Existem diversas definições relacionadas a SOA. Para Xiao, Wijegunaratne e Qiang (2016), consiste em uma abordagem que surgiu através da necessidade de criar uma arquitetura distribuída baseada em componentes. Essa arquitetura possibilita maior agilidade do negócio e permite a configuração de uma solução a fim de atender uma necessidade particular de negócio e de fornecer mudanças a medida que as necessidades de negócio evoluem..

Já para MacKenzie et al. (2006), SOA é um paradigma para organizar e utilizar capacidades distribuídas que podem estar sob controle de diferentes domínios de propriedade. A SOA fornece uma maneira uniforme de oferecer, descobrir, interagir e utilizar capacidades para produzir efeitos desejados consistentes com pré-condições e expectativas mensuráveis.

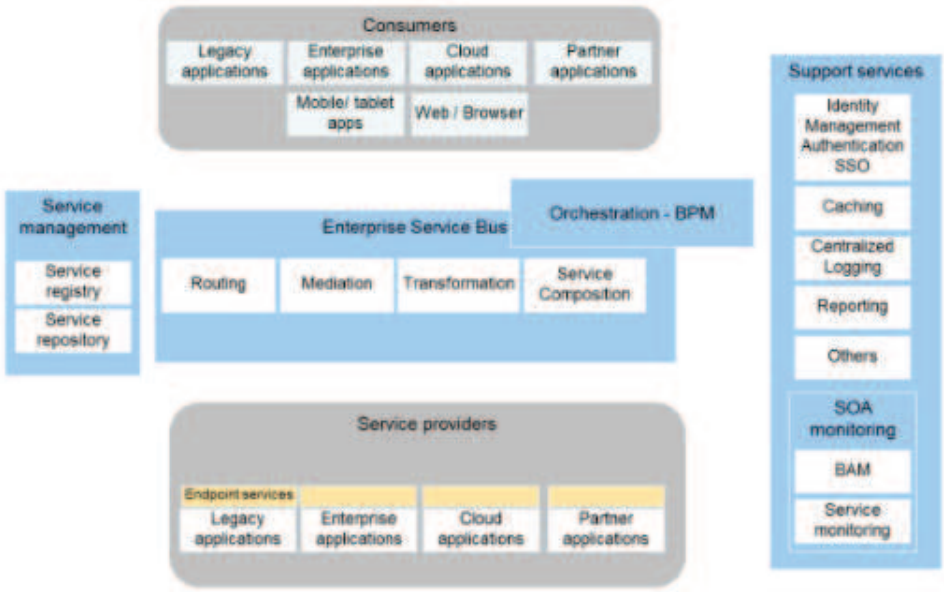
Empresas como a IBM definem SOA como sendo um framework de aplicação que permite quebrar aplicações em funções e processos de negócios individuais chamados de serviços.

As principais características do SOA segundo Xiao, Wijegunaratne e Qiang (2016), são: componentização utilizando serviços, baixo acoplamento, composição de serviços, abstração da plataforma e de tecnologias de infraestrutura, alto Desempenho, gerenciamento do ciclo de vida dos serviços, reutilização dos serviços e governança dos serviços.

Os principais componentes de uma arquitetura orientada a serviços são: serviços; ESB e monitoramento; gerenciamento de serviços, conforme a Figura 6.

- **Serviços**

Figura 6 – Componentes de uma Arquitetura Orientada a Serviços (SOA)



Fonte: Xiao, Wijegunaratne e Qiang (2016)

Um serviço é uma funcionalidade do negócio auto-contida que encapsula algumas funcionalidades que sejam significativas para o negócio.

- **Enterprise Service Bus (ESB)**

  É um software de infraestrutura que permite alta interoperabilidade entre os serviços. O ESB permite o roteamento entre os serviços, a mediação entre provedores e consumidores de serviços, a transformação de mensagens e a composição de serviços. O principal objetivo desse componente é permitir a disponibilização de serviços em um local único e centralizado.

- **Monitoramento e Gerenciamento de Serviços**

  É um software que possibilita a descoberta de serviços a partir de um registro de serviços. Esse software também permite a governança, o gerenciamento do ciclo de vida de serviços e o monitoramento da execução dos serviços.

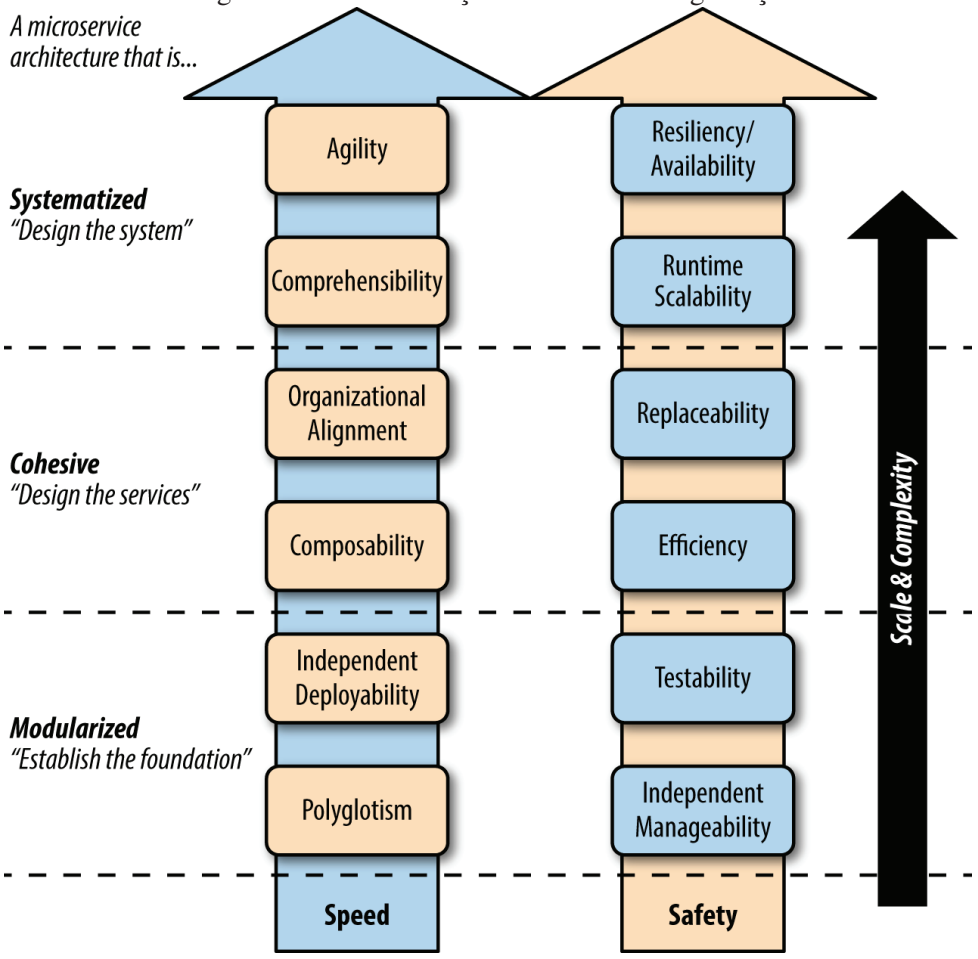
2.6 Arquitetura de Microsserviços

Conforme Dragoni et al. (2017), microsserviços é a segunda geração de serviços baseada em SOC e SOA. Trata-se de uma abordagem que utiliza conceitos bases do SOC, como a separação de preocupações, e remove as complexidades geradas pelo SOA, como utilização de ESB e orquestração de serviços centrada no processo de negócio, focando somente na programação de serviços simples com uma única responsabilidade.

Microsserviços é considerada uma abordagem simplificada para construção de sistemas dis-

tribuídos. Nos últimos anos, vem crescendo o interesse tanto na indústria como na academia por esse tema. Conforme Nadareishvili et al. (2016), um dos principais interesses na adoção de microsserviços deve-se à necessidade do equilíbrio entre velocidade e segurança no desenvolvimento de software. Segundo os autores, os softwares tendem a se tornarem mais complexos a medida que sua escala na forma de escopo, volume e interações do usuário aumenta. Para eles, a arquitetura de microsserviços está sendo utilizada para conseguir uma entrega mais rápida e uma maior segurança quando a escala do sistema aumenta, tal como pode ser observado na Figura 7.

Figura 7 – Microsserviço: Velocidade x Segurança



Fonte: Nadareishvili et al. (2016)

O termo *microsserviços* surgiu em 2011 em um workshop arquitetural como uma forma de descrever ideias comuns em padrões de arquitetura de software (FOWLER; LEWIS, 2014). Portanto, pode-se afirmar que microsserviços é uma arquitetura que ganhou popularidade recentemente, embora não exista ainda um consenso sobre o termo.

Para Dragoni et al. (2017), "microsserviço é processo coesivo e independente que interage via mensagem". Além disso, os autores definem *arquitetura de microsserviço* como sendo "uma aplicação distribuída onde todos os seus módulos são microsserviços".

Para Newman (2015), microsserviços podem ser considerados uma forma de realizar de-

composição funcional de um sistema grande (monolítico) em serviços totalmente autônomos, com alta coesão, baixo acoplamento, única responsabilidade e com um contexto bem delimitado. Segundo o autor, a decomposição funcional de uma aplicação e a equipe de desenvolvimento são os pontos-chave para a construção de uma arquitetura de microsserviços bem sucedidas. Newman (2015) relata ainda que a decomposição funcional permite maior agilidade, confiabilidade, escalabilidade, manutenibilidade, adaptabilidade e substituibilidade.

2.6.1 Definição de Microsserviços

Para Newman (2015), os microsserviços devem ser *pequenos, focados em fazer uma coisa bem* e *autônomos*.

- **Pequeno e Focado em Fazer uma Coisa Bem**

  Bases de código crescem ao implementar novas funcionalidades. No decorrer do tempo, torna-se difícil a manutenibilidade do software, pois a sua base de código está grande, complexa e também porque as funcionalidades semelhantes estão espalhadas por toda parte do código. Esse cenário descrito normalmente ocorre em aplicações monolíticas, devido ao fato de não ter uma delimitação do contexto do negócio em que a funcionalidade se encaixa.

  Microsserviços enfatizam a utilização do *princípio de responsabilidade única* que significa: "reúna as coisas que mudam pelo mesmo motivo e separe as coisas que mudam por motivos diferentes"(MARTIN; MARTIN, 2006). A utilização desse princípio somada à delimitação de contexto de negócio permite que o microsserviço execute uma única tarefa da melhor forma, conforme pode ser observado na Figura 8.

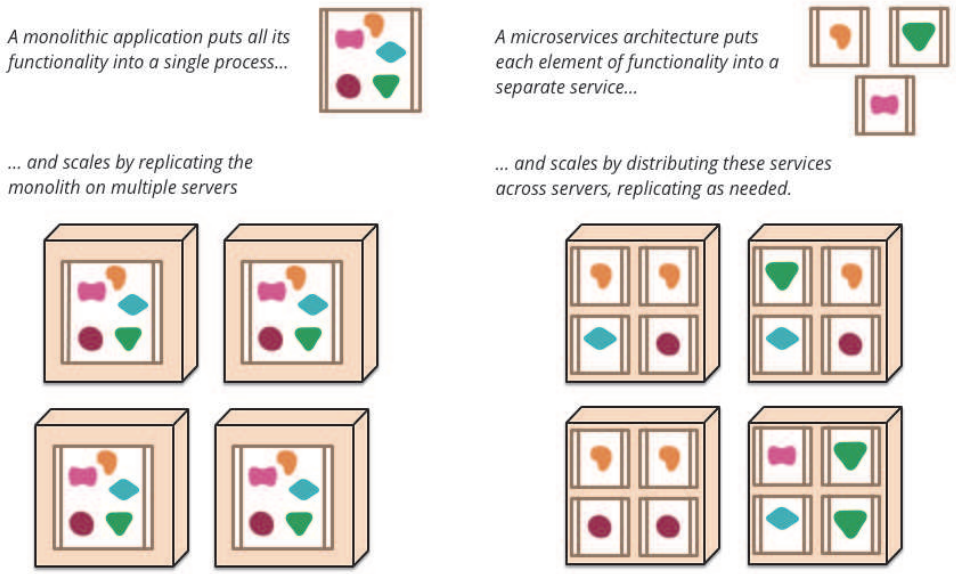
  Com relação a um microsserviço ter um tamanho pequeno, esse tipo de característica levanta diversas dúvidas, pois analisar o tamanho de um microsserviço pela quantidade de linhas de código é inviável, tendo em vista que algumas linguagens são mais verbosas do que as outras. Então, uma medida que pode ser utilizada para verificar se o microsserviços é pequeno é avaliar se uma equipe pequena conseguiria manter esse microsserviço. Se a base de código está tão grande que essa equipe não consegue mantê-la, então, o microsserviço não é pequeno e necessita ser quebrado em outros microsserviços.

- **Autônomo**

  Autônomo significa dizer que os microsserviços devem ser independentes, possuir uma alta coesão e um baixo acoplamento com outros microsserviços. Microsserviços necessitam ser implantados de forma independente em uma (PaaS) ou em um processo sistema operacional.

  Microsserviços precisam ser independente uns dos outros, ou seja, caso um microsserviço seja alterado, os microsserviços que o consomem não devem ser impactados. Assim, cada

Figura 8 – Decompondo uma aplicação monolítica em microsserviços



Fonte: https://www.martinfowler.com/articles/microservices.html

microsserviço tem que possuir seu próprio ciclo de vida, utilizar a rede para comunicação e disponibilizar uma API que permita a comunicação de forma totalmente desacoplada, conforme pode ser observado na Figura 9.

Portanto, para ter um microsserviço autônomo, é necessário que ele seja modelado da melhor forma e tenha a sua API exposta da melhorar forma para que ele tenha um ótimo desacoplamento.

2.6.2 Principais Benefícios de Microsserviços

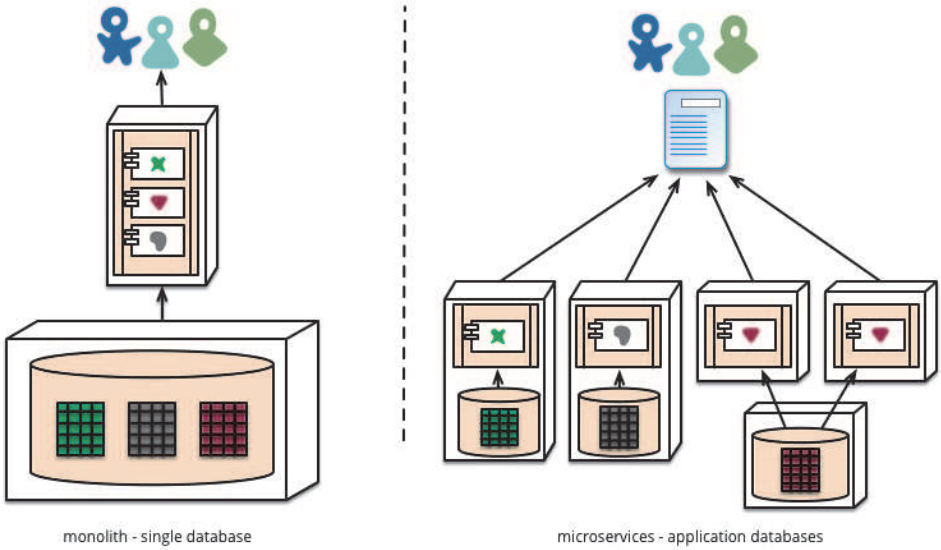
Newman (2015) cita sete benefícios-chave ao utilizar microsserviços: heterogeneidade tecnológica, resiliência, escalabilidade, fácil implantação, alinhamento organizacional, componibilidade e otimizando a substituição.

- Heterogeneidade Tecnológica

  A autonomia que os microsserviços trazem permite as equipes de desenvolvimento a adotar com mais frequência e velocidade uma nova tecnologia. A independência entre os microsserviços permite que desenvolvedores e arquitetos tenham mais liberdade para definir as tecnologias de que eles necessitam para solucionar um determinado problema.

  A Figura 10 demonstra um exemplo disso, no qual cada microsserviço representado pelos

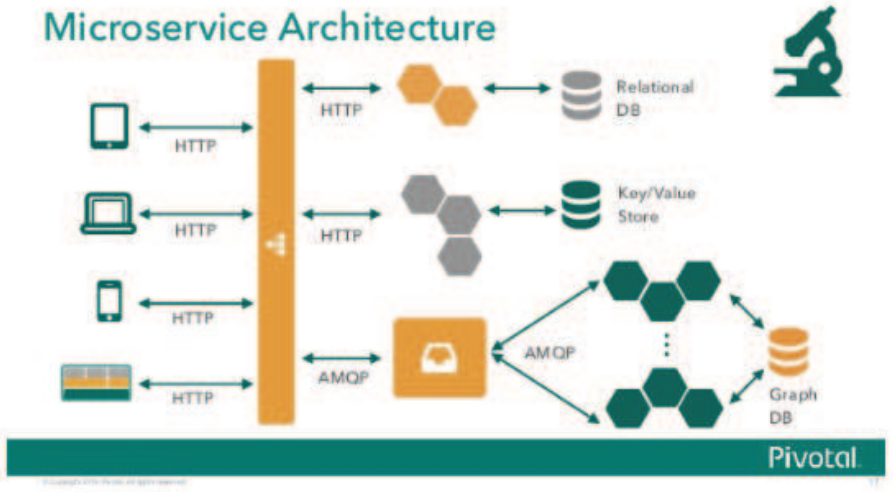
Figura 9 – Microsserviços: independentes e autônomos



Fonte: https://www.martinfowler.com/articles/microservices.html

os losangos na imagem, utilizam uma tecnologia de banco de dados e diferentes protocolos para comunicação. Nesse cenário, cada microsserviço poderá escolher a melhor tecnologia para resolver o problema sem que tenha que ficar dependente de uma única solução, o que permite a prática de uma arquitetura evolucionária.

Figura 10 – Heterogeneidade tecnológica com microsserviços



Fonte: https://www.slideshare.net/mstine/microservices-cf-summit

- Resiliência

  Ao desenvolver aplicações distribuídas, umas das certezas que existe é a de que, em um determinado momento, tanto a rede como a máquina poderão falhar.

  A arquitetura de microsserviços tende a ser tolerante a falhas, isso ocorre devido a sua característica de independência. Cada microsserviço poderá tratar de forma isolada a

interrupção que possa ocorrer, evitando assim a degradação da funcionalidade do sistema por completo como ocorre em aplicações monolíticas.

- Escalabilidade

  Os autores Abbott e Fisher (2015) descrevem um modelo de escalabilidade em três dimensões, conforme pode ser observado na Figura 11. Neste modelo, a abordagem mais comum para escalar uma aplicação é por meio da execução de múltiplas cópias idênticas da aplicação, acessíveis através de um balanceador de carga. Conhecida como escalabilidade horizontal, corresponde à escalabilidade no eixo X (X-axis scaling). A escalabilidade no eixo Z (Z-axis scaling) é similar a do eixo X, em que cada servidor executa uma cópia idêntica do código. A grande diferença é que cada servidor é responsável por seu conjunto dos dados. As abordagens Z-axis scaling e X-axis scaling ampliam a capacidade e a disponibilidade da aplicação. Entretanto, nenhuma delas resolve os problemas do aumento da complexidade do desenvolvimento e da própria aplicação.

  Para solucionar esses problemas, é preciso aplicar a escalabilidade de eixo Y (Y-axis scaling). A terceira dimensão de escalabilidade é denominada decomposição funcional ou (Y-axis scaling) aplicada sobre o eixo Y. Enquanto a escalabilidade de eixo Z divide elementos semelhantes, a escalabilidade de eixo Y divide elementos diferentes. A camada de aplicação, (Y-axis scaling) divide uma aplicação monolítica em um conjunto de microsserviços.

  Ao dividir uma aplicação em microsserviços permite-se uma escalabilidade de forma granular, ou seja, se um microsserviço necessitar de mais recurso computacional, como memória ou CPU, somente ele será escalado e não todos os componentes tal como observado em aplicações monolíticas. Segundo Alshuqayran, Ali e Evans (2016), microsserviço é considerada uma arquitetura apropriada para computação em nuvem, pois ela permite tirar melhor proveito da elasticidade e do provisionamento de recursos sob demanda disponíveis em uma plataforma de computação em nuvem.

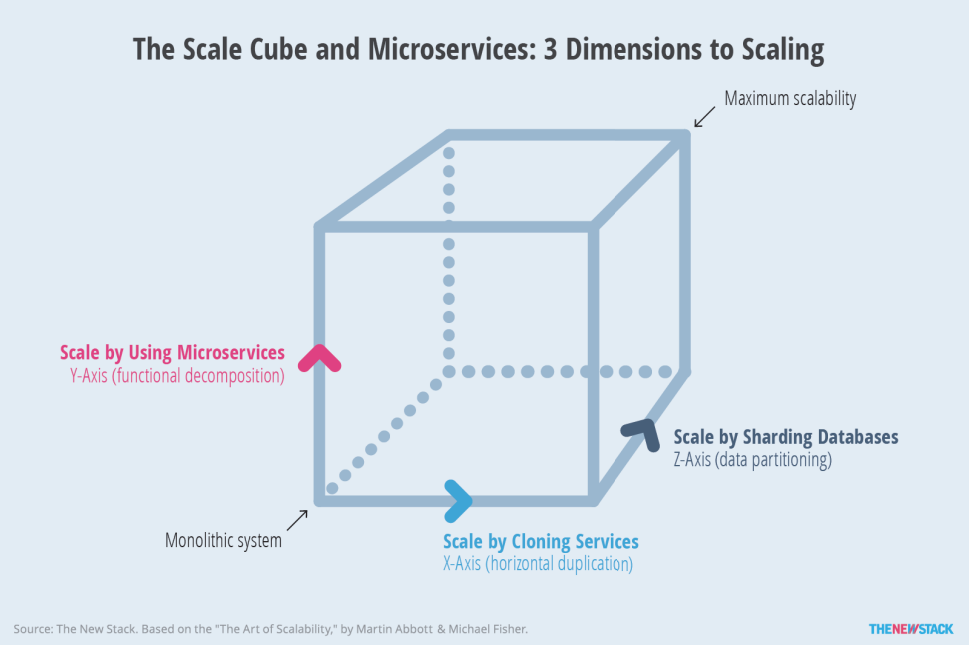
- Fácil Implantação

  Microsserviços podem ser implantados de forma mais simples, pois são independentes e autônomos uns dos outros. Essa característica de autonomia permite que um sistema em geral tenha um menor tempo de indisponibilidade quando é realizada uma mudança em determinada funcionalidade.

  Como cada microsserviço possui uma única responsabilidade e um melhor gerenciamento no seu ciclo de vida, ao implantar uma nova funcionalidade, é mais simples voltar atrás quando um problema ocorre em uma arquitetura de microsserviços, pois, caso ocorra um erro, somente o microsserviço que casou o erro precisa ser reparado.

- Alinhamento Organizacional

Figura 11 – Microsserviços: Três dimensões de escalabilidade



Fonte: https://content.pivotal.io/blog/supercharging-your-scale-cube-caching-microservices-on-pivotal-cloud-foundry

Microsserviços permitem um melhor alinhamento organizacional devido a suas características de tamanho, autonomia e delimitação de contexto. Um microsserviço tende a possuir uma base de código menor e códigos menos complexos, permitindo assim criar times de desenvolvimento mais focados em um determinado contexto de negócio e com uma maior produtividade.

- Componibilidade

  Umas das promessas de sistemas distribuídos e SOA é a oportunidade de reuso de funcionalidades. Microsserviços permitem a composição de funcionalidades no nível máximo, pois cada microsserviço pode ser consumido de diferentes formas e para diferentes propósitos. Microsserviços são agnósticos a uma determinada tecnologia, eles podem ser consumidos por qualquer tipo de aplicação.

- Otimizando a Substituição

  Microsserviços permitem de forma menos traumática a reescrita ou até a remoção de uma funcionalidade de um sistema. Isso ocorre porque eles são menores e autônomos. Assim, desenvolvedores ficam mais encorajados a dar manutenção em uma base de código menor do que a uma base de código grande de uma aplicação monolítica.

3 TRABALHOS RELACIONADOS

Neste capítulo, será realizada uma análise comparativa dos trabalhos relacionados utilizados nesta pesquisa. Essa análise tem por objetivo identificar critérios em comum entre os trabalhos relacionados levantados a partir de um estudo sobre o estado da arte na temática de decomposição de aplicações monolíticas em microsserviços. Para tanto, este capítulo está organizado da seguinte forma: na seção 3.1, será descrita a metodologia para a escolha dos trabalhos relacionados; na seção 3.2, será analisado o estado da arte deste trabalho; na seção 3.3, serão comparados os trabalhos relacionados com base nos critérios comparativos definidos e, por fim, na seção 3.4, serão informadas as oportunidades de pesquisa identificadas nos trabalhos.

3.1 Metodologia para a Escolha dos Trabalhos

Conforme discutido neste trabalho, existem ainda muitas dúvidas referentes à temática sobre a decomposição de aplicações monolíticas em microsserviços, tendo em vista que o tema ainda é muito novo no contexto acadêmico e que poucas pesquisas foram realizadas sobre o assunto. A partir dos estudos realizados, identificou-se que ainda existem algumas lacunas a serem preenchidas por novas investigações a respeito desse tema.

Diante dessa problemática, este trabalho pesquisou as principais bibliotecas digitais de publicações de pesquisa na área da computação, a fim de encontrar trabalhos referentes especificamente à decomposição de aplicações monolíticas em microsserviços. Para tanto, foram consultadas as principais bibliotecas digitais, conforme demonstrado na Tabela 1.

Tabela 1 – Tabela com as bibliotecas digitais consultadas por esta pesquisa

Biblioteca Digital	**Endereço Web**
ACM Digital Library	https://dl.acm.org
CiteSeer Library	http://citeseerx.ist.psu.edu
Google Scholar	https://scholar.google.com.br
IEEE Explorer	https://ieeexplore.ieee.org
Science Direct	https://www.sciencedirect.com
Springer Link	https://link.springer.com
Wiley Online Library	https://onlinelibrary.wiley.com

Para realizar a busca nas bibliotecas citadas, foi utilizada a *query* de pesquisa com as seguintes palavras-chave:

(*microservices*) **AND** (*"monolithic application"* **OR** *"monolith application*") **AND** (*"application decomposition"* **OR** *"decompose application"* **OR** *"extract application"* **OR** *"exctracting application"* **OR** *"extraction application"*)

Com base nos resultados obtidos na busca, ainda foram aplicados alguns filtros para encontrar os artigos com maior relação com o objetivo deste trabalho. O primeiro filtro aplicado foi considerar somente artigos que possuíam no título ou no abstract os termos citados ante-

riormente. Depois foram filtrados somente trabalhos que tinham como objetivo fim realizar a decomposição de uma aplicação monolítica em microsserviços, utilizando algum tipo de técnica, processo ou metodologia. O motivo de restringir especificamente a esse assunto é para permitir a comparação entre a técnica proposta neste trabalho com os trabalhos encontrados. Além das restrições aplicadas referentes à temática dos artigos, foram aplicados mais dois filtros: o primeiro refere-se à seleção de trabalhos que foram escritos no idioma inglês e o último filtro aplicado foi referente a selecionar somente artigos publicados entre os anos de 2014 e 2018, já que, segundo Pahl e Jamshidi (2016), antes de 2014, não existia um consenso sobre o termo microsserviço. A partir dos critérios de busca e dos filtros aplicados, foram selecionados **10 artigos** para ser realizada a análise comparativa entre eles.

A seção 3.2 descreverá brevemente cada artigo selecionado e a seção 3.3 apresentará os critérios utilizados para comparar os trabalhos a fim de identificar as oportunidades de pesquisa.

3.2 Análise do Estado da Arte

Nesta seção, serão descritos resumidamente os 10 trabalhos selecionados que representam o estado da arte em decomposição de aplicações monolíticas em microsserviços.

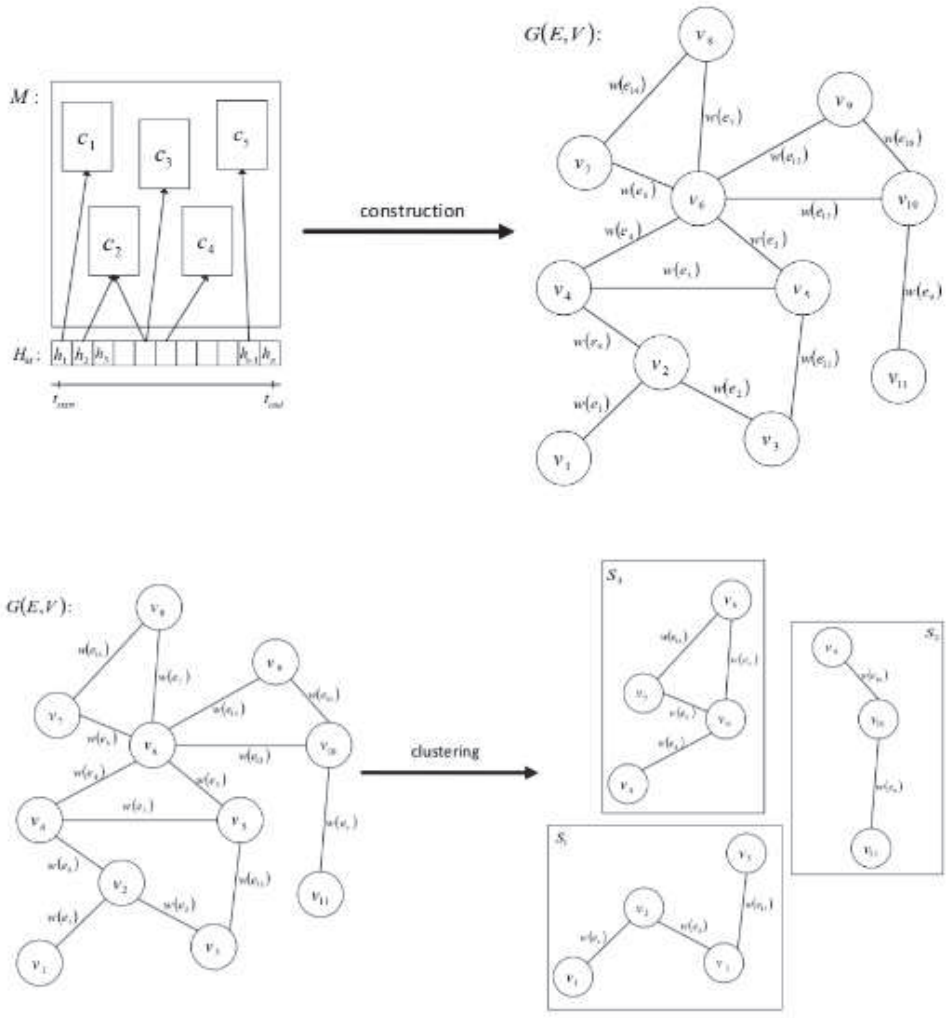
3.2.1 *Extraction of Microservices from Monolithic Software Architectures*

No trabalho de Mazlami, Cito e Leitner (2017), os autores propuseram um processo de extração semiautomático que utiliza um algoritmo baseado em grafo para decomposição de aplicações monolíticas em microsserviços. O modelo proposto apresenta três estágios de extração, são eles: **estágio do monolítico**, **estágio do grafo** e **estágio dos microsserviços**. Entre esses estágios existem duas etapas de transformação: transformação do monólito em grafo e a transformação do grafo em recomendação de microsserviços, conforme pode ser observado na Figura 12.

O ponto de partida do processo inicia-se, quando o modelo analisa um repositório de código fonte versionado através de um sistema de controle de versão (Git). A partir da análise do código fonte, é gerada uma representação da aplicação monolítica baseada no modelo **M = (CM, HM, DM)**. O **M** no modelo significa a aplicação monolítica; o **CM** é o conjunto de classes da aplicação monolítica; o **HM** é o histórico de mudanças das classes da aplicação e o **DM** é o conjunto de desenvolvedores que contribuíram na implementação da aplicação. O histórico das alterações das classes (HM) e os desenvolvedores que realizaram essas alterações (DM) foram obtidos através da ferramenta de controle de versão Git.

Com base nos dados coletados das classes, do histórico de mudanças e dos desenvolvedores, foi gerado um grafo não direcionado ponderado, no qual os vértices representam as classes do monolítico e as arestas são representadas por pesos que identificam o grau de acoplamento entre duas classes (vértices) da aplicação. Para calcular o peso inserido nas arestas do grafo,

Figura 12 – Etapa de criação e decomposição do Grafo



Fonte: Mazlami, Cito e Leitner (2017)

os autores utilizaram três tipos de estratégias: a primeira refere-se a analisar quais as classes foram alteradas em um determinado evento no histórico de alteração da aplicação. As classes que foram modificadas em um determinado evento no histórico de alteração apresentam um maior peso de acoplamento entre elas e logo devem ficar juntas dentro de um microsserviço. A segunda estratégia refere-se à análise semântica das classes, ou seja, os autores realizam uma análise em cada classe do sistema em busca de termos em comum entre os domínios da aplicação. O objetivo dessa análise é avaliar o quanto uma classe está relacionada a um determinado domínio da aplicação e qual o nível de acoplamento dela dentro daquele domínio. A última estratégia utilizada para calcular os pesos das arestas é verificar o quanto que um desenvolvedor está acoplado no desenvolvimento das classes da aplicação, ou seja, quantos desenvolvedores alteraram aquela classe durante o desenvolvimento do sistema.

Com base nos resultados obtidos através do uso das três estratégias, é calculado e atribuído o valor do peso para cada aresta do grafo gerado. A partir desse grafo, é aplicado o algoritmo

proposto pelos autores que se baseia no algoritmo de *Kruskal*. O algoritmo proposto no trabalho gera subgrafos que são obtidos através da decomposição do grafo com base nos maiores pesos encontrados. Cada subgrafo gerado é uma recomendação de microsserviço, conforme pode ser observado na Figura 12.

Para avaliar o algoritmo proposto, os autores executaram-no em 21 aplicações *open source*, em diferentes tipos de linguagem, como: *Java*, *Python* e *Ruby*. Nessas aplicações, foi verificado o desempenho dos microsserviços recomendados quando comparado ao monolítico e também foi verificado o quanto que o domínio de dados de um determinado microsserviço foi replicado entre os outros microsserviços. Segundo os autores, os resultados obtidos foram satisfatórios, porém indicaram como limitação a falta de análise do código fonte a nível de método e funções. Para eles, a falta dessa verificação não possibilitou a geração de microsserviços com uma melhor granularidade. Como trabalho futuro, os autores sugerem a análise de código a nível de métodos e também a possibilidade de configurar a recomendação.

3.2.2 *From Monolith to Microservices: A Dataflow-Driven Approach*

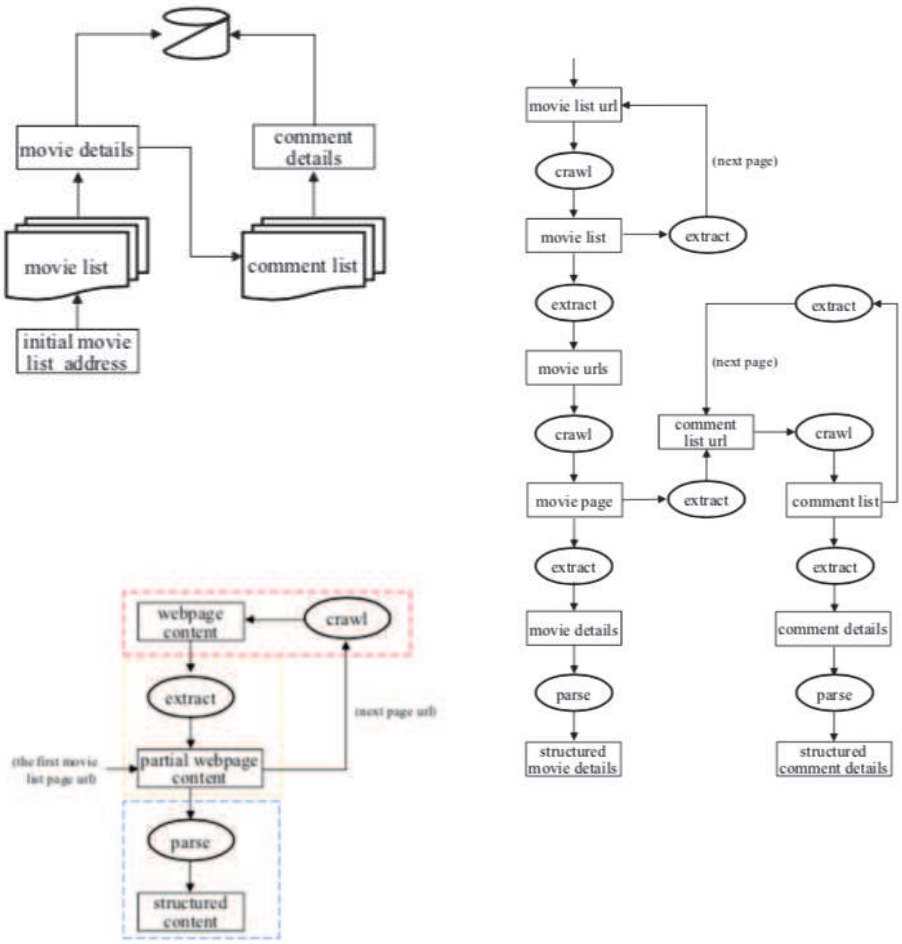
No trabalho de Chen, Li e Li (2017), os autores desenvolveram um processo semiautomático baseado num algoritmo que utiliza como entrada de dados um diagrama de fluxo de dados (DFD) para realizar a decomposição das regras de negócio da aplicação em microsserviços. Esse processo é composto por três etapas, incluindo uma manual, que é referente ao desenho do diagrama chamado de purificação de fluxo de dados, e duas etapas referentes à decomposição dirigida por fluxo de dados.

A primeira etapa, conforme mencionado anteriormente, refere-se à construção do diagrama de fluxo de dados purificado. Nessa etapa, os autores propõem a construção de um diagrama de fluxo de dados padrão (DFD) como início do processo. Após a construção do DFD, são aplicadas regras predefinidas desenvolvidas pelos autores para gerar o DFD purificado. Esse diagrama purificado é um DFD que tem como foco a semântica dos dados e as suas operações, ou seja, não considera componentes referentes ao armazenamento de dados e entidades externas, componentes esses padrões nos diagramas DFD. A segunda etapa é referente à condensação dos diagramas DFD purificados para um fluxo de dados decomponível (*Decomposable DFD*). Nessa etapa, os autores aplicam um algoritmo que verifica o DFD purificado e combina as mesmas operações com os mesmos tipos de saída de dados para um fluxo de dados que pode ser decomposto. Esses fluxos de dados gerados são os potenciais candidatos a microsserviços. A última etapa refere-se à identificação dos microsserviços a partir da representação do DFD purificado no formato de grafo. Para gerar a recomendação dos microsserviços, é aplicado o algoritmo proposto pelos autores que realiza a decomposição do grafo com base nos módulos, nas operações e na saída de dados do DFD decomponível.

Para avaliar o processo de decomposição proposto, os autores do trabalho realizaram um estudos de caso numa aplicação de informações de filme, a partir do qual foram selecionados

dois módulos de regras de negócio da aplicação. Os objetivos dessa avaliação foram, primeiro, explicar o passo a passo de como aplicar na prática a técnica proposta, conforme pode ser observado na Figura 13, e o segundo objetivo foi comparar os resultados da técnica proposta com os da técnica de *Service Cutter* (GYSEL et al., 2016).

Figura 13 – Decomposição dirigida por fluxo de dados



Fonte: Chen, Li e Li (2017)

Segundo os autores, os resultados apresentados, ao aplicar o DFD no projeto de informações de filmes, foram satisfatórios. O número de microsserviços gerados bem como a disposição do modelo de domínio foram melhores apresentados pela técnica de fluxo de dados quando comparados aos do *Service Cutter*, pois o Service Cutter acabou separando componentes que não poderiam estar separados em microsserviços diferentes. Como limitação do trabalho, os autores indicaram a necessidade de um especialista para definição e construção dos DFDs e também a falta de uma avaliação da técnica em projetos reais de grande porte. Como trabalhos futuros, eles ressaltaram a necessidade de implementar um mecanismo que possibilite o refinamento ou ajuste de granularidade dos microsserviços gerados.

3.2.3 *From Monolithic to Microservices: An Experience Report from the Banking Domain*

Neste trabalho, os autores Bucchiarone et al. (2018) propuseram uma arquitetura para realizar a migração do sistema bancário do Danske Bank's de uma arquitetura monolítica para uma arquitetura baseada em microsserviços. Segundo os autores, a migração das funcionalidades para microsserviços ocorreu uma de cada vez e a definição de quais seriam migradas primeiro foi obtida através de entrevistas com as partes interessadas do sistema (*FX Traders*). A partir dessas entrevistas e da análise do código fonte da aplicação, os autores começaram a definir quais as funcionalidades deveriam ficar juntas ou separadas e quais delas seriam migradas para microsserviços.

Conforme pode ser observado na Figura 14, os autores definiram uma arquitetura como base de microsserviços para migração das aplicações bancárias. Segundo eles, a modernização das aplicações para esse novo tipo de arquitetura trouxe alguns benefícios tanto a nível de processo como a nível de aspecto arquitetural. Os benefícios alcançados durante essa modernização foram: processo de integração e implantação contínua dos serviços, através do uso *pipelines* de automação; alta disponibilidade dos serviços obtidos pelo uso de orquestradores de *containers* (*Docker Swarm*); e integração entre os serviços através da troca de mensagens por coreografia (*RabbitMQ*).

Figura 14 – Arquitetura de microsserviços proposta no Danske Bank's

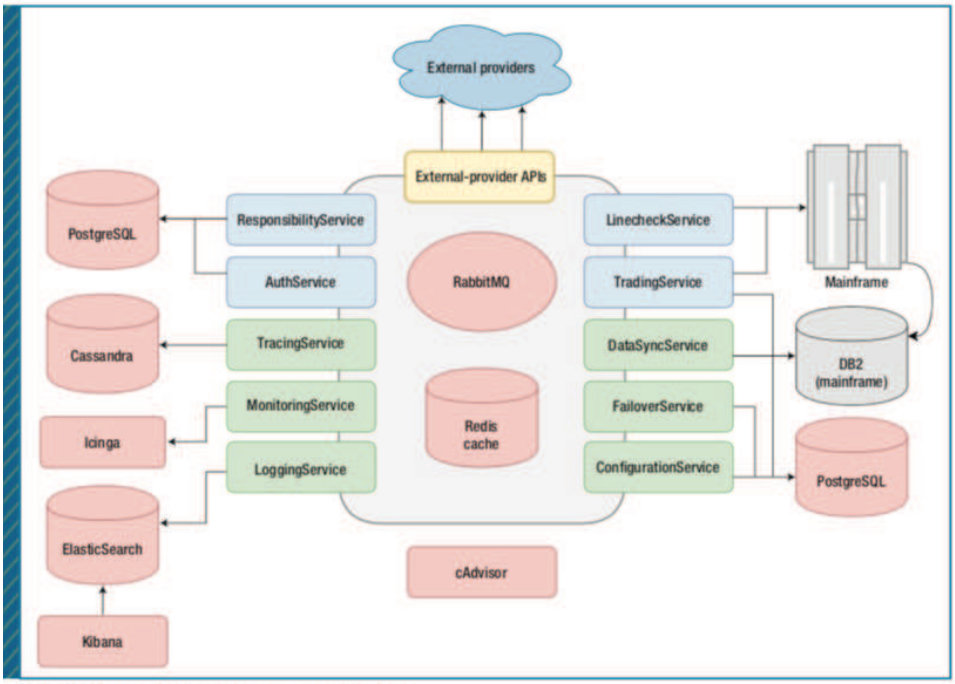


FIGURE 2. The new FX Core microservice architecture.

Fonte: Bucchiarone et al. (2018)

Os autores citaram ainda que vários problemas que ocorriam com aplicações monolíticas foram resolvidos com o uso de uma arquitetura de microsserviços como, por exemplo: a flexi-

bilidade de escolha de uma tecnologia para resolver um determinado problema que um serviço possuía; a facilidade na implantação de serviços ao utilizar *docker containers* e a centralização de logs e métricas de monitoramento. Para eles, a centralização de logs e métricas de monitoramento permitiu às equipes de desenvolvimento uma melhor visão sobre os serviços e, em consequência, a atuação proativa dessas equipes, quando um comportamento não esperado ocorresse.

Por fim, os autores relatam que microsserviços também trazem diversos desafios que as aplicações monolíticas não apresentam. Dentre eles, ressaltam que os microsserviços começaram a ter valor dentro da empresa, quando conseguiram resolver problemas relacionados ao monitoramento, à tolerância à falha e à concorrência. Isso contribuiu para uma melhor visibilidade desse tipo de arquitetura.

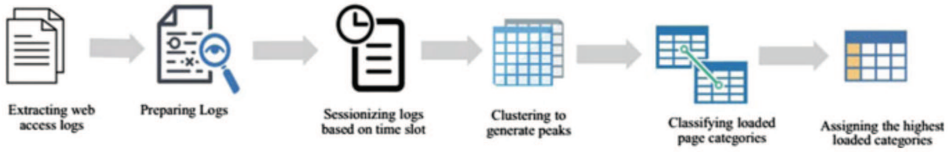
3.2.4 *GranMicro: A Black-Box Based Approach for Optimizing Microservices Based Applications*

Neste trabalho, os autores Mustafa et al. (2018) propõem um processo manual a fim de encontrar a melhor granularidade dos microsserviços ao decompor uma aplicação monolítica. A técnica proposta pelos autores para encontrar a melhor granularidade dos microsserviços é baseada na mineração do log de acesso web da aplicação. Esse processo é composto por seis etapas, conforme pode ser observado na Figura 15. A primeira etapa é referente à extração do log de acesso do servidor web da aplicação; a segunda etapa refere-se à remoção de URLs inválidas do log de acesso; a terceira etapa determina quais os principais períodos de acesso, ou seja, em quais os períodos de tempo que ocorreram a maior carga de trabalho da aplicação; na quarta etapa, são definidos e agrupados os períodos de tempo de acesso a fim de gerar os picos de acesso, isto é, qual a carga de trabalho que ocorreu em um determinado período (picos); na quinta etapa, são identificadas e classificadas as páginas web da aplicação que fizeram parte dos picos de acesso, conforme pode ser observado na tabela da Figura 15; já na última etapa, gera-se a recomendação de microsserviços baseada nas páginas com maior percentual de acesso.

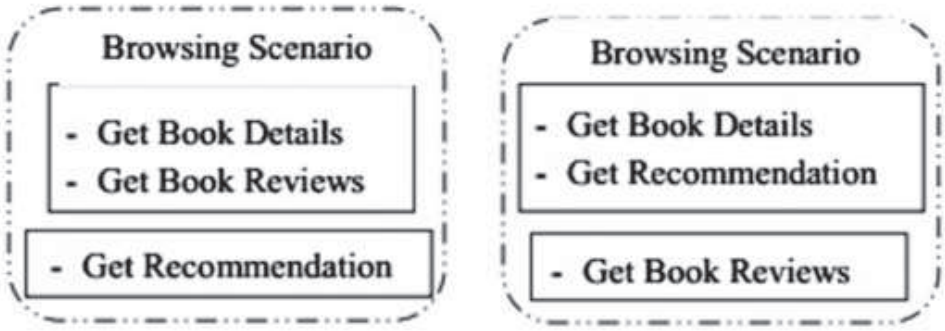
Os autores relatam nesse trabalho que, para alcançar uma granularidade ideal de microsserviços, é necessário levar em conta requisitos não funcionais, como desempenho. Eles citam ainda que a maioria das técnicas utilizadas para decomposição em microsserviços analisa somente o modelo de domínio, sem levar em consideração questões de desempenho. Nesse trabalho, eles mostram um comparativo entre decompor uma aplicação sem o uso da técnica GranMicro e com a técnica GranMicro. Como alvo da decomposição, os autores escolheram uma aplicação desenvolvida por eles chamada Micro-BookShop. Na Figura 15, são apresentados dois cenários (*Browsing Scenario*), o primeiro deles mostra a decomposição da aplicação sem o uso da técnica proposta (caixa da esquerda) e o segundo cenário apresenta o uso da técnica Gran-Micro (caixa da direita).

Os autores destacam que, ao levar em consideração a carga de trabalho no momento da de-

Figura 15 – Processo de decomposição proposto pelo GranMicro



	Reviews page	Cart page	Recomm.
Peak1 (5000 U/M)	50%	20%	30%
Peak2 (10000 U/M)	38%	18%	44%
Peak3 (15000 U/M)	52%	23%	25%
AVG.	46.6%	20.3%	33%



Fonte: Mustafa et al. (2018)

composição, identificou-se que a página web referente a *Book Reviews* deveria ser recomendada como um microsserviço completamente isolado das outras páginas, pois essa página foi a que maior apresentou carga de trabalho em média, conforme pode ser observado na tabela da Figura 15. Eles mostram que, se não for levada em consideração a questão de carga de trabalho e o desempenho, a recomendação de microsserviços pode ser prejudicada, pois uma funcionalidade pode afetar o desempenho de outras funcionalidades, se estiverem juntas.

Para validar o processo de decomposição proposto e verificar a influência de requisitos não funcionais, como desempenho na recomendação de microsserviços, os autores utilizaram uma aplicação de teste desenvolvida por eles chamada Micro-BookShop. Nessa aplicação, foram executados testes de carga com os seguintes cenários: 5000 usuários por minuto; 10000 usuários por minuto; e 15000 usuários por minuto. Ao executar esses teste, os autores identificaram que a decomposição gerada pela técnica, quando comparada com uma decomposição sem o uso da técnica, permitiu uma redução no uso de CPU e também no tempo de resposta das páginas, o que evidencia uma grande contribuição ao considerar aspectos de desempenho no momento da recomendação.

Por fim, eles concluem que o Gran-Micro apresentou dois pontos fortes, o primeiro é que se

trata de uma técnica que não necessita utilizar o modelo de negócio e segundo é que a técnica leva em consideração requisitos não funcionais como desempenho. Como trabalhos futuros, eles propõem implementar o processo proposto em código e também executar uma avaliação numa aplicação web grande.

3.2.5 *Microservices Architecture: Case on the Migration of Reservation-based Parking System*

Neste trabalho, os autores Yugopuspito, Panduwinata e Sutrisno (2017) propõem uma arquitetura para realizar a migração de uma aplicação monolítica para uma arquitetura baseada em microsserviços. Como alvo da migração, os autores escolheram um sistema de reserva de vagas de estacionamento num supermercado. O objetivo dessa migração é permitir que o sistema possua uma maior escalabilidade e também possibilite que novas funcionalidades sejam desenvolvidas e mantidas com menor esforço. O sistema apresentado como estudo de caso contém quatro domínios principais: *Membership*, *Reservation*, *Parking* e *Payment* e todos eles utilizam uma única base de dados para o armazenamento. Para realizar o desacoplamento dos dados entre os domínios, os autores propuseram a decomposição de cada domínio em um microsserviço. Assim, cada microsserviço terá a sua própria base de dados e a integração entre eles será realizada através de APIs REST, conforme pode ser observado na Figura 16.

Segundo os autores, todos os serviços foram implementados seguindo boas práticas, como: princípio da responsabilidade única; segregação de interfaces; e também a aplicação da metodologia de desenvolvimento baseada no *Domain Driven Design* para identificação dos contextos. Ao iniciar as implementações de integrações entre os serviços, os autores identificaram alguns problemas relacionados à comunicação diretamente via API REST. Eles perceberam que a comunicação direta entre os microsserviços através de suas API REST estavam gerando um alto acoplamento novamente entre os serviços e também estavam dificultando a manutenção a medida que novos serviços eram desenvolvidos. Tendo em vista esse problema, os autores implementaram um padrão chamado *API Gateway*, que possibilita uma única forma de entrada para os clientes que desejam consumir os microsserviços. Através desse padrão utilizado, os autores resolveram problemas referentes à segurança e ao monitoramento dos microsserviços.

Com relação à segurança entre os microsserviços e ao consumo deles, os autores utilizaram os protocolos de autorização OAuth2 e OpenID. Esses protocolos foram utilizados para que aplicação *mobile* conseguisse consumir através do protocolo HTTP de forma segura os dados expostos pelas APIs REST dos microsserviços. Por fim, os autores sugeriram o uso de um *service registry* e um balanceador de carga para permitir que os microsserviços fizessem o *auto-scale*, conforme a demanda de utilização aumentasse ou diminuísse, tal como pode ser observado na Figura 16.

Por fim, os autores concluíram que, ao realizar a migração da arquitetura monolítica para microsserviços, o processo de negócio da aplicação ficou mais simples e também ficou mais

Figura 16 – Arquitetura de microsserviço da aplicação de reserva de vagas

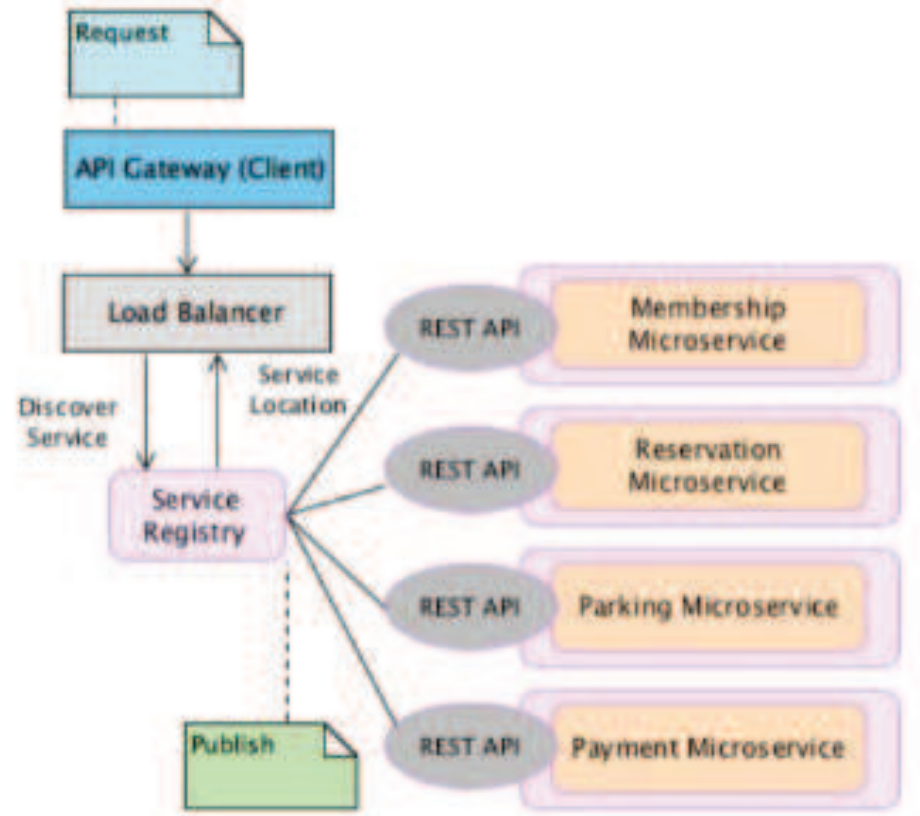
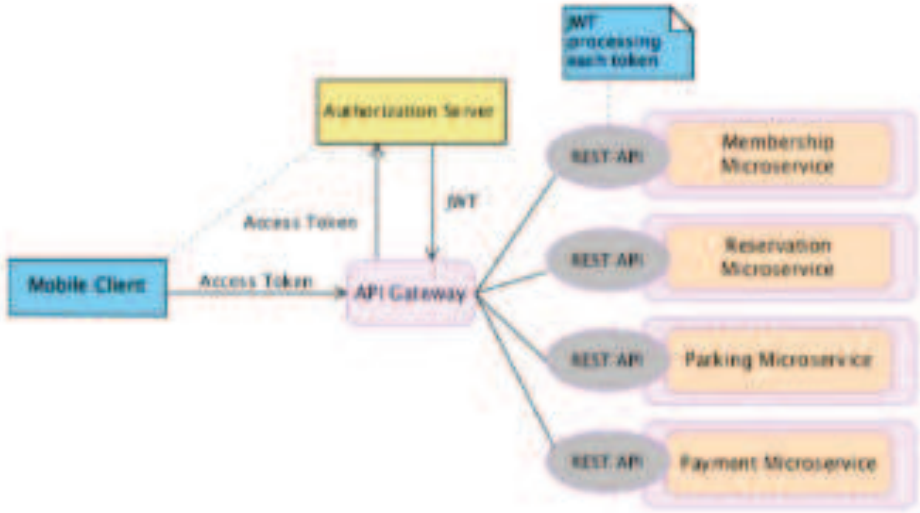


Figure 5. Service discoveries.



Fonte: Yugopuspito, Panduwinata e Sutrisno (2017)

fácil dar manutenção no sistema.

3.2.6 *Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture*

Neste trabalho, os autores Balalaie, Heydarnoori e Jamshidi (2016) relatam as suas experiências e lições aprendidas durante a migração e refatoração arquitetural do Backtory, uma aplicação monolítica baseada em *Mobile Backend as a Service* (MBaaS) para uma arquitetura de microsserviços. Os objetivos do trabalho são demonstrar as lições aprendidas ao migrar uma aplicação monolítica para microsserviços e também disponibilizar um catálogo de padrões de migração para uma arquitetura de microsserviços. Os autores desse trabalho destacam que a migração de uma aplicação monolítica para microsserviços traz uma série de benefícios, como a adaptabilidade a mudanças tecnológicas, a redução de *lock-in* de tecnologia e também, segundo eles, o mais importante, a diminuição do *time-to-market* e a melhoria na estruturação dos times de desenvolvimento ao redor da capacidade de negócio do serviço.

O principal motivo para migração do *Backtory* para uma arquitetura de microsserviço foi a necessidade de resolver um problema relacionado a um requisito de prover um chat como serviço. Além disso, os autores relatam que, com esse novo modelo de arquitetura, seria possível alcançar algumas necessidades, como reusabilidade no uso de serviços, necessidade de governança dos dados de forma descentralizada, automatização de implantação dos serviços e melhor escalabilidade dos serviços oferecidos. A partir das necessidades levantadas, os autores propuseram uma nova arquitetura do *Backtory*, utilizando o estilo arquitetural baseado em microsserviços. Segundo eles, a base para fazer a migração da aplicação monolítica para microsserviços foi a aplicação de práticas como *Domain Driven Design* e o uso do padrão de delimitação de contexto (*Bounded Context*). Nessa migração para arquitetura de microsserviços, foram utilizadas novas tecnologias, como *Spring Boot*, *Spring Cloud Server* e o Netflix OSS que é um *framework* que prove uma série de componentes necessários para implementação de uma arquitetura de microsserviços, como *service registry* (*Eureka Server*), *load balancer* (Ribbon), *circuit breaker* (*Hystrix*) e *Edge Server* (Zuul). Além de mudança a nível de tecnologia, foram também aplicadas práticas de DevOps como a entrega contínua.

Após a definição de tecnologias e práticas, os autores elencaram dez passos para realizar a migração e a refatoração da arquitetura monolítica do *Backtory* para uma arquitetura de microsserviços. São eles: preparando o pipeline de integração contínua; transformando *DeveloperData* para serviço; introduzindo a entrega contínua; introduzindo *Edge Server*; introduzindo colaboração de serviço dinâmica; introduzindo *ResourceManager*; introduzindo serviço de chat e *DeveloperInfoService*; clusterização de serviços; interconectando times de desenvolvimento e operações e mudando estrutura dos times, conforme pode ser observado na Figura 17.

Ao concluir essas dez etapas de migração, os autores relataram cinco lições aprendidas que podem colaborar com a migração para microsserviços. A primeira lição refere-se à dificuldade de implantar microsserviços num ambiente de desenvolvimento; a segunda é relacionada à manutenção dos contratos de serviços; a terceira lição trata sobre a necessidade de ter desenvolvedores sênior para trabalhar com microsserviços; a quarta lição relata sobre a necessidade

Figura 17 – Etapas de migração do backtory

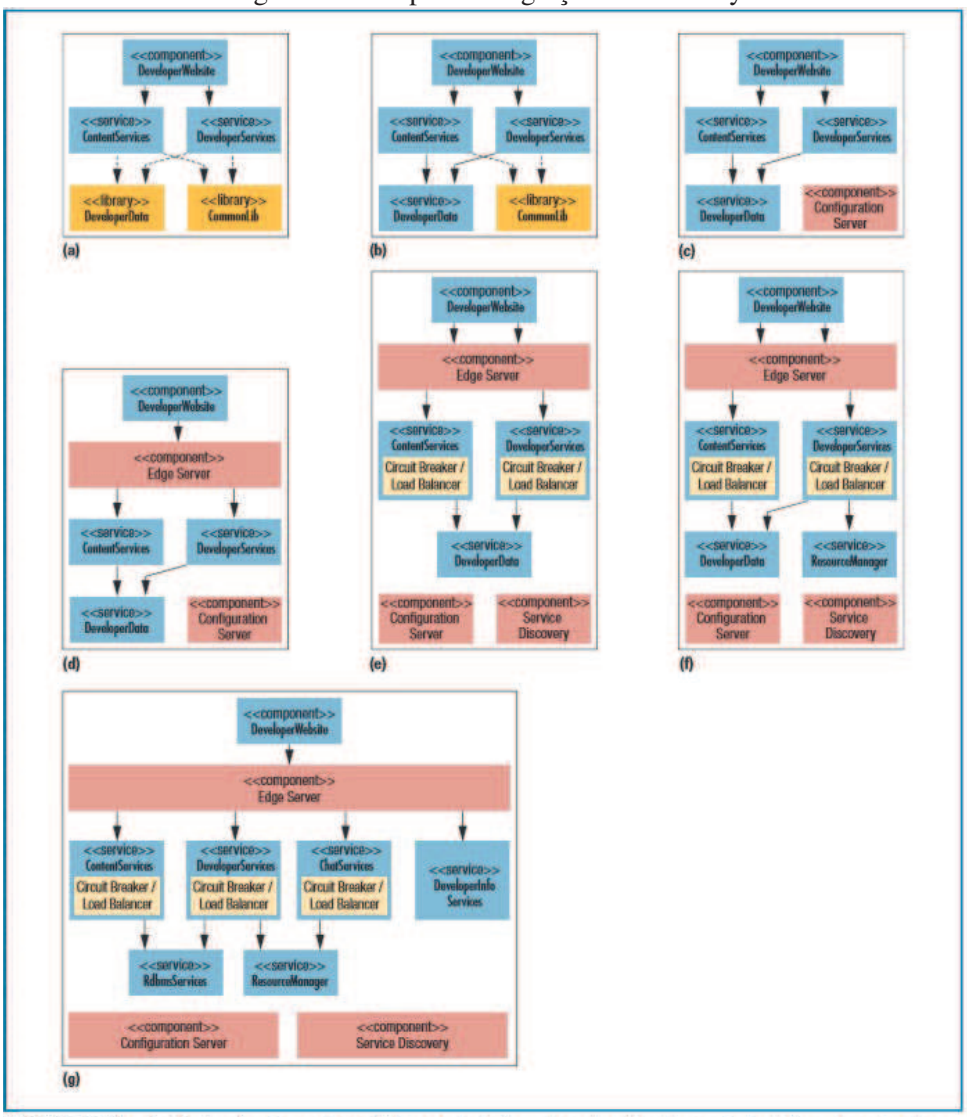


FIGURE 2. Migrating Backory to microservices. Solid arrows indicate service calls; dashed arrows indicate library dependencies. (a) Backtory's architecture before the migration. (b) Transforming **DeveloperData** to a service. (c) Introducing the Configuration Server. (d) Introducing the Edge Server. (e) Introducing dynamic service collaboration. (f) Introducing **ResourceManager**. (g) Backtory's target architecture after the migration.

Fonte: Balalaie, Heydarnoori e Jamshidi (2016)

de um *template* para padronizar o desenvolvimento de microsserviços; e, por fim, a última lição é que microsserviços não é uma "bala de prata" e que, ao adotar esse tipo de arquitetura, várias complexidades serão introduzidas. Assim, ao relatar as lições aprendidas, os autores compartilharam um catálogo de padrões de projetos que visam auxiliar as equipes de desenvolvimento no processo de migração de suas aplicações monolíticas para microsserviços, conforme Figura 18.

Figura 18 – Catálogo de padrões de migração para microsserviços

TABLE 1

Migration patterns related to this article.[8]

Pattern name	DevOps impact
Enable the Continuous Integration (CI)	CI is the first step toward continuous delivery (CD), a DevOps practice.
Recover the Current Architecture Decompose the Monolith Decompose the Monolith Based on Data Ownership Change Code Dependency to Service Call	These patterns enable decomposition of the system into smaller services, which leads to smaller teams.
Introduce Service Discovery Introduce Service Discovery Client Introduce Internal Load Balancer Introduce External Load Balancer	Dynamic discovery of services removes the need for manual wiring, thereby promoting more independent deployment pipelines.
Introduce Circuit Breaker	Failing fast can decrease the coupling between services, thereby contributing to independent service deployments.
Introduce Configuration Server	Separating configuration from code is a CD best practice.
Introduce Edge Server	The Edge Server not only allows the development team to more easily change the system's internal structure but also permits the operations team to better monitor each service's overall status.
Containerize the Services	Containers can produce the same environment in both production and development, thus reducing conflicts between the development and operations teams.
Deploy into a Cluster and Orchestrate Containers	Cluster management tools reduce the difficulties around deployment of many instances from different services in production, thus reducing the operations team's resistance to the development team's changes.
Monitor the System and Provide Feedback	Performance monitoring enables systematic collection of performance data and sharing to enhance decision making. For example, the development team can use such information to refactor the architecture if it discovers a performance anomaly in the system.

Fonte: Balalaie, Heydarnoori e Jamshidi (2016)

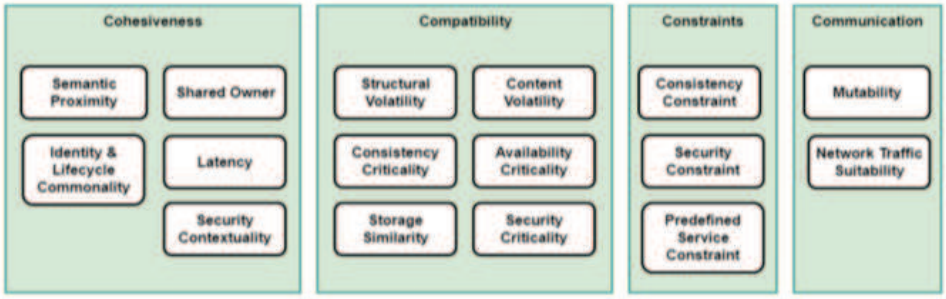
3.2.7 *Service Cutter: A Systematic Approach to Service Decomposition*

Neste trabalho, os autores Gysel et al. (2016) propõem uma ferramenta para realizar a decomposição de aplicações monolíticas em serviços. Para realizar a decomposição das aplicações, os autores definiram um catálogo com 16 critérios referentes a decisões de decomposição em serviços. Esse catálogo é utilizado como base para a tomada de decisão no momento de agrupar ou separar os serviços, conforme pode ser observado na Figura 19.

Esses critérios de decomposição foram identificados através de estudos realizados no contexto da indústria e da academia. Cada critério definido possui uma documentação que descreve o que ele representa no *Service Cutter* e também qual o tipo de artefato de especificação (*System Specification Artifact*) que precisa ser provido pelo usuário para que ele seja avaliado no momento da decomposição, conforme pode ser observado na tabela da Figura 19. A partir do entendimento de cada critério, o usuário da ferramenta *Service Cutter* precisa prover um arquivo JSON com modelo de entidade de relacionamento da aplicação ou um modelo baseado em entidades e agregados do padrão do *Domain Driven Design* ou um modelo de caso de uso.

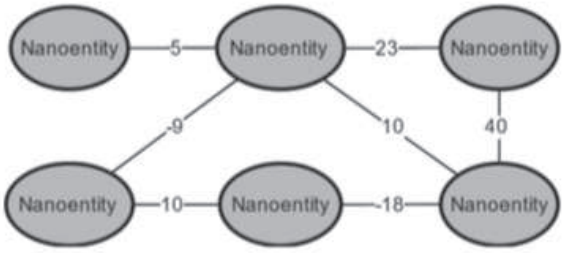
Cada critério definido na técnica trabalha com um tipo de artefato de especificação diferente, conforme exibido na segunda tabela da Figura 19. Além disso, cada critério possui um escore pré-definido referente à decomposição e um valor de prioridade que é informado pelo usuário num formato de tamanho de camisas (XS, S, M, L, XL e XXL). Com a definição das prioridades,

Figura 19 – Service Cutter: 16 critérios de decomposição



CC-1 Identity and Lifecycle Commonality	
Description	Nanoentities that belong to the same identity and therefore share a common lifecycle (create, read, update, delete)
System Specification Artifacts (SSAs)	– Entity-Relationship Models – Domain-Driven Design Entity pattern instances
Literature	Entity definition in Domain-Driven Design [5]: *Some objects are not defined primarily by their attributes* *They represent a thread of identity that runs through time and often across distinct representations*
Type	Cohesiveness

Coupling criterion	Score	Priority	Result
CC-1: Semantic Proximity	4	1	$4 * 1 = 4$
CC-7: Availability Criticality	2.5	5	$2.5 * 5 = 12.5$
CC-9: Consistency Constraint	8	3	$8 * 3 = 24$
Total weight			$4 + 12.5 + 24 = 40.5$



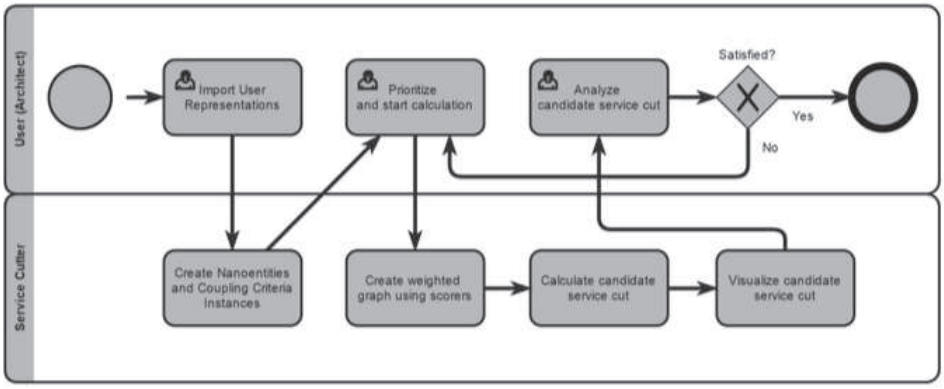
Fonte: Gysel et al. (2016)

é calculado um peso para cada critério que, depois, será utilizado no grafo gerado pelo *Service Cutter*. Esse grafo é a representação do grau de acoplamento entre as entidades da aplicação, ou seja, cada vértice representa uma nano entidade e cada aresta representa o grau de acoplamento entre essas entidades, conforme pode ser observado na Figura 19.

Após definido o grafo, é executado um algoritmo de clusterização que realiza o agrupamento e a separação das entidades da aplicação. O resultado desse algoritmo é a recomendação dos serviços. O passo a passo do processo realizado pelo *Service Cutter* pode ser observado na Figura 20.

Para avaliar a ferramenta proposta, conforme exibida na Figura 21, os autores escolheram

Figura 20 – Service Cutter: Processo de decomposição



Fonte: Gysel et al. (2016)

duas aplicações hipotéticas: uma que simula um sistema de negociação e outra que está disponível no livro *Domain Driven Design*, de Evans e Szpoton (2015). Para avaliar os resultados gerados pela ferramenta, foram convidados 20 profissionais da área de TI, como engenheiros e arquitetos com experiência em SOA. Para os participantes da pesquisa, o *Service Cutter* apresentou um resultado promissor, no que se refere à identificação de possíveis impactos arquiteturais ao decompor uma aplicação em serviços.

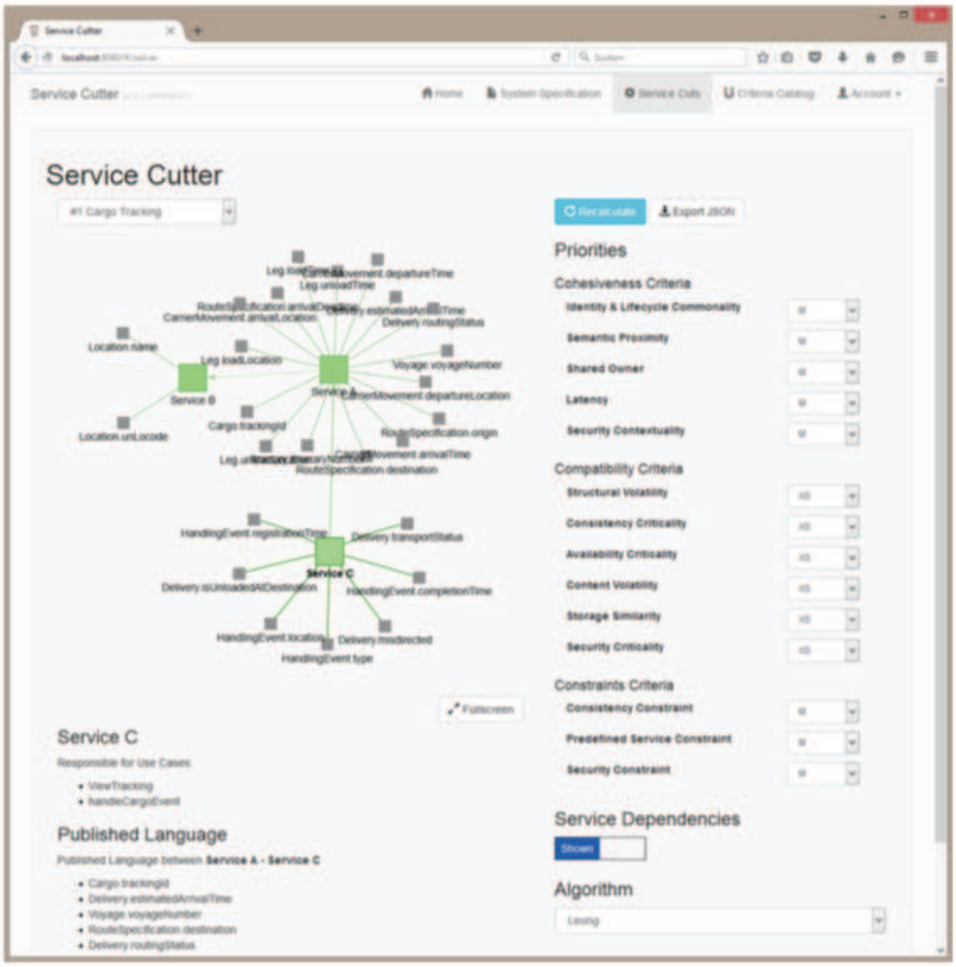
Os autores comentam que a ferramenta ainda apresenta algumas limitações como, por exemplo, o alto grau de esforço para gerar os JSON com as informações de domínio e caso de uso da aplicação. Como trabalhos futuros, os autores sugerem a integração do *Service Cutter* com ferramentas de modelagem UML para conseguir extrair os dados de modelo de dados e casos de uso de forma automática.

3.2.8 *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems*

Neste trabalho, os autores Levcovitz, Terra e Valente (2016) propõem um processo manual para a identificação de microsserviços em uma aplicação monolítica. Eles assumem que uma aplicação possui três principais partes: *client side*, *server side* e *database*. Para eles, uma aplicação possui pequenos subsistemas (SS) de forma bem estruturada e cada subsistema possui um conjunto de responsabilidade de negócios. Com base nessas definições, eles assumem que um sistema é representado por uma tríplice (**F,B,D**), na qual **F** significa as fachadas, ou seja, o ponto de entrada para a chamada de funções de negócios; **B** significa as funções de negócio, ou seja, o local onde estão implementadas as regras de negócio e o **D** representa o conjunto de tabelas do banco de dados, conforme pode ser observado na Figura 22.

Com base nessas definições, os autores sugerem seis passos para alcançar a extração de microsserviços a partir de uma aplicação monolítica: o primeiro passo refere-se à identificação entre os subsistemas, as áreas de negócio e as tabelas de banco de dados; o segundo passo tem o objetivo de criar o grafo de dependência entre as fachadas, as funções de negócios e as tabelas do

Figura 21 – Service Cutter: Interface visual

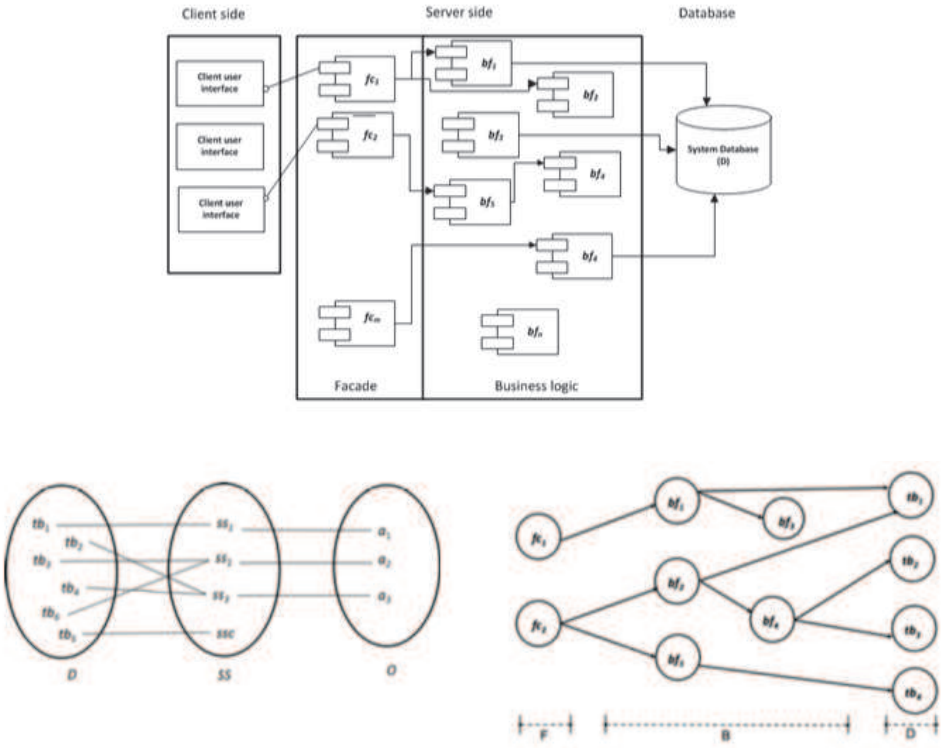


Fonte: Gysel et al. (2016)

banco de dados; o terceiro busca identificar no grafo as dependências entre fachadas e tabelas do banco de dados; o quarto passo identifica para cada subsistema o relacionamento entre funções de negócio e tabelas; o quinto passo é responsável por recomendar os microsserviços com base no grafo de dependência gerado. Nesse passo, os autores analisam manualmente o código fonte das fachadas e as funções de negócio, a fim de verificar quais tabelas estão relacionadas com determinadas regras de negócio. Após realizada essa análise do código, é gerado um relatório para cada microsserviço com as seguintes informações: nome, propósito, entrada/saída de dados, funcionalidades e dados/tabelas. O último passo executado é a criação de um *API Gateway* que tem como objetivo intermediar o acesso entre a camada cliente e servidora da aplicação. O propósito desse componente, segundo os autores, é permitir que a migração para microsserviços seja mais transparente possível para os clientes.

Para validar o processo proposto, os autores desse trabalho executaram cada passo descrito em um sistema bancário no Brasil. Conforme relatado por eles, os resultados foram satisfatórios, pois conseguiram mapear todos os subsistemas da aplicação bancária e também recomendar quatro microsserviços dos cinco subsistemas avaliados. Como trabalhos futuros, os autores

Figura 22 – Aplicação x grafo de dependência



Fonte: Levcovitz, Terra e Valente (2016)

sugerem a implementação de uma técnica que permita medir o quanto esses microsserviços realmente trouxeram de valor para a área de atuação.

3.2.9 *Towards the Understanding and Evolution of Monolithic Applications as Microservices*

Neste trabalho, os autores Escobar et al. (2016) propõem um processo semiautomático que produz um conjunto de diagramas para ajudar os desenvolvedores a entender e particionar o código de uma aplicação monolítica em microsserviços. Segundo os autores, o foco principal da proposta é agrupar e separar Enterprise Java Beans (EJB) de acordo com os dados que eles manipulam. Para realizar a separação e o agrupamento dos EJBs, os autores definiram um algoritmo que analisa as entidades na camada de dados que os EJBs utilizam. Para implementar o processo, os autores definiram três fases principais: a primeira refere-se ao que eles chamam de injeção de dados, essa etapa consiste em obter o modelo de representação da aplicação, ou seja, nela, é processado o código fonte da aplicação, através do uso da ferramenta Modisco. O resultado gerado pelo Modisco é um metamodelo baseado no padrão KDM que será utilizado como entrada da próxima etapa. Na segunda etapa, é executada uma **query** no metamodelo KDM, que identifica os principais componentes do modelo, como classes, interfaces e métodos. A partir da identificação desses componentes, são gerados dois tipos de grupos (**clusters**) *cluster* de EJB e o *cluster* de microsserviços. O grupo de EJBs contém todos EJBs e as entidades do

sistema que possuem algum tipo de relacionamento.

Com base nesse relacionamento obtido, é gerado um grafo de dependência entre os componentes identificados que será processado por um algoritmo de agrupamento(*clustering*). Esse algoritmo é o responsável pela separação e pelo agrupamento dos componentes e tem como resultado a criação de dois grupos chamados de esquerdo e direito. Para cada par de grupo esquerdo e direito gerado, é calculado um percentual de relacionamento entre os grupos, o qual é utilizado como critério para recomendar os grupos de microsserviços.

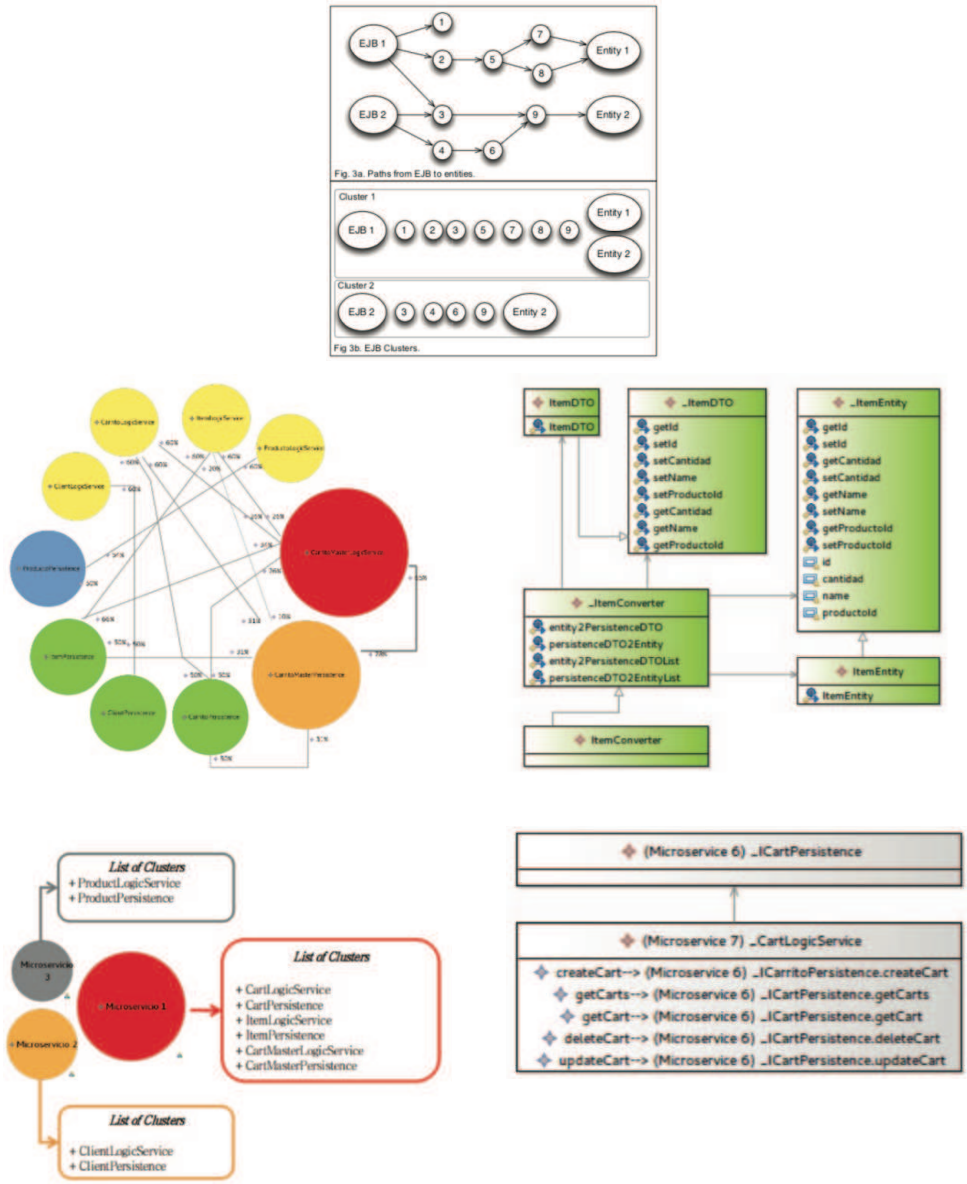
Uma vez processado o grafo e gerados os grupos de EJB e microsserviços, é executada a última fase do processo que se refere à exibição dos dados. Nessa fase, são gerados quatro diagramas: o primeiro apresenta o relacionamento entre os grupos EJBs gerados; o segundo apresenta um diagrama das classes que são compartilhadas entre os grupos EJBs; o terceiro apresenta os microsserviços gerados com os seus respectivos grupos de EJBs e o último diagrama exibe o relacionamento entre os microsserviços gerados. Os grupos gerados e os diagramas disponibilizados podem ser observados na Figura 23

Para avaliar o processo proposto, os autores aplicaram o processo em duas aplicações do tipo JavaEE. A primeira é uma aplicação de comércio eletrônico hipotética chamada de marketplace e a segunda é uma aplicação de gerenciamento de processos acadêmicos da Universidade dos Andes. Conforme relato dos autores, os resultados obtidos nas avaliações foram satisfatórios, tendo em vista que os desenvolvedores e arquitetos conseguiriam, através dos diagramas gerados, conduzir a decomposição das aplicações. Os autores relatam duas limitações do trabalho, a primeira referente à técnica funcionar somente em aplicações que utilizam a linguagem Java e a segunda ocorre devido ao processo utilizar como entrada de dados a análise estática de código. Segundo eles, o uso de análise estática do código impossibilita a verificação de relacionamentos que somente são capturados em tempo de execução da aplicação. Como trabalhos futuros, os autores sugerem, através das informações dos diagramas, gerar clientes para as API REST como recomendação para a integração entre diferentes microsserviços.

3.2.10 *Using Microservices for Legacy Software Modernization*

Neste trabalho, os autores Knoche e Hasselbring (2018) propõem um processo de modernização manual para a migração de uma aplicação monolítica de gerenciamento de clientes desenvolvida na linguagem de programação Cobol para uma arquitetura de microsserviços. Os autores enumeram cinco passos para realizar a modernização, conforme pode ser observado na Figura 24. No primeiro passo, os autores definem o que eles chamam de serviço de fachada externo. O objetivo desse passo é definir as operações dos serviços que serão expostos para as aplicações clientes consumirem Figura 24 (b). O segundo passo refere-se à implementação dos serviços de fachada. O objetivo desse passo é garantir que os serviços de fachada implementados realizem o mesmo comportamento das operações já existentes na aplicação legada. Nesse passo, os autores utilizaram técnicas de teste de software para garantir o mesmo resultado

Figura 23 – Processo de agrupamento x diagramas propostos



Fonte: Escobar et al. (2016)

implementado nas operações legadas. O terceiro passo refere-se à migração das aplicações clientes para utilizarem os serviços de fachadas implementados. O objetivo dessa fase é fazer com que as aplicações clientes deixem de acessar os recursos da aplicação legada diretamente, como banco de dados, e realizem o acesso através dos serviços de fachada. O intuito desse passo é também diminuir os pontos de entrada da aplicação legada.

O quarto passo do processo de modernização busca estabelecer os serviços de fachadas internos, conforme pode ser observado na Figura 24 (e). O objetivo desse passo é organizar a aplicação legada internamente, ou seja, fazer com que as funcionalidades acessem os serviços de fachadas internos ao invés de acessar as outras funcionalidades diretamente. O último passo refere-se à troca das implementações dos serviços de fachadas por microsserviços, ou seja, os serviços que foram implementados podem ser migrados para microsserviços, pois tanto os

Figura 24 – Processo de modernização de uma aplicação monolítica

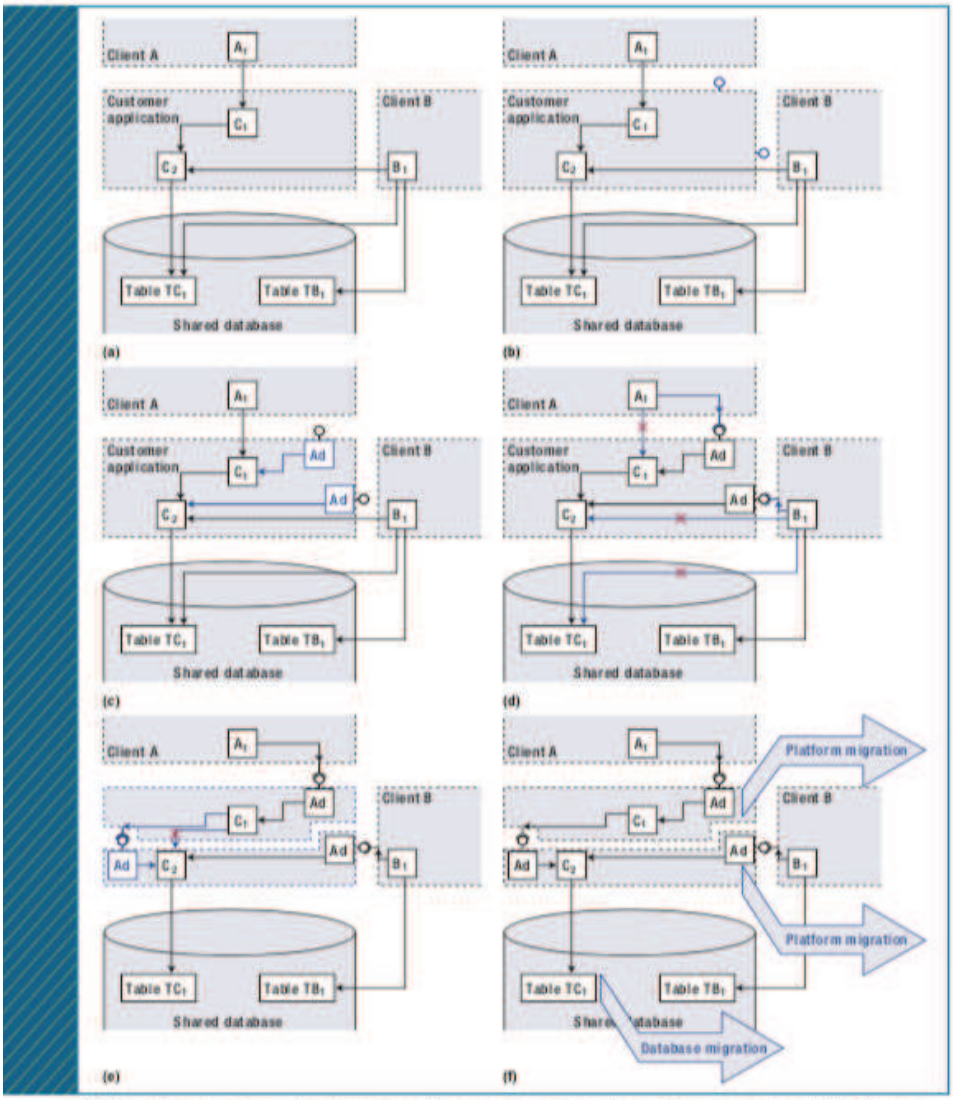


FIGURE 1. Overview of the modernization process. (a) The initial situation. (b) Defining an external service facade. (c) Adapting the service facade. (d) Migrating clients to the service facade. (e) Establishing internal service facades. (f) Replacing the service implementations by microservices. Changes in the respective process steps are highlighted in blue.

Fonte: Knoche e Hasselbring (2018)

clientes que consomem os dados da aplicação legada quanto a estrutura interna da aplicação utiliza a interface de acesso a serviços para se comunicar.

Por fim, os autores relatam que o processo de migração da aplicação de Cobol para Java durou aproximadamente quatro anos para migrar todas as aplicações clientes, mais a aplicação central que, no caso, é a aplicação de gerenciamento de clientes. Os autores relatam ainda que foram criadas um total de 23 operações de serviços e que algumas operações continuam acessando o código em Cobol, a fim de manter a compatibilidade. Segundo eles, o motivo de alguns serviços não serem microsserviços deve-se ao fato relativo a problemas relacionados ao controle de transação em um ambiente distribuído. Eles afirmam também que o controle de transação distribuída é um dos principais desafios que ainda precisa ser tratado num ambiente que utiliza uma arquitetura baseada em microsserviços, tendo em vista que algumas aplicações

necessitam que os dados estejam completamente íntegros.

3.3 Comparação dos Trabalhos Relacionados

Nesta seção, será realizada a comparação dos trabalhos selecionados no item 3.2 com base nos critérios comparativos definidos a seguir. Para uma melhor compreensão, serão descritos brevemente os grupos de critérios utilizados na análise comparativa dos trabalhos e também será disponibilizada uma tabela comparativa que irá exibir a relação entre os trabalhos selecionados e os critérios comparativos. Por fim, será discutida a relação dos trabalhos com os critérios comparativos.

Os critérios deste trabalho foram agrupados em nove grupos, são eles: **principal contribuição**; **método de avaliação**; **contexto de avaliação**; **tipo da técnica**; **grau de automatização**; **análise do código**; **granularidade dos microsserviços**; **nível de recomendação** e **ferramenta**. Cada grupo informado anteriormente possui critérios associados, os quais serão apresentados abaixo.

- **Principal Contribuição**

  Neste grupo, será avaliada qual a contribuição científica que o trabalho propôs.

  1. Algoritmo
  2. Arquitetura
  3. Ferramenta
  4. Processo

- **Método de Avaliação**

  Neste grupo, será avaliado qual o método de avaliação foi utilizado para avaliar a técnica proposta pelos trabalhos.

  1. Estudo de Caso
  2. Experimento Controlado
  3. Pesquisa

- **Contexto de Avaliação**

  Neste grupo, será avaliado se a proposta do trabalho foi avaliada num ambiente acadêmico ou na indústria ou em ambos.

  1. Academia
  2. Indústria

- **Tipo da Técnica**

  Neste grupo, será avaliado se a proposta do trabalho utilizou a teoria de grafos ou de conjuntos para implementação da técnica.

  1. Teoria de Grafos
  2. Teoria de Conjuntos

- **Grau de Automatização**

  Neste grupo, será avaliado o grau de automatização da técnica proposta pelo trabalho.

  1. Manual
  2. Semiautomática
  3. Automática

- **Análise do Código**

  Neste grupo, será avaliado o tipo de análise que foi utilizada na decomposição da aplicação. Será avaliado se foram utilizadas ferramentas para análise de código estática da aplicação ou se aplicação foi instrumentada em tempo de execução (análise de código dinâmica) ou se ela foi analisada através de diagramas.

  1. Estática
  2. Dinâmica
  3. Diagramas

- **Granularidade dos Microsserviços**

  Neste grupo, será avaliado se a técnica proposta permite que o usuário configure o grau de granularidade ou a decomposição dos microsserviços a serem gerados.

  1. Permite Configuração

- **Nível de Recomendação**

  Neste grupo, será avaliado se a técnica proposta apresenta informações das funcionalidades, das classes e dos métodos que fazem parte do microsserviço recomendado.

  1. Funcionalidade
  2. Classe
  3. Método

- **Ferramenta** Neste grupo, será avaliado se foi desenvolvido algum tipo de ferramenta como proposta nos trabalhos.

1. IDE Plugin
2. Protótipo
3. Sistema

Com base nos critérios definidos, a Tabela 2 apresenta uma visão geral sobre a relação entre os trabalhos relacionados e os critérios comparativos escolhidos nesta pesquisa.

Tabela 2 – Tabela comparativa dos trabalhos relacionados

Critérios		**Trabalhos**									
		Mazlami, Cito e Leitner (2017)	Chen, Li e Li (2017)	Bucchiarone et al. (2018)	Mustafa et al. (2018)	Yugopuspito, Panduwinata e Sutrisno (2017)	Balalaie, Heydarnoori e Jamshidi (2016)	Gysel et al. (2016)	Levcovitz, Terra e Valente (2016)	Escobar et al. (2016)	Knoche e Hasselbring (2018)
Principal Contribuição	Algoritmo	+	+	-	-	-	-	-	-	-	-
	Arquitetura	-	-	+	-	+	-	-	-	-	-
	Ferramenta	-	-	-	-	-	-	+	-	-	-
	Processo	-	-	-	+	-	+	-	+	+	+
Método de Avaliação	Estudo de Caso	+	+	+	+	+	+	+	+	+	+
	Experimento Controlado	-	-	-	-	-	-	-	-	-	-
	Pesquisa	-	-	-	-	-	-	+	-	-	-
Contexto de Avaliação	Indústria	+	+	+	-	+	+	+	+	+	+
	Academia	-	-	-	+	-	-	+	-	+	-
Tipo da Técnica	Teoria de Grafos	+	+	ø	-	ø	-	+	-	+	-
	Teoria de Conjuntos	-	-	ø	-	ø	-	-	-	-	-
Grau de Automatização	Manual	-	-	+	+	+	+	-	+	-	+
	Semiautomática	+	+	-	-	-	-	+	-	+	-
	Automática	-	-	-	-	-	-	-	-	-	-
Análise do Código	Estática	+	-	ø	ø	ø	ø	+	∼	+	-
	Dinâmica	-	-	ø	ø	ø	ø	-	-	-	-
	Diagramas	-	+	ø	ø	ø	ø	-	-	-	-
Granularidade dos Microsserviços	Permite Configuração	-	-	ø	-	ø	-	∼	-	∼	-
Nível de Recomendação	Funcionalidade	-	-	ø	-	ø	-	-	-	-	-
	Classe	+	-	ø	-	ø	-	+	-	+	-
	Método	-	-	ø	-	ø	-	-	-	+	-
Ferramenta	IDE Plugin	-	-	ø	-	ø	-	-	-	-	-
	Protótipo	+	+	ø	+	ø	-	-	-	+	-
	Sistema	-	-	ø	-	ø	-	+	-	-	-

Legenda: **(+)** Suportado (∼) Suportado Parcialmente **(-)** Não Suportado **(ø)** Não se Aplica
Fonte: Elaborado pela autor.

Com base nos dados exibidos na Tabela 2, observa-se que cinco trabalhos apresentaram como principal contribuição um processo para decomposição de aplicações monolíticas em mi-

crosserviços. Dois trabalhos sugeriram como proposta algoritmo e arquitetura. Conforme Pahl e Jamshidi (2016) citou no seu estudo de mapeamento sistemático de microsserviços, poucos trabalhos propõem uma ferramenta para decomposição de aplicações monolíticas em microsserviços. Nessa análise comparativa, pôde-se observar essa situação, conforme Pahl e Jamshidi (2016) relata, somente o trabalho de Gysel et al. (2016), propõe uma ferramenta para decomposição de aplicações monolíticas em microsserviços.

Com relação aos métodos de avaliação dos trabalhos selecionados, todos utilizaram casos de estudo como alvo da avaliação de suas propostas. Somente o trabalho de Gysel et al. (2016) executou uma pesquisa para validar a qualidade das recomendações dos microsserviços gerados pela ferramenta *Service Cutter*. Nenhum dos trabalhos utilizou uma avaliação baseada em experimentos controlados, o que identifica uma oportunidade de pesquisa na área de decomposição de aplicações. A maioria dos trabalhos foram avaliados no contexto da indústria, somente os trabalhos de Mustafa et al. (2018), Gysel et al. (2016) e Escobar et al. (2016) realizaram avaliações com projetos e participantes da academia.

Sobre o tipo de teoria aplicada na criação da técnica, observa-se que a grande maioria utilizou a teoria de grafos. Isso ocorre devido alguns trabalhos, como de Mazlami, Cito e Leitner (2017) e Chen, Li e Li (2017), utilizarem algoritmos para recomendação de microsserviços. Outros trabalhos, como de Gysel et al. (2016), utilizaram um algoritmo de grafos para recomendar os microsserviços em sua ferramenta proposta, o mesmo também ocorreu no trabalho de Escobar et al. (2016), que utilizou também um algoritmo de clusterização para fazer a separação do componentes da aplicação alvo escolhida. No entanto, nenhum dos trabalhos apresentados utilizou teoria de conjuntos para verificar a similaridade entre classes e funcionalidades ao decompor uma aplicação monolítica em microsserviços, o que apresenta uma oportunidade de pesquisa a ser avaliada, tendo em vista que algoritmos de grafos são mais complexos de implementar quando comparados a algoritmos de similaridade de conjuntos.

A respeito do grau de automatização das técnicas propostas, a maioria é realizada de forma manual. Apenas os trabalhos de Mazlami, Cito e Leitner (2017), Chen, Li e Li (2017), Gysel et al. (2016) e Escobar et al. (2016) apresentaram uma certa automação dos passos a serem executados pelas técnicas propostas. Observou-se que trabalhos que possuem um mínimo de automação utilizam em sua técnica algum tipo de algoritmo. Evidência-se também que nenhuma das técnicas propostas é 100% automática, o que indica uma grande oportunidade de pesquisa.

Para atingir o objetivo de decompor uma aplicação monolítica em microsserviço, observou-se que os trabalhos de Mazlami, Cito e Leitner (2017), Gysel et al. (2016) e Escobar et al. (2016) utilizaram ferramentas para a análise de código estática das aplicações monolíticas. O motivo pelo qual usaram essas ferramenta foi para poder encontrar os relacionamentos entre as classes e os componentes da aplicação-alvo. O trabalho de Levcovitz, Terra e Valente (2016) analisou o código da aplicação-alvo, porém sem o uso de ferramenta de análise estática, foi realizada uma análise através de uma IDE. Já o trabalho de Chen, Li e Li (2017) utilizou diagramas de

fluxo de dados (DFD) para analisar o código da aplicação. Nenhum dos trabalhos analisou o código da aplicação-alvo em tempo de execução, o que evidencia uma oportunidade de pesquisa. Segundo Escobar et al. (2016), ao não analisar o código em tempo de execução, a confiabilidade dos dados referente à dependência entre classes pode ser falha, o que pode comprometer a recomendação dos microsserviços.

Ainda relacionado à forma de como são decompostas as aplicações monolíticas, observou-se que as técnicas propostas não permitem a configuração de granularidade em que os microsserviços são gerados, tanto em nível de funcionalidade como a nível de código. Os trabalhos de Gysel et al. (2016) e Escobar et al. (2016) apresentam um parâmetro para definir qual a quantidade de cortes que o algoritmo baseado em grafo irá realizar no grafo de dependência, porém esse parâmetro não possui uma relação direta com o grau de granularidade em que os microsserviços serão gerados. Esse parâmetro é uma variável de controle do algoritmo de clusterização utilizado nos trabalhos.

Ao analisar os resultados de recomendação dos microsserviços gerados pelos trabalhos, observa-se que nenhum dos trabalhos recomenda os microsserviços baseado nas funcionalidades da aplicação. Outros trabalhos, como o de Levcovitz, Terra e Valente (2016), apresentam somente uma quantidade de microsserviços gerados, e o trabalho de Chen, Li e Li (2017) apresenta um diagrama com fluxo de dados recomendado a ser decomposto. Já os trabalhos de Mazlami, Cito e Leitner (2017), Gysel et al. (2016) e Escobar et al. (2016) apresentaram as classes a serem migradas nos microsserviços. Somente o trabalho de Escobar et al. (2016) explicita quais os métodos fazem parte de cada classe a ser migrada por cada microsserviço recomendado.

Por fim, destaca-se que apenas o trabalho de Gysel et al. (2016) apresentou como ferramenta um sistema para a decomposição de aplicações monolíticas em microsserviços. Alguns trabalhos, como o de Mustafa et al. (2018), propuseram protótipos para serem implementados e nenhum dos trabalhos propôs um plugin para IDE, o que gera uma oportunidade de pesquisa.

3.4 Oportunidades de Pesquisa

Após a análise comparativa dos estudos apresentados neste capítulo, foram identificadas oportunidades de pesquisa que podem contribuir para a temática de decomposição de aplicações monolíticas em microsserviços. Essas oportunidades de pesquisa são exibidas como possíveis contribuições científicas desta pesquisa e levam em consideração as características e as fragilidades identificadas nos trabalhos relacionados.

- **Experimentos Controlados**

  As análises realizadas demonstraram que não foram encontrados experimentos controlados sobre a temática de decomposição de aplicações monolíticas em microsserviços. Assim, torna-se importante também realizar experimentos controlados a fim de identifi-

car quais as variáveis que podem impactar na decomposição de aplicações monolíticas em microsserviços.

- **Algoritmo Utilizando Teoria de Conjuntos**

  Com base nas análises realizadas, não foram encontrados estudos que utilizem algoritmos que façam uso da teoria de conjuntos. Tendo em vista que, para decompor uma aplicação monolítica em microsserviço, é recomendado utilizar princípios como o de responsabilidade única, conforme (MARTIN; MARTIN, 2006), a teoria de conjuntos seria um bom método para encontrar funcionalidades que são similares e que, no caso, deveriam ficar juntas no momento de uma decomposição para microsserviços.

- **Técnica Automática**

  Pôde-se verificar, nas análises realizadas, que nenhum trabalho propôs uma técnica 100% automática. Isso indica uma grande oportunidade de pesquisa, tendo em vista que o processo de decomposição de uma aplicação monolítica em microsserviço exige um grande esforço para ser realizado (MUSTAFA et al., 2018).

- **Análise de Código Dinâmica**

  Com base nas análises realizadas, não foram encontrados trabalhos que analisam o código da aplicação em tempo de execução. Isso evidência uma grande oportunidade de pesquisa, pois alguns tipos de relacionamentos que os componentes da aplicação apresentam somente podem ser obtidos em tempo de execução. Com esse tipo de análise, é possível verificar o real comportamento dos componentes da arquitetura da aplicação, o que possibilita uma decomposição mais assertiva.

- **Permitir Configuração da Granularidade dos Microsserviços**

  Com base nas análises realizadas, nenhum trabalho permite mudar a recomendação dos microsserviços gerados sem ter que manipular os dados informados da aplicação. A possibilidade de mudar o grau de granularidade dos microsserviços gerados pode permitir que o usuário execute diversos cenários de recomendação. Com esse recurso de configuração de granularidade, o usuário poderia verificar qual a recomendação se adequará melhor a sua necessidade de negócio e também permitirá o usuário avaliar o quanto a decomposição gerada pode impactar em questões de desempenho da aplicação, tendo em vista que, se for gerado um serviço para cada funcionalidade do sistema, o uso da rede, por exemplo, poderia ser tornar um gargalo de desempenho na aplicação, pois microsserviços é um tipo de arquitetura distribuída.

- **Nível de Recomendação por Funcionalidade**

  Com base nas análises realizadas, nenhum dos trabalhos gera recomendação de microsserviços apresentando as funcionalidades envolvidas. Isso torna-se uma grande oportunidade

de pesquisa, pois, quando se necessita decompor uma aplicação, normalmente, o usuário tem interesse em decompor uma aplicação ao redor de uma capacidade de negócio que, no caso, está representado por uma funcionalidade da aplicação (EVANS; SZPOTON, 2015).

- **IDE Plugin**

  Com base nas análises realizadas, nenhum dos trabalhos gerou uma ferramenta como um plugin para IDE. Tendo em vista que profissionais de desenvolvimento de software usam normalmente uma IDE, essa seria uma ótima oportunidade para centralizar em uma única ferramenta o desenvolvimento e o acompanhamento de uma decomposição de uma aplicação monolítica.

  A partir das oportunidades de pesquisa levantadas, os itens **Algoritmo Utilizando Teoria de Conjuntos**; **Análise de Código Dinâmica**; **Permitir Configuração da Granularidade dos Microsserviços**; e **Nível de Recomendação por Funcionalidade** serão alvos de pesquisa deste trabalho. O Capítulo 4 desta pesquisa irá detalhar como essas oportunidades de pesquisas serão implementadas.

4 A TÉCNICA DE MONÓLISE

Neste capítulo será apresentada a técnica de Monólise, conforme denominada pelo autor desta dissertação. O desenvolvimento dessa técnica visa atender ao principal objetivo da pesquisa, tal como descrito na seção 1.3 do trabalho, ou seja, propor uma técnica para decomposição de aplicações monolíticas em microsserviços.

Para tanto, este capítulo está organizado em quatro seções: a seção 4.1 descreve a visão geral da técnica, a seção 4.2 expõe suas características-chave, já a seção 4.3 detalha o algoritmo utilizado na decomposição e, por fim, a seção 4.4 explica o aspecto de implementação da técnica.

4.1 Visão Geral da Técnica

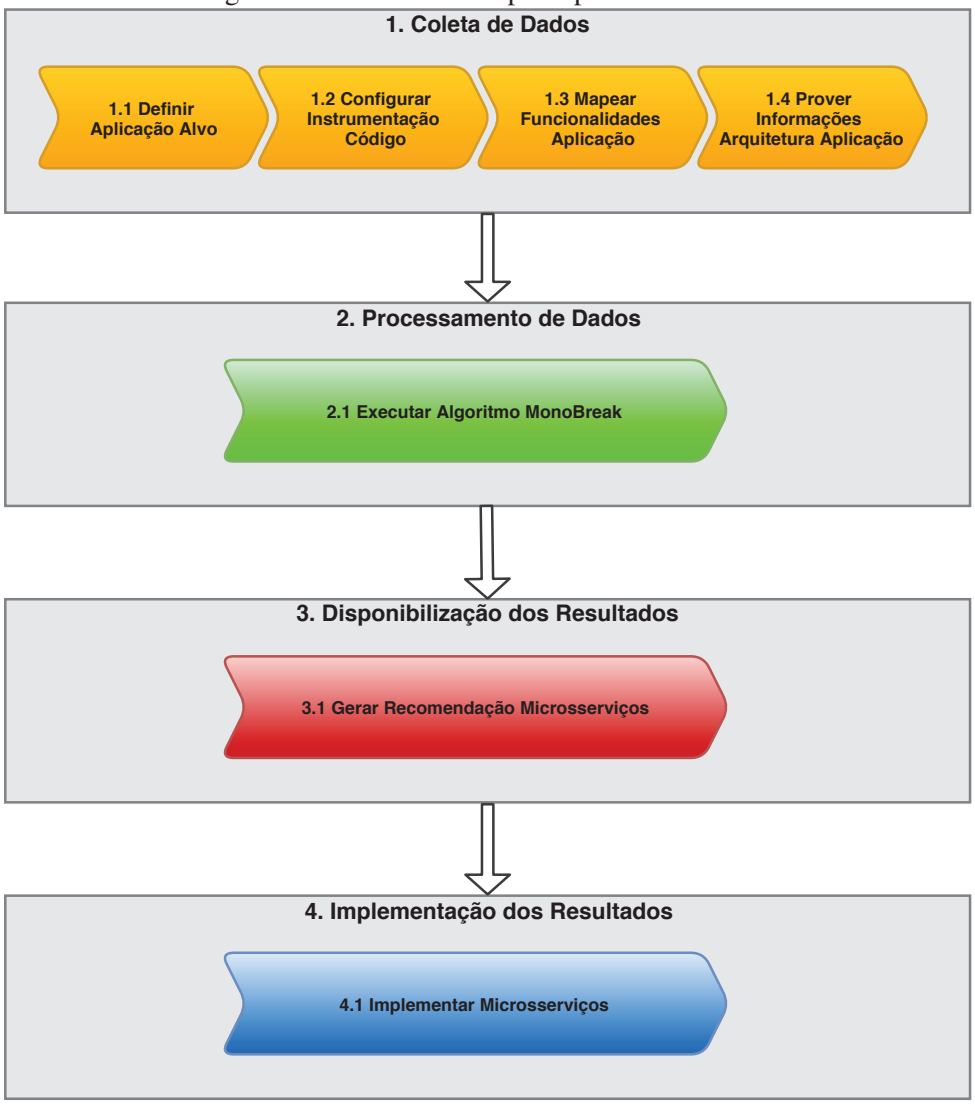
Essa técnica foi denominada de Monólise, considerando, segundo o dicionário Priberam da Língua Portuguesa, o significado dos radicais gregos *mónos*, que significa único, e *lúsis*, que quer dizer a ação de separar. Assim, Monólise consiste no processo de separação das funcionalidades de uma aplicação monolítica em microsserviços e tem como objetivo ajudar as equipes de desenvolvimento de software na jornada de modernização de suas aplicações monolíticas em microsserviços.

Para apoiar os times de desenvolvimento na decomposição de suas aplicações, essa técnica conta com um algoritmo chamado MonoBreak, o qual permite aos times de desenvolvimento simular diversos cenários de decomposições de suas aplicações monolíticas em microsserviços. Essas simulações possibilitam aos times encontrar o melhor grau de granularidade que eles entendem ser o ideal para a decomposição das funcionalidades da aplicação monolítica em microsserviços. Com a Monólise, o time poderá verificar se um microsserviço será composto por uma única funcionalidade (granularidade fina) ou se será composto por mais de uma funcionalidade (granularidade grossa). Cabe destacar ainda que as funcionalidades que fazem parte de um microsserviço não possuem dependência com outras funcionalidades, o que permite que um microsserviço seja autônomo, independente e auto-contido.

Como já discutido nesta pesquisa, o uso de culturas agéis como DevOps, práticas agéis (entrega contínua) e uso de arquitetura nativas para nuvem como microsserviços permite que as empresas entreguem softwares com mais valor de negócio, melhor experiência do usuário e redução do *time-to-market* para seus clientes. Realizar a decomposição das funcionalidades de uma aplicação monolítica ao redor de uma capacidade de negócio que permita a entrega contínua e que garanta a independência e autonomia das funcionalidades em microsserviços é um grande desafio (NEWMAN, 2015). A fim de resolver esse desafio, esta pesquisa desenvolveu uma técnica que é composta por quatro etapas: **coleta de dados, processamento de dados, disponibilização dos resultados e implementação dos resultados**, conforme pode ser observado na Figura 25. Para uma melhor compreensão sobre a técnica, será descrita brevemente cada

etapa da técnica com seus respectivos passos.

Figura 25 – Monólise: Etapas e passos da técnica



Fonte: Elaborado pelo autor

4.1.1 Etapa 1 - Coleta de Dados

A coleta de dados é a primeira etapa da Monólise, seu principal objetivo é coletar os dados que servirão como parâmetros de entrada para a execução do algoritmo de decomposição de aplicações monolíticas em microsserviços, o MonoBreak. Essa etapa é composta por quatro passos: **definir aplicação-alvo; configurar instrumentação de código; mapear as funcionalidades da aplicação; e prover as informações da arquitetura da aplicação**, conforme pode ser observado na Figura 25. Cada passo dessa etapa está descrito abaixo.

- **Passo 1.1 - Definir Aplicação-Alvo**

Neste passo, o usuário da técnica deverá escolher a aplicação monolítica alvo que pretende ser decomposta em microsserviços. Para a escolha dessa aplicação existem alguns requisitos que deverão ser cumpridos, como: a aplicação escolhida tem que ser implementada com uma linguagem de programação orientada a objetos e a sua arquitetura de software deverá utilizar o padrão de projeto MVC.

- **Passo 1.2 - Configurar Instrumentação de Código**

  Neste passo, o usuário da técnica deverá configurar uma ferramenta de instrumentação de código de sua escolha. O objetivo disso é garantir que o usuário consiga, através dessa ferramenta, capturar o fluxo de execução das chamadas dos métodos das classes da aplicação, quando uma funcionalidade estiver sendo executada por ele. A partir dos dados gerados por essa ferramenta, será possível o MonoBreak identificar quais as classes e os métodos utilizados por uma determinada funcionalidade do sistema.

- **Passo 1.3 - Mapear as Funcionalidades da Aplicação**

  Neste passo, o usuário da técnica irá mapear as funcionalidades da aplicação e escolherá quais delas serão decompostas. A escolha dessas funcionalidades pode estar relacionada a diversos tipos de necessidades, como: permitir a entrega contínua de uma funcionalidade e consequentemente a redução do *time-to-market*; melhorar o desempenho e a escalabilidade dela; facilitar a troca de tecnologia, dentre outras situações. Essa técnica não especifica um motivo para decomposição, fica a critério do usuário definir a razão pela qual ele necessita decompor uma determinada funcionalidade em microsserviço. Sendo assim, esse passo tem como principais objetivos identificar as funcionalidades a serem decompostas e obter o rastro de execução das classes e dos métodos que a funcionalidade utiliza ao ser executada, sendo esse rastro capturado e armazenado através da ferramenta de instrumentação de código. Resumidamente, nessa etapa, o usuário deverá, primeiramente, escolher as funcionalidades, depois, utilizar cada uma delas como se estivesse usando-a em um ambiente produtivo e, após concluir o uso de cada funcionalidade, o usuário deverá salvar o resultado da instrumentação de código em um arquivo com o seguinte padrão “nome_Funcionalidade.trace”. Cada funcionalidade escolhida e executada terá um arquivo com a extensão .trace, o qual conterá todo o rastro de execução de classes e métodos da respectiva funcionalidade. Esses arquivos são um dos parâmetros de entrada do algoritmo MonoBreak.

- **Passo 1.4 - Prover Informações da Arquitetura da Aplicação**

  Neste passo, o usuário da técnica deverá prover um arquivo com as informações referentes à arquitetura da aplicação escolhida e também informar um percentual de limite de decomposição que o algoritmo irá aplicar no momento da decomposição das funcionalidades em microsserviços. O objetivo dessas informações é permitir que o algoritmo possa “entender” como que a aplicação foi projetada, quais são as camadas de softwares

da aplicação e qual o peso de relevância dessas camadas quando vinculadas ao valor de negócio da aplicação. Também nesse arquivo o usuário deverá informar qual o limite de percentual de decomposição (granularidade) em que os microsserviços deverão ser gerados para que o MonoBreak realize a recomendação de microsserviços. Todos os dados informados nesse passo são baseados no questionário da Figura 26. Esse arquivo gerado pelo usuário é um parâmetro de entrada do algoritmo MonoBreak.

Figura 26 – Monólise: Questionário sobre a configuração da arquitetura da aplicação-alvo

Questão
1) Qual o nome da aplicação?
2) Qual o nome do **pacote** que contém as classes de **Model** da aplicação?
3) Qual o nome do **pacote** que contém as classes de **DAOs** ou **Repositories** da aplicação?
4) Qual o nome do **pacote** que contém as classes de **Services** da aplicação?
5) Qual o nome do **pacote** que contém as classes de **DTOs** da aplicação?
6) Qual o nome do **pacote** que contém as classes de **Controllers** da aplicação?
7) Qual o valor do **peso** do pacote de **Model** da aplicação?
8) Qual o valor do **peso** do pacote de **DAO** ou **Repositories** da aplicação?
9) Qual o valor do **peso** do pacote de **Services** da aplicação?
10) Qual o valor do **peso** do pacote de **DTOs** da aplicação?
11) Qual o valor do **peso** do pacote de **Controllers** da aplicação?
12) Qual o **valor** do **limite (threshold)** de **decomposição** da aplicação?

Fonte: Elaborado pelo autor

4.1.2 Etapa 2 - Processamento de Dados

O processamento de dados é a segunda etapa da Monólise, seu objetivo é executar o algoritmo MonoBreak utilizando os parâmetros de entrada fornecidos no processo de coleta de dados. Essa etapa é composta pelo passo **executar o algoritmo MonoBreak**, conforme pode ser observado na Figura 25. Esse passo está brevemente descrito abaixo.

- **Passo 2.1 - Executar o Algoritmo MonoBreak**

  Nesse passo, o usuário da técnica executa o algoritmo MonoBreak com base nos parâmetros de entrada informados na etapa anterior. O objetivo do MonoBreak é realizar a decomposição das funcionalidade da aplicação monolítica em microsserviços. Ele necessita executar em seu fluxo principal seis sub-rotinas, sendo cinco delas de processamento de dados e uma sub-rotina de disponibilização de resultados. A primeira e a segunda sub-rotina referem-se à transformação dos dados dos parâmetros de entradas do algoritmo em estrutura de dados internos do algoritmo. Resumidamente, nessas duas primeiras etapas, o algoritmo converte os arquivos de rastro de execução de funcionalidade e o arquivo

com os dados da arquitetura da aplicação para uma estrutura de dados interna que o algoritmo manipula; a terceira sub-rotina classifica as classes dos rastros de execução das funcionalidades com base nas camadas definidas no arquivo de arquitetura da aplicação. Nessa sub-rotina, o algoritmo compara os nomes dos pacotes definidos no arquivo de arquitetura da aplicação com o nome do pacote definido antes do nome da classe no arquivo de rastro de execução. À medida que a comparação for verdadeira, o algoritmo categoriza a classe como parte da camada de modelo (*model*), repositório ou *Data Access Object* (DAO), serviço (*service*), *Data Transfer Object* (DTO) ou controladora (*controller*), atribuindo também o peso da camada para classe em questão; a quarta sub-rotina do algoritmo é gerar uma tabela de similaridade (matriz de similaridade) entre todas as funcionalidades. Nessa sub-rotina, o algoritmo cria uma matriz de similaridade entre as funcionalidades e aplica para cada combinação um cálculo que leva em consideração os pesos das camadas e o quanto as classes entre duas funcionalidades são iguais. Essa matriz de similaridade é a entrada de dados para a quinta sub-rotina, que tem como objetivo verificar quais as funcionalidades apresentam o percentual de similaridade maior ou igual ao limite de percentual de decomposição informado no arquivo de configuração da arquitetura da aplicação. Em síntese, nessa última sub-rotina de processamento de dados, o MonoBreak percorre a tabela de similaridade em busca de funcionalidades que possuem um percentual de similaridade igual ou maior ao *threshold* de decomposição. Ao encontrar essas funcionalidades, o MonoBreak verifica se elas já existem em outros microsserviços e, caso a intersecção exista, ele as agrupa em um único microsserviço; caso contrário, ele cria um microsserviço para uma funcionalidade. O MonoBreak será detalhado na seção 4.3.

4.1.3 Etapa 3 - Disponibilização dos Resultados

A disponibilização dos resultados é a terceira etapa da Monólise, seu objetivo é exibir o resultado da recomendação de decomposição em microsserviços baseada nas funcionalidades escolhidas pelo usuário. Essa etapa é composta pelo passo **gerar recomendação de microsserviços**, conforme pode ser observado na Figura 25. Na sequência está descrito brevemente o passo de gerar recomendação de microsserviços.

- **Passo 3.1 - Gerar Recomendação de Microsserviços**

  Conforme mencionado na etapa anterior, o MonoBreak é composto por seis sub-rotinas sendo cinco de processamento de dados e uma de disponibilização de resultados. Nessa etapa, será executada a última sub-rotina do MonoBreak, a qual é responsável por indicar os microsserviços que foram recomendados. Nessa sub-rotina, o MonoBreak apresenta quais funcionalidades fazem parte de cada microsserviço e indica para cada funcionalidade quais as classes e os métodos que o usuário necessitária migrar para a nova aplicação baseado na arquitetura de microsserviços. Basicamente, o resultado disponibilizado é

uma lista com os microsserviços recomendados, os nomes das funcionalidades que fazem parte daquele microsserviço, as classes e os métodos que fazem parte de cada funcionalidade. Será através desse resultado obtido que o usuário poderá iniciar a migração das suas funcionalidades escolhidas para uma nova aplicação baseada em uma arquitetura de microsserviço.

4.1.4 Etapa 4 - Implementação dos Resultado

A implementação de resultados é a última etapa da técnica da Monólise, seu objetivo é permitir que o usuário migre as funcionalidades de seus respectivos microsserviços recomendados para uma aplicação baseada nas melhores práticas de implementação de uma arquitetura de microsserviços. Essa etapa é composta pelo passo **implementar microsserviços**, conforme pode ser observado na Figura 25. O passo de implementação de microsserviço será descrito brevemente a seguir.

- **Passo 4.1 - Implementação de Microsserviços**

  A partir do resultado gerado pela recomendação dos microsserviços, esse passo tem como objetivo permitir que o usuário analise os resultados alcançados e implemente de fato os microsserviços, utilizando as classes e os métodos de cada funcionalidades que o MonoBreak recomendou. O propósito desse passo é demostrar na prática como que seriam implementados os microsserviços recomendados e quais as boas práticas a nível de padrões de projeto e tecnologias em que o usuário dever prestar atenção no momento da implementação de uma arquitetura baseada em microsserviços. O detalhamento dessa implementação será apresentado no Capítulo 5 desta pesquisa. Na sequência, serão apresentadas as principais características dessa técnica quando comparada a demais técnicas propostas em outros trabalhos.

4.2 Principais Características da Técnica

Nesta seção, serão apresentadas as principais características que diferenciam essa técnica de outros trabalhos que sugerem a decomposição de aplicações monolíticas em microsserviços. Pode-se destacar como diferencial quatro características dessa técnica: **técnica agnóstica à linguagem de programação; possibilidade de modificar o comportamento do algoritmo; algoritmo extensível; e algoritmo sensível à arquitetura da aplicação**. As quatro características apresentadas serão discutidas abaixo.

- **Técnica Agnóstica à Linguagem de Programação**

  Conforme apresentado na seção 4.1, a Monólise exige que a aplicação monolítica escolhida seja desenvolvida em uma linguagem de programação orientada a objetos. Além disso, a arquitetura utilizada na aplicação deve ser implementada seguindo o padrão de

projeto MVC. O diferencial da Monólise comparada a outras técnicas é que ela é agnóstica à linguagem de programação orientada a objetos utilizada pela aplicação-alvo da decomposição, ou seja, se o usuário desejar aplicar a técnica em uma aplicação que utilize as linguagens como Java, .Net, Python, PHP ou JavaScript, isso é possível, pois a Monólise, juntamente com o algoritmo MonoBreak, não utiliza nenhum recurso específico de uma determinada linguagem de programação, o que torna a técnica reusável.

- **Possibilidade de Modificar o Comportamento do Algoritmo MonoBreak**

  Ao utilizar o MonoBreak para recomendação de microsserviços, o usuário tem a flexibilidade de mudar dinamicamente o comportamento da lógica de recomendação do algoritmo. Isso é possível graças ao parâmetro de entrada referente às configurações da arquitetura da aplicação e também ao recurso de limite (*threshold*) de decomposição das funcionalidades em microsserviços. Através dos parâmetros de pesos por camada e limite de decomposição, conforme descritos na Figura 26, o usuário pode modificar as recomendações e optar, por exemplo, se ele deseja que a decomposição da funcionalidade em microsserviços possua uma granularidade fina, ou seja, um microsserviço por funcionalidade, ou se quer uma granularidade grossa, um agrupamento de funcionalidades por microsserviços. Essa alteração no comportamento do algoritmo permite que o usuário realize diferentes tipos de recomendações, o que possibilita encontrar o modelo ideal de decomposição de suas funcionalidades para microsserviços.

- **Algoritmo Extensível**

  Outro aspecto importante do algoritmo MonoBreak é que ele é extensível. Isso significa que, se o usuário desejar incluir no critério de comparação entre as funcionalidade um requisito como, por exemplo, o tempo de execução das funcionalidades (*performance*) como um critério de decomposição, isso é totalmente possível, pois a lógica que é executada para realizar o cálculo de similaridade entre as funcionalidades é totalmente desacoplada do processamento do MonoBreak. Sendo assim, o usuário poderia adicionar diversos tipos de métricas para que o algoritmo as usasse como critério de decisão ao separar ou agrupar as funcionalidades em microsserviços. Por exemplo, o usuário poderia modificar a métrica de decomposição para agrupar ou separar as funcionalidades com base no tempo de resposta que uma funcionalidade necessita para ser completamente concluída.

- **Algoritmo Sensível à Arquitetura da Aplicação**

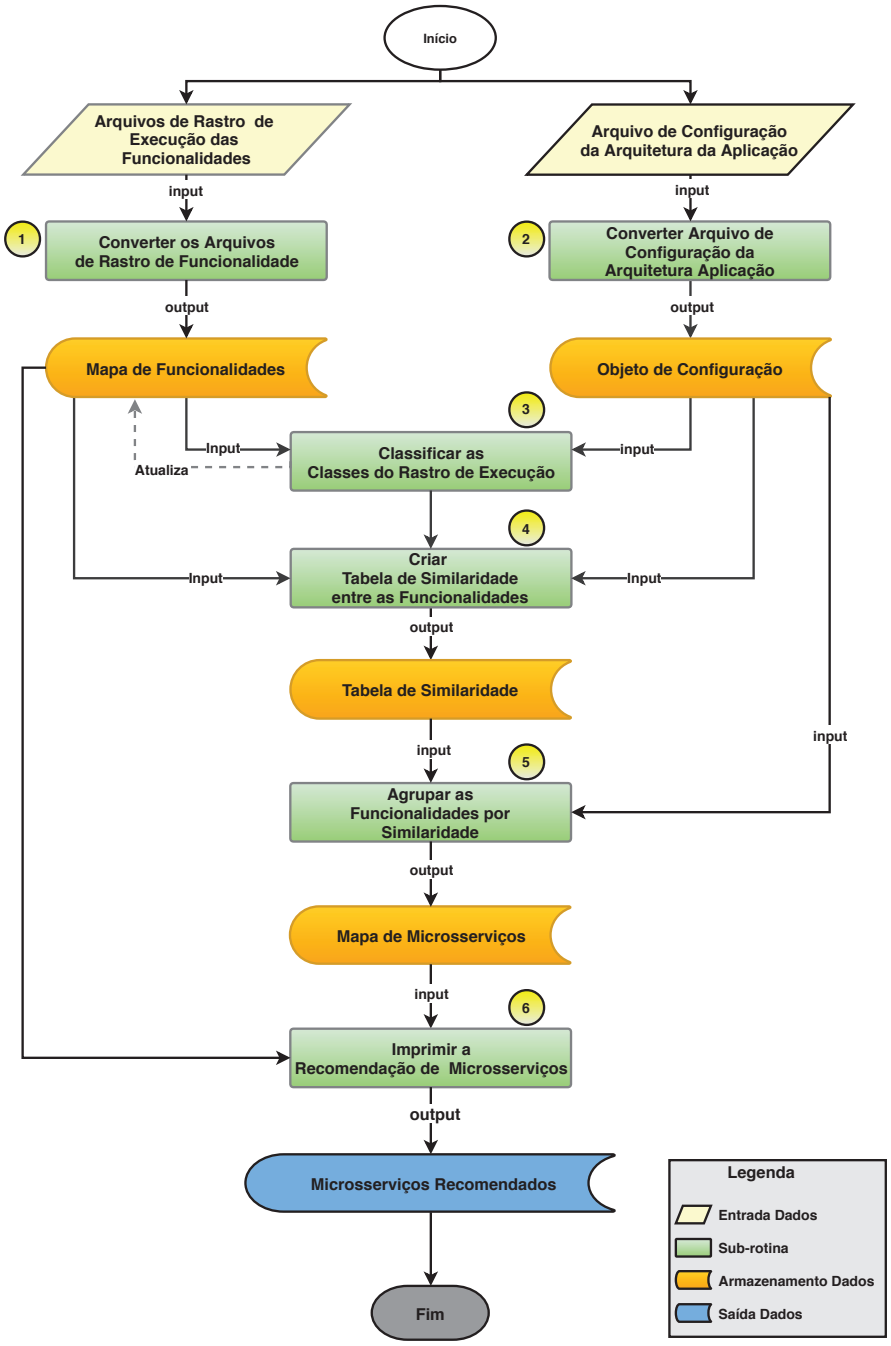
  Essa é a característica principal dessa técnica, o MonoBreak é sensível à arquitetura da aplicação escolhida. Conforme discutido na seção anterior, o MonoBreak recebe parâmetros de entrada que possibilitam ao usuário informar qual o peso de relevância que, por exemplo, uma camada da arquitetura da aplicação tem no momento do cálculo de similaridade entre as funcionalidades. Um exemplo de como o algoritmo é sensível à arquitetura

da aplicação é quando uma aplicação possui camadas e componentes de software que são utilitários ou auxiliares ao valor de negócio do sistema, também conhecidos como componentes ou interesses transversais. Uma aplicação normalmente apresenta componentes de software (classes) com preocupações transversais ao interesse de negócio da aplicação (GULIA; DEV; PATEL, 2015). Esses componentes normalmente estão representados por requisitos como, por exemplo, segurança, log e controle de transação. Ao capturar as classes desses componentes transversais na execução das funcionalidades, percebeu-se que essas classes transversais estavam influenciando os cálculos de similaridade entre as funcionalidades de forma negativa, pois isso estava aumentando o grau de similaridade entre as funcionalidades devido ao fato de essas classes transversais estarem espalhadas por todas as funcionalidades da aplicação. Para garantir um valor de similaridade mais correto entre as funcionalidades, o algoritmo necessitaria ter um recurso que pudesse aumentar o grau de similaridade para caso de classes que estão diretamente ligadas ao valor de negócio da funcionalidade, ou seja , classes de modelo de dados e classes de serviços com regras de negócio. Foi então a partir do uso de pesos de relevância por camadas que se conseguiu diferenciar quais são as classes que realmente importam de ser comparadas no momento do cálculo e quais não têm muita importância, no caso, as classes transversais. Outro ponto importante desse algoritmo é que ele verifica e recomenda todas as classes e todos os métodos de todas as camadas da aplicação, ou seja, ela não analisa somente a camada de modelo de dados da aplicação, mas sim todas as camadas, o que permite que a decomposição seja feita de ponta-a-ponta da aplicação.

4.3 Algoritmo MonoBreak

Nesta seção, será apresentado o algoritmo MonoBreak e as suas principais sub-rotinas. O objetivo dessa seção é explicar o funcionamento de cada sub-rotina do algoritmo e como que essas sub-rotinas se relacionam para entregar o resultado da recomendação em microsserviços. Para explicar o MonoBreak, essa seção foi organizadas em sete seções, são elas: **fluxo principal**; **converter arquivos de rastro das funcionalidades**; **converter arquivo de configuração da arquitetura aplicação**; **classificar as classes do rastro de execução das funcionalidades**; **criar tabela de similaridade entre as funcionalidades**; **agrupando as funcionalidades por similaridade** e **imprimir recomendação dos microsserviços**. As seis sub-rotinas descritas no fluxograma da Figura 27 fazem parte do fluxo principal do MonoBreak e cada uma delas será apresentada abaixo com seu respectivo pseudocódigo e diagrama de funcionamento. O fluxo principal será o primeiro discutido, tendo em vista que ele é o agrupador de todas as seis sub-rotinas.

Figura 27 – MonoBreak: Fluxograma do Algoritmo



Fonte: Elaborado pelo autor

4.3.1 Fluxo Principal

Para realizar a recomendação dos microsserviços, o MonoBreak necessita de dois parâmetros de entrada. O primeiro deles é um conjunto de arquivos de rastro de execução das funcionalidades a serem decompostas e o segundo parâmetro é referente ao arquivo de configurações da arquitetura da aplicação escolhida.

A partir dos parâmetros informados, o MonoBreak executa seis sub-rotinas, cinco delas

referentes ao processamento de dados: **convertFunctionalitiesFiles**; **convertApplicationConfigFile**; **setApplicationLayer**; **createSimilarityTable**; **groupFunctionalitiesBySimilarity**; e uma referente à disponibilização do resultado **printMicroservices**, conforme pode ser observado no Algoritmo 1.

Cada sub-rotina descrita abaixo está representada no fluxograma da Figura 27, a diferença é que no fluxograma não é utilizado o nome da sub-rotina a nível de código e sim o que a rotina realiza de fato no MonoBreak.

Algoritmo 1: MonoBreak - Fluxo Principal

```
1 function monobreak (functionalityFiles, appInfoFile)
2     functionalities ← convertFunctionalitiesFiles(functionalityFiles);
3     config ← convertApplicationConfigFile(appInfoFile);
4     setApplicationLayer(functionalities,config);
5     similarityTable ← createSimilarityTable (functionalities,config);
6     microservice ← groupFunctionalitiesBySimilarity
        (similarityTable,config);
7     printMicroservices(microservices,functionalities);
8 end
```

4.3.2 Sub-rotina 1 - Converter Arquivos de Rastro das Funcionalidades

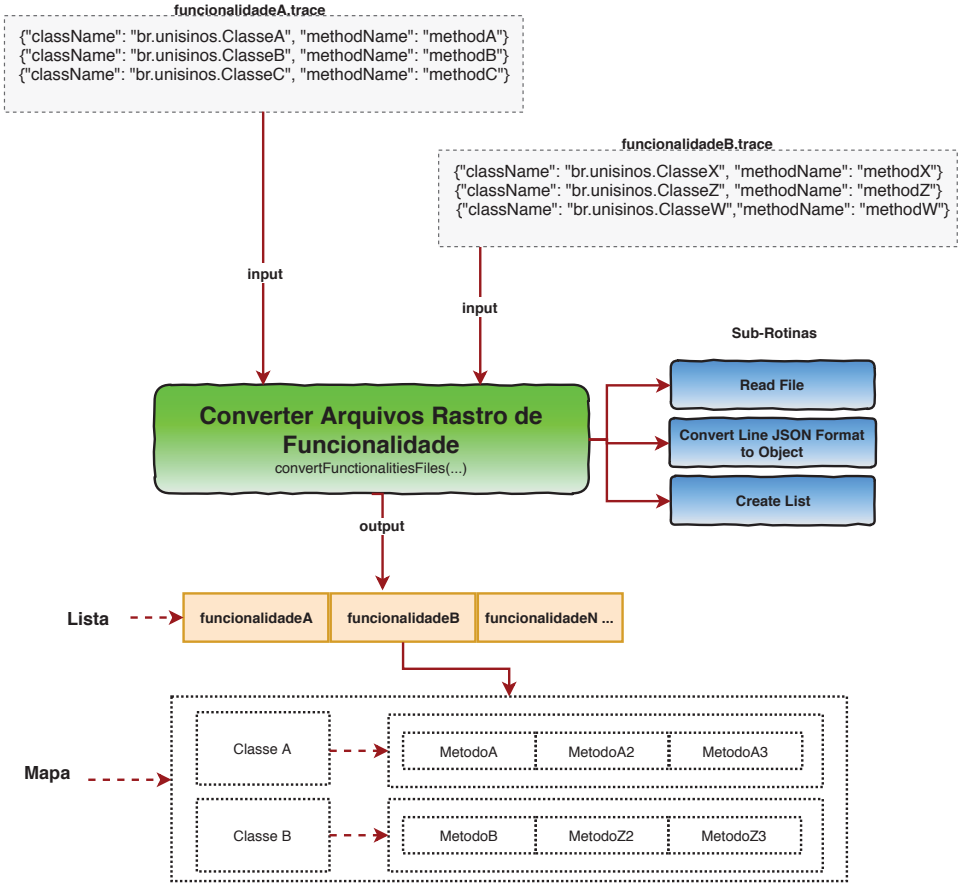
A sub-rotina **convertFunctionalitiesFiles** é a primeira executada pelo MonoBreak, conforme pode ser observado no Algoritmo 1. Essa sub-rotina recebe como parâmetro de entrada uma lista de arquivos de rastro de execução de cada funcionalidade e converte os dados contidos no arquivo em uma estrutura de dados interna do algoritmo. O processo de conversão inicia-se na linha 5 do Algoritmo 2, quando a sub-rotina percorre a lista de arquivos. Uma vez o arquivo carregado, o algoritmo extrai o nome do arquivo (linha 6) e atribui o nome da funcionalidade. Após definir o nome da funcionalidade, na linha 7, a sub-rotina lê o conteúdo do arquivo e o converte para uma lista de linhas. Essas linhas são percorridas pela sub-rotina (linha 9) que converte o texto no formato JSON para uma estrutura de dados interna do algoritmo (linha 10). Essa conversão trata-se da criação de um objeto chamado *Class* que irá conter o nome da classe e o nome do método que foi executado pelo usuário na funcionalidade.

Após a sub-rotina ler todas as linhas do arquivo da funcionalidade em questão, ela gera uma lista com todas as classes e seus respectivos métodos que foram executados por uma determinada funcionalidade (linha 11). Depois de criar a lista de classes, o algoritmo vincula a lista ao mapa de funcionalidades (linha 13). Essa lógica ocorre para todos os arquivos de funcionalidade informados no algoritmo. Ao converter todos os arquivos de rastro de execução de funcionalidade, o algoritmo retorna como resultado o mapa de funcionalidades com as respectivas classes e os métodos conforme pode ser observado na Figura 28.

Algoritmo 2: MonoBreak - Sub-rotina 1

```
1  function convertFunctionalitiesFiles (functionalityFiles)
2      foreach file in functionalityFiles do
3          functionalityName ← file.name;
4          lines ← readFile(file);
5          classes ← array[];
6          foreach line in lines do
7              class ← convertLineToClass(line);
8              classes.add(class);
9          end
10         functionalitiesMap.put(functionalityName, classes);
11     end
12     return functionalitiesMap
13 end
```

Figura 28 – MonoBreak: Diagrama Sub-rotina 1



Fonte: Elaborado pelo autor

4.3.3 Sub-rotina 2 - Converter Arquivo de Configuração da Arquitetura Aplicação

A sub-rotina **convertApplicationConfigFile** é a segunda executada pelo MonoBreak, conforme pode ser observado no Algoritmo 1. Essa sub-rotina recebe como parâmetro de entrada

um arquivo JSON com as informações da arquitetura da aplicação e o converte para uma estrutura de dados interna do algoritmo. O processo de conversão do arquivo JSON inicia na linha 2, conforme pode ser observado no Algoritmo 3. Para cada atributo encontrado no arquivo JSON, a sub-rotina gera um atributo num objeto Config. O objeto Config é utilizado internamente pelo MonoBreak para "entender" como a aplicação monolítica foi projetada, ou seja, quais as camadas que ela apresenta e quais são os nomes dos pacotes dessas camadas. Além dessas informações, esse arquivo também apresenta configurações relacionadas ao valor do peso que uma determinada camada da aplicação possui quando comparado ao valor de negócio da funcionalidade e também uma configuração relacionada ao percentual de limite de decomposição *(threshold)* que o MonoBreak utilizará para realizar a decomposição das funcionalidades em microsserviços, ou seja, qual percentual de similaridade que o MonoBreak utilizará quando necessitar agrupar ou decompor uma funcionalidade num microsserviço. Toda a lógica descrita anteriormente está representada na Figura 29.

As configurações geradas com base nas doze questões do questionário apresentado na Figura 26 serão melhores discutidas a fim de deixar compreensível o quanto essas configurações impactam no funcionamento do MonoBreak.

Algoritmo 3: MonoBreak - Sub-rotina 2

```
1 function convertApplicationConfigFile (appInfoFile)
2 |   config ← parseJSONFile(appInfoFile);
3 |   return config
4 end
```

- **Pergunta 1 - Qual o nome da aplicação?**

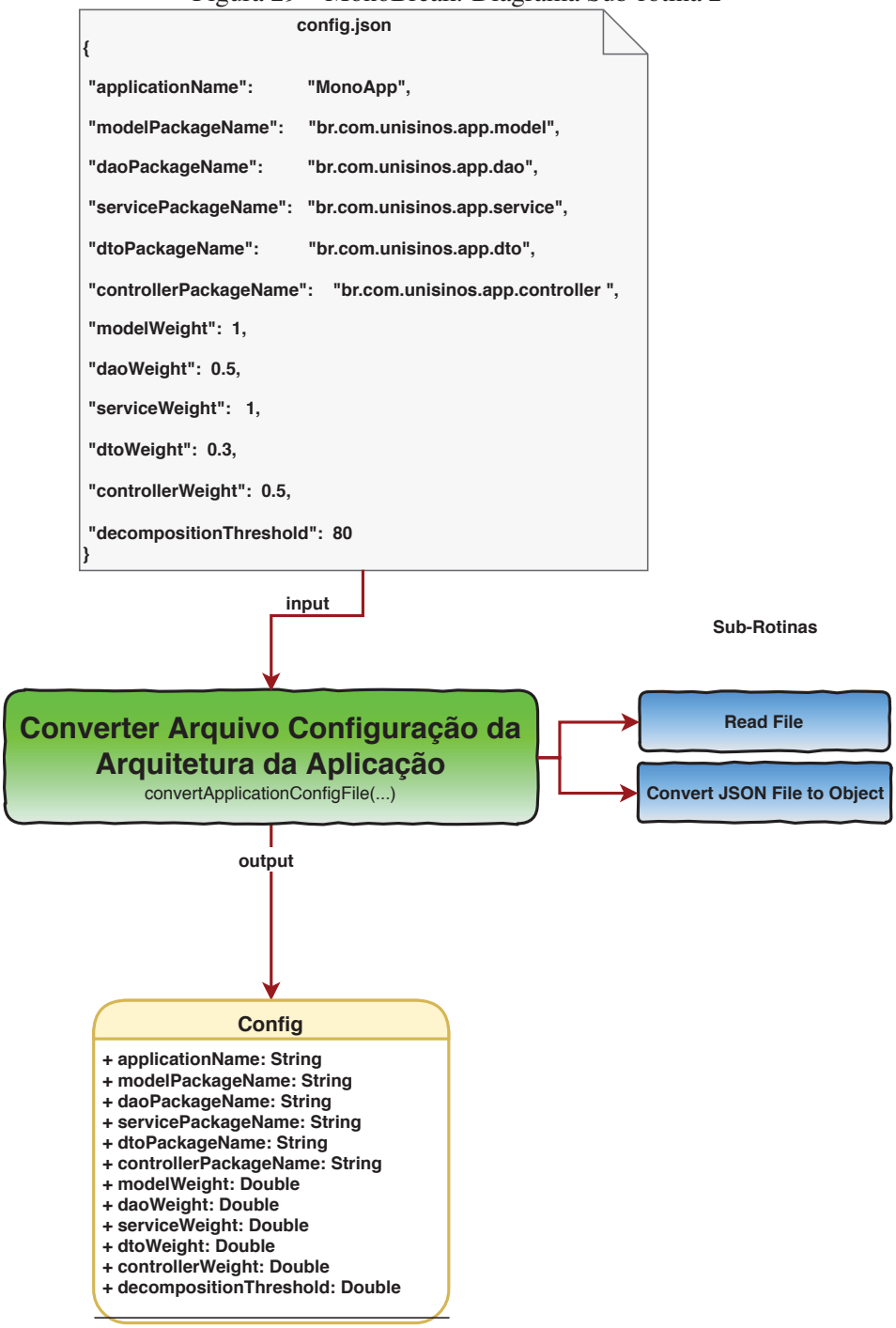
  O intuito dessa pergunta é permitir que o usuário forneça o nome da aplicação monolítica que ele escolheu como alvo dessa técnica de decomposição. Outro propósito dessa questão é possibilitar, no futuro, agrupar as recomendações de microsserviços pelo nome da aplicação, pensando numa possível ferramenta para gerar decomposições para várias aplicações simultâneas.

- **Pergunta 2 - Qual o nome do pacote que contém as classes de *Model* da aplicação?**

  O objetivo dessa pergunta é fazer com que o usuário informe o nome do pacote da sua aplicação que contém as classes responsáveis pelo domínio de negócio da aplicação. Normalmente, as classes de modelo são responsáveis por armazenar os dados que são persistidos no banco de dados da aplicação. Outra finalidade dessa questão é permitir que o MonoBreak consiga classificar as classes que foram executadas por uma funcionalidade e identificar quais delas fazem parte do pacote da camada de domínio da aplicação.

- **Pergunta 3 - Qual o nome do pacote que contém as classes de *DAOs* ou *Repositories* da aplicação?**

Figura 29 – MonoBreak: Diagrama Sub-rotina 2



Fonte: Elaborado pelo autor

O intuito dessa pergunta é fazer como que o usuário informe o nome do pacote da sua aplicação que contém as classes responsáveis pelo acesso e pela persistência de dados da aplicação. Normalmente, essas classes de repositórios, também conhecidas como DAOs, são responsáveis por realizar as operações de criar, atualizar, deletar e consultar as informações de uma base de dados da aplicação. Assim, essa pergunta visa permitir que o MonoBreak consiga classificar as classes executadas por uma funcionalidade e identificar

quais delas fazem parte do pacote da camada de acesso a dados da aplicação.

- **Pergunta 4 - Qual o nome do pacote que contém as classes de *Services* da aplicação?**

  O objetivo dessa pergunta é fazer com que o usuário informe o nome do pacote da sua aplicação que contém as classes que são responsáveis pela lógica de negócio da aplicação. Normalmente, essas classes de serviços são responsáveis por realizar a lógica de negócio da aplicação, ou seja, a camada mais importante da aplicação, pois é nela que está o valor de negócio do software. Essa pergunta tem a finalidade de permitir que o MonoBreak consiga classificar as classes que foram executadas por uma funcionalidade e identificar quais delas fazem parte do pacote da camada de serviços e regras de negócios da aplicação.

- **Pergunta 5 - Qual o nome do pacote que contém as classes de *DTOs* da aplicação?**

  O intuito dessa pergunta é fazer como que o usuário informe o nome do pacote da sua aplicação que contém as classes responsáveis pela transferência de dados entre as camada de visualização e a camada de negócio e vice-versa. Algumas arquiteturas, às vezes, não utilizam essa camada. Sendo então uma camada opcional, algumas aplicações utilizam a camada de domínio como uma camada de transporte, o que acaba gerando um alto grau de acoplamento entre o consumidor e o produtor da mensagem. Assim, o objetivo dessa pergunta é permitir que o MonoBreak consiga classificar as classes que foram executadas por uma funcionalidade e identificar quais delas fazem parte do pacote da camada de DTOs.

- **Pergunta 6 - Qual o nome do pacote que contém as classes de *Controllers* da aplicação?**

  O intuito dessa pergunta é fazer como que o usuário informe o nome do pacote da sua aplicação que contém as classes que são as responsáveis pelo controle de dados da interface com o usuário. Normalmente, essas classes de controladores são utilizadas em aplicações WEB que usam o padrão de projeto MVC. Algumas aplicações, às vezes, não utilizam essa camada, pois não possuem interface com o usuário, ou utilizam linguagens de programação ou frameworks da camada de visão que estão totalmente desacoplados da aplicação. Portanto, o objetivo dessa pergunta é permitir que o MonoBreak consiga classificar as classes que foram executadas por uma funcionalidade e identificar quais delas fazem parte do pacote da camada controladora.

- **Pergunta 7 - Qual o valor do peso do pacote de Model da aplicação?**

- **Pergunta 8 - Qual o valor do peso do pacote de DAO ou Repositories da aplicação?**

- **Pergunta 9 - Qual o valor do peso do pacote de Services da aplicação?**

- **Pergunta 10 - Qual o valor do peso do pacote de DTOs da aplicação?**

- **Pergunta 11 - Qual o valor do peso do pacote de Controllers da aplicação?**

  As perguntas de 7 a 11 têm um mesmo propósito: identificar qual o peso de cada camada da aplicação quando se deseja avaliar o quanto ela é importante para o valor de negócio da aplicação. Nessas perguntas, o objetivo é entender onde estão as principais classes e os componentes do sistema relacionados à importância de negócio do software. Isso ocorre com base nas camadas de arquitetura de software que esse questionário utiliza, como *Model*, DAOs, *Services*, DTOs e *Controllers*. Normalmente, as camadas de modelo (*Model*) e serviços (*Services*) são as que possuem maior peso de importância dentro de uma funcionalidade, pois sãos elas que dão significado ao negócio; é nelas que estão representados os dados e as regras do negócio.

  O objetivo dessas perguntas, relacionadas ao peso de cada camada no contexto do MonoBreak é, portanto, permitir que o cálculo de similaridade executado para comparar as funcionalidades tenha o máximo de assertividade. Se não forem utilizados pesos no momento do cálculo de similaridade entre as funcionalidades, o resultado pode não ser correto, tendo em vista que, às vezes, as aplicações podem apresentar classes que são comuns a todas as funcionalidades, também conhecidas como classes ou componentes transversais. Se, por exemplo, o peso de uma classe que executa um comportamento transversal, como segurança, tiver o mesmo peso de uma classe que executa uma lógica de negócio, isso poderia aumentar a similaridade entre as funcionalidades, pois esses tipos de classes transversais estão espalhados por toda as funcionalidades do sistema, o que acarretaria um valor de similaridade alto ao comparar as classes que fazem parte da funcionalidade A com a funcionalidade B. Diante desse cenário, tem-se a necessidade do usuário informar os pesos para cada camada, já que somente assim ele poderá calibrar a similaridade entre as funcionalidades, tendo em vista que o processo de decomposição utiliza os valores de similaridade para agrupar ou separar as funcionalidades em microsserviços. A correta configuração dos pesos por um especialista da aplicação permitirá que o cálculo de similaridade entre as funcionalidades seja mais assertivo em uma situação de decomposição de uma aplicação.

- **Pergunta 12 - Qual o valor do limite (*threshold*) de decomposição da aplicação?**

  O valor de limite (*threshold*) de decomposição é o parâmetro mais importante do MonoBreak, pois é ele que influencia a forma em que serão decompostas as funcionalidades em microsserviços. É esse parâmetro, por exemplo, que possibilita ao algoritmo ter um comportamento dinâmico. Através do percentual de similaridade informado nesse parâmetro, o algoritmo irá avaliar quais as funcionalidades são mais similares às outras, ou seja, quais funcionalidade devem estar juntas e quais devem estar separadas. Por exem-

plo, duas funcionalidades possuem um percentual de similaridade entre elas de 79% e o *threshold* de decomposição configurado é de 70%. Nesse caso, as funcionalidades serão agrupadas e será criado somente um microsserviço, pois o valor de similaridade entre elas é maior do que o percentual de *threshold* de decomposição. No entanto, se o percentual do *theshold* de decomposição for de 80%, as funcionalidades não serão agrupadas, sendo gerado um microsserviço por funcionalidade. A possibilidade que o MonoBreak oferece de realizar diversas simulações para identificar o melhor cenário de decomposição e consequentemente a melhor recomendação de microsserviços deve-se ao fato do parâmetro de *threshold* de decomposição, pois é ele que permite a geração de microsserviços com granularidade mais fina (um microsserviço por funcionalidade) ou microsserviços com granularidade mais grossa (duas ou mais funcionalidades por microsserviços).

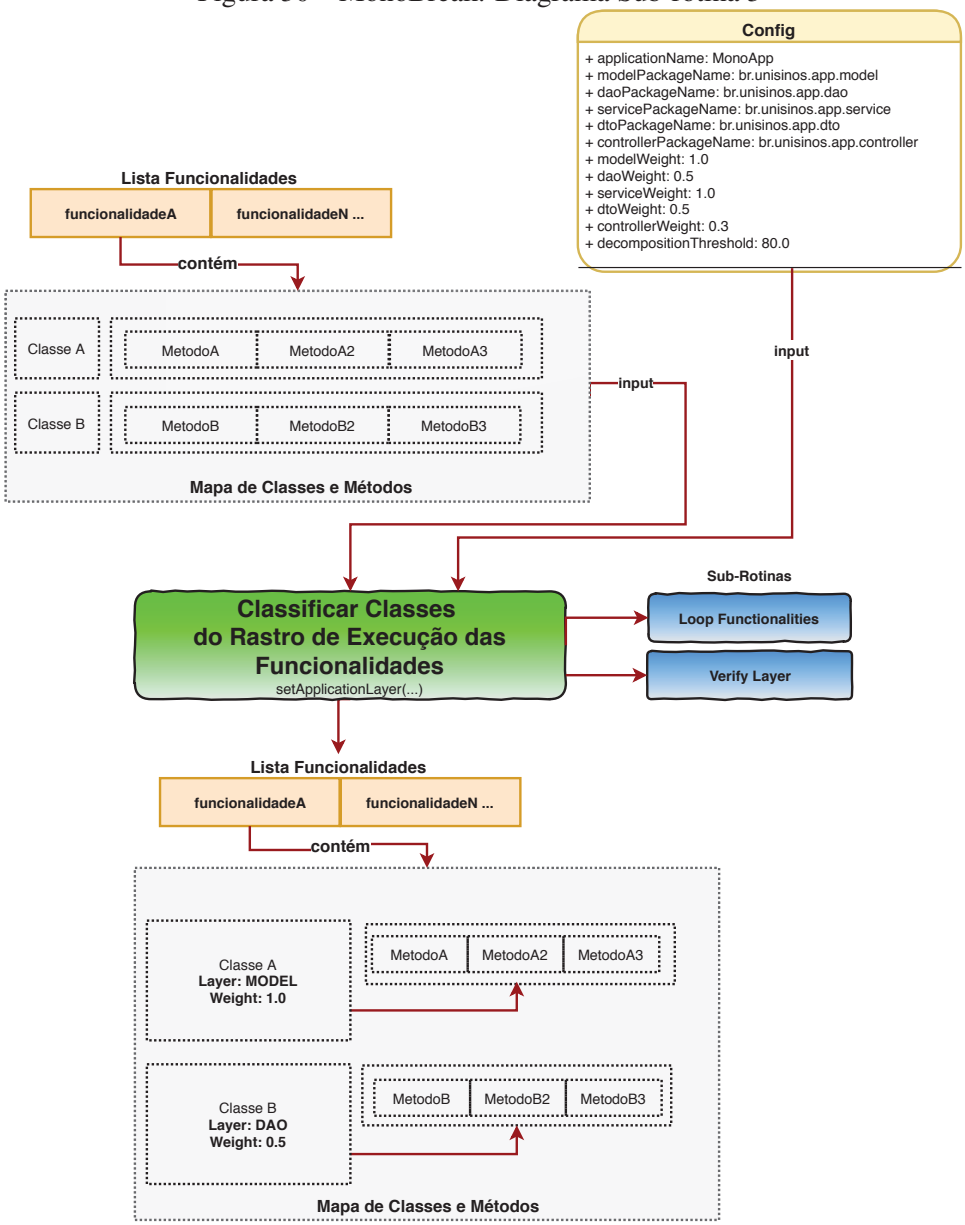
4.3.4 Sub-rotina 3 - Classificar as Classes do Rastro Execução das Funcionalidades

A sub-rotina **setApplicationLayer** é a terceira executada pelo MonoBreak, conforme pode ser observado no Algoritmo 1. Essa sub-rotina recebe como parâmetro de entrada o mapa de funcionalidade gerado pela sub-rotina 1 e o objeto de configuração da arquitetura da aplicação gerado pela sub-rotina 2. O objetivo dessa sub-rotina é percorrer cada classe de cada funcionalidade a ser decomposta e categorizá-la em qual camada da aplicação a classe pertence. Para identificar a camada da aplicação correta para cada classe, a sub-rotina utiliza o nome do pacote da camada da aplicação informado no parâmetro de configuração da arquitetura da aplicação conforme pode ser observado no Algoritmo 4. O processo de classificação inicia na linha 2, quando o algoritmo percorre a lista de funcionalidade e acessa a lista de classes vinculadas à funcionalidade (linha 3). Para cada classe acessada é verificado se o nome do pacote da classe é igual ao nome do pacote informado no parâmetro de configuração da arquitetura da aplicação para camada *MODEL*, *REPOSITORY*, *SERVICE*, *DTO* ou *CONTROLLER*, (linhas 4 a 18). Se a comparação for verdadeira, a classe é definida como parte da camada selecionada (linhas 5, 8, 11, 14, 17) e recebe um peso de importância conforme a camada selecionada (linhas 6, 9, 12, 15, 18). Por outro lado, se o pacote da classe não for igual a um dos nomes do pacote informado pelo parâmetro de configuração da arquitetura da aplicação, a classe será categorizada como parte de uma camada denominada *OTHERS* e terá um peso de importância igual 0,1 (linhas 20 e 21). Essa sub-rotina é muito importante para o MonoBreak, pois é ela que identifica as classes transversais do sistema e também permite que seja calculado o valor de similaridade entre funcionalidades baseadas no peso da camada. Esse processo também está representado na Figura 30.

Algoritmo 4: MonoBreak - Sub-rotina 3

```
1  function setApplicationLayer (functionalities, config)
2      foreach f in functionalities do
3          foreach class in f.classes do
4              if class.packageName == config.modelPackageName then
5                  class.layer← MODEL;
6                  class.weight← config.modelWeight;
7              else if class.packageName == config.repositoryPackageName then
8                  class.layer← REPOSITORY;
9                  class.weight← config.repositoryWeight;
10             else if class.packageName == config.servicePackageName then
11                 class.layer← SERVICE;
12                 class.weight← config.serviceWeight;
13             else if class.packageName == config.dtoPackageName then
14                 class.layer← DTO;
15                 class.weight← config.dtoWeight;
16             else if class.packageName == config.controllerPackageName then
17                 class.layer← CONTROLLER;
18                 class.weight← config.controllerWeight;
19             else
20                 class.layer← OTHERS;
21                 class.weight← 0, 1;
22         end
23     end
24 end
```

Figura 30 – MonoBreak: Diagrama Sub-rotina 3



Fonte: Elaborado pelo autor

4.3.5 Sub-rotina 4 - Criar Tabela de Similaridade entre as Funcionalidades

A sub-rotina **createSimilarityTable** é a quarta executada pelo MonoBreak, conforme pode ser observado no Algoritmo 1. Essa sub-rotina recebe como parâmetro de entrada o mapa de funcionalidade atualizado pela sub-rotina 3 e o objeto de configuração da arquitetura da aplicação gerado pela sub-rotina 2. O objetivo dessa sub-rotina é gerar uma tabela de similaridade entre as funcionalidades, ou seja, gerar uma matriz que represente o cruzamento entre todas as funcionalidades com seus respectivos percentuais de similaridade que uma funcionalidade possui comparados às outras. O processo de criação da tabela de similaridade inicia na linha 2 conforme pode ser observado no Algoritmo 5. A partir da lista de funcionalidade que se deseja

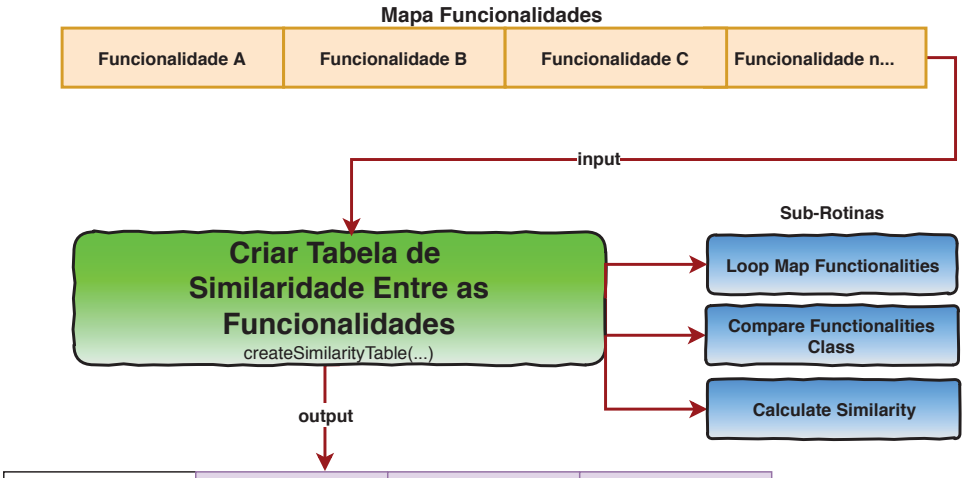
decompor, o algoritmo gera uma matriz com todas as possibilidades de combinação entre as funcionalidades, descartando somente aquelas que são iguais na linha e na coluna da matriz (linhas 3, 4 e 5). A primeira estrutura de repetição das funcionalidades é denominada de f1, e a segunda estrutura de repetição das funcionalidades é denominada f2. Para cada funcionalidade em f1 são obtidas todas as classes da funcionalidade (linha 7). Para cada classe acessada é somado o peso definido para a camada de que a classe faz parte (linhas 7 e 8). Ao obter o somatório dos pesos das camadas das classes da funcionalidade f1, o próximo passo é descobrir quais classes são comuns entre as funcionalidades f1 e f2. Nesse cenário, é aplicada uma função de intersecção de conjuntos para verificar as classes em comum entre as funcionalidades (linha 11). O resultado dessa intersecção é armazenado na lista classesEquals (linha 11). A partir dessa lista de intersecção obtida, o algoritmo percorre essa lista, classe a classe, e soma o seus pesos, procedimento esse semelhante ao aplicado na funcionalidade 1 (linhas 12 e 13). Ao obter o somatório dos pesos das classes iguais entre as funcionalidades, o próximo passo no algoritmo é realizar o cálculo de similaridade apresentado na linha 15.

Algoritmo 5: MonoBreak - Sub-rotina 4

```
1  function createSimilarityTable (functionalities, config)
2      foreach f1 in functionalities do
3          foreach f2 in functionalities do
4              if f1.name != f2.name then
5                  sumWeightClassFOne ← 0;
6                  foreach c in f1.classes do
7                      sumWeightClassFOne ← SumWeightClassFOne + c.weight;
8                  end
9                  sumWeightClassEquals ← 0;
10                 classesEquals ← intersection (f1.classes, f2.classes);
11                 foreach c in classesEquals do
12                     sumWeightClassEquals ← SumWeightClassEquals + c.weight;
13                 end
14                 similarity ← (classesEquals.size * sumWeightClassEquals) /
                     (f1.classes.size * sumWeightClassFOne) * 100;
15                 similarityTable.put(f1.name, f2.name, similarity);
16             end
17         end
18     end
19     return similarityTable
20 end
```

Após aplicada a fórmula de similaridade para cada funcionalidade, a sub-rotina retorna como resultado uma estrutura de dados que representa uma tabela (matriz) com todos os percentuais de similaridade entre todas as funcionalidades, conforme pode ser observado na Figura 31. Com base nesses percentuais de similaridade, o MonoBreak determina se as funcionalidades deverão ser decompostas ou agrupadas em um ou mais microsserviços. A próxima sub-rotina irá

Figura 31 – MonoBreak: Diagrama Sub-rotina 4



	Funcionalidade A	Funcionalidade B	Funcionalidade C
Funcionalidade A	-	20%	30%
Funcionalidade B	30%	-	85%
Funcionalidade C	10%	50%	-

Fonte: Elaborado pelo autor

explicar a lógica de recomendação.

4.3.6 Sub-rotina 5 - Agrupando as Funcionalidades por Similaridade

A sub-rotina **groupFunctionalitiesBySimilarity** é a quinta executada pelo MonoBreak, conforme pode ser observado no Algoritmo 1. Essa sub-rotina recebe como parâmetro de entrada a tabela de similaridade gerada pela sub-rotina 4 e o objeto de configuração da arquitetura da aplicação gerado pela sub-rotina 2. O objetivo dessa sub-rotina é agrupar as funcionalidades que possuem um percentual de similaridade igual ou maior ao limite de decomposição (*threshold*) configurado, conforme pode ser observado no Algoritmo 6. O processo de agrupamento das funcionalidades inicia quando o algoritmo percorre as linhas e colunas da tabela de similaridade (linhas 6 e 8). Para cada funcionalidade posicionada na linha da tabela, o algoritmo adiciona-a numa lista de similaridade (linha 7) e depois percorre todas as colunas vinculadas àquela linha, verificando se o valor de similaridade informado é igual ou maior ao valor informado no parâmetro de limite de decomposição (*threshold*) (linha 9). Se a condição for verdadeira, ou seja, se o valor da similaridade da funcionalidade da linha (i) com coluna (j) for igual ou maior ao *threshold* definido pelo usuário, a funcionalidade da coluna é adicionada à lista de similaridade (linha 10). Ao percorrer todas as colunas para uma determinada linha da tabela o algoritmo verifica se já existem recomendações criadas no mapa de microsserviços (linha 13). Caso tenha sido criado pelo menos um microsserviço no mapa, o algoritmo cria uma lista com

todas as funcionalidades de todos os microsserviços gerados até o momento (linhas 13 e 14). Com as listas de similaridades e de todas as funcionalidades geradas, o próximo passo do algoritmo é utilizar uma função de intersecção para verificar se existem funcionalidades em comum entre a lista de similaridade com a lista de todas as funcionalidades (linha 18). Se o tamanho da lista resultante da intersecção for igual a zero (linha 19), quer dizer que as funcionalidades entre os dois conjuntos (similaridade atual x total funcionalidade) não contêm nenhuma funcionalidade em comum. Sendo assim, o algoritmo coloca no mapa de microsserviços uma chave com nome Microsserviços concatenada com número gerado incrementalmente e vincula a essa chave do mapa as funcionalidades desse microsserviços (linhas 20 a 22). Caso o tamanho da lista de intersecção entre os conjuntos de similaridade e de todas as funcionalidades seja diferente de zero (linha 23), o algoritmo então irá procurar em cada microsserviço se pelo menos uma de suas funcionalidades pertence ao conjunto gerado pela linha e coluna em questão (linhas 24 e 25). Caso o tamanho da intersecção seja maior do que zero (linha 26), as funcionalidade são agrupadas ao microsserviço corrente e então adicionadas novamente ao mapa (linhas 27 e 28). O resultado final dessa sub-rotina é apresentar um mapa com os microsserviços gerados com suas respectivas funcionalidades. Todo esse processo descrito está resumido na Figura 32.

4.3.7 Sub-rotina 6 - Imprimir Recomendação de Microsserviços

A sub-rotina **printMicroservices** é a sexta executada pelo MonoBreak, conforme pode ser observado no Algoritmo 1. Essa sub-rotina recebe como parâmetro de entrada o mapa de microsserviços gerado pelo Algoritmo 6 e a lista de funcionalidades atualizada pelo Algoritmo 4. O objetivo dessa sub-rotina é exibir os microsserviços recomendados com as suas respectivas funcionalidades, suas classes e seus métodos, conforme pode ser observado no Algoritmo 7. O processo de impressão de recomendação dos microsserviços inicia na linha 2, quando o algoritmo percorre o mapa de microsserviços. Para cada microsserviço acessado no mapa, o algoritmo imprime seu nome (linha 3). Ao acessar cada microsserviço, o algoritmo percorre todas as funcionalidades vinculadas a ele (linha 4) e imprime o nome de cada funcionalidade (linha 5). Para cada funcionalidade, o algoritmo busca pelo nome a funcionalidade na lista de funcionalidades (linha 6) e, ao encontrá-la, percorre a lista de classes e métodos, imprimindo seus respectivos nomes (linhas 7 a 10). Sendo assim, esse é o resultado final do algoritmo MonoBreak, conforme pode ser observado na Figura 33.

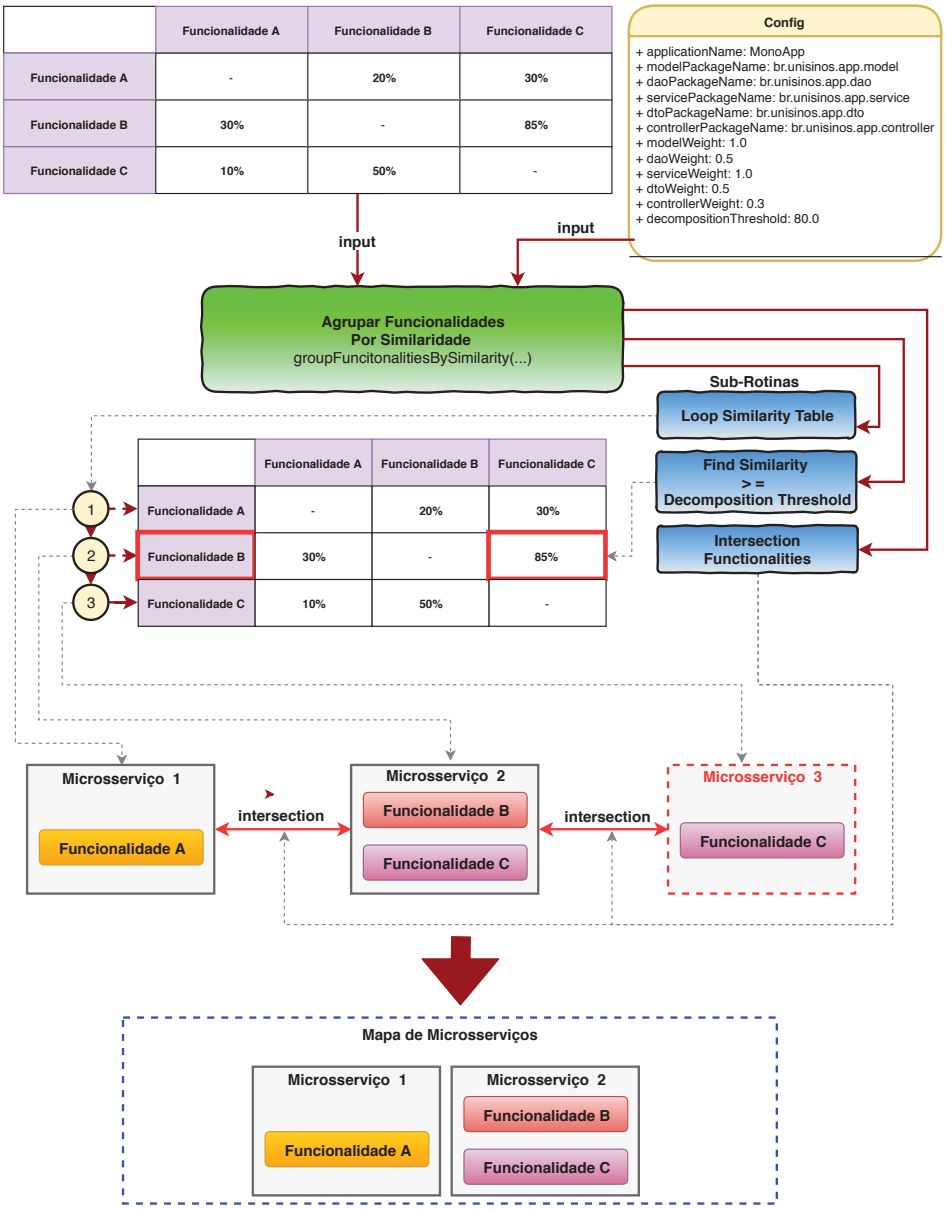
4.4 Aspecto de Implementação da Técnica

Nesta seção serão apresentados os aspectos relacionados à implementação da técnica, ou seja, como que a técnica da Monólise foi implementada nesta pesquisa, quais tecnologias e ferramentas foram utilizadas para o seu desenvolvimento. Para explicar a implementação da Monólise serão abordadas novamente as etapas que compõem a técnica, a saber: **coleta de**

Algoritmo 6: MonoBreak - Sub-rotina 5

```
1  function groupFunctionalitiesBySimilarity (similarityTable, config)
2      microserviceMap ← array[][];
3      similarities ← null;
4      allFunctionalities ← null;
5      count ← 0;
6      foreach row in similarityTable.rows do
7          similarities.add(row.name);
8          foreach column in row.columns do
9              if column.similarity >= config.decompositionThreshold then
10                 similarities.add(column.name);
11             end
12         end
13         foreach m in microserviceMap.keys do
14             foreach functionality in m.values do
15                 allFunctionalities.add(functionality);
16             end
17         end
18         intersectionList ← intersection (similarities, allFunctionalities);
19         if intersectionList.size == 0 then
20             count ← count + 1;
21             id ← "Microservice" + count ;
22             microserviceMap.put(id, similarities);
23         else
24             foreach m in microserviceMap.keys do
25                 intersections ← intersection (similarities, m.values);
26                 if intersectionList.size > 0 then
27                     unions ← union (similarities, m.values);
28                     microserviceMap.put(m, unions);
29                 end
30             end
31         end
32     end
33     return microserviceMap
34 end
```

Figura 32 – MonoBreak: Diagrama Sub-rotina 5



Fonte: Elaborado pelo autor

dados; **processamento de dados**, **disponibilização de resultados** e **implementação de resultados**, conforme pode ser observado na Figura 25. Cada etapa será abordada, nesta seção, com um enfoque mais voltado à parte de implementação da técnica.

Algoritmo 7: MonoBreak - Sub-rotina 6

```
1  function printMicroservices (microserviceMap, functionalities)
2      foreach m in microserviceMap.keys do
3          print (m);
4          foreach f in m.keys do
5              print ("Functionality : " + f);
6              functionality ← findFunctionalityByName (functionalities, f);
7              foreach c in functionality.classes do
8                  print ("Class : " + c);
9                  foreach m in c.methods do
10                     print ("Method : " + m);
11                 end
12             end
13         end
14     end
15 end
```

4.4.1 Etapa 1 - Coleta de Dados

Conforme discutido na seção 4.1 deste trabalho, a etapa de coleta de dados da técnica de Monólise apresenta quatro passos: **definir aplicação**; **configurar instrumentação de código**; **mapear funcionalidades da aplicação** e **prover informações da arquitetura da aplicação**.

Para executar esses passos, o usuário necessita utilizar tecnologias e ferramentas, sendo assim, esta seção irá apresentar as ferramentas e tecnologias utilizadas por este trabalho para realizar cada passo definido.

- **Passo 1.1 - Definir Aplicação-Alvo**

  Neste passo, o usuário da técnica deverá escolher uma aplicação monolítica que utilize uma linguagem de programação orientada a objetos, conforme apresentado na Figura 34. Ao escolher essa aplicação, o usuário deverá se certificar de que ela utiliza na sua implementação de arquitetura de software o padrão de projeto MVC, pois o MonoBreak necessita que a aplicação esteja organizada em camadas de softwares bem definidas.

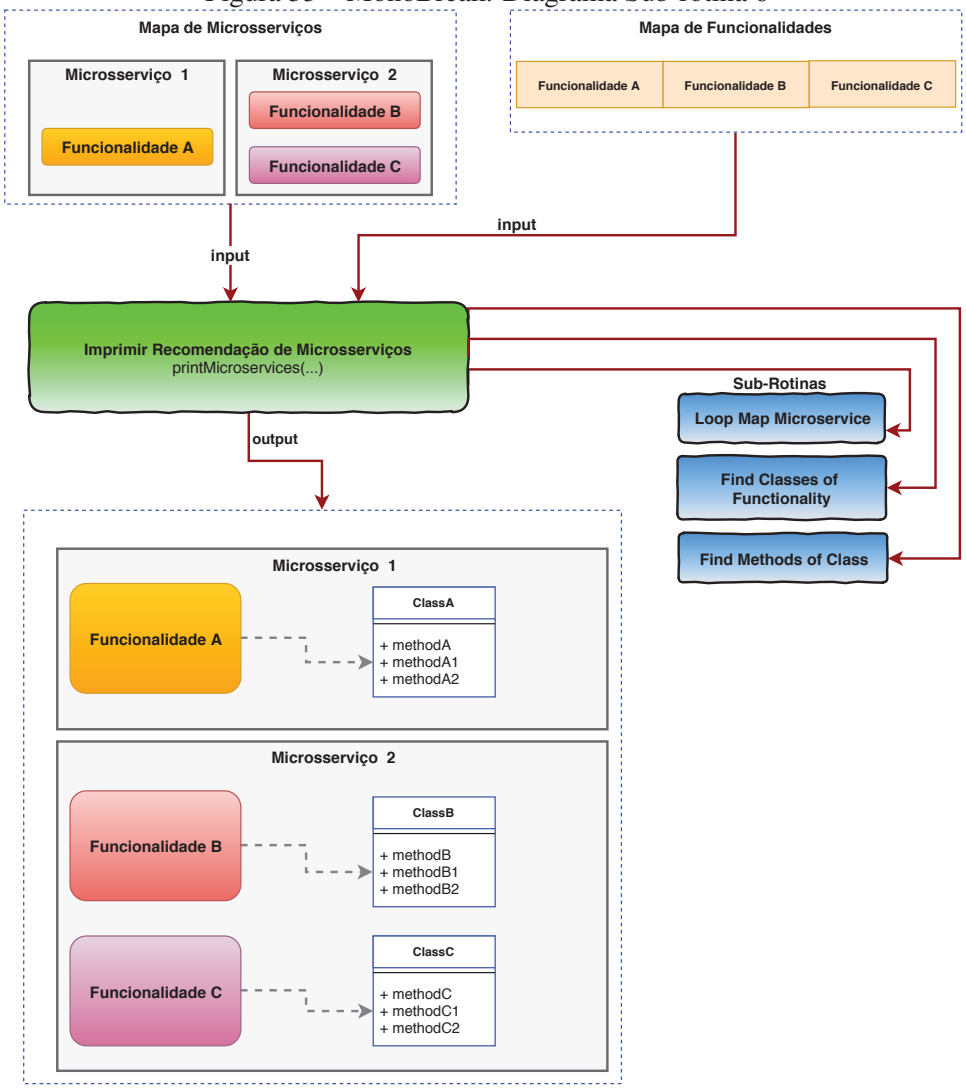
  Para esta pesquisa foi escolhida uma aplicação Web desenvolvida na linguagem Java[1], ela que será o alvo de avaliação desta técnica, mais detalhes podem ser conferidos no Capítulo 5.

- **Passo 1.2 - Configurar Instrumentação de Código**

  Uma vez definida a aplicação monolítica alvo da decomposição é necessário escolher a ferramenta que instrumentará o código da aplicação. A necessidade de utilizar uma ferramenta de instrumentação deve-se ao fato do MonoBreak precisar saber quais as classes

[1]Para saber mais sobre Java, acesse https://www.java.com/pt_BR/about/whatis_java.jsp

Figura 33 – MonoBreak: Diagrama Sub-rotina 6



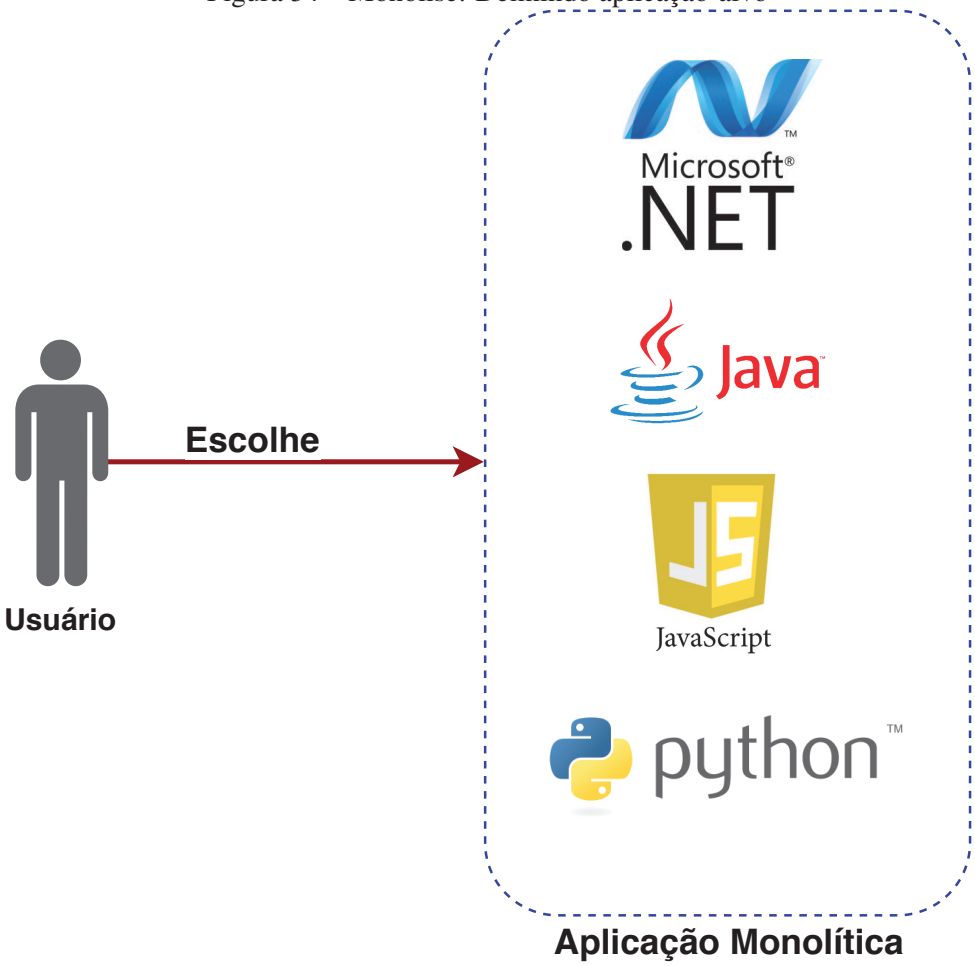
Fonte: Elaborado pelo autor

e os métodos do sistema que são executados ao utilizar determinada funcionalidade do sistema. Para entender o impacto, ao extrair as funcionalidades da aplicação monolítica e para compreender como que as funcionalidades estão acopladas umas às outras, faz-se necessário do uso de uma ferramenta que possibilite realizar um "raio-x"da aplicação. Não foi considerada nessa pesquisa uma etapa manual para capturar a execução das classes e dos métodos do sistema devido ao alto grau de esforço e complexidade que isso exige, por isso optou-se pela utilização da ferramenta que facilita esse trabalho.

Então, com vistas a obter o rastro de execução das classes e dos métodos das funcionalidades da aplicação e permitir o acompanhamento dessa captura em tempo real, foi proposto nessa técnica utilizar bibliotecas e ferramentas para esse fim, tendo em vista que uso desse tipo de ferramenta é um modelo padrão utilizado por diversos tipos de linguagem de programação, como Java, .Net, Python, PHP e etc.

Conforme mencionado na etapa anterior, a aplicação escolhida para avaliação desta pes-

Figura 34 – Monólise: Definindo aplicação-alvo



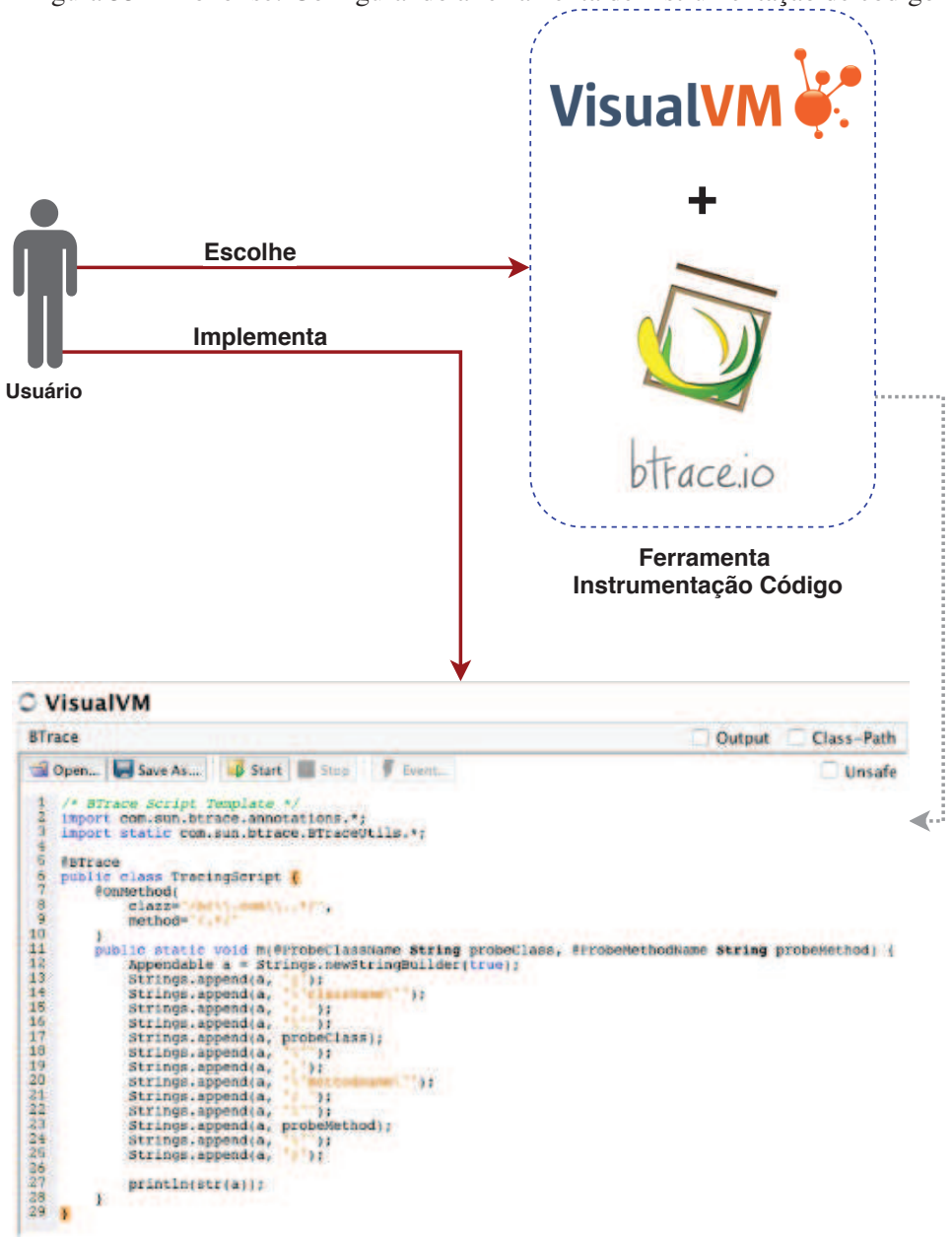
Fonte: Elaborado pelo autor

quisa foi desenvolvida em Java. Portanto, para instrumentar o código da aplicação foi utilizada a ferramenta **VisualVM**[2] e o framework **BTrace**[3], conforme pode ser observado Figura 35. O VisualVM é uma ferramenta para monitoramento de aplicações Java; nela, é possível monitorar a performance da aplicação, o seu consumo de memória e CPU, dentre outras funcionalidades. Uma vantagem do VisualVM é que ele é uma ferramenta de código aberto, o que possibilita ao desenvolvedor adicionar novos plugins. Nesse cenário da pesquisa foi adicionado ao VisualVM o plugin que possibilita a integração com o framework de instrumentação de código BTrace. O BTrace é um framework que permite realizar a instrumentação de código fonte de aplicações que executam em uma máquina virtual Java, também conhecida como JVM. Para instrumentar o código fonte de uma aplicação Java, o BTrace oferece uma API que permite ao usuário escrever scripts para capturar informações referentes ao nomes das classes, dos pacotes e métodos que foram executados quando um usuário utilizou uma funcionalidade do sistema. Com o uso desse framework somado ao ambiente oferecido pelo VisualVM será possível ao usuário cap-

[2]Para saber mais sobre VisualVM, acesse https://visualvm.github.io/
[3]Para saber mais sobre BTrace, acesse https://github.com/btraceio/btrace

turar e armazenar o rastro de execução de cada funcionalidade executada no sistema.

Figura 35 – Monólise: Configurando a ferramenta de instrumentação de código



Script BTrace

Fonte: Elaborado pelo autor

- **Passo 1.3 - Mapear as Funcionalidades da Aplicação**

  Conforme apresentado na Figura 36, o usuário da técnica necessitará seguir pelo menos seis fases para concluir este passo. Na primeira, deve-se escolher a aplicação monolítica; na segunda, é preciso mapear as funcionalidades da aplicação, fazendo o levantamento de quais estão disponíveis. Já na terceira fase, é necessário definir as funcionalidades que serão decompostas em microsserviços. Na quarta, o usuário executa as funcionalidades, como se estivesse utilizando o sistema em produção. Na penúltima fase, há a

instrumentação do código da funcionalidade em uso, conforme já citado no passo anterior. Ao utilizar a funcionalidade, a ferramenta de instrumentação de código BTrace com o VisualVM captura o fluxo de execução da aplicação e gera, no console do VisualVM, o nome dos pacotes, das classes e dos métodos executados pela funcionalidade em uso. A geração desses dados no console do VisualVM é baseada na implementação do script BTrace informado na Figura 36. Cada linha gerada corresponde a uma classe e a um método que foi executado no momento do uso da funcionalidade, utilizando o formato JSON. Por exemplo, dentro do atributo *className*, está informado o pacote e o nome da classe executada, no atributo *methodName*, está o nome do método da classe que foi executada. Esse conceito de linha por linha demonstra a pilha de chamadas de métodos que foi executada no momento de uso de uma funcionalidade. Por fim, na última fase, o usuário deve copiar o conteúdo gerado pelo BTrace no console do VisualVM e mover para um arquivo com extensão .trace, gerando um arquivo para cada funcionalidade executada.

- **Passo 1.4 - Prover Informações da Arquitetura da Aplicação**

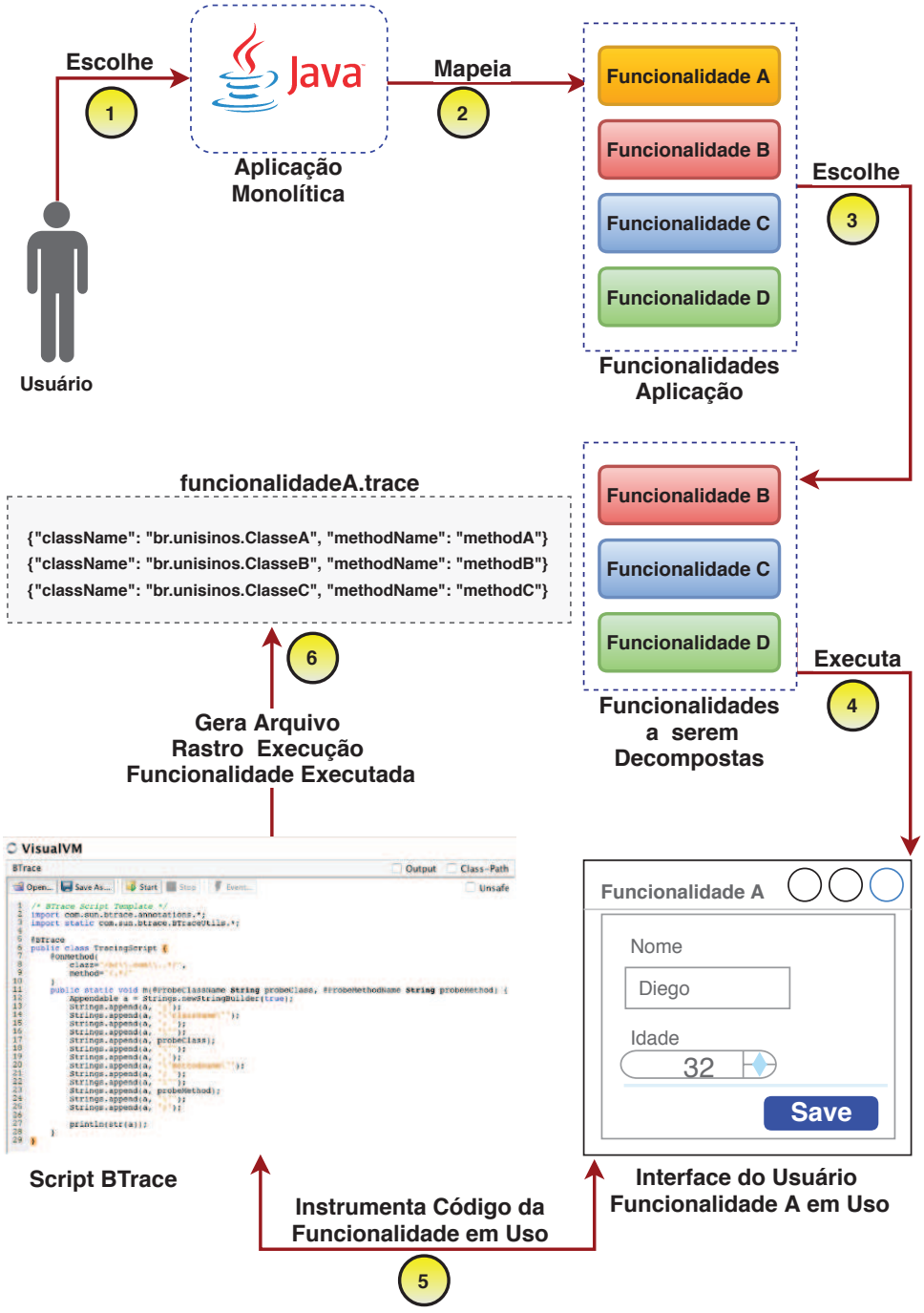
  Nesse passo o usuário da técnica deverá primeiramente responder um questionário com doze perguntas, as quais referem-se a questões específicas de como a arquitetura de software da aplicação foi projetada. É nesse passo que o usuário deverá informar o nome dos pacotes das camadas da aplicação, o peso de relevância das camadas no contexto de negócio e também definir o percentual do limite de decomposição que o MonoBreak utilizará para realizar a recomendação dos microsserviços. A partir do questionário respondido, conforme Figura 37, o usuário deverá gera um arquivo com os dados no formato JSON[4], para que o MonoBreak possa utilizar no momento da decomposição.

4.4.2 Etapa 2 e 3 - Processamento de Dados e Disponibilização dos Resultados

Para processar os dados e disponibilizar os resultados, o algoritmo MonoBreak utilizou a linguagem de programação Java 8 e os frameworks Google Guava Libraries[5] e Spring Boot[6], conforme pode ser observado na Figura 38. O Java foi escolhido para implementação do algoritmo, porque a aplicação-alvo desta pesquisa foi implementada em Java e também porque o autor do trabalho possui maior conhecimento com essa linguagem. Toda a lógica de programação do MonoBreak foi construída utilizando as estruturas de dados fornecidas pelo Java e pelo Google Guava Libraries. O Guava, por exemplo, oferece uma ótima API para trabalhar como operações de conjuntos como intersecção e união, além de oferecer uma estrutura de dados específica para montar e manipular tabelas multi-dimensionais. O MonoBreak também utilizou na sua implementação o framework Spring Boot, que possibilita, por exemplo, que aplicação seja executada em modo de linha de comando ou pelo Web Browser. O uso do Spring simplificou a

[4]Para saber mais sobre JSON, acesse https://www.json.org/json-pt.html
[5]Para saber mais sobre Google Guava, acesse https://github.com/google/guava
[6]Para saber mais sobre Spring Boot, acesse https://spring.io/projects/spring-boot

Figura 36 – Monólise: Mapeando as funcionalidades da aplicação



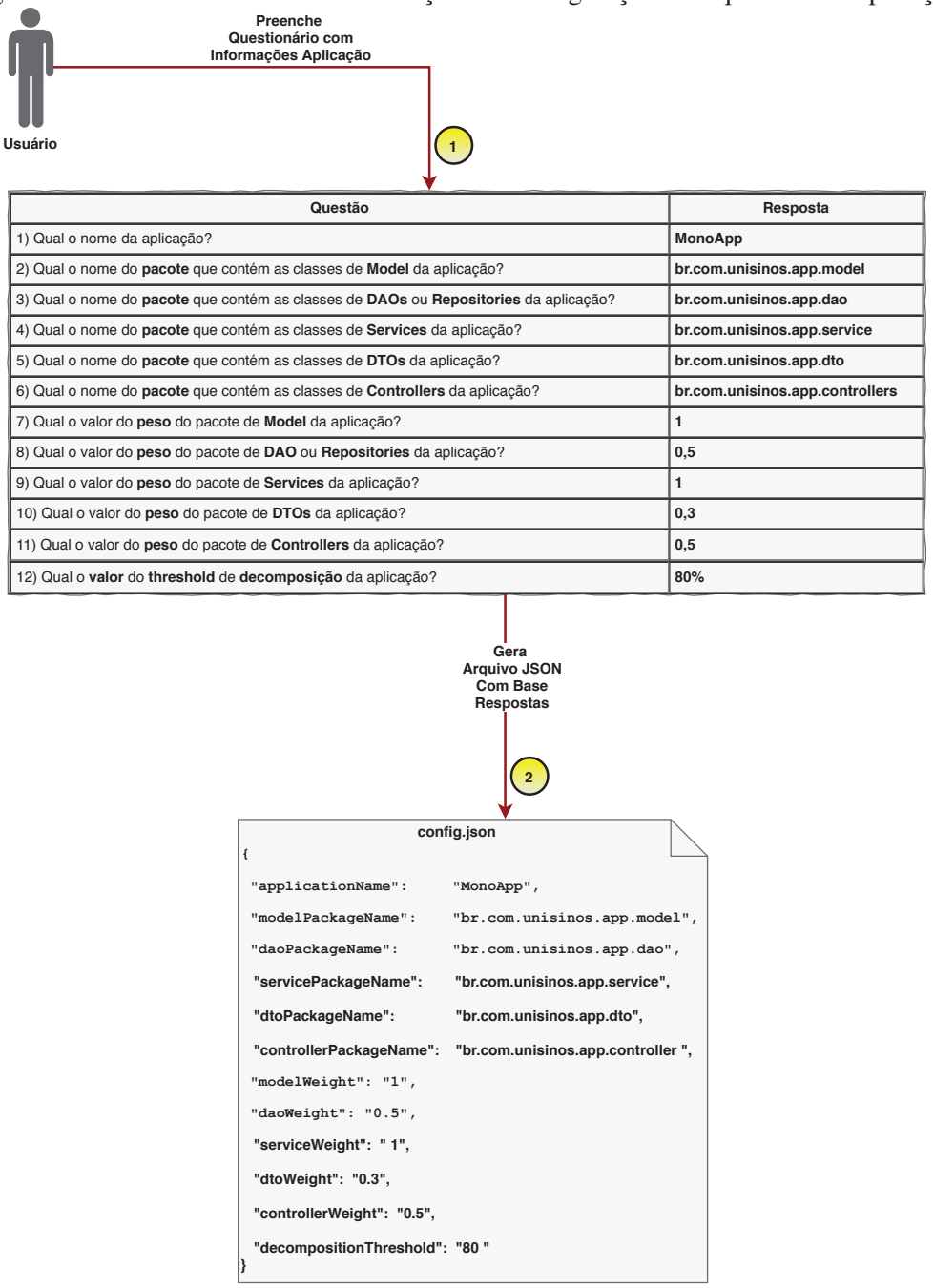
Fonte: Elaborado pelo autor

manipulação de arquivos no formato texto puro e JSON.

4.4.3 Etapa 4 - Implementação dos Resultados

A implementação dos resultados é a última etapa da técnica de Monólise e é nessa etapa que devem ser implementados os microsserviços recomendados pelo MonoBreak. Neste estudo, os microsserviços que forem recomendados serão implementados com a linguagem Java

Figura 37 – Monólise: Provendo informações da configuração da arquitetura da aplicação

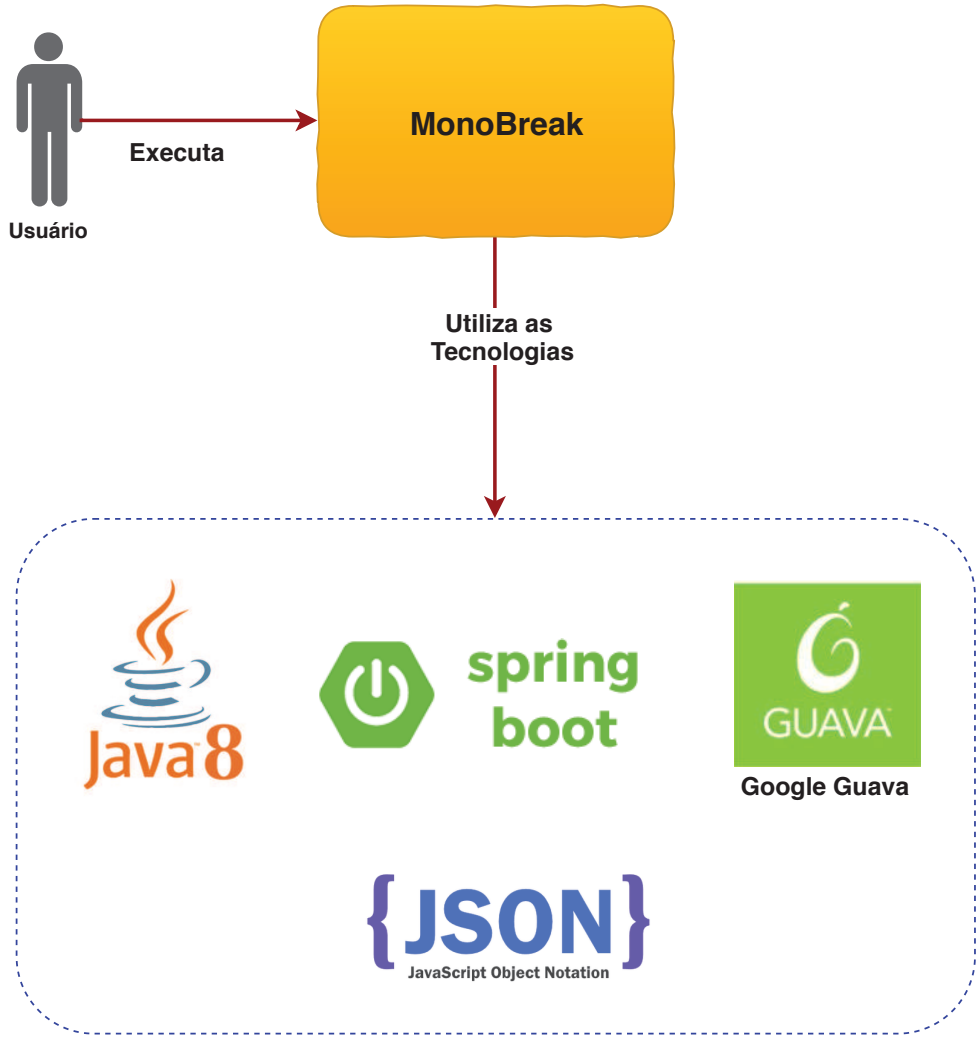


Fonte: Elaborado pelo autor

8 e com o framework Spring Boot, conforme pode ser observado na Figura 39. O Spring Boot atualmente é o framework mais recomendado para desenvolver aplicações Java para uma arquitetura baseada em microsserviço. Isso ocorre devido às suas diversas integrações com diferentes tecnologias e ferramentas. Caso um microsserviço necessitar armazenar dados num banco de dados relacional, será utilizado o MySQL[7]; por outro lado, se existir um requisito que necessite de uma banco de dados baseado em documentos, será utilizado o banco NOSQL MongoDB[8].

[7]Para saber mais sobre MySQL, acesse https://www.mysql.com
[8]Para saber mais sobre MongoDB, acesse https://www.mongodb.com

Figura 38 – MonoBreak: Tecnologias e frameworks utilizados



Fonte: Elaborado pelo autor

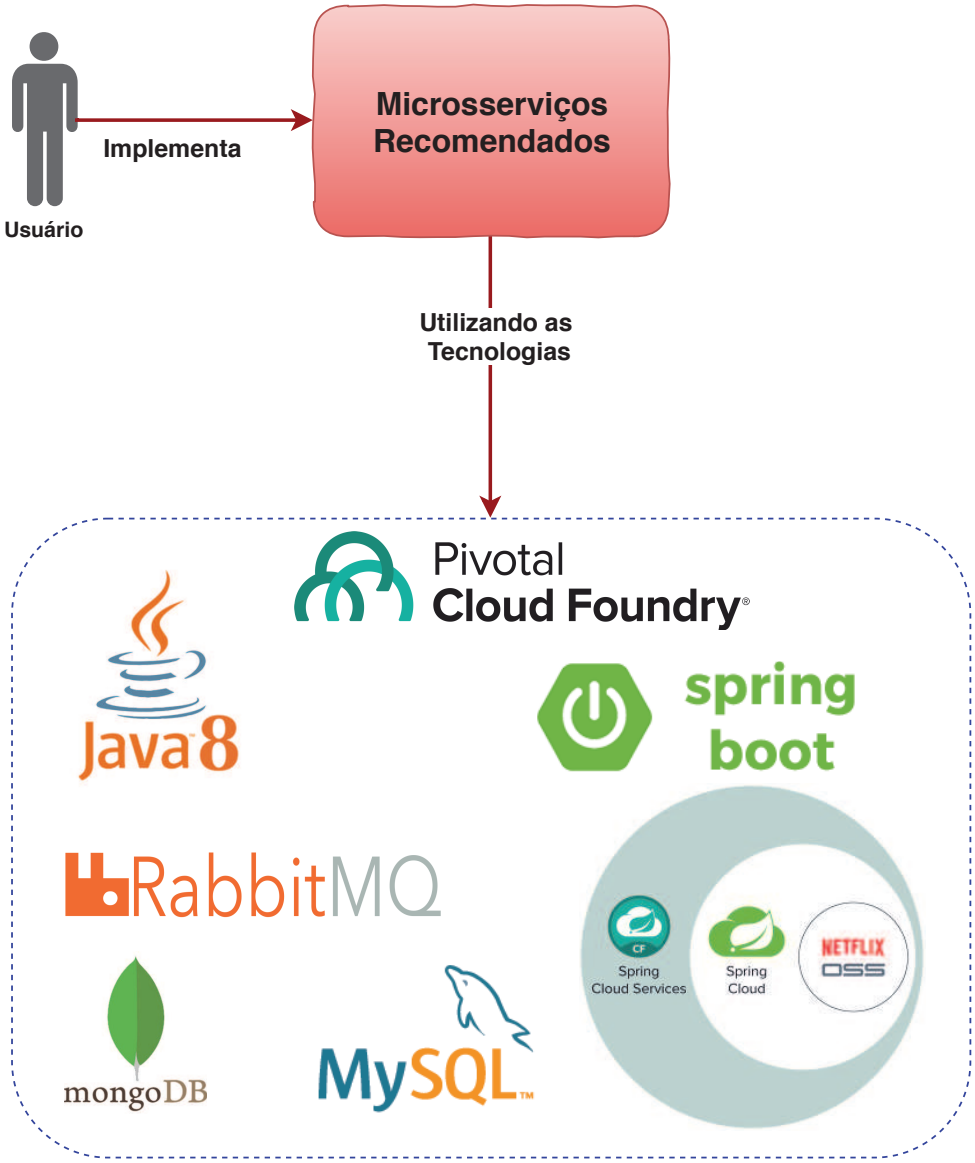
Para microsserviços que necessitarem trocar mensagens entre si será utilizado o RabbitMQ[9], o RabbitMQ é um **middleware** de mensageria que permite a troca de mensagem através do formato *publish* e *subscribe*. Todos os microsserviços construídos nesta pesquisa serão executados dentro do PaaS Pivotal Cloud Foundry[10]. O objetivo de utilizar essa plataforma é que ela é preparada para execução de aplicações nativas para nuvem, ou seja, ela oferece diversas ferramentas que são necessárias para o desenvolvimento de aplicações baseadas em microsserviços, como registro de serviços, servidor de configuração, roteamento de serviços e também integração nativa com Spring Boot e ferramentas como banco de dados e mensageria citadas

[9]Para saber mais sobre RabbitMQ, acesse https://www.rabbitmq.com
[10]Para saber mais sobre Pivotal Cloud Foundry, acesse https://pivotal.io/platform

anteriormente.

Os detalhes de implementação e uso dessas tecnologias serão discutidos no Capítulo 5 deste trabalho.

Figura 39 – Tecnologias e frameworks dos microsserviços



Fonte: Elaborado pelo autor

5 AVALIAÇÃO

Este capítulo descreve como a técnica de Monólise (proposta no Capítulo 4) foi avaliada através de um estudo de caso. A avaliação consistiu na comparação da decomposição realizada pela Monólise com a decomposição executada por um especialista na aplicação-alvo utilizada no estudo de caso. Essa comparação permitiu avaliar a efetividade da Monólise através de oito cenários realísticos de decomposição.

O estudo de caso elaborado foi estruturado em quatro fases, as quais são discutidas nas próximas seções. Na seção 5.1, a primeira fase é apresentada, a qual consiste na escolha e descrição da aplicação monolítica utilizada na avaliação; na seção 5.2, demonstra-se como será realizada a decomposição da aplicação-alvo; já na seção 5.3, detalha-se a avaliação da técnica proposta e, por fim, na seção 5.4, são discutidos os resultados apresentados.

5.1 Fase 1 - Desenvolvimento e Escolha da Aplicação

Nesta seção, será detalhada a aplicação-alvo escolhida como estudo de caso que avaliará a técnica de Monólise.

Para tanto, esta seção foi organizada em quatro tópicos: o primeiro irá descrever o que é a aplicação-alvo; o segundo irá apresentar brevemente as funcionalidades que a aplicação possui; já o terceiro tópico irá explicar as tecnologias e os *frameworks* utilizados na implementação da arquitetura de software da aplicação; por fim, o último tópico irá informar o motivo pela escolha dessa aplicação.

5.1.1 Aplicação-Alvo

A aplicação escolhida como estudo de caso para avaliação da técnica de mónolise foi o Sistema de Mapeamento de Competência (SMC). O SMC é um sistema que tem por objetivo permitir a seus usuários tanto desenvolver quanto ser objeto de avaliações durante a execução de projetos, ou seja, enquanto participam de um projeto, todos os colaboradores, a partir de competências pré-definidas, terão a possibilidade não só de se avaliarem como também de avaliar a seus colegas.

Além do processo de avaliação, o sistema contará ainda com um módulo destinado à formação profissional dos colaboradores. Nele, os usuários poderão acompanhar a formação profissional de seus funcionários fora da empresa como, por exemplo, se eles têm realizado cursos na área, certificações, etc. Esse módulo é útil principalmente aos Recursos Humanos e às chefias.

Através do SMC, o usuário poderá cadastrar: cargos, projetos, equipes que compõem esses projetos, temas (Competências), perguntas vinculadas aos temas e também definir pesos para os cargos em um determinado tema. Além disso, o usuário poderá cadastrar idiomas, escolaridade e todas as informações necessárias para a parte administrativa do sistema, tais como: usuários,

perfis, menus e permissões.

O processo de mapeamento de competência inicia com a criação de uma pesquisa, na qual tem de se definir o projeto que se deseja avaliar, quais as competências serão mapeadas e qual a forma de avaliação a ser aplicada, isto é, se será uma avaliação 180 graus ou 360 graus e se terá a autoavaliação. Na avaliação 180º, somente os cargos de chefia avaliam a seus subordinados; já na avaliação 360º, todos os envolvidos avaliam-se e são avaliados, inclusive os cargos superiores. Tendo em vista essas explicações, pode-se confirmar que o sistema tem como público-alvo todo o tipo de empresa ou instituição que deseje mapear as competências de seus colaboradores. Por isso, nesse sistema, têm-se, no mínimo, três tipos de usuários: Administrador, Colaborador (Avaliado ou Avaliador) e RH (Recursos Humanos).

- **Administrador**

  O administrador, como a denominação indica, seria o responsável por administrar o sistema, isto é, por criar os usuários, definir os perfis e as funcionalidades que estariam disponíveis para um determinado perfil.

- **Colaborador**

  O Colaborador será o responsável por cadastrar a sua formação profissional e participar das pesquisas, respondendo ao questionário de perguntas. As funcionalidades do SMC serão detalhadas na próxima seção.

- **Recursos Humanos (RH)**

  O RH seria responsável por definir os cargos, cadastrar os projetos com suas equipes, cadastrar os temas com suas perguntas e seus pesos, cadastrar os idiomas e as escolaridades. Após realizada essa etapa de cadastro, o RH poderá criar pesquisas para que os colaboradores possam se autoavaliar ou ser avaliados. O RH ainda conta com o recurso de relatórios, através do qual poderá acompanhar o andamento das pesquisas, divulgar para cada colaborador o seu desempenho nas avaliações e consultar a formação profissional de cada colaborador.

5.1.2 As Funcionalidades do SMC

O SMC apresenta 21 funcionalidades, sendo 12 de **CRUD** (verde), 6 de processo (azul) e 3 de relatórios (amarelo), conforme pode ser observado na Figura 40. As funcionalidades pertencentes ao grupo CRUD são as que não possuem regras de negócio complexas, apenas apresentam operações básicas, como criar (*CREATE*), ler (*READ*), atualizar (*UPDATE*) e deletar (*DELETE*), elas normalmente são cadastros básicos do sistema. Já as funcionalidades do grupo de processos são as mais complexas da aplicação, pois nelas estão implementadas as principais regras de negócio para os usuários, ou seja, elas apresentam maior valor de negócio do sistema. As funcionalidades do grupo de relatórios são as responsáveis por disponibilizar

os resultados das pesquisas de mapeamento de competência. Após a imagem, há uma sucinta descrição do que cada funcionalidade representa no SMC:

Figura 40 – Funcionalidades do SMC

Fonte: Elaborado pelo autor

- **F001 – Autenticação e Autorização de Usuários**

  Funcionalidade que permite aos usuários do sistema acessar às suas informações de forma segura e confiável, através de credenciais únicas (login e senha). É essa funcionalidade que realiza todo o controle de acesso dos usuários aos módulos do sistema.

- **F002 - Esqueci Minha Senha**

  Funcionalidade que disponibiliza uma nova senha de acesso, através de envio de e-mail para os usuários do sistema.

- **F003 - Alterar Senha**

  Funcionalidade que permite ao usuário alterar a sua senha de acesso.

- **F004 - Cadastro de Cargo**

  Funcionalidade que permite consultar, inserir, atualizar e deletar um cargo do sistema.

- **F005 - Cadastro de Usuário**

  Funcionalidade que permite consultar, inserir, atualizar e deletar um usuário do sistema.

- **F006 - Cadastro de Perfil**

  Funcionalidade que permite consultar, inserir, atualizar e deletar um perfil do sistema.

- **F007 - Cadastro de Perfil x Usuário**

  Funcionalidade que permite vincular os usuários a um determinado perfil do sistema.

- **F008 - Cadastro de Perfil x Menu**

  Funcionalidade que permite vincular os menus a um determinado perfil do sistema.

- **F009 - Cadastro de Parâmetro**

  Funcionalidade que permite ao usuário consultar, atualizar e inserir parâmetros de configuração do sistema.

- **F010 - Cadastro de Projeto**

  Funcionalidade que permite consultar, inserir, atualizar e excluir projetos do sistema. Além desses recursos, tal funcionalidade possibilita vincular usuários que irão compor a equipe do projeto.

- **F011 - Cadastro de Tema**

  Funcionalidade que permite consultar, inserir, atualizar e excluir tema do sistema. Além desses recursos, ela possibilita vincular perguntas e pesos por cargo ao tema.

- **F012 - Cadastro de Pesquisas**

  Funcionalidade que permite consultar, inserir, atualizar e excluir uma pesquisa. Além desses recursos, tal funcionalidade possibilita vincular temas, ordenar as perguntas dos temas escolhidos bem como publicar a pesquisa para o acesso aos usuários.

- **F013 - Cadastro de Idiomas**

  Funcionalidade que permite consultar, inserir, atualizar e deletar idiomas do sistema.

- **F014 - Cadastro de Escolaridades**

  Funcionalidade que permite consultar, inserir, atualizar e deletar escolaridade do sistema.

- **F015 - Cadastro Formação Profissional**

  Funcionalidade que permite inserir e atualizar uma formação profissional no sistema. Além desses recursos, possibilita também vincular históricos acadêmicos e idiomas à formação profissional.

- **F016 - Consulta Formação Profissional**

  Funcionalidade que permite acesso a todas as formações profissionais dos usuários do sistema.

- **F017 - Avaliação de Equipes**

  Funcionalidade que permite ao usuário, através de um questionário de perguntas e respostas, avaliar a si próprio ou aos demais colegas.

- **F018 - Acompanhamento de Pesquisas**

  Funcionalidade que permite aos usuários acompanhar o andamento das pesquisas.

- **F019 - Relatório de Acompanhamento de Pesquisa**

  Funcionalidade que permite ao usuário exportar para um arquivo PDF o andamento das pesquisas.

- **F020 - Relatório de Mapeamento de Competência**

  Funcionalidade que permite ao usuário exportar para um arquivo PDF um gráfico, comparando o desempenho atingido na avaliação e o desempenho ideal para o seu cargo. Esse comparativo sempre levará em conta o cargo do avaliado.

- **F021 - Relatório de Ranking de Competências**

  Funcionalidade que permite ao usuário exportar para um arquivo PDF um ranking de quais são as melhores e as piores competências de um determinado usuário.

5.1.3 Arquitetura de Software do SMC

Nesta seção, serão apresentadas as tecnologias e os *frameworks* utilizados na implementação do SMC. Também serão discutidos os componentes que fazem parte da arquitetura de software do SMC.

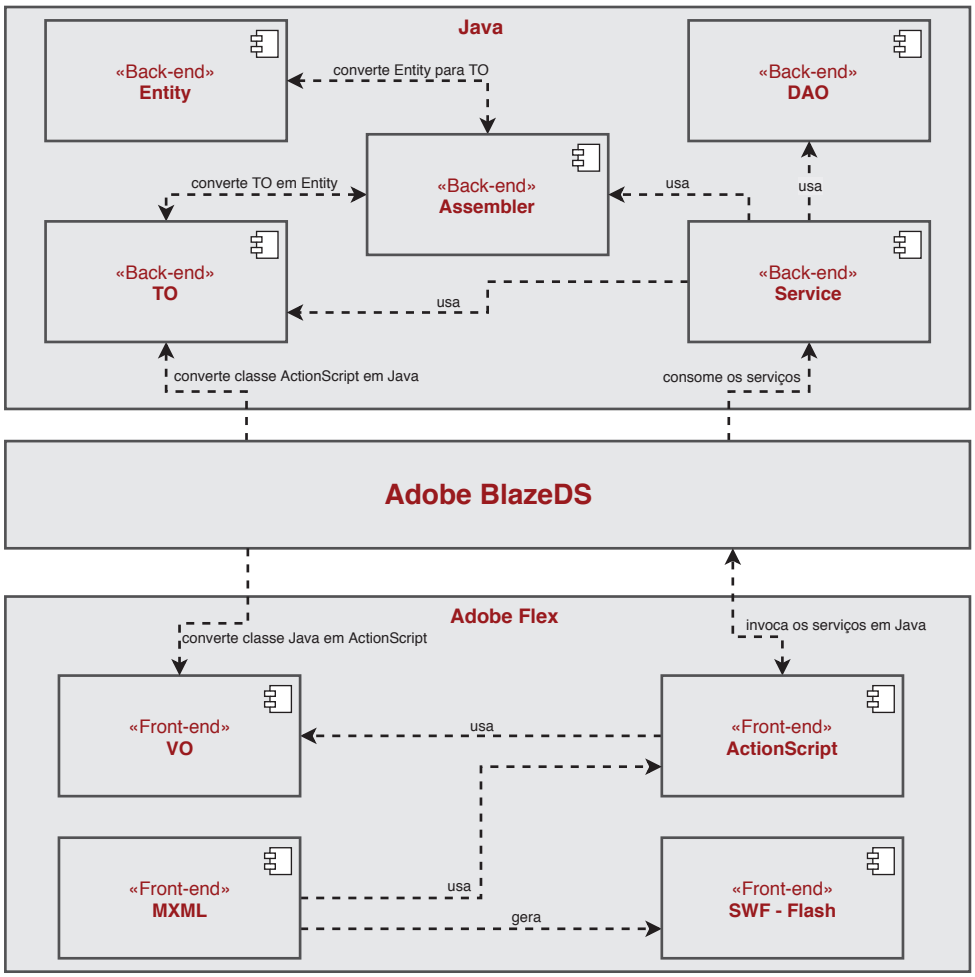
O SMC é uma aplicação Web que utiliza o padrão de arquitetura MVC. Sua arquitetura de software é monolítica, ou seja, todos os componentes de software e as funcionalidades executam em um mesmo artefato de execução; no caso do SMC, esse artefato é do tipo WAR[1].

[1]Para saber mais sobre WAR, acesse https://docs.oracle.com/javaee/7/tutorial/packaging003.htm.

O SMC possui duas camadas arquiteturais bem definidas a de *Back-end* e a de *Front-end*. Na camada de *Back-end*, estão implementados: as regras de negócio (*Service*); o acesso a dados (DAO); o modelo de dados (*Entity*); a transferência de objetos de dados (TO) e a conversão de dados (*Asssembler*). Já na camada de *Front-end*, estão implementados os componentes responsáveis pela interface com o usuário, ou seja, é nela que se encontram os componentes de objeto de valor (VO); a interface com usuário (MXML e *SWF Flash*) e a parte de controle da interface com o usuário (*ActionScript*).

Cada componente citado anteriormente está representado no diagrama de componentes da arquitetura do SMC, conforme Figura 41. Para um melhor entendimento sobre a arquitetura de software, cada componente está descrito na sequência:

Figura 41 – Diagrama de componentes da arquitetura do SMC



Fonte: Elaborado pelo autor

- **Back-end - Entity**

  Este componente contém o modelo de dados da aplicação. É nessa camada que será realizado o mapeamento objeto relacional com o banco de dados. Para realizar tal mapeamento, será utilizado o *framework* JPA[2].

[2]Para saber mais sobre JPA, acesse https://docs.oracle.com/javaee/5/tutorial/doc/?wp406143&PersistenceIntro.html.

- **Back-end - DAO (Data Access Object)**

  Componente responsável pelo acesso a dados. É nele que são executadas as operações de CRUD na base de dados do SMC.

- **Back-end - Assembler**

  Componente responsável pela conversão entre objetos de transferência de dados (TO) para entidades (*Entity*) e vice-versa.

- **Back-end - TO (Transfer Object)**

  Componente responsável por armazenar os dados que serão transferidos entre as camadas *back-end* e *front-end*.

- **Back-end - Service**

  Componente que contém as regras de negócio do SMC. É esse componente que expõe os dados para camada *front-end*.

- **Adobe BlazeDS**

  Middleware que permite a troca de mensagens entre a camada *back-end* em Java com a camada *front-end* em Adobe Flex.

- **Front-end - VO (Value Object)**

  Classe em ActionScript que recebe os valores da interface do usuário.

- **Front-end - ActionScript**

  Scripts que permitem a manipulação das telas, isto é, são esses scripts que realizam a lógica das interfaces.

- **Front-end - MXML**

  Linguagem que permite a criação das interfaces do usuário.

- **Front-end - SWF - Flash**

  Formato gerado a partir da compilação dos MXML. É esse o componente que apresenta a interface do usuário.

Conforme mencionado anteriormente, o SMC é uma aplicação WEB desenvolvida com duas linguagens de programação. Para a camada *back-end*, foi utilizado o Java[3] juntamente com o *Spring Framework*[4]. Para a camada *front-end*, foram utilizadas as linguagens *ActionScript*[5] e

[3]Para saber mais sobre o Java, acesse https://www.java.com/pt_BR/about/whatisjava.jsp.
[4]Para saber mais sobre o Spring Framework, acesse https://spring.io.
[5]Para saber mais sobre o ActionScript, acesse https://www.adobe.com/devnet/actionscript/learning.html.

MXML[6]. Ambas as linguagens fazem parte do *framework* Adobe Flex[7]. O objetivo do Adobe Flex é disponibilizar a interface do usuário baseada na tecnologia Adobe Flash[8].

Para o armazenamento dos dados do SMC, foi utilizado o banco de dados MySQL[9]. Já para a integração entre as camadas de *back-end* e *front-end*, foi utilizado o *middleware* BlazeDS[10]. A implantação e execução do SMC é realizada através de um *Servlet Container*. Para essa aplicação, foi escolhido o *Apache Tomcat*[11]. A Figura 42 apresenta todas as tecnologias citadas anteriormente. Na próxima seção, será discutido o motivo pela escolha do SMC como estudo de caso desta pesquisa.

5.1.4 Por que da escolha do SMC como estudo de caso?

O SMC foi escolhido como estudo de caso desta pesquisa devido a três razões: a primeira delas relaciona-se ao fato de que o SMC é uma **aplicação monolítica**; a segunda diz respeito a ele utilizar uma **linguagem de programação orientada a objetos** (Java); e a terceira é porque o SMC utiliza o **padrão de arquitetura MVC**. Esses três motivos são requisitos obrigatórios para que se possa aplicar a técnica de Monólise, conforme descrito na seção 4.1 deste trabalho.

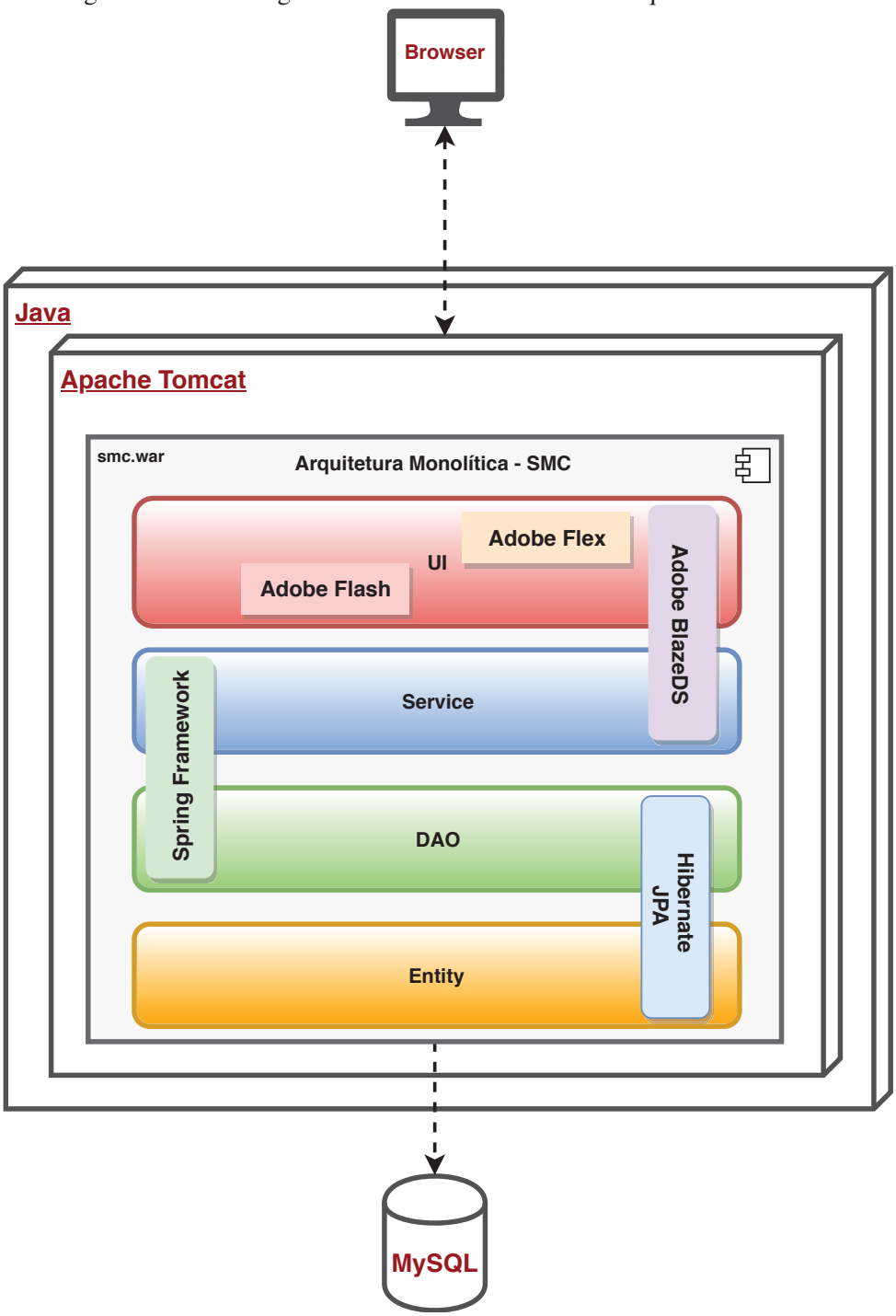
Além das razões citadas, o SMC apresenta também outros aspectos importantes que reforçam ainda mais a sua escolha, como o fato de ser uma aplicação bem documentada que segue boas práticas de projeto arquitetural e implementação (através do uso de padrões de projeto e boas práticas de programação). Trata-se de uma aplicação que possui funcionalidades e características de aplicações reais utilizadas na indústria.

Para realizar a decomposição manual do SMC, por exemplo, é necessário um especialista do modelo de domínio da aplicação. Nesse caso, como a aplicação foi desenvolvida pelo próprio autor desta pesquisa, isso será possível de ser atingido e, consequentemente, será possível comparar a técnica de Monólise com a decomposição realizada manualmente por um especialista da aplicação. Vale lembrar ainda que, para utilizar a técnica de Monólise, o usuário necessita conhecer minimamente a arquitetura da aplicação a ser decomposta em microsserviços, tendo em vista que a Monólise precisa de algumas informações referentes à arquitetura da aplicação. Sendo assim, como o autor é ao mesmo tempo usuário da técnica e também especialista da aplicação, isso possibilita executar tanto a decomposição manual como a decomposição gerada pela técnica.

Outro ponto que reforça a escolha do SMC é que ele é uma aplicação legada, desenvolvida em 2010. A arquitetura de software do SMC em conjunto com as tecnologias escolhidas prejudicam a manutenibilidade do software e impossibilitam a escalabilidade horizontal, o que não contribui para o uso de forma efetiva do ambiente em nuvem.

[6]Para saber mais sobre MXML, acesse https://www.adobe.com/devnet/flex/articles/fcf_mxml_actionscript.html.
[7]Para saber mais sobre Adobe Flex, acesse https://www.adobe.com/br/products/flex.html.
[8]Para saber mais sobre Adobe Flash, acesse https://www.adobe.com/devnet/swf.html.
[9]Para saber mais sobre MySQL, acesse https://www.mysql.com.
[10]Para saber mais sobre BlazeDS, acesse https://en.wikipedia.org/wiki/BlazeDS.
[11]Para saber mais sobre Apache Tomcat, acesse http://tomcat.apache.org.

Figura 42 – Tecnologias e frameworks utilizados na arquitetura do SMC



Fonte: Elaborado pelo autor

Tendo em vista que microsserviços é uma arquitetura nativa para nuvem e que esse estilo arquitetural necessita utilizar tecnologias mais modernas, habilitadas para uso em nuvem, o SMC é uma boa oportunidade para avaliar os impactos de uma modernização de uma aplicação legada para uma arquitetura de microsserviços.

5.2 Fase 2 - Aplicando a Decomposição na Aplicação-Alvo

Nesta seção, serão apresentados os passos necessários para a decomposição da aplicação-alvo escolhida e também será descrito como que alguns dos microsserviços recomendados foram implementados. Para tanto, essa seção foi organizada em três tópicos: o primeiro apresenta como será realizada a decomposição do SMC de forma manual; o segundo tópico mostra a utilização da técnica de Monólise aplicada no SMC e, por fim, o último tópico apresenta como os microsserviços gerados por cada técnica foram implementados.

5.2.1 Decomposição Manual

Para decompor manualmente o SMC, esta pesquisa seguiu as metodologias e os padrões de projeto que são considerados boas práticas no contexto da indústria. Para realizar a decomposição, foram utilizadas as recomendações descritas em (STINE, 2015) e (NEWMAN, 2015). Este trabalho segue essas recomendações por causa de duas razões: (1) pela utilização de um padrão de desenvolvimento no qual o projeto é orientado pelo domínio (DDD); e (2) pela utilização do princípio de responsabilidade única (SRP), que permite identificar os candidatos a microsserviços. Na sequência, serão informados os principais padrões e as metodologias aplicadas na decomposição do SMC.

Conforme descrito por Stine (2015), para decompor uma aplicação monolítica em microsserviços, são necessários pelo menos três passos. O primeiro deles é evitar que a aplicação monolítica cresça ainda mais em complexidade, ou seja, caso seja implementada uma nova funcionalidade, ela deverá ser desenvolvida fora da aplicação monolítica como um microsserviço. O segundo passo refere-se à integração da aplicação monolítica com a nova funcionalidade desenvolvida em microsserviço.

Segundo Stine (2015), qualquer integração necessária entre a aplicação monolítica e o novo microsserviço deve ser realizada através do padrão *Anti-Corruption Layer* (EVANS; SZPOTON, 2015). O objetivo dessa camada é evitar que o modelo de domínio do microsserviço possua um acoplamento com o modelo de domínio da aplicação monolítica. A camada de anticorrupção tem o papel de realizar a mediação e a transformação do modelo de dados e dos protocolos entre as aplicações.

Por fim, o terceiro passo está relacionado à forma de como extrair as funcionalidades da aplicação monolítica para microsserviços. Para Stine (2015), existem diversos fatores que levam à decomposição de uma aplicação como, por exemplo, permitir a entrega do software mais rápido ao mercado. Para conseguir atingir a decomposição das funcionalidades de uma aplicação monolítica para microsserviço, o autor recomenda utilizar o padrão de projeto *StranglerApplication* (FOWLER, 2004) e aplicar o conceito de *bounded context* (EVANS; SZPOTON, 2015), a fim de conseguir a melhor delimitação de contexto do modelo de domínio de cada funcionalidade.

Esta pesquisa também fez o uso das recomendações descritas por Newman (2015). Para

o autor, o uso de abordagens de desenvolvimento de software, como DDD (*Domain Driven Design*), ajudam na jornada de decomposição de aplicações monolíticas para microsserviços. Ele ressalta também a importância de encontrar a delimitação de contexto (*bounded context*) do modelo de domínio de negócio da aplicação a ser decomposta. O uso desse conceito possibilita a decomposição de uma aplicação monolítica ao redor da capacidade de negócio. Newman (2015) ainda cita que os microsserviços devem ser implementados com base no princípio da responsabilidade única (SRP) (MARTIN; MARTIN, 2006). Isso significa dizer que cada microsserviço gerado deverá conter somente funcionalidades que mudem pelo mesmo motivo, ou seja, caso uma funcionalidade dentro de um microsserviço mude por um motivo diferente das outras, ela deve ser decomposta em um novo microsserviço.

A fim de extrair as funcionalidades do SMC e obter os microsserviços, essa técnica aplicou todos os padrões e conceitos descritos, dando em ênfase ao uso do conceito de delimitação de contexto. Ao aplicar o conceito de contexto delimitado, desenvolvido por Evans e Szpoton (2015), foram identificados **sete contextos** dentro do SMC, o que gerou então **sete recomendações de microsserviços**, conforme pode ser observado na Figura 43. Cada retângulo cinza no mapa representa um contexto específico de negócio do SMC com as suas respectivas funcionalidades. Nesse mapa de contexto, é possível observar a integração entre os contextos, ou seja, um contexto necessita de informações de outro contexto, o que leva à necessidade de troca de mensagens e gerenciamento de dados distribuídos.

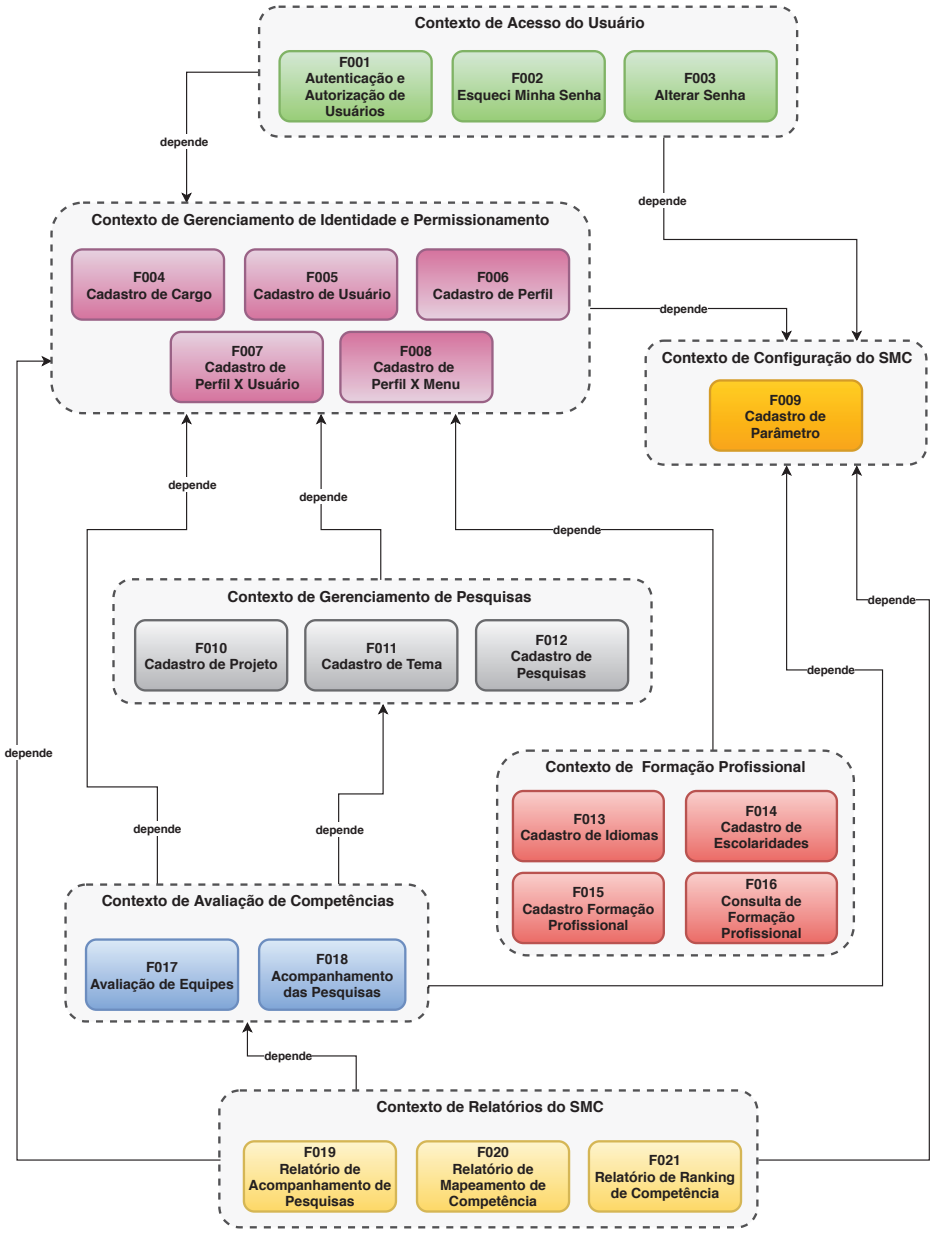
Ao aplicar o conceito de delimitação de contexto e também o uso do princípio de responsabilidade única, setes microsserviços foram gerados. No entanto, cabe destacar que o conhecimento e a experiência empírica do autor desta pesquisa sobre o SMC contribuiu para esse resultado. Tal afirmação se justifica tendo em vista que o autor foi quem desenvolveu o SMC, o que colaborou fortemente para identificação da melhor granularidade dos microsserviços gerados, conforme Figura 44.

5.2.2 Decomposição Utilizando Monólise

Para realizar a decomposição do SMC utilizando a técnica de Monólise, serão seguidas as etapas e os passos informados na Figura 25, da seção 4.4. Sendo assim, abaixo, serão apresentadas cada etapa com seus respectivos passos: **coleta de dados**, **processamento de dados**, **disponibilização dos resultados** e **implementação dos resultados**.

- **Etapa 1 - Coleta de Dados**
  - Passo 1.1 - Definir Aplicação-Alvo
    A aplicação-alvo escolhida foi o SMC.
  - Passo 1.2 - Configurar Instrumentação de Código
    Como o SMC é uma aplicação Java, será utilizado o *VisualVM* e o *BTrace* como ferramenta e *framework* de instrumentação de código.

Figura 43 – Mapa com as delimitações de contexto do SMC

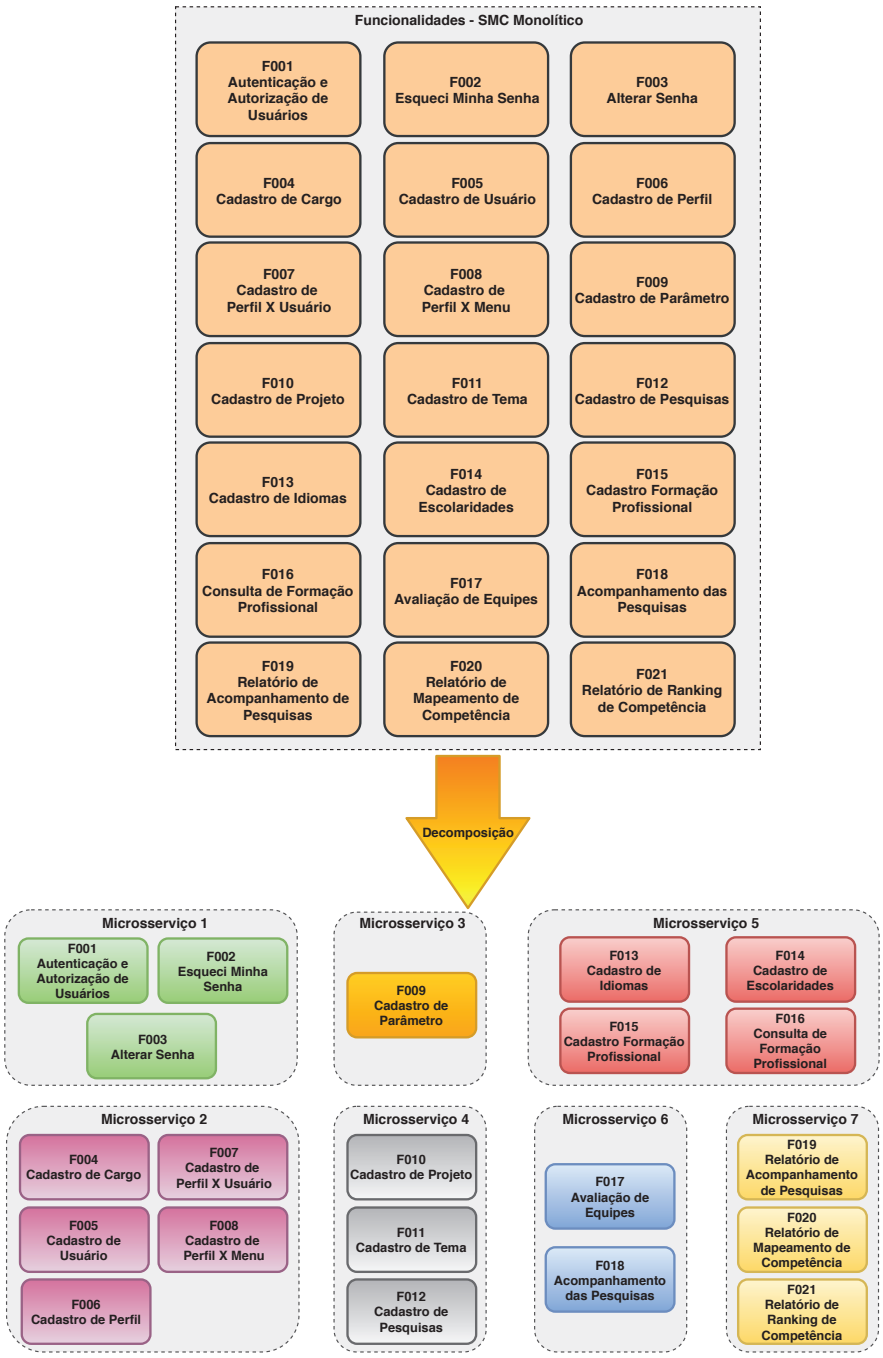


Fonte: Elaborado pelo autor

– Passo 1.3 - Mapear as Funcionalidades da Aplicação

Conforme exibido na Figura 40, o SMC apresenta 21 funcionalidades e, neste estudo, todas elas foram decompostas. Cada funcionalidade foi executada para gerar o arquivo de rastro de execução, conforme pode ser observado no Apêndice B. Esse arquivo faz parte dos parâmetros de entrada do algoritmo MonoBreak. No Apêndice A, é possível verificar um exemplo do arquivo de rastro de execução da funcionalidade de cadastro de formação profissional gerado a partir da sua utilização.

Figura 44 – Funcionalidades do SMC decompostas em microsserviços



Fonte: Elaborado pelo autor

– Passo 1.4 - Prover Informações da Arquitetura da Aplicação

Nesse passo, foi criado um arquivo JSON com as informações da arquitetura de software do SMC, conforme pode ser observado na Figura 47. É esse arquivo que contém os parâmetros de pesos por camada e o limite de decomposição (*threshold*), utilizados pelo algoritmo MonoBreak quando é realizada a decomposição das funcionalidades do SMC.

- **Etapa 2 - Processamento de Dados**

  - Passo 2.1 - Executar o Algoritmo MonoBreak

    Nesse passo, foram enviados os parâmetros de entrada para o MonoBreak; são eles: o arquivo de rastro de execução de cada funcionalidade (Apêndice A) e o arquivo com as informações da arquitetura do SMC (Figura 47).

- **Etapa 3 - Disponibilização dos Resultados**

  - Passo 3.1 - Gerar Recomendação de Microsserviços

    Nesse passo, o MonoBreak apresenta a recomendação dos microsserviços com base nas 21 funcionalidades selecionadas. O resultado da recomendação pode ser observado no Apêndice C.

- **Etapa 4 - Implementação dos Resultados**

  - Passo 4.1 - Implementação de Microsserviços

    Conforme mencionado neste capítulo, para a implementação dos microsserviços tanto utilizando a técnica de Monólise como a técnica manual, serão utilizadas as mesmas tecnologias e os mesmos *frameworks*. Sendo assim, foi desenvolvida uma seção 5.2.3 que tratará especificamente do aspecto de implementação dos microsserviços nesta avaliação.

5.2.3 Implementação dos Microsserviços

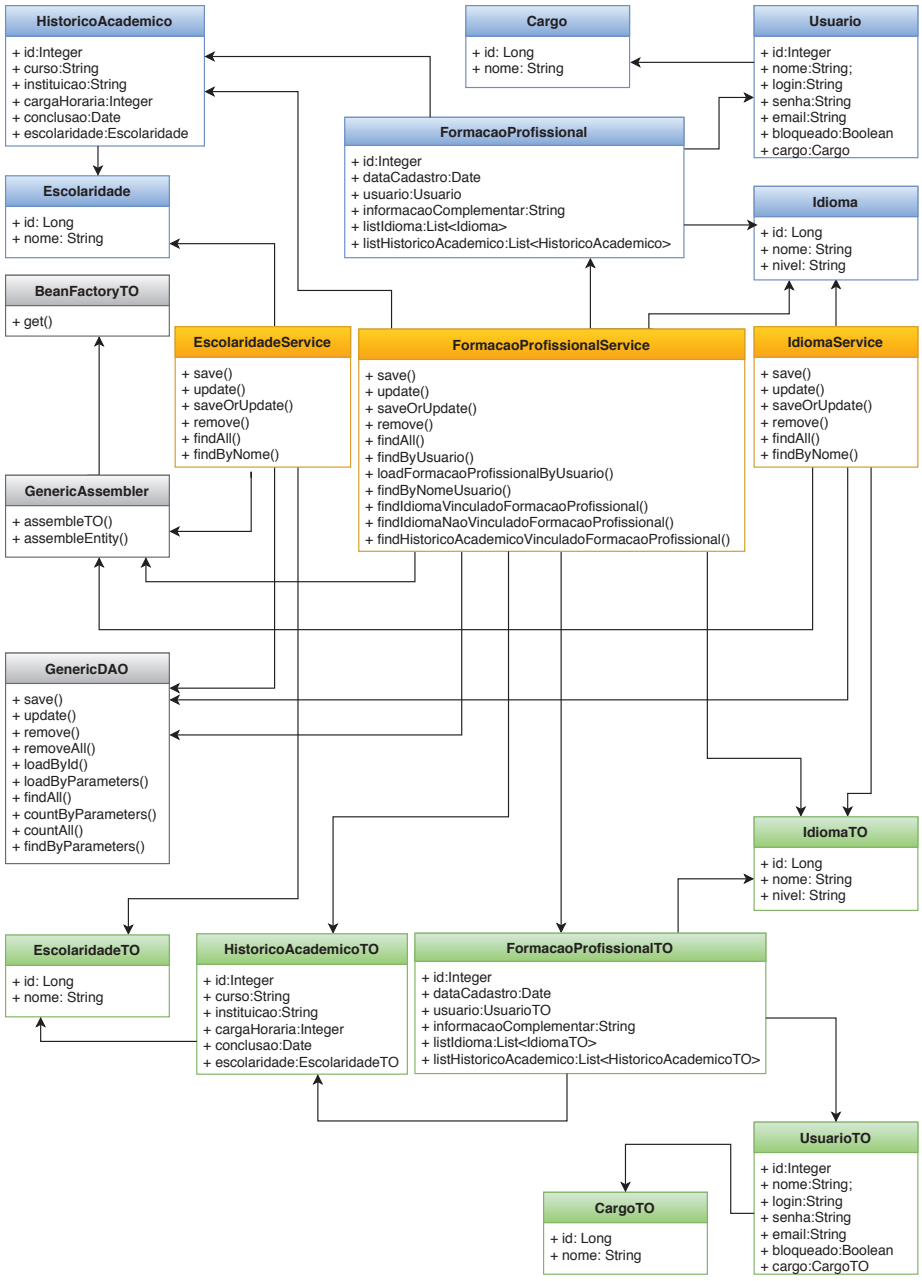
Todos os microsserviços recomendados foram implementados com *Java* e *Spring Boot*. Havendo a necessidade de uso de tecnologias de banco de dados e mensageria, os microsserviços poderão utilizar o *MySQL* ou o *MongoDB* como base de dados e o *RabbitMQ* como *broker* de troca de mensagens. Todos os microsserviços executam na PaaS *Pivotal Cloud Foundry* e utilizam as tecnologias *cloud-native* oferecidas pela própria plataforma, como o *Spring Cloud Services*[12]. A implementação dos microsserviços seguiu as boas práticas definidas pela metodologia *The Twelve-Factor App* (WIGGINS, 2011). Essa metodologia possui 12 práticas que permitem que uma aplicação seja considerada apta para rodar na nuvem, ou seja, é considerada uma *cloud-native application*.

Com base nas definições de tecnologias e nas boas práticas, o primeiro passo executado para implementar os microsserviços foi migrar as classes da aplicação monolítica SMC para a nova aplicação baseada em microsserviço. Como demonstração do processo, foi escolhida a funcionalidade de cadastro de formação profissional (F015), conforme descrita na Figura 44.

[12]Para saber mais sobre Spring Cloud Services, acesse https://docs.pivotal.io/spring-cloud-services/1-5/common/.

Para representar as classes e os métodos que necessitam ser migrados, foi criado um diagrama de classe da funcionalidade cadastro de formação profissional, conforme pode ser observado na Figura 45. Todas as classes e os métodos gerados pelo algoritmo MonoBreak, exibido no apêndice A, estão representados no diagrama de classe.

Figura 45 – Classes a serem migradas do cadastro de formação profissional



Fonte: Elaborado pelo autor

Após migradas as classes para a nova aplicação baseada em microsserviço, é necessário implementar as boas práticas descritas pela metodologia *The Twelve-Factor App*. Conforme pode ser observado na Figura 46, o cadastro de formação profissional foi desenvolvido em *Java*, uti-

lizando o *framework SpringBoot* (caixa laranja). Esse microsserviço expõe suas capacidades de negócio através do protocolo HTTP, utilizando o conceito de *REST* (FIELDING; TAYLOR, 2000). Com a disponibilização de uma API REST, outros microsserviços e interface de usuário poderão consultar (*GET*), criar (*POST*), atualizar (*PUT*) e deletar (*DELETE*) as formações profissionais. Os dados da formação profissional serão armazenados num banco de dados orientado a documentos, o *MongoDB*.

Já os outros componentes destacados em verde e vermelho na Figura 46 são serviços utilitários ao microsserviço de formação profissional. Por exemplo, o *Eureka Server*[13] é um registro de serviço que permite que os microsserviços registrem seu endereço IP e DNS. Esse tipo de solução possibilita a descoberta de outros serviços através de um identificador único, dispensando, assim, que a aplicação conheça o endereço IP ou o DNS da aplicação requisitada. Essa solução é importante numa arquitetura de microsserviços, tendo em vista que os microsserviços podem realizar o processo de *scale-in* e *scale-out*, conforme a demanda de acesso a aplicação, e também evita que os microsserviços contenham dados *hard-code* de endereço de acesso de outros serviços.

O *Spring Cloud Config*[14] é outro componente da arquitetura, que tem por objetivo armazenar de forma centralizada dados de configuração dos microsserviços. Por exemplo, dados de acesso ao banco de dados do microsserviço, ao invés de ficarem armazenados dentro do microsserviço, ficariam armazenados dentro dessa solução. Isso evita problemas de *restart* da aplicação, quando uma configuração for modificada, e também simplifica o gerenciamento de configurações por ambientes de desenvolvimento, teste e produção. O *Hystrix Circuit Breaker*[15] é uma biblioteca para tolerância a falhas em sistemas distribuídos. Seu papel nessa arquitetura é garantir que, caso um microsserviço esteja fora do ar ou o tempo de resposta exceda seu tempo limite, isso não gera um falha em cascata dentro do ambiente. Com essa biblioteca, é possível definir programaticamente o que deve ser realizado quando uma aplicação não está no ar, ou seja, é possível definir um *fallback*.

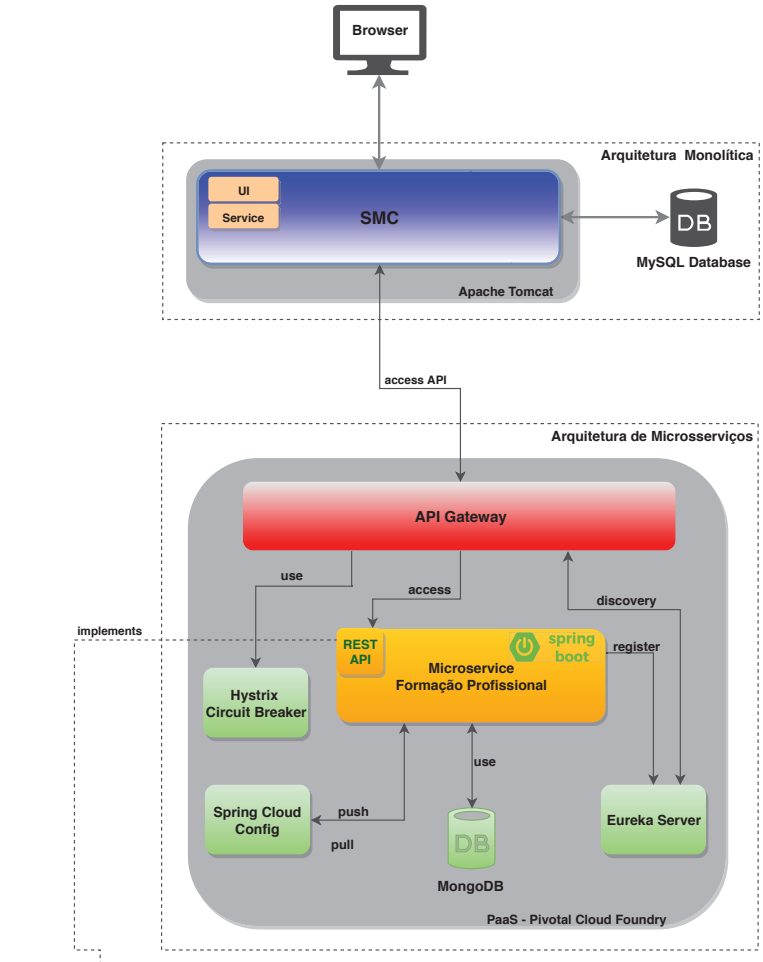
Por fim, o componente *API Gateway* (RICHARDSON, 2016) nessa arquitetura tem como objetivo ser a "porta de entrada"para o acesso aos microsserviços. Nele, será implementado o roteamento de acesso aos microsserviços, o controle de segurança, o *cache* de dados e a composição de microsserviços. Numa arquitetura de microsserviços, o *API Gateway* é, normalmente, o componente que expõe as APIs dos microsserviços, permite realizar a composição de serviços assim como a tradução de diferentes protocolos dos microsserviços.

5.3 Fase 3 - Avaliação da Decomposição

Nesta seção, serão apresentados os critérios utilizados para realização da avaliação da técnica proposta. Para tanto, essa seção foi organizada em três tópicos: o primeiro apresenta o

[13]Para saber mais sobre Eureka Server, acesse https://github.com/spring-cloud/spring-cloud-netflix.
[14]Para saber mais sobre Spring Cloud Config, acesse https://cloud.spring.io/spring-cloud-config.
[15]Para saber mais sobre Hystrix Circuit Breaker, acesse https://github.com/Netflix/Hystrix.

Figura 46 – Arquitetura de microsserviço do cadastro de formação profissional



Rest API - Formação Profissional	
Recurso	**Verbo HTTP**
/formacao_profissional/	POST
/formacao_profissional/{id}	GET
/formacao_profissional/{id}	PUT
/formacao_profissional/{id}	DELETE
/formacao_profissional/search?usuario=xxx	GET

Fonte: Elaborado pelo autor

método de avaliação utilizado para avaliar a técnica proposta; o segundo apresenta os cenários propostos que serão utilizados na avaliação e, por fim, o último apresenta as métricas utilizadas na avaliação.

5.3.1 Método de Avaliação

Para realizar a avaliação, foram criados dois tipos de decomposição, a decomposição ideal (DI) e a decomposição gerada pela técnica (DG). A DI refere-se ao uso da técnica de decomposição manual, ou seja, é quando um especialista no domínio da aplicação executa a decomposição manualmente, seguindo as melhores práticas do mercado. Já a DG refere-se ao uso da técnica de Monólise proposta por esta pesquisa.

O objetivo dessa avaliação é verificar a efetividade da decomposição em microsserviços gerada pela técnica de Monólise, quando comparada à técnica manual. Será validado o quanto a decomposição da técnica de Monólise gerada é semelhante à decomposição ideal gerada pela técnica manual. Para executar a avaliação, foram propostas duas métricas referentes a projeto de software (WUST, 2005) e duas métricas referentes à quantidade de microsserviços e de funcionalidades recomendadas.

Com essas quatro métricas coletadas, será possível mensurar os resultados da recomendação de microsserviços entre as técnicas manual e de Monólise. Para realizar esse comparativo entre as técnicas, serão calculadas as medidas de precisão (*precision*), revocação (*recall*) e média harmônica (*f-measure*) para verificar o quanto a técnica de Monólise se assemelha à decomposição ideal.

5.3.2 Cenários de Avaliação

Com o objetivo de avaliar a equivalência dos resultados gerados pelas técnicas, foram criados oito cenários de avaliação. Esses oito cenários foram construídos com base no resultado da delimitação de contexto realizada pela técnica manual, conforme apresentado na Figura 43. O objetivo é verificar se a técnica é capaz de indicar a mesma recomendação ideal de microsserviços e funcionalidades, conforme apresentado pela técnica manual na Figura 44.

- **Cenário 1 - Contexto de Acesso do Usuário**

  Esse cenário tem por objetivo recomendar as funcionalidades do contexto de acesso do usuários. As funcionalidades que serão decompostas são: **F001 – Autenticação e Autorização de Usuários**; **F002 - Esqueci Minha Senha** e **F003 - Alterar Senha**. Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha as três funcionalidades.

- **Cenário 2 - Contexto de Gerenciamento de Identidade e Permissionamento**

  Esse cenário tem por objetivo recomendar as funcionalidades do contexto de gerenciamento de identidade e permissionamento. As funcionalidades que serão decompostas são: **F004 - Cadastro de Cargo**; **F005 - Cadastro de Usuário**; **F006 - Cadastro de Perfil**; **F007 - Cadastro de Perfil x Usuário** e **F008 - Cadastro de Perfil x Menu**.

Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha as cinco funcionalidades.

- **Cenário 3 - Contexto de Configuração do SMC**

  Esse cenário tem por objetivo recomendar a funcionalidade do contexto de configuração do SMC. A funcionalidade que será decomposta é: **F009 - Cadastro de Parâmetro**. Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha uma funcionalidade.

- **Cenário 4 - Contexto de Gerenciamento de Pesquisas**

  Esse cenário tem por objetivo recomendar as funcionalidades do contexto de gerenciamento de pesquisa. As funcionalidades que serão decompostas são: **F010 - Cadastro de Projeto**; **F011 - Cadastro de Tema** e **F012 - Cadastro de Pesquisas**. Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha as três funcionalidades.

- **Cenário 5 - Contexto de Formação Profissional**

  Esse cenário tem por objetivo recomendar as funcionalidades do contexto de formação profissional. As funcionalidades que serão decompostas são: **F013 - Cadastro de Idiomas**; **F014 - Cadastro de Escolaridades**; **F015 - Cadastro de Formação Profissional** e **F016 - Consulta de Formação Profissional**. Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha as quatro funcionalidades.

- **Cenário 6 - Contexto de Avaliação de Competência**

  Esse cenário tem por objetivo recomendar as funcionalidades do contexto de avaliação de competência. As funcionalidades que serão decompostas são: **F017 - Avaliação de Equipes** e **F018 - Acompanhamento de Pesquisas**. Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha as duas funcionalidades.

- **Cenário 7 - Contexto de Relatórios do SMC**

  Esse cenário tem por objetivo recomendar as funcionalidades do contexto de relatórios do SMC. As funcionalidades que serão decompostas são: **F019 - Relatório de Acompanhamento de Pesquisa**; **F020 - Relatório de Mapeamento de Competência** e **F021 - Relatório de Ranking de Competências**. Espera-se que para esse cenário seja recomendado **um microsserviço** que contenha as três funcionalidades.

- **Cenário 8 - Todos os Contextos**

  Esse cenário tem por objetivo recomendar todos os contextos e suas funcionalidades de uma única vez, ou seja, será recomendada a decomposição das 21 funcionalidades do SMC de uma única vez. O objetivo é validar como que a técnica recomenda os microsserviços com um número maior de funcionalidades. Espera-se para esse cenário que sejam

recomendados **7 microsserviços** e que contenha as **21 funcionalidades** distribuídas para cada contexto.

5.3.3 Métricas Utilizadas

Para mensurar o quanto a técnica proposta por esta pesquisa é semelhante à técnica de decomposição manual, foram propostas quatro métricas. As métricas **NumClasses** e **NumMetodos** referem-se a métricas que verificam a quantidade de classes e métodos que existem nas funcionalidades a serem decompostas. Já as métricas **NumMicrosserviços** e **NumFuncionalidades** referem-se respectivamente ao número de microsserviços recomendados e ao número de funcionalidades inseridas num determinado cenário de avaliação. A Tabela 3 apresenta um resumo de cada métrica que será utilizada pela avaliação.

Tabela 3 – Tabela com as métricas utilizadas para a avaliação da técnica proposta

Métrica	**Descrição**
NumClasses	Total de classes das funcionalidades a serem decompostas
NumMetodos	Total de métodos das classes das funcionalidades a serem decompostas
NumMicrosserviços	Total de microsserviços recomendados
NumFuncionalidades	Total de funcionalidades parte dos microsserviços recomendados

Para realizar a análise dos resultados gerados pelas técnicas, ou seja, verificar a semelhança entre os resultados da decomposição em microsserviços, serão utilizadas as medidas de precisão (*precision*), revocação (*recall*) e média harmônica (*f-measure*). A fórmula da precisão verifica os valores positivos corretos previstos na recomendação de microsserviços gerados pela técnica manual dividido pelo número de todos os resultados. A revocação calcula os resultados que são corretos divididos pelo número de resultados que deveriam ser apresentados. Por fim, a média harmônica é a medida que combina a precisão e a revocação (FRAKES; BAEZA-YATES, 1992). Abaixo, serão apresentadas as fórmulas de cada medida.

$$precision = \frac{|DG \cap DI|}{|DG|} \tag{5.1}$$

$$recall = \frac{|DG \cap DI|}{|DI|} \tag{5.2}$$

$$f - measure = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{5.3}$$

5.4 Fase 4 - Análise da Precisão da Decomposição

Nesta seção, serão discutidos os resultados da decomposição gerada pela técnica de Monólise (DG) e os gerados pela técnica manual, também conhecida como decomposição ideal (DI).

Para tanto, essa seção está organizada em dois tópicos: o primeiro apresenta o resultado de cada cenário e o segundo discute os resultados apresentados.

5.4.1 Resultados

Abaixo, serão apresentadas as tabelas com os resultados da decomposição gerada pela técnica de Monólise (DG) e os resultados da decomposição ideal (DI) gerada pela técnica manual.

Para coletar os dados referentes às métrica de **NumClasses** e **NumMetodos** para decomposição ideal (DI), foi utilizada a ferramenta de análise de código *SonarQube*[16]. O SonarQube oferece diversas métricas a nível de código fonte, inclusive informações como quantidade de classes e métodos que uma aplicação possui.

Para coletar as métrica de **NumClasses** e **NumMetodos** para decomposição gerada pela técnica, foi implementado um algoritmo que exporta as estruturas de dados do algoritmo MonoBreak para arquivos do tipo CSV. O algoritmo utilizado está disponível no Apêndice D. As métricas **NumMicrosserviços** e **NumFuncionalidades** para decomposição ideal foram obtidas manualmente, já para técnica gerada, foi utilizado o algoritmo apresentado no Apêndice D.

Antes de apresentar os dados, é importante ressaltar que a técnica de Monólise foi executada com a configuração de ***threshold* de decomposição no valor 85% de similaridade**. Esse percentual de limite de decomposição foi o melhor resultado encontrado próximo da decomposição ideal (DI). A Figura 47 apresenta o arquivo de configuração utilizado pelo algoritmo MonoBreak ao realizar a decomposição. Abaixo, serão apresentadas as tabelas com os resultados de cada cenário.

Figura 47 – Arquivo JSON de configuração da arquitetura do SMC

```
{
 "applicationName": "SMC",
 "modelPackageName": "br.com.sape.model.entity",
 "daoPackageName": "br.com.sape.model.dao",
 "servicePackageName": "br.com.sape.business.service",
 "dtoPackageName":"br.com.sape.model.dto",
 "modelWeight": 1.0,
 "daoWeight": 0.5,
 "serviceWeight": 1.0,
 "dtoWeight": 0.3,
 "decompositionThreshold": 85
}
```

Fonte: Elaborado pelo autor

[16]Para saber mais sobre SonarQube, acesse https://www.sonarqube.org.

Tabela 4 – Tabela com os resultados das métricas do Cenário 1

Cenário 1						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	22	22	22	1	1	1
NumMetodos	150	150	150	1	1	1
NumMicrosserviços	1	1	1	1	1	1
NumFuncionalidades	3	3	1	1	1	1

Tabela 5 – Tabela com os resultados das métricas do Cenário 2

Cenário 2						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	23	23	23	1	1	1
NumMetodos	158	158	158	1	1	1
NumMicrosserviços	1	1	1	1	1	1
NumFuncionalidades	5	5	1	1	1	1

Tabela 6 – Tabela com os resultados das métricas do Cenário 3

Cenário 3						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	6	6	6	1	1	1
NumMetodos	28	28	28	1	1	1
NumMicrosserviços	1	1	1	1	1	1
NumFuncionalidades	1	1	1	1	1	1

Tabela 7 – Tabela com os resultados das métricas do Cenário 4

Cenário 4						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	50	29	21	0,42	0,72	0,53
NumMetodos	382	250	132	0,34	0,52	0,42
NumMicrosserviços	3	1	0	0	0	0
NumFuncionalidades	3	3	3	1	1	1

Tabela 8 – Tabela com os resultados das métricas do Cenário 5

Cenário 5						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	18	18	18	1	1	1
NumMetodos	151	151	151	1	1	1
NumMicrosserviços	1	1	1	1	1	1
NumFuncionalidades	4	4	4	1	1	1

5.4.2 Discussão dos Resultados

Nesta seção, serão discutidos os resultados apresentados na seção 5.4.1. Na sequência, serão discutidos os oito cenários avaliados a fim de verificar a precisão da decomposição gerada pela

Tabela 9 – Tabela com os resultados das métricas do Cenário 6

Cenário 6						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	25	25	25	1	1	1
NumMetodos	227	227	227	1	1	1
NumMicrosserviços	1	1	1	1	1	1
NumFuncionalidades	2	2	2	1	1	1

Tabela 10 – Tabela com os resultados das métricas do Cenário 7

Cenário 7						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	15	15	15	1	1	1
NumMetodos	107	107	107	1	1	1
NumMicrosserviços	1	1	1	1	1	1
NumFuncionalidades	3	3	3	1	1	1

Tabela 11 – Tabela com os resultados das métricas do Cenário 8

Cenário 8						
Métrica	**DG**	**DI**	**DG ∩ DI**	***Precision***	***Recall***	***f-measure***
NumClasses	71	60	60	0,84	1	0,92
NumMetodos	591	507	507	0,85	1	0,92
NumMicrosserviços	8	7	5	0,62	0.71	0,67
NumFuncionalidades	21	21	21	1	1	1

técnica de Monólise (DG) *versus* à decomposição ideal (DI).

Com base nos dados referentes ao cenário 1 (Tabela 4), a técnica de Monólise apresentou **100% de precisão** ao realizar a decomposição das funcionalidades. A técnica proposta indicou o mesmo número de microsserviço, funcionalidades, classes e métodos recomendados pela técnica manual. Essa mesma precisão também ocorreu no cenário 2 (Tabela 5); no cenário 3 (Tabela 6); no cenário 5 (Tabela 8); no cenário 6 (Tabela 9) e no cenário 7 (Tabela 10).

Assim, para os cenários referentes aos contextos de acesso do usuário (**cenário 1**), contexto de gerenciamento de identidade e permissionamento (**cenário 2**), contexto de configuração do SMC (**cenário3**), contexto de formação profissional (**cenário 5**), contexto de avaliação de competência (**cenário 6**) e contexto de relatórios do SMC (**cenário 7**), a técnica de Monólise, através do algoritmo MonoBreak, conseguiu decompor o SMC com a mesma configuração de microsserviços e funcionalidades, conforme descrito na Figura 44.

Dos oito cenários propostos, a técnica de Monólise não atingiu os resultados de decomposição ideal somente no **cenário 4** e no **cenário 8**. No cenário 4, ela teve o pior desempenho, conforme pode ser observado na Tabela 7. A técnica de Monólise recomendou **três microsserviços**, quando o esperado era apenas **um microsserviço** como resultado da decomposição ideal. No cenário de contexto de gerenciamento de pesquisa, esperava-se que a decomposição gerada pela técnica recomendasse **um microsserviço com três funcionalidades** (F010 cadastro de

projeto; F011 cadastro de tema e F012 cadastro de pesquisa), conforme descrito na Figura 44. Porém, a Monólise recomendou **um microsserviço para cada funcionalidade**, o que resultou em **três microsserviços**.

Tabela 12 – Tabela de similaridade das 21 funcionalidades do SMC

	F001	**F002**	**F003**	**F004**	**F005**	**F006**	**F007**	**F008**	**F009**	**F010**	**F011**	**F012**	**F013**	**F014**	**F015**	**F016**	**F017**	**F018**	**F019**	**F020**	**F021**
F001	-	17,80	17,80	5,48	21,54	1,05	18,74	6,73	1,05	30,91	5,48	46,06	1,05	1,05	12,91	12,91	46,06	36,74	3,99	46,06	46,06
F002	32,34	-	32,34	2,72	81,70	0,45	18,12	0,45	6,79	18,12	2,72	10,87	0,45	0,45	10,87	10,87	27,17	27,17	0,45	10,87	10,87
F003	100	100	-	8,40	68,91	1,40	56,02	1,40	1,40	56,02	8,40	33,61	1,40	1,40	33,61	33,61	33,61	33,61	1,40	33,61	33,61
F004	57,29	15,62	15,62	-	100	10,94	57,29	10,94	10,94	57,29	100	57,29	10,94	10,94	57,29	57,29	57,29	57,29	2,60	57,29	57,29
F005	19,35	40,41	11,02	8,60	-	8,60	35,48	8,60	6,05	16,85	8,60	11,60	0,94	0,94	11,60	11,60	22,98	22,98	0,90	11,60	11,60
F006	10,94	2,60	2,60	10,94	100	-	100	100	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	0,00	10,94	10,94
F007	40,69	21,65	21,65	11,90	85,71	20,78	-	20,78	2,27	40,69	11,90	28,03	2,27	2,27	28,03	28,03	28,03	28,03	2,16	28,03	28,03
F008	21,77	0,81	0,81	3,39	30,97	30,97	30,97	-	3,39	3,39	3,39	3,39	3,39	3,39	3,39	3,39	3,39	3,39	0,00	3,39	3,39
F009	10,94	39,06	2,60	10,94	70,31	10,94	10,94	10,94	-	10,94	10,94	10,94	10,94	10,94	10,94	10,94	70,31	70,31	0,00	10,94	10,94
F010	78,28	25,25	25,25	13,89	47,47	2,65	47,47	2,65	2,65	-	13,89	78,28	2,65	2,65	32,70	32,70	59,09	59,09	5,68	78,28	78,28
F011	8,10	2,21	2,21	14,14	14,14	1,55	8,10	1,55	1,55	8,10	-	58,32	1,55	1,55	8,10	8,10	34,46	34,46	0,37	8,10	8,10
F012	19,25	2,50	2,50	2,29	5,40	0,44	5,40	0,44	0,44	12,92	16,50	-	0,44	0,44	5,40	5,40	66,50	59,17	1,67	23,56	23,56
F013	10,94	2,60	2,60	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	-	10,94	100	57,29	10,94	10,94	0,00	10,94	10,94
F014	10,94	2,60	2,60	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	10,94	-	100	100	10,94	10,94	0,00	10,94	10,94
F015	11,33	5,25	5,25	4,81	11,33	0,92	11,33	0,92	0,92	11,33	4,81	11,33	8,40	8,40	-	87,01	11,33	11,33	0,87	11,33	11,33
F016	13,02	6,03	6,03	5,53	13,02	1,06	13,02	1,06	1,06	13,02	5,53	13,02	5,53	9,65	100	-	13,02	13,02	1,01	13,02	13,02
F017	20,31	6,59	2,64	2,42	11,27	0,46	5,69	0,46	2,97	10,29	10,29	70,15	0,46	0,46	5,69	5,69	-	90,73	1,76	20,31	20,31
F018	17,85	7,27	2,91	2,66	12,43	0,51	6,27	0,51	3,27	11,34	11,34	68,80	0,51	0,51	6,27	6,27	100	-	1,94	17,85	17,85
F019	43,01	2,69	2,69	2,69	10,75	0,00	10,75	0,00	0,00	24,19	2,69	43,01	0,00	0,00	10,75	10,75	43,01	43,01	-	100	100
F020	62,86	8,16	8,16	7,48	17,62	1,43	17,62	1,43	1,43	42,18	7,48	76,94	1,43	1,43	17,62	17,62	62,86	50,14	12,65	-	100
F021	62,86	8,16	8,16	7,48	17,62	1,43	17,62	1,43	1,43	42,18	7,48	76,94	1,43	1,43	17,62	17,62	62,86	50,14	12,65	100	-

O motivo da geração de três microsserviços no cenário 4 deve-se ao fato de as funcionalidades não terem um percentual de similaridade igual ou maior a **85%**, ou seja, elas estavam abaixo do *threshold* de decomposição configurado para ser utilizado pelo MonoBreak. A Tabela 12 apresenta a tabela de similaridade gerada e utilizada pelo MonoBreak para decompor as 21 funcionalidades do SMC nesta avaliação.

É possível observar que, nas linhas e colunas da tabela de similaridade apresentadas na Tabela 12, referentes às funcionalidades **F010**; **F011** e **F012**, todas estão com percentual de similaridade abaixo ao *threshold* de decomposição configurado para ser utilizado pelo MonoBreak, que é de **85%**. Observa-se que o maior percentual de similaridade encontrado para esse cenário foi de **78,28%**, entre a funcionalidade F010 (linha) com a funcionalidade F012 (coluna). Esse valor apresentado não foi suficiente para fazer com que o MonoBreak agrupasse essas funcionalidades em um único microsserviço.

Outro impacto gerado pela quebra em três microsserviços referente ao cenário 4 é que o número de classes e de métodos recomendados pela técnica foi maior do que o recomendado pela decomposição ideal, conforme Tabela 7. Esse aumento no número de classes e de métodos ocorreu, porque, como foram gerados três microsserviços, as classes e os métodos transversais das

funcionalidades desse cenário tiveram que ser replicados entre os microsserviços, o que acarretou nesse aumento. Caso fosse gerado somente um microsserviço, as classes não precisariam ser replicadas. Em resumo, a técnica de Monólise no cenário 4 não apresentou uma precisão, isso ocorreu devido à recomendação de três microsserviços serem completamente diferentes do único microsserviço recomendado na decomposição ideal.

Um outro cenário que não apresentou uma precisão de 100% foi o cenário 8, conforme pode ser observado na Tabela 11. Esse cenário apresentou um melhor resultado quando comparado com o cenário 4 em termos de precisão. Na decomposição ideal, foram recomendados **7 microsserviços para as 21 funcionalidades** escolhidas, conforme apresentado na Tabela 11, porém a técnica de Monólise acabou gerando **8 microsserviços**, um a mais do recomendado.

Essa diferença na recomendação dos microsserviços no cenário 8 ocorreu devido às funcionalidades do cenário 4 (Contexto de Gerenciamento de Pesquisas). Quando o MonoBreak analisou as funcionalidades **F010**, **F011** e **F012**, num contexto com todas as funcionalidades do SMC juntas, o algoritmo recomendou a separação dessas funcionalidades em microsserviços diferentes, conforme havia ocorrido no cenário 4, porém com uma pequena diferença. Quando o MonoBreak analisou a funcionalidade de cadastro de cargo (F004), o algoritmo identificou que existia uma similaridade de 100% com a funcionalidade de cadastro de tema (F011), conforme pode ser observado na tabela de similaridade apresentada na Tabela 12. A consequência dessa similaridade alta fez com que a funcionalidade F011 fosse agrupada juntamente com a funcionalidade F004, ou seja, a funcionalidade F011 foi movida para o microsserviço gerado para o contexto de gerenciamento de identidade e permissionamento, conforme descrito na Figura 44.

A movimentação da funcionalidade de cadastro de tema (F011) para o contexto de gerenciamento de identidade e permissionamento fez com que esse contexto ficasse com as seguintes funcionalidades **F004**, **F005**, **F006**, **F007**, **F008** e a nova funcionalidade **F011**. Após essa movimentação da funcionalidade F011, o MonoBreak tomou a mesma decisão descrita no cenário 4, ou seja, ele recomendou um microsserviço para a funcionalidade **F010** e um para a funcionalidade **F012**, que são respectivamente: cadastro de projeto e cadastro de pesquisa. Em resumo, o MonoBreak moveu a funcionalidade de cadastro de tema (F011) para o contexto de gerenciamento de identidade e permissionamento, criou um novo microsserviço para cadastro de projeto e manteve a funcionalidade de cadastro de pesquisa no contexto de gerenciamento de pesquisa. Esse foi o motivo pelo qual a técnica de Monólise gerou 8 microsserviços ao invés de 7 recomendados pela técnica manual, conforme pode ser observado na Figura 48. No final da decomposição gerada pela técnica de Monólise, 5 dos microsserviços recomendados são iguais aos 7 recomendados pela decomposição ideal. Outro resultado importante de destacar do cenário 8 ainda é que, devido ao novo microsserviço gerado de cadastro de projeto, houve um aumento no número de classes e métodos, quando comparado com a decomposição ideal, a diferença ocorreu novamente devido à replicação de classe e métodos transversais às funcionalidades.

Os resultados apresentados nos cenários 1, 2, 3, 5, 6 e 7 demonstraram semelhanças en-

tre realizar a decomposição manualmente e realizá-la utilizando a técnica de Monólise. Já no cenário 4, os resultados demonstraram diferenças significativas, o que exigiu a reflexão sobre quais seriam os motivos disso ocorrer. Ao analisar os baixos valores de similaridade das funcionalidades **F010**, **F011** e **F012** do **cenário 4**, buscou-se descobrir qual o motivo que levou a decomposição considerada ideal (manual) a agrupar essas três funcionalidades, já que, segundo o princípio de responsabilidade única (MARTIN; MARTIN, 2006) apresentado nesta pesquisa, deve-se seguir a máxima:"reúna as coisas que mudam pelo mesmo motivo e separe as coisas que mudam por motivos diferentes".

Se aplicarmos o princípio da responsabilidade única nessas funcionalidades, podemos chegar à conclusão de que elas deveriam estar separadas, pois as funcionalidades F010, F011 e F012 apresentam baixa similaridade entre elas, ou seja, não mudariam pelos mesmos motivos, tendo em vista que não são similares. Então, se aplicarmos esse princípio estritamente, é possível concluir que o MonoBreak gerou a recomendação da forma correta, pois ele separou as funcionalidades que mudariam por motivos diferentes, ou seja, por baixa similaridade.

Diante dessa conclusão, fica a seguinte questão: o quanto o conhecimento empírico de quem está decompondo a aplicação monolítica influencia no momento da decomposição? O que prevalece nesse momento: seguir princípios e boas práticas convencionadas ou aplicar a decomposição conforme a necessidade do negócio da aplicação? A partir desses questionamentos, é possível observar que existem semelhanças e diferenças ao decompor uma aplicação monolítica através de técnicas manuais e semiautomáticas. Ao aplicar uma decomposição manualmente, o usuário utiliza o conhecimento empírico que possui sobre a aplicação para realizar a decomposição, já quando a técnica é executada com um algoritmo, esse conhecimento empírico não é possível de ser passado para a lógica de decisão do algoritmo, o que acaba gerando resultados em alguns cenários semelhantes e, em outros, diferentes. Por isso, decompor aplicações monolíticas em microsserviços é ainda um tema de pesquisa tanto no contexto da indústria quanto no da academia, já que existem diversas situações que precisam ser analisadas antes de modernizar as aplicações para uma arquitetura baseada em microsserviços.

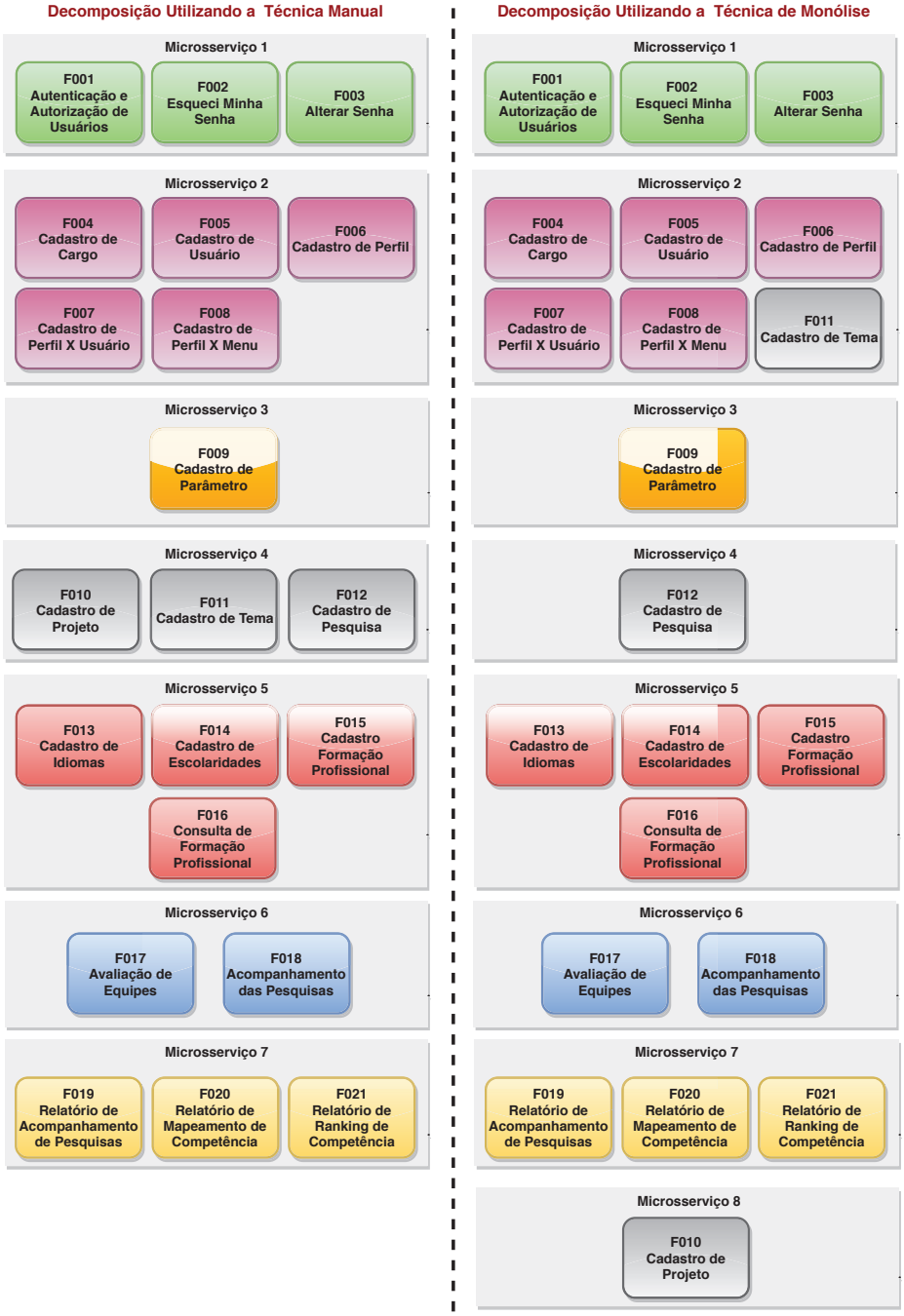
Com base nos cenários e nos resultados apresentados, é possível concluir que a técnica de Monólise tem potencialidade para realizar a decomposição de aplicações monolíticas em microsserviços de forma semiautomática. No caso deste estudo, mesmo com as diferenças apresentadas no cenário 4 e no cenário 8, a técnica conseguiu atingir uma precisão de **67%** no cenário mais completo da avaliação, o 8. Ainda assim, cabe destacar a potencialidade de continuidade na investigação sobre técnicas semiautomáticas para decomposição de aplicações monolíticas em microsserviços, tendo em vista que, para alcançar uma recomendação de microsserviços, são necessários diversos conhecimentos não só de princípios e técnicas mas também é preciso levar em consideração o conhecimento empírico sobre a aplicação.

Com base nas avaliações realizadas, é possível concluir que a primeira questão de pesquisa deste trabalho foi devidamente respondida com o uso da técnica de mónolise, pois, através dela, foi possível decompor uma aplicação monolítica em microsserviços de forma semiautomática.

A segunda questão de pesquisa também foi respondida neste trabalho, pois o algoritmo Mono-Break, através do seu recurso de decomposição por similaridade, permitiu ao usuário da técnica ajustar a granularidade dos microsserviços gerados à decomposição ideal.

Por fim, o resultado dessa avaliação permitiu verificar as semelhanças e diferenças ao decompor uma aplicação monolítica em microsserviços de forma manual e a partir de uma técnica semiautomática. Contudo, para responder à terceira questão de pesquisa, cabe enfatizar, conforme discutido neste trabalho, que o uso de princípios, boas práticas ou técnicas semiautomáticas não garantem uma decomposição perfeita, já que existem outras questões que podem influenciar na forma de realizar a decomposição como, por exemplo, o conhecimento empírico sobre o contexto da aplicação. Tal constatação sinaliza que há desafios a serem estudados na área de Engenharia de Software no que diz respeito a encontrar a melhor granularidade de microsserviços ao decompor uma aplicação monolítica.

Figura 48 – Resultado da decomposição: Técnica Manual x Técnica de Monólise



Fonte: Elaborado pelo autor

6 CONCLUSÃO

Este trabalho investigou o tema de decomposição de aplicações monolíticas em microsserviços, tendo em vista que se trata de um assunto relativamente recente tanto no âmbito da academia quanto no âmbito da indústria e que apresenta lacunas a serem preenchidas por investigações científicas. Para tratar do assunto em foco, esta pesquisa apresentou a técnica de Monólise, que consiste no processo de separação das funcionalidades de uma aplicação monolítica em microsserviços. A Monólise tem como objetivo ajudar as equipes de desenvolvimento de software na jornada de modernização de suas aplicações monolíticas em microsserviços. Para apoiar os times de desenvolvimento na decomposição de suas aplicações, a técnica conta com um algoritmo chamado MonoBreak, o qual permite aos times de desenvolvimento simular diversos cenários de decomposições de suas aplicações monolíticas em microsserviços. Essas simulações possibilitam encontrar o melhor grau de granularidade que os profissionais entendem ser o ideal para a decomposição das funcionalidades de suas aplicações.

Esta pesquisa, além da introdução e da conclusão, foi organizada em quatro capítulos principais. No Capítulo 2, foram apresentados os principais conceitos sobre arquitetura de software, computação em nuvem e estilos arquiteturais, como: arquitetura monolítica, arquitetura orientada a serviços e arquitetura de microsserviços. Já no Capítulo 3, discutiu-se o estado da arte referente à decomposição de aplicações monolíticas em microsserviços e também realizou-se uma análise comparativa entre diferentes trabalhos a partir de critérios definidos por esta investigação científica.

O Capítulo 4 apresentou a visão geral das técnica, seus principais diferenciais e também demonstrou como que o algoritmo MonoBreak e a técnica foram implementados. O Capítulo 5 comparou a decomposição realizada pela técnica Monólise com a decomposição executada por um especialista na aplicação-alvo utilizada no estudo de caso. Essa comparação possibilitou verificar a efetividade da Monólise a partir de oito cenários realísticos de decomposição, os quais foram responsáveis por comprovar a precisão da técnica proposta.

Por fim, o desenvolvimento desta pesquisa demonstrou que a técnica de Monólise apresenta-se com uma grande potencialidade na área de Engenharia de Software referente à decomposição de aplicações. A Monólise, através do algoritmo MonoBreak, permitiu definir o grau de granularidade dos microsserviços gerados e também identificar quais as funcionalidades, as classes e os métodos que necessitam ser migrados ao implementar um determinado microsserviço. Ademais, o trabalho também comprovou o que vem sendo discutido por diversas pesquisas na área no que diz respeito à influência do conhecimento empírico do desenvolvedor no momento da decomposição, ou seja, ele também influencia no modo como a decomposição é realizada. As considerações do estudo evidenciam que essa técnica poderá ser um motivador para encorajar desenvolvedores e arquitetos na jornada de modernização de suas aplicações monolíticas em microsserviços bem como diminuir possíveis erros cometidos nessa atividade por profissionais com pouca experiência em decomposição de aplicações.

6.1 Contribuições

A realização dessa pesquisa trouxe três grandes contribuições científicas referente à temática de decomposição de aplicações monolíticas em microsserviços.

- **Técnica Proposta**

  A técnica de Monólise proposta nesta pesquisa preencheu uma série de lacunas deixadas pelos trabalhos que compõem o estado da arte em decomposição de aplicações monolíticas em microsserviços, como: criação de um algoritmo que possibilita a definição da granularidade dos microsserviços a serem gerados; análise dinâmica de código da aplicação-alvo a ser decomposta; algoritmo sensível à arquitetura da aplicação; recomendação de microsserviços a nível de funcionalidade, classes e métodos; e utilização da teoria de conjuntos como base para análise de similaridade entre as funcionalidades no momento da decomposição. Todos esse recursos descritos são lacunas identificadas nos trabalhos relacionados desta pesquisa que foram desenvolvidos na técnica de Monólise e no algoritmo MonoBreak. A criação da técnica de Monólise proporciona um processo estruturado e semiautomático que pode ser utilizado tanto no contexto da indústria quanto no da academia no que diz respeito à modernização de aplicações monolíticas para microsserviços. Essa técnica apresenta um grande potencial para resolver os problemas referentes à identificação da melhor granularidade de microsserviços, conforme descrito nos trabalhos de Mustafa et al. (2018), Gysel et al. (2016) e Mazlami, Cito e Leitner (2017).

- **Análise Crítica do Estado da Arte**

  Esta pesquisa executou uma análise crítica sobre o estado da arte relacionado à decomposição de aplicações monolíticas em microsserviços. A partir dessa análise, foi possível identificar diversas lacunas das quais os trabalhos não tratavam em sua avaliações. Por exemplo, nenhum dos trabalhos analisado realizou experimentos controlados para validação de suas técnicas. Os trabalhos também não utilizaram a técnica de análise de código dinâmica, embora, segundo Escobar et al. (2016), esse tipo de análise possa gerar uma melhor recomendação dos microsserviços. Foi importante realizar essa análise crítica dos trabalhos, pois identificou-se diversas oportunidades de pesquisa na área referente à decomposição de aplicações monolíticas em microsserviços.

- **Conhecimento Empírico sobre Decomposição de Aplicações em Microsserviços**

  O uso de uma aplicação real como estudo de caso nesta pesquisa aumentou o conhecimento empírico sobre a decomposição de aplicações monolítica em microsserviços. A avaliação realizada neste trabalho inclusive reforça o que foi descrito por Gysel et al. (2016), isto é, o resultado da decomposição de uma aplicação monolítica em microsserviços está totalmente relacionado com a experiência de quem está executando a de-

composição. Durante a execução da avaliação, foi possível identificar as semelhanças e as diferenças entre executar uma decomposição manual e uma baseada em uma técnica semiautomática, no caso, a Monólise.

6.2 Limitações e Trabalhos Futuros

Conforme discutido no decorrer deste trabalho, o estudo desenvolvido apresentou contribuições científicas referentes à temática de decomposição de aplicações monolíticas em microsserviços. No entanto, observou-se que ainda é necessário investigar outras questões relativas ao tema, tendo em vista que a jornada de decomposição de uma aplicação em microsserviços apresenta diversas variáveis, como o conhecimento empírico de quem está realizando-a, e que podem interferir nos resultados da realização de decomposições de aplicações. Assim, serão apresentadas abaixo as limitações desta pesquisa e possíveis trabalhos futuros.

- **Automatizar Passos Manuais**

  Conforme discutido neste trabalho, a técnica proposta apresenta alguns passos que são executados manualmente. Para gerar o arquivo de rastro de execução de cada funcionalidade a ser decomposta mais o arquivo de configuração referente à arquitetura da aplicação, o usuário da técnica precisa prover manualmente esses arquivos, o que torna a tarefa trabalhosa. A fim de simplificar esse processo, seria interessante prover uma forma automática que possibilitasse gerar esses arquivos através de uma interface WEB, habilitando o usuário a controlar o início e o fim da captura do fluxo de execução da aplicação. Isso, consequentemente, possibilitaria gerar o arquivo de rastro de funcionalidade de forma automática. Para gerar o arquivo de configuração da arquitetura, poderia ser utilizado um formulário através de uma aplicação WEB para fornecer as informações da configuração da arquitetura da aplicação-alvo a ser decomposta.

- **Adicionar Novos Critérios de Comparação**

  A Monólise realiza a decomposição das funcionalidades a partir da similaridade entre as classes das funcionalidades que se deseja decompor. Esse único critério referente à similaridade de funcionalidade pode não ser interessante em cenários em que se decompõe a aplicação baseada em requisitos não funcionais, como desempenho da aplicação. Para tanto, seria importante disponibilizar na técnica de Monólise a possibilidade de alterar qual o tipo de critério será utilizado para gerar os microsserviços. Por exemplo, a decomposição da aplicação será baseada no requisito de manutenibilidade ou no de desempenho. Essa possibilidade de escolha por parte do usuário permitiria a técnica ser empregada em diferentes cenários de decomposição.

- **Gerar Implementação dos Microsserviços Recomendados**

Conforme apresentado neste trabalho, a técnica proposta apenas sugere um modelo de implementação de microsserviços. Seria interessante a técnica recomendar e gerar os microsserviços com uma implementação básica. Assim, o usuário conseguiria implantar e testar a recomendação num cenário real, a fim de validar se a recomendação gerada seria apropriada para o cenário em que se espera executar os microsserviços. Isso possibilitaria aos usuários validar de forma efetiva se a recomendação gerada apresentaria um problema relacionado ao desempenho ao utilizar esses microsserviços num ambiente produtivo.

- **Realizar Estudos Experimentais**

  Conforme apresentado no Capítulo 3, nenhum trabalho em decomposição de aplicações monolíticas em microsserviços realizou um experimento controlado referente ao tema de decomposição de aplicações. Seria importante realizar experimentos controlados para verificar o impacto que uma decomposição realizada com granularidade muito baixa ou alta e o quanto acarreta, por exemplo, em questões de custo e desempenho no uso de microsserviços na nuvem. Outra questão importante relacionada à técnica proposta neste trabalho seria realizar uma pesquisa qualitativa com profissionais da área para verificar a qualidade da recomendação dos microsserviços gerados pela técnica, já que a técnica é validada somente contra uma decomposição ideal gerada por um especialista na aplicação. Seria válido, portanto, verificar a opinião de outros profissionais e também executar essa técnica em outras aplicações para averiguar a sua eficácia.

REFERÊNCIAS

ABBOTT, M. L.; FISHER, M. T. **The art of scalability**: scalable web architecture, processes, and organizations for the modern enterprise. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2015.

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: SERVICE-ORIENTED COMPUTING AND APPLICATIONS (SOCA), 2016 IEEE 9TH INTERNATIONAL CONFERENCE ON, 2016. **Anais...** [S.l.: s.n.], 2016. p. 44–51.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: migration to a cloud-native architecture. **IEEE Software**, [S.l.], v. 33, n. 3, p. 42–52, May 2016.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. 3rd. ed. [S.l.]: Addison-Wesley Professional, 2012.

BUCCHIARONE, A. et al. From monolithic to microservices: an experience report from the banking domain. **IEEE Software**, [S.l.], v. 35, n. 3, p. 50–55, May 2018.

CHEN, R.; LI, S.; LI, Z. From monolith to microservices: a dataflow-driven approach. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p. 466–475.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In: **Present and ulterior software engineering**. [S.l.]: Springer, 2017. p. 195–216.

ERL, T. **Soa principles of service design (the prentice hall service-oriented computing series from thomas erl)**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.

ERL, T.; PUTTINI, R.; MAHMOOD, Z. **Cloud computing**: concepts, technology & architecture. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013.

ESCOBAR, D. et al. Towards the understanding and evolution of monolithic applications as microservices. In: XLII LATIN AMERICAN COMPUTING CONFERENCE (CLEI), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p. 1–11.

EVANS, E.; SZPOTON, R. **Domain-driven design**. [S.l.]: Helion, 2015.

FIELDING, R. T.; TAYLOR, R. N. **Architectural styles and the design of network-based software architectures**. [S.l.]: University of California, Irvine Doctoral dissertation, 2000. v. 7.

FOWLER, M. **Patterns of enterprise application architecture**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

FOWLER, M. **Stranglerapplication**. Disponível em: <https://www.martinfowler.com/bliki/StranglerApplication.html>. Acesso em: 15 maio 2018.

FOWLER, M.; LEWIS, J. Microservices. **Viittattu**, [S.l.], v. 28, p. 2015, 2014.

FRAKES, W. B.; BAEZA-YATES, R. **Information retrieval**: data structures & algorithms. [S.l.]: prentice Hall Englewood Cliffs, NJ, 1992. v. 331.

GAMMA, E. et al. **Design patterns**: elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

GULIA, P.; DEV, A.; PATEL, S. Comparative analysis of object oriented programming and aspect oriented programming approach. In: INTERNATIONAL CONFERENCE ON COMPUTING FOR SUSTAINABLE GLOBAL DEVELOPMENT (INDIACOM), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 1836–1842.

GYSEL, M. et al. Service cutter: a systematic approach to service decomposition. In: SERVICE-ORIENTED AND CLOUD COMPUTING, 2016, Cham. **Anais...** Springer International Publishing, 2016. p. 185–200.

KNOCHE, H.; HASSELBRING, W. Using microservices for legacy software modernization. **IEEE Software**, [S.l.], v. 35, n. 3, p. 44–49, May 2018.

LEVCOVITZ, A.; TERRA, R.; VALENTE, M. T. Towards a technique for extracting microservices from monolithic enterprise systems. **CoRR**, [S.l.], v. abs/1605.03175, 2016.

MACKENZIE, C. M. et al. Reference model for service oriented architecture 1.0. **OASIS standard**, [S.l.], v. 12, p. 18, 2006.

MARTIN, R. C.; MARTIN, M. **Agile principles, patterns, and practices in c# (robert c. martin)**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

MAZLAMI, G.; CITO, J.; LEITNER, P. Extraction of microservices from monolithic software architectures. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES (ICWS), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p. 524–531.

MUSTAFA, O. et al. Granmicro: a black-box based approach for optimizing microservices based applications. In: FROM SCIENCE TO SOCIETY, 2018, Cham. **Anais...** Springer International Publishing, 2018. p. 283–294.

NADAREISHVILI, I. et al. **Microservice architecture**: aligning principles, practices, and culture. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2016.

NEWMAN, S. **Building microservices**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015.

PAHL, C.; JAMSHIDI, P. Microservices: a systematic mapping study. In: INTERNATIONAL CONFERENCE ON CLOUD COMPUTING AND SERVICES SCIENCE - VOLUME 1 AND 2, 6., 2016, Portugal. **Proceedings...** SCITEPRESS - Science and Technology Publications: Lda, 2016. p. 137–146. (CLOSER 2016).

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **ACM SIGSOFT Software engineering notes**, [S.l.], v. 17, n. 4, p. 40–52, 1992.

RICHARDSON, C. **Api gateway**. Disponível em: <http://microservices.io/patterns/apigateway.html>. Acesso em: 06 junho 2018.

SHAW, M.; GARLAN, D. **Software architecture**: perspectives on an emerging discipline. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

STINE, M. **Migrating to cloud-native application architectures**. [S.l.]: O'Reilly, 2015.

SZYPERSKI, C. **Component software**: beyond object-oriented programming. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

WANG, A.; TONSE, S. **Announcing ribbon**: tying the netflix mid-tier services together. [S.l.]: January, 2013.

WIGGINS, A. **The twelve-factor app**. Disponível em: <https://12factor.net/>. Acesso em: 06 junho 2018.

WITTIG, A.; WITTIG, M. **Amazon web services in action**. 1st. ed. Greenwich, CT, USA: Manning Publications Co., 2015.

WUST, J. **Sdmetrics**: the software design metrics tool for uml. 2005.

XIAO, Z.; WIJEGUNARATNE, I.; QIANG, X. Reflections on soa and microservices. In: INTERNATIONAL CONFERENCE ON ENTERPRISE SYSTEMS (ES), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p. 60–67.

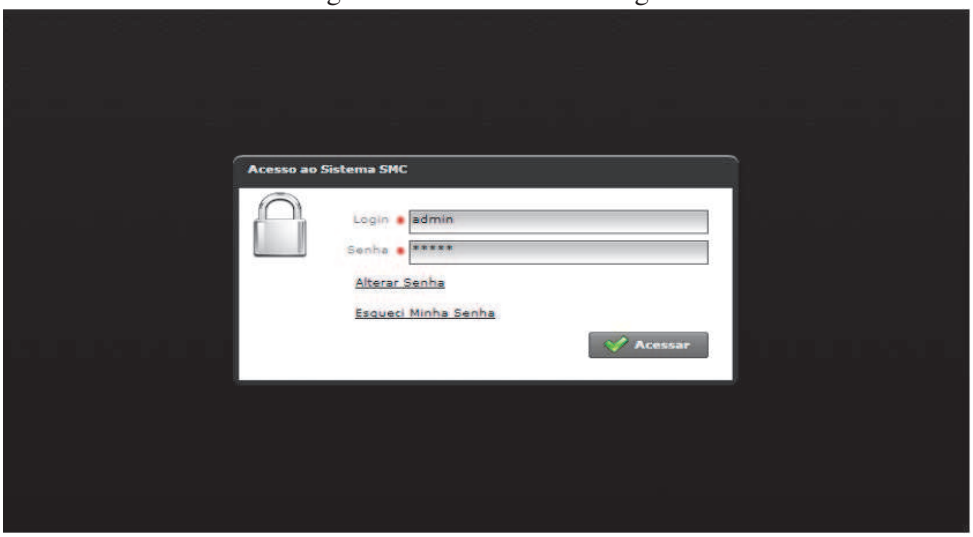
YUGOPUSPITO, P.; PANDUWINATA, F.; SUTRISNO, S. Microservices architecture: case on the migration of reservation-based parking system. In: IEEE 17TH INTERNATIONAL CONFERENCE ON COMMUNICATION TECHNOLOGY (ICCT), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p. 1827–1831.

APÊNDICE A – ARQUIVO DE RASTRO DE EXECUÇÃO DA FUNCIONALIDADE (F015)


```
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "<init>"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "setListHistoricoAcademico"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "setListIdioma"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "setInformacaoComplementar"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "setDataCadastro"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "<init>"}
{"className": "br.com.sape.model.dto.CargoTO","methodName": "<init>"}
{"className": "br.com.sape.model.dto.CargoTO","methodName": "setNome"}
{"className": "br.com.sape.model.dto.CargoTO","methodName": "setId"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setCargo"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setNome"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setNovaSenha"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setSenha"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setEmail"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setLogin"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setBloqueado"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "setId"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "setUsuario"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "setId"}
{"className": "br.com.sape.business.service.FormacaoProfissionalService","methodName": "loadFormacaoProfissionalByUsuario"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "<init>"}
{"className": "br.com.sape.model.assembler.GenericAssembler","methodName": "assembleEntity"}
{"className": "br.com.sape.model.assembler.GenericAssembler","methodName": "assembleEntity"}
{"className": "br.com.sape.util.BeanFactoryTO","methodName": "<init>"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "getInformacaoComplementar"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "setInformacaoComplementar"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "getUsuario"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "getUsuario"}
{"className": "br.com.sape.util.BeanFactoryTO","methodName": "get"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "<init>"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "setUsuario"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getSenha"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setSenha"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getBloqueado"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setBloqueado"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getNome"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setNome"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getId"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setId"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getLogin"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setLogin"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getCargo"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "getCargo"}
{"className": "br.com.sape.util.BeanFactoryTO","methodName": "get"}
{"className": "br.com.sape.model.entity.Cargo","methodName": "<init>"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setCargo"}
{"className": "br.com.sape.model.dto.CargoTO","methodName": "getNome"}
{"className": "br.com.sape.model.entity.Cargo","methodName": "setNome"}
{"className": "br.com.sape.model.dto.CargoTO","methodName": "getId"}
{"className": "br.com.sape.model.entity.Cargo","methodName": "setId"}
{"className": "br.com.sape.model.dto.UsuarioTO","methodName": "getEmail"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "setEmail"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "getId"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "setId"}
{"className": "br.com.sape.model.dto.FormacaoProfissionalTO","methodName": "getDataCadastro"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "setDataCadastro"}
{"className": "br.com.sape.business.service.FormacaoProfissionalService","methodName": "findByUsuario"}
{"className": "br.com.sape.model.entity.FormacaoProfissional","methodName": "getUsuario"}
{"className": "br.com.sape.model.entity.Usuario","methodName": "getId"}
{"className": "br.com.sape.model.dao.GenericDAO","methodName": "loadByParameters"}
{"className": "br.com.sape.business.service.FormacaoProfissionalService","methodName": "findHistoricoAcademicoVinculadoFormacaoProfissional"}
{"className": "br.com.sape.business.service.FormacaoProfissionalService","methodName": "findIdiomaVinculadoFormacaoProfissional"}
{"className": "br.com.sape.business.service.FormacaoProfissionalService","methodName": "findIdiomaNaoVinculadoFormacaoProfissional"}
{"className": "br.com.sape.business.service.IdiomaService","methodName": "findAll"}
{"className": "br.com.sape.model.dao.GenericDAO","methodName": "findAll"}
... Suprimido o conteudo devido ao tamanho
```

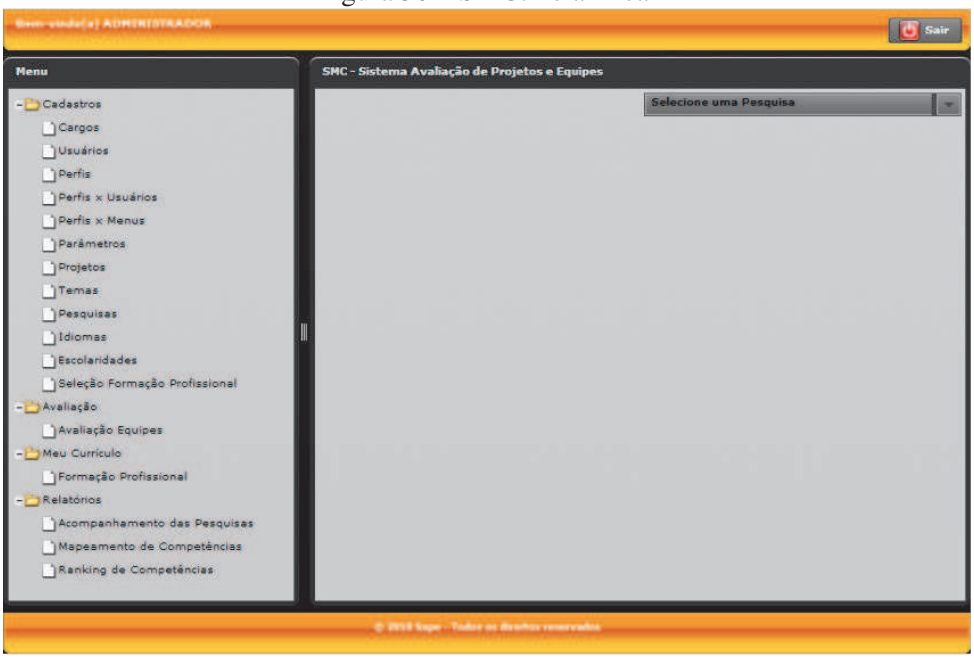
APÊNDICE B – TELAS DO SISTEMA SMC

Figura 49 – SMC: Tela de login



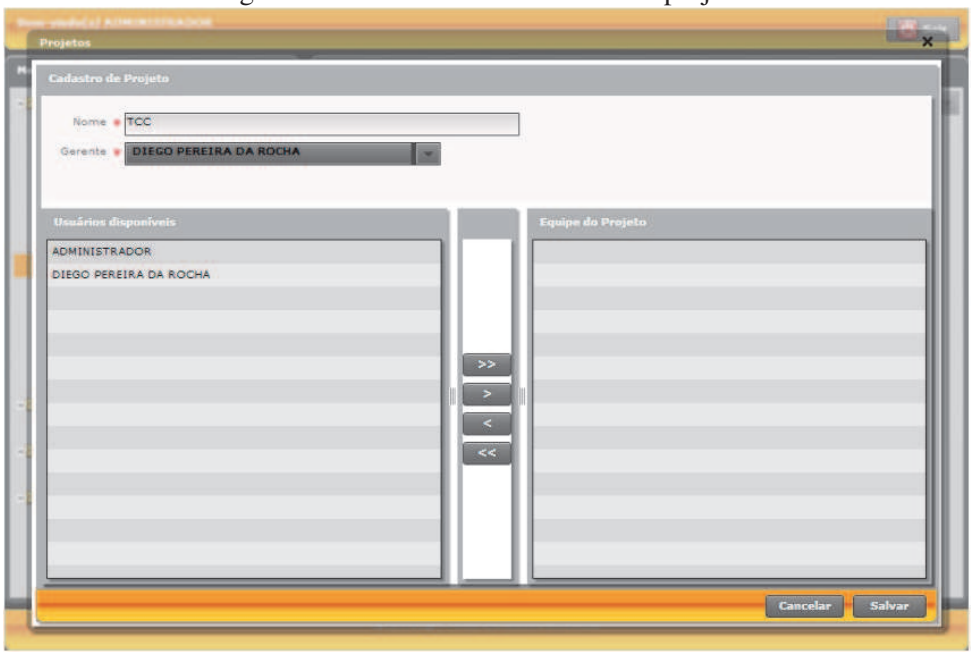
Fonte: Elaborado pelo autor

Figura 50 – SMC: Tela inical



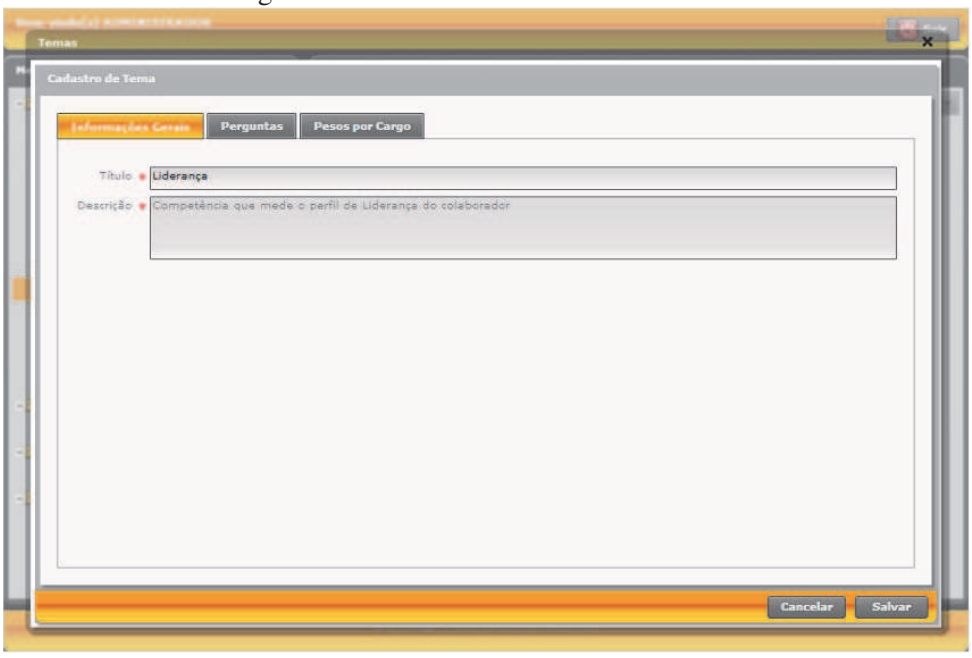
Fonte: Elaborado pelo autor

Figura 51 – SMC: Tela de cadastro de projetos



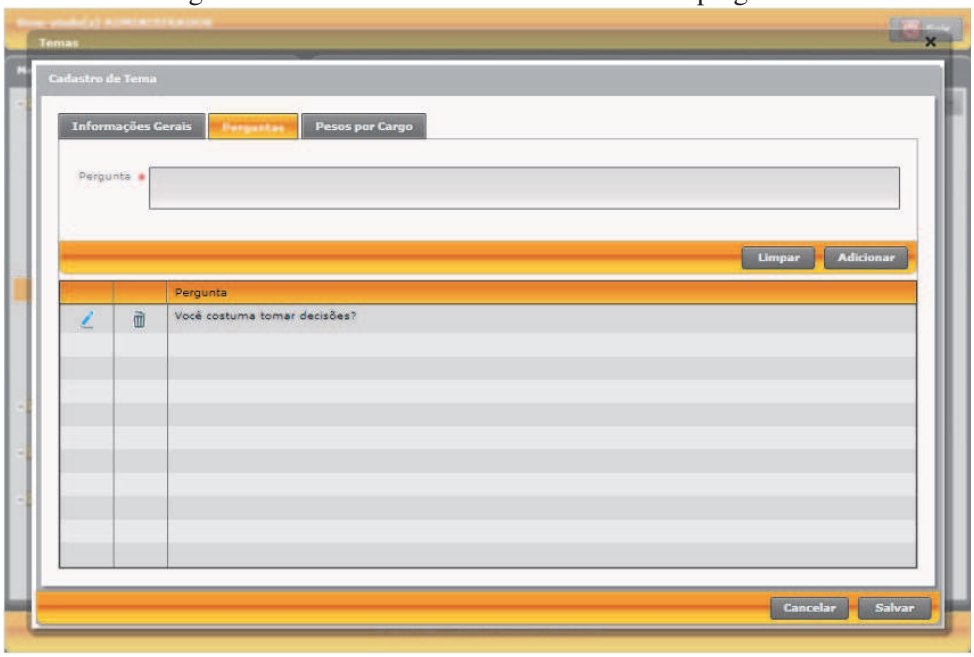
Fonte: Elaborado pelo autor

Figura 52 – SMC: Tela de cadastro de tema



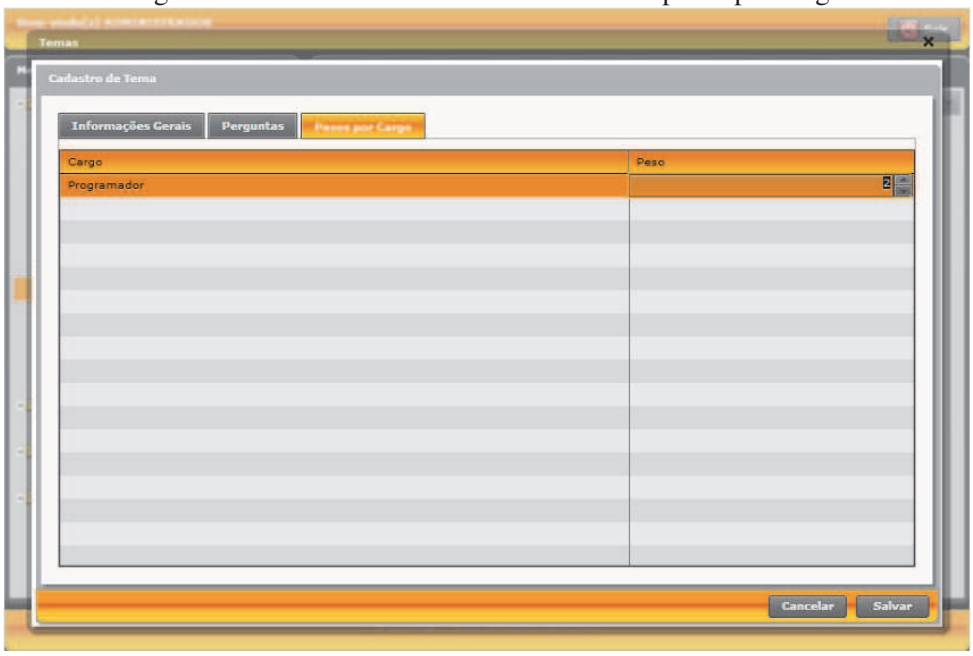
Fonte: Elaborado pelo autor

Figura 53 – SMC: Tela de cadastro de tema e perguntas



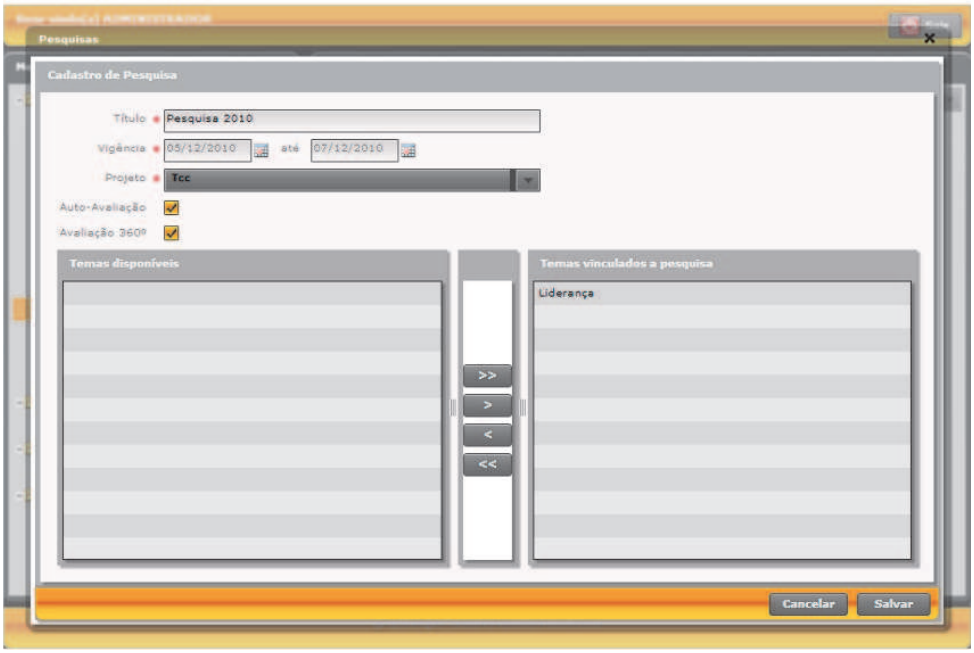
Fonte: Elaborado pelo autor

Figura 54 – SMC: Tela de cadastro de tema e pesos por cargo



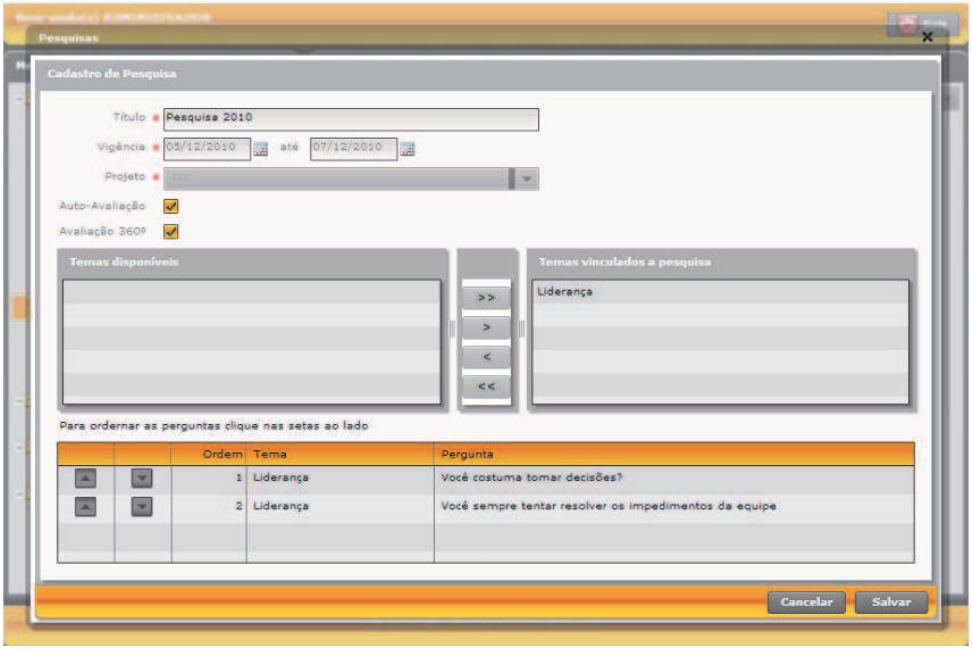
Fonte: Elaborado pelo autor

Figura 55 – SMC: Tela de cadastro de pesquisa



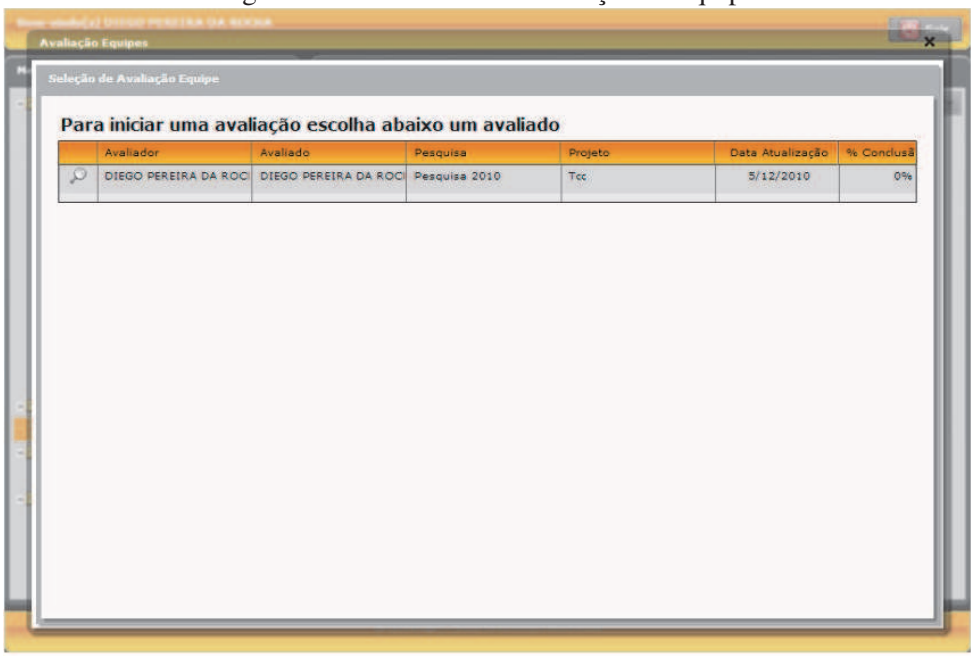
Fonte: Elaborado pelo autor

Figura 56 – SMC: Tela de cadastro de pesquisa e ordem perguntas



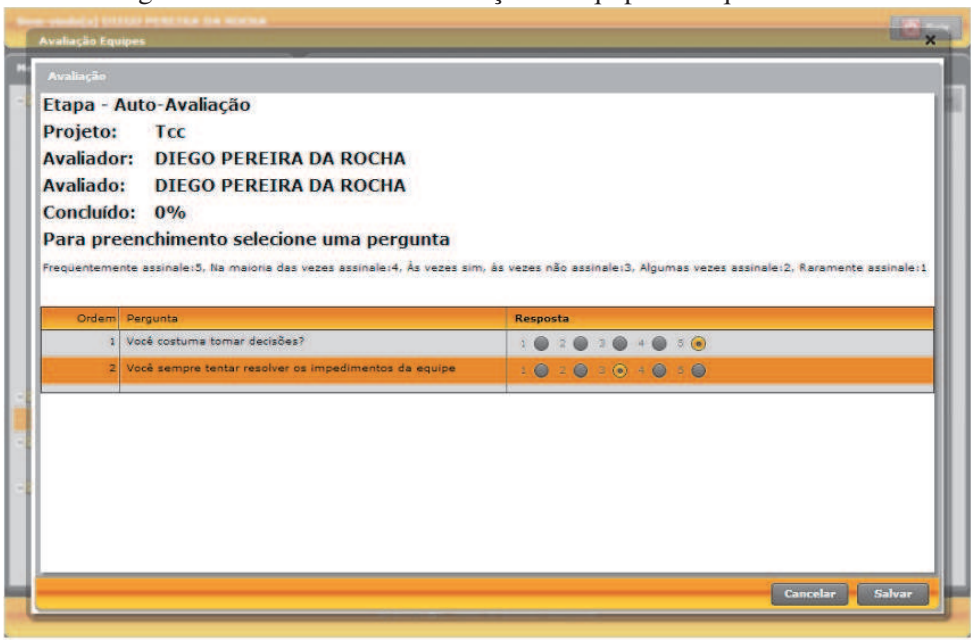
Fonte: Elaborado pelo autor

Figura 57 – SMC: Tela de avaliação de equipe



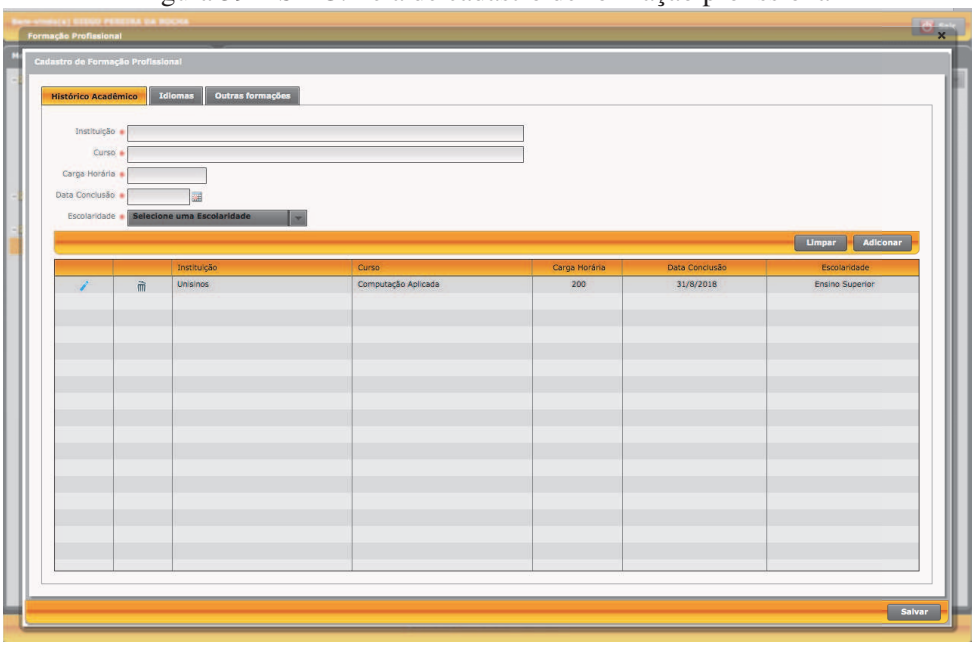
Fonte: Elaborado pelo autor

Figura 58 – SMC: Tela de avaliação de equipe com questionário



Fonte: Elaborado pelo autor

Figura 59 – SMC: Tela de cadastro de formação profissional



Fonte: Elaborado pelo autor

APÊNDICE C – MONÓLISE: RESULTADO DA DECOMPOSIÇÃO DO SMC


```
[ {
    "id" : "Microservice 1",
    "functionalities" : [ "F001", "F002", "F003" ],
    "classes" : [ {
        "name" : "br.com.sape.model.dto.PesquisaTO",
        "methods" : [ {
            "name" : "getAutoAvaliacao"
        }, {
          "name" : "<init>"
        }, {
          "name" : "setProjeto"
        }, {
          "name" : "setDataFim"
        }, {
          "name" : "setTitulo"
        }, {
          "name" : "setAvaliacao360"
        }, {
          "name" : "setId"
        }, {
          "name" : "setDataInicio"
        }, {
          "name" : "setPublicada"
        }, {
          "name" : "setAutoAvaliacao"
        }, {
          "name" : "getProjeto"
        }, {
          "name" : "getDataFim"
        }, {
          "name" : "getListTema"
        }, {
          "name" : "getTitulo"
        }, {
          "name" : "getAvaliacao360"
        }, {
          "name" : "getDataInicio"
        }, {
          "name" : "getId"
```

```
    }, {
      "name" : "getPublicada"
    }, {
      "name" : "getListPesquisaPergunta"
    } ]
  }, {
    "name" : "br.com.sape.model.dto.UsuarioTO",
    "methods" : [ {
      "name" : "<init>"
    }, {
      "name" : "setCargo"
    }, {
      "name" : "setNome"
    }, {
      "name" : "setNovaSenha"
    }, {
      "name" : "setSenha"
    }, {
      "name" : "setEmail"
    }, {
      "name" : "setLogin"
    }, {
      "name" : "setBloqueado"
    }, {
      "name" : "setId"
    }, {
      "name" : "getLogin"
    }, {
      "name" : "getSenha"
    }, {
      "name" : "getNovaSenha"
    }, {
      "name" : "getNome"
    }, {
      "name" : "getBloqueado"
    }, {
      "name" : "getId"
    }, {
      "name" : "getCargo"
    }, {
      "name" : "getEmail"
```

```
    } ]
  }, {
    "name" : "br.com.sape.business.service.LoginService",
    "methods" : [ {
      "name" : "alterarSenha"
    }, {
      "name" : "esqueciSenha"
    }, {
      "name" : "login"
    }, {
      "name" : "montaMenuUsuario"
    } ]
  }, {
    "name" : "br.com.sape.business.service.UsuarioService",
    "methods" : [ {
      "name" : "findByLoginSenha"
    }, {
      "name" : "update"
    }, {
      "name" : "findByLogin"
    } ]
  }, {
    "name" : "br.com.sape.util.MD5",
    "methods" : [ {
      "name" : "gerarMD5"
    } ]
  }, {
    "name" : "br.com.sape.model.dao.GenericDAO",
    "methods" : [ {
      "name" : "loadByParameters"
    }, {
      "name" : "update"
    }, {
      "name" : "findByParameters"
    }, {
      "name" : "findAll"
    } ]
  }, {
    "name" : "br.com.sape.model.entity.Usuario",
    "methods" : [ {
      "name" : "<init>"
```

```
    }, {
      "name" : "setId"
    }, {
      "name" : "setBloqueado"
    }, {
      "name" : "setCargo"
    }, {
      "name" : "setEmail"
    }, {
      "name" : "setListPerfil"
    }, {
      "name" : "setLogin"
    }, {
      "name" : "setNome"
    }, {
      "name" : "setSenha"
    }, {
      "name" : "getBloqueado"
    }, {
      "name" : "getCargo"
    }, {
      "name" : "getEmail"
    }, {
      "name" : "getListPerfil"
    }, {
      "name" : "getLogin"
    }, {
      "name" : "getNome"
    }, {
      "name" : "getSenha"
    }, {
      "name" : "getId"
    } ]
  }, {
    "name" : "br.com.sape.model.entity.Cargo",
    "methods" : [ {
      "name" : "<init>"
    }, {
      "name" : "setId"
    }, {
      "name" : "setNome"
```

```
    }, {
      "name" : "getNome"
    }, {
      "name" : "getId"
    } ]
  }, {
    "name" : "br.com.sape.model.assembler.GenericAssembler",
    "methods" : [ {
      "name" : "assembleTO"
    } ]
  }, {
    "name" : "br.com.sape.util.BeanFactoryTO",
    "methods" : [ {
      "name" : "<init>"
    }, {
      "name" : "get"
    } ]
  }, {
    "name" : "br.com.sape.model.dto.CargoTO",
    "methods" : [ {
      "name" : "<init>"
    }, {
      "name" : "setNome"
    }, {
      "name" : "setId"
    }, {
      "name" : "getNome"
    }, {
      "name" : "getId"
    } ]
  }, {
    "name" : "br.com.sape.business.service.MenuService",
    "methods" : [ {
      "name" : "findByPerfil"
    }, {
      "name" : "findSubmenuByMenuPai"
    } ]
  }, {
    "name" : "br.com.sape.model.entity.Menu",
    "methods" : [ {
      "name" : "<init>"
```

```
    }, {
      "name" : "setId"
    }, {
      "name" : "setAtivo"
    }, {
      "name" : "setListMenu"
    }, {
      "name" : "setListPerfil"
    }, {
      "name" : "setMenuPai"
    }, {
      "name" : "setNome"
    }, {
      "name" : "setOrdem"
    }, {
      "name" : "setUrl"
    }, {
      "name" : "getAtivo"
    }, {
      "name" : "getListMenu"
    }, {
      "name" : "getListPerfil"
    }, {
      "name" : "getMenuPai"
    }, {
      "name" : "getNome"
    }, {
      "name" : "getOrdem"
    }, {
      "name" : "getUrl"
    }, {
      "name" : "getId"
    } ]
  }, {
    "name" : "br.com.sape.business.service.PesquisaService",
    "methods" : [ {
      "name" : "findAllPublicada"
    } ]
  }, {
    "name" : "br.com.sape.model.entity.Pesquisa",
    "methods" : [ {
```

```
    "name" : "<init>"
  }, {
    "name" : "setId"
  }, {
    "name" : "setAutoAvaliacao"
  }, {
    "name" : "setAvaliacao360"
  }, {
    "name" : "setDataFim"
  }, {
    "name" : "setDataInicio"
  }, {
    "name" : "setListPesquisaPergunta"
  }, {
    "name" : "setListTema"
  }, {
    "name" : "setProjeto"
  }, {
    "name" : "setPublicada"
  }, {
    "name" : "setTitulo"
  }, {
    "name" : "getAutoAvaliacao"
  }, {
    "name" : "getAvaliacao360"
  }, {
    "name" : "getDataFim"
  }, {
    "name" : "getDataInicio"
  }, {
    "name" : "getListPesquisaPergunta"
  }, {
    "name" : "getListTema"
  }, {
    "name" : "getProjeto"
  }, {
    "name" : "getPublicada"
  }, {
    "name" : "getTitulo"
  }, {
    "name" : "getId"
```

```
    } ]
  }, {
    "name" : "br.com.sape.model.entity.Projeto",
    "methods" : [ {
      "name" : "<init>"
    }, {
      "name" : "setId"
    }, {
      "name" : "setGerente"
    }, {
      "name" : "setListUsuario"
    }, {
      "name" : "setNome"
    }, {
      "name" : "getGerente"
    }, {
      "name" : "getListUsuario"
    }, {
      "name" : "getNome"
    }, {
      "name" : "getId"
    } ]
  }, {
    "name" : "br.com.sape.model.dto.ProjetoTO",
    "methods" : [ {
      "name" : "<init>"
    }, {
      "name" : "setNome"
    }, {
      "name" : "setId"
    }, {
      "name" : "setGerente"
    }, {
      "name" : "getListEquipe"
    }, {
      "name" : "getNome"
    }, {
      "name" : "getId"
    }, {
      "name" : "getGerente"
    } ]
```

```
}, {
  "name" : "br.com.sape.util.GeradorSenha",
  "methods" : [ {
    "name" : "geraSenha"
  } ]
}, {
  "name" : "br.com.sape.business.service.EmailService",
  "methods" : [ {
    "name" : "send"
  }, {
    "name" : "getMailSession"
  } ]
}, {
  "name" : "br.com.sape.business.service.ParametroService",
  "methods" : [ {
    "name" : "getParametroByNome"
  } ]
}, {
  "name" : "br.com.sape.model.entity.Parametro",
  "methods" : [ {
    "name" : "<init>"
  }, {
    "name" : "setId"
  }, {
    "name" : "setDescricao"
  }, {
    "name" : "setNome"
  }, {
    "name" : "setValor"
  }, {
    "name" : "getDescricao"
  }, {
    "name" : "getNome"
  }, {
    "name" : "getValor"
  }, {
    "name" : "getId"
  } ]
}, {
  "name" : "br.com.sape.business.service.EmailService\$AutenticaSMTP",
  "methods" : [ {
```

```
            "name" : "<init>"
          } ]
        } ]
    },
    {
      "id" : "Microservice 2",
      "functionalities" : [ "F004", "F005", "F006", "F007", "F008", "F011" ],
      "classes" : [ {...}]
    },
    {
      "id" : "Microservice 3",
      "functionalities" : [ "F009" ],
      "classes" : [ {...}]
    },
    {
      "id" : "Microservice 4",
      "functionalities" : [ "F012" ],
      "classes" : [ {...}]
    },
    {
      "id" : "Microservice 5",
      "functionalities" : [ "F013", "F014", "F015", "F016" ],
      "classes" : [ {...}]
  },
  {
      "id" : "Microservice 6",
      "functionalities" : [ "F017", "F018" ],
      "classes" : [ {...}]
  },
  {
      "id" : "Microservice 7",
      "functionalities" : [ "F019", "F020", "F021" ],
      "classes" : [ {...}]
  },
  {
      "id" : "Microservice 8",
      "functionalities" : [ "F010" ],
      "classes" : [ {...}]
  }
]
```

APÊNDICE D – ALGORITMO QUE EXPORTA ESTRUTURA DE DADOS DO MONOBREAK

```
1  package br.unisinos.monolith.report;
2
3  import br.unisinos.monolith.algorithm.SimilarityAlgorithm;
4  import br.unisinos.monolith.algorithm.model.Functionality;
5  import br.unisinos.monolith.algorithm.model.Microservice;
6  import br.unisinos.monolith.algorithm.model.MonolithClass;
7  import br.unisinos.monolith.algorithm.model.MonolithMethod;
8  import br.unisinos.monolith.utils.CSVUtils;
9  import com.fasterxml.jackson.databind.ObjectMapper;
10 import com.fasterxml.jackson.databind.SerializationFeature;
11 import com.google.common.collect.Sets;
12 import com.google.common.collect.Table;
13 import org.slf4j.Logger;
14 import org.slf4j.LoggerFactory;
15
16 import java.io.File;
17 import java.io.FileWriter;
18 import java.io.IOException;
19 import java.util.*;
20
21 public class MonoBreakReport {
22
23     private final static Logger LOGGER = LoggerFactory.getLogger(
          MonoBreakReport.class);
24     private final String fullPath;
25
26     public MonoBreakReport(String prefixFileName, String path) {
27         this.fullPath = new StringBuilder(path).append(prefixFileName
              ).append("_").toString();
28     }
29
30     private FileWriter createFile(String fileName) {
31         try {
32             StringBuilder sb = new StringBuilder(fullPath);
33             sb.append(fileName);
34             return new FileWriter(sb.toString());
35         } catch (IOException e) {
36             LOGGER.error(e.getMessage(), e);
37             throw new IllegalArgumentException(String.format("Erro ao
```

```
                criar o %s", fileName));
38          }
39      }
40
41      /**
42       * Exporta para CSV as funcionalidades e a quantidade de classes
             e metodos
43       *
44       * @param functionalities
45       */
46      public void exportCSVFunctionalities(List<Functionality>
           functionalities) {
47          FileWriter fileWriter = createFile("functionalities.csv");
48
49          functionalities.stream().forEach(functionality -> {
50              String className = functionality.getName();
51              String numberOfClasses = Integer.toString(functionality.
                   getClasses().size());
52              Integer totalMethods = 0;
53
54              for (MonolithClass monolithClass : functionality.
                   getClasses()) {
55                  totalMethods = totalMethods + monolithClass.
                       getMethods().size();
56              }
57
58              String numberOfMethods = Integer.toString(totalMethods);
59
60              CSVUtils.writeLine(fileWriter, Arrays.asList(
                   functionality.getName(), numberOfClasses,
                   numberOfMethods));
61          });
62      }
63
64      /**
65       * Exporta para CSV os microsservicos com o numero de
             funcionalidades classes e metodos
66       *
67       * @param microservices
68       */
69      public void exportSummarizeMicrosevices(List<Microservice>
```

```
        microservices) {
70       FileWriter fileWriter = createFile("summarizeMicrosevices.csv
             ");
71
72       CSVUtils.writeLine(fileWriter, Arrays.asList("Microservice",
             "Classes", "Methods", "Functionalities"));
73
74       int numberOfMicroservices = microservices.size();
75       int numberOfClasses;
76       int numberOfMethods = 0;
77       int numberOfFunctionalities = 0;
78
79       Map<MonolithClass, Set<MonolithMethod>> classes = new
             LinkedHashMap<>();
80
81       for (Microservice microservice : microservices) {
82           numberOfFunctionalities += microservice.
                 getFunctionalities().size();
83
84           for (MonolithClass monolithClass : microservice.
                 getClasses()) {
85               if (!classes.containsKey(monolithClass)) {
86                   classes.put(monolithClass, monolithClass.
                         getMethods());
87               } else {
88                   Set<MonolithMethod> m1 = classes.get(
                         monolithClass);
89                   Set<MonolithMethod> m2 = monolithClass.getMethods
                         ();
90                   Sets.SetView<MonolithMethod> union = Sets.union(
                         m1, m2);
91                   classes.put(monolithClass, union);
92               }
93           }
94       }
95
96       for (Map.Entry<MonolithClass, Set<MonolithMethod>> entry :
             classes.entrySet()) {
97           numberOfMethods += entry.getValue().size();
98       }
99
```

```
100         numberOfClasses = classes.size();
101
102         CSVUtils.writeLine(fileWriter, Arrays.asList(Integer.toString
                (numberOfMicroservices),
103                 Integer.toString(numberOfClasses),
104                 Integer.toString(numberOfMethods),
105                 Integer.toString(numberOfFunctionalities)));
106     }
107
108     /**
109      * Exporta para CSV a tabela de similaridade
110      *
111      * @param similarityTable
112      */
113     public void exportCSVTableSimilarity(Table<String, String,
            SimilarityAlgorithm.Similarity> similarityTable) {
114         FileWriter fileWriter = createFile("similarityTable.csv");
115
116         StringBuilder sb = new StringBuilder();
117
118         similarityTable.rowMap().forEach((row, columns) -> {
119             sb.append(row).append(",");
120         });
121
122         CSVUtils.writeLine(fileWriter, Arrays.asList("", sb.toString
                ()));
123
124         similarityTable.rowKeySet().forEach(row -> {
125             StringBuilder col = new StringBuilder();
126             similarityTable.rowKeySet().forEach(column -> {
127                 SimilarityAlgorithm.Similarity similarity =
                        similarityTable.get(row, column);
128                 col.append(similarity != null ? similarity.
                        getSimilarityWeightValue().toString() : "-")
129                         .append(",");
130             });
131             CSVUtils.writeLine(fileWriter, Arrays.asList(row, col.
                    toString()));
132         });
133     }
134
```

```
135     /**
136      * Exporta os resultados dos microservicos para JSON
137      *
138      * @param microservices
139      */
140     public void exportJSONMicroservices(List<Microservice>
            microservices) {
141         ObjectMapper objectMapper = new ObjectMapper();
142         objectMapper.enable(SerializationFeature.INDENT_OUTPUT);
143 
144         StringBuilder pathName = new StringBuilder(fullPath);
145         pathName.append("microservices.json");
146         try {
147             objectMapper.writeValue(new File(pathName.toString()),
                    microservices);
148         } catch (IOException e) {
149             LOGGER.error(e.getMessage(), e);
150         }
151     }
152 }
```