API SLICER: Uma Abordagem Baseada em Features para Decomposição de APIs monolíticas em Microsserviços

Carlos Xavier[1]
Kleinner Farias[2]

Resumo: Muitas abordagens relacionados a decomposição de aplicações monolíticas foram propostas em estudos recentes. Entretanto tais estudos não são sensíveis a decomposição de APIs monolíticas em microsserviços, resultando na decomposição manual das APIs monolíticas, com base apenas na experiência dos desenvolvedores. Este artigo apresenta a API *Slicer*, uma abordagem baseada no nível de similaridade entre *features*, que utiliza o rastro de execução para entender a API monolítica e gerar sugestões de microsserviços. A API *Slicer* se destaca por: (1) mostrar quais funcionalidades devem se transformar em microsserviços e quais devem permanecer na API monolítica; (2) ser agnóstica a tecnologia, contanto que a aplicação alvo seja orientada a objetos; e (3) a similaridade ser feita com base na lista de pacotes que o usuário fornece no início do processo de recomendação. A abordagem API *Slicer* tenta resolver através da decomposição, o problema que toda a aplicação monolítica sofrerá com o tempo: crescer de forma indeterminada, até se tornar complexa, de difícil manutenção e evolução. Os desenvolvedores vão se beneficiar com a API *Slicer*, por ela permitir a decomposição semiautomática de uma API monolítica, além de sua decomposição ter sido criado com base em um estudo empírico. A abordagem foi avaliada por meio de um estudo de caso, onde 3 aplicações alvo foram testadas para verificar a efetividade da recomendação de microsserviços. Para cada aplicação alvo, foram criados dois cenários, um mostrando as recomendações de microsserviços e outro indicando quais serviços deveriam permanecer na API monolítica. Além disso, em cada um dos cenários, o nível de similaridade variou de 10% a 90%, para verificar a partir de qual porcentagem, as abordagens API *Slicer* e *Monólise* alcançariam o resultado desejado. A *Monólise* é uma abordagem de decomposição de aplicações monolíticas em microsserviços, proposta na literatura. As métricas utilizadas para comparar o resultado gerado com o ideal, foram a *precision* e *recall*. E os resultados indicaram que a abordagem API *Slicer* teve o maior nível de precisão em duas das três aplicações alvo utilizadas por este estudo, se mostrando uma alternativa viável.

Palavras-chave: API Slicer, Decomposição, API Monolítica, Microsserviços

Abstract: Many approaches related to decomposing monolithic applications have been proposed in recent studies. However, such studies are not sensitive to the decomposition of monolithic APIs into microservices, resulting in the manual decomposition of monolithic APIs, based only on the experience of the developers; lack of theoretical foundation to follow best practices and possible maintenance problems, which the microservice set out to solve. This article introduces the API Slicer, a similarity level-based approach between features, which uses the execution trace to understand the monolithic API and generate suggestions of microservices. The API Slicer stands out for: (1) showing which functionalities should be transform into microservices and which should remain in the monolithic API; (2) be agnostic the technology,

[1]Graduando em Sistemas de Informação pela Unisinos. Email: carlosf.s.x@hotmail.com
[2]Possui doutorado em Informática pela Pontifícia Universidade Católica do Rio de Janeiro (2012), mestrado em Ciência da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul (2008), graduação em Ciência da Computação pela Universidade Federal de Alagoas (2006) e em Tecnologia da Informação pelo Instituto Federal de Alagoas. Email: kleinnerfarias@unisinos.br

as long as the target application is object-oriented; and (3) the similarity is made based on the list of packages that the user provides at the beginning of the recommendation process. The API *Slicer* approach tries to solve through decomposition, the problem that every application monolithic will suffer over time: grow indefinitely, until it becomes complex, difficult to maintain and evolve. Developers will benefit from the API Slicer because it allow semi-automatic decomposition of a monolithic API in addition to its decomposition have been created on the basis of an empirical study. The approach was evaluated through a case study, where 3 target applications were tested to verify the effectiveness of the microservices recommendation. For each target application, two scenarios were created, one showing microservices recommendations and another indicating which services should remain in the monolithic API. In addition, in each of the scenarios, the similarity level varied from 10% to 90%, to verify from what percentage, the API *Slicer* and *Monólise* approaches would achieve the ideal result. The metrics used to compare the generated result with the ideal, were precision and recall. And the results indicated that the API *Slicer* approach had the highest level of precision in two of the three target applications used by this study, proving to be a viable alternative

Keywords: API *Slicer*, Decomposition, Microservices, Monolithic Application

1 INTRODUÇÃO

Sistemas corporativos estão inseridos em ambientes dinâmicos e de alta volatilidade. Os sistemas corporativos são compostos por APIs . API (Application Programming Interface) tem como objetivo fazer a integração entre sistemas. A proporção que sistemas dinâmicos sofrem alterações, as APIs precisam se readequar às novas regras de negócio. A cada readequação, as APIs precisam adicionar mais regras de negócio e funcionalidades, fazendo com as APIs cresçam em tamanho e complexidade. A longo prazo o aumento de tamanho e complexidade transforma uma API em uma API monolítica.

API monolítica é uma API em que as *features* possuem entrelaçamento e espalhamento ao longo dos módulos que implementam os endpoints. A Figura 1 apresenta a API monolítica. A funcionalidade login (circulo vermelho) e a funcionalidade comentário (losângo verde) estão entrelaçadas nas classes `JWtTokenProvider`, `CustomerDetailService` e `UserRepository`. Isso resulta em (1) forte acoplamento entre classes; (2) problemas de manutenção para os desenvolvedores, visto que se a funcionalidade login ser alterada, a funcionalidade comentário também deverá ser alterada; (3) possível inserção de bugs a cada nova alteração.

A decomposição é o ato de dividir uma aplicação em partes menores. Na maioria dos casos, esse processo é feito de forma manual, fazendo uso apenas da experiência do arquiteto de software (SHEIKH; BS, 2020). Uma aplicação monolítica é usualmente caracterizada como um único artefato de software executável, constituído de módulos altamente acoplados e requisitos implementados de forma entrelaçada e distribuída entre os módulos da aplicação (URDANGARIN; FARIAS; BARBOSA, 2021a). Isso implica em: (1) problema de manutenção, à medida que a aplicação monolítica vai ficando maior; (2) dificuldade de inserir novas tecnologias; (3)

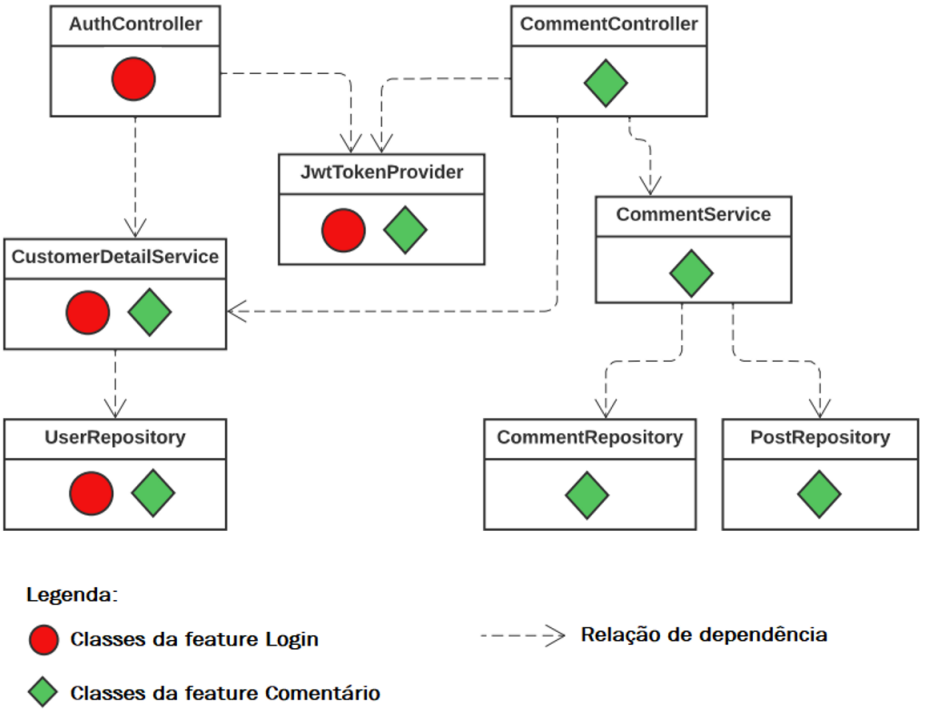
maior complexidade para introduzir uma nova funcionalidade, pois há um risco grande de haver efeitos indesejados em funcionalidades já existentes. (JÚNIOR, 2017)

A literatura atual possui várias abordagens de decomposição de aplicações monolíticas em microsserviços. Dentre as abordagens, são destacadas a decomposição utilizando, dentre outros fatores: o acoplamento e coesão (ASSUNÇÃO et al., 2022) e (FILIPPONE et al., 2021); a estrutura do banco de dados junto com o código da aplicação (ZHAO; ZHAO, 2021); a análise estática do código, considerando classes, métodos e histórico de alterações (SANTOS; PAULA, 2020); os logs da aplicação (SHEIKH; BS, 2020); e os *traces* de execução (ROCHA, 2018). Entretanto tais estudos não são sensíveis a decomposição de APIs monolíticas em microsserviços. Isso resulta na decomposição manual das APIs monolíticas, com base apenas na experiência dos desenvolvedores; falta de embasamento teórico para seguir as melhores práticas e possíveis problemas de manutenção, que o microsserviço se propôs a resolver.

Portanto o presente artigo propõe a API *Slicer*, uma abordagem baseada em *features*, para decompor APIs monolíticas em microsserviços. A abordagem API *Slicer* é dividida em 7 etapas: (1) leitura do *trace* de execução (arquivo contendo os métodos e classes chamadas, quando uma funcionalidade foi executada); (2) leitura dos pacotes que devem ser considerados na decomposição; (3) leitura do nível de similaridade (valor fornecido pelo usuário, que é utilizado como valor de corte, onde se duas funcionalidades alcançarem ou ultrapassarem esse valor de corte, ambas ficarão no mesmo microsserviço, do contrário, irão para microsserviços individuais; (4) processar os arquivos para identificar as *features*, (5) verificar a similaridade entre *features*, (6) executar os agrupamentos necessários e (7) gerar as recomendações de microsserviços. Os desenvolvedores vão se beneficiar com a API *Slicer*, por a abordagem permitir a decomposição semiautomática de uma API monolítica e sua decomposição ter sido criada com base em um estudo empírico. A abordagem foi avaliada por meio de um estudo de caso, onde 3 aplicações alvo foram testadas para verificar a efetividade da recomendação de microsserviços. Para cada aplicação alvo, foram criados dois cenários, um mostrando as recomendações de microsserviços e outro indicando quais serviços deveriam permanecer na API monolítica. Além disso, em cada um dos cenários, o nível de similaridade variou de 10% a 90%, para verificar a partir de qual porcentagem, as abordagens API *Slicer* e *Monólise* alcançariam o resultado desejado. As métricas utilizadas para comparar o resultado gerado com o ideal, foram a *precision* e *recall*. E os resultados indicaram que a abordagem API *Slicer* teve o maior nível de precisão em duas das três aplicações alvo utilizadas por este estudo, se mostrando uma alternativa viável.

O estudo está dividido conforme a seguinte estrutura: a Seção 2 conterá o referencial teórico, com os principais conceitos para entendimento do estudo proposto; a Seção 3 abordará os trabalhos relacionados, explorando o processo de seleção utilizado e também realizando um comparativo destes com o presente; A Seção 4 tratará sobre a visão do processo, arquitetura utilizada, algoritmos criados e aspectos de implementação. Seção 5 abordará a descrição da aplicação alvo, métricas, procedimento de avaliação, resultados e limitações do presente trabalho. E por fim, a Seção 6 traça algumas conclusões e trabalhos futuros.

Figura 1 – Exemplo de uma API monolítica



Fonte: Elaborado pelo autor.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção aborda os conceitos teóricos usados durante a construção e desenvolvimento do estudo.

2.1 Microsserviço

Microsserviços são serviços que expõe um contrato, para que outros serviços possam consumi-lo, sem que haja forte dependência entre eles, visto que nenhum dos dois conhece a implementação um do outro. Ele tem como premissa ser pequeno (micro), para que uma equipe pequena consiga gerenciar seu código. E seu crescimento é delimitado pela funcionalidade de negócio que ele se propõe a atender, para que ele não cresça indefinidamente a ponto de sofrer os problemas de uma aplicação monolítica, ou seja, grande, complexo e com alto risco ao fazer uma alteração. (NEWMAN, 2021)

As principais características dos microsserviços são: (1) heterogeneidade tecnológica, ou seja, cada microsserviço criado pode ser feito com uma stack de tecnologia que mais faz sentido para o negócio. (2) Autonomia, o microsserviço pode ser implantado e reimplantado sem depender de outros serviços para que isso aconteça. O que permite que mais recursos possam ser adicionados e tecnologias alteradas mais facilmente, desde que o contrato utilizado pelos consumidores não seja alterado. E (3) comunicação leve, os serviços se comunicam via REST,

usando o HTTP ou um serviço de mensageria assíncrona, o que evita o alto acoplamento entre os mesmos (GUPTA, 2015).

2.2 Sistema monolítico

Segundo Lucio et al. (2017), um sistema monolítico é uma aplicação onde a interface com o usuário, a lógica de negócio e persistência de banco de dados se localizam em uma única unidade. E essa unidade é independente e abrange não só um, mas todos os passos para completar uma funcionalidade macro. Já para Urdangarin, Farias e Barbosa (2021a), aplicações monolíticas são usualmente caracterizadas como um único artefato de software executável, constituído de módulos altamente acoplados e requisitos implementados de forma entrelaçada e distribuída entre os módulos da aplicação.

As principais vantagens do sistema monolítica são: (1) compreensão total do fluxo macro, visto que a aplicação abrange todas as etapas da funcionalidade, (2) bom desempenho, por não precisar fazer várias chamadas a serviços externos e (3) independência de outros serviços, pois todas as funcionalidades que a aplicação precisa estão dentro dela (WAHLSTRÖM, 2019). E como desvantagens pode-se destacar: (1) problema de manutenção, à medida que o monolito aumenta de tamanho, (2) dificuldade de inserir novas tecnologias, (3) maior complexidade para introduzir uma nova funcionalidade, pois pode haver efeitos indesejados em funcionalidades já existentes. (JÚNIOR, 2017)

2.3 Desenvolvimento orientado a feature

Desenvolvimento orientado a feature é um paradigma para a construção, customização e síntese de sistemas de software. A ideia básica do desenvolvimento orientado a *feature*, é decompor sistemas de software em *features*, para fornecer opções de configuração e facilitar a geração de softwares com base em uma seleção de *features*. (THüM et al., 2014).

Do ponto de vista abstrato, uma feature é um aspecto, qualidade ou característica visível ao usuário proeminente ou distintivo de um sistema de software. Já do ponto de vista técnico, a feature é uma estrutura que estende e modifica a estrutura de um determinado programa, a fim de satisfazer um requisito de um *stakeholder*, implementar e encapsular uma decisão de projeto. E oferecer uma opção de configuração (APEL; KÄSTNER, 2009). Este estudo olhará a feature do ponto de visto técnico, ou seja, como uma funcionalidade, pois é implementada para satisfazer os requisitos de negócio. As razões pelas quais foi utilizado a ideia de feature para decompor APIs monolíticas em microsserviços foram: (1) obtenção do comportamento completo do software, em relação a cada funcionalidade que ele possui; (2) possibilidade de reutilização da feature para outros sistemas; (3) facilidade de manutenção, visto que o desenvolvedor precisará se preocupar com uma feature e não com um sistema inteiro;

3 TRABALHOS RELACIONADOS

Essa seção tem como propósito discutir os trabalhos relacionados, apresentar uma análise comparativa dos mesmos, bem como apontar oportunidades de pesquisa. Os trabalhos relacionados foram selecionados a partir de dois repositórios digitais: *Google Scholar* e *IEEE*. Esses repositórios foram utilizados porque possuem um vasto conteúdo sobre engenharia de software. Os autores Lazzari e Farias (2022) demonstraram a utilidade de tais repositórios para o propósito desta seção. No caso do repositório da IEEE, foi aplicada uma string de busca, filtrando pelo período de 2019 à 2021, como pode ser visto abaixo:

("All Metadata":monolithic) AND ("All Metadata":decomposition) AND ("All Metadata": microsservice)

Uma string de busca é um conjunto de termos utilizados para fazer uma pesquisa mais restrita e direcionada. Dessa pesquisa foram identificados 16 trabalhos, mas apenas 5 foram selecionados por duas razões: (1) similaridade de propósito e (2) ano que o estudo foi feito.

3.1 Análise dos trabalhos relacionados

(ASSUNÇÃO et al., 2022). Este estudo apresenta o *toMicroservice*, uma abordagem multi-objetivo, composta por 5 objetivos, para identificar microsserviços de sistemas monolíticos. Os objetivos utilizados foram: acoplamento, coesão, modularização de recursos, sobrecarga de rede e reutilização. Uma ferramenta foi criada e aplicada em um sistema monolítico da indústria de óleo e gás para identificar se os objetivos escolhidos são conflitantes e se a abordagem tem melhores resultados do que a busca aleatório. A avaliação foi feita através do teste de correlação de Spearman. Os resultado mostraram que os objetivos coesão e acoplamento são inversamente proporcionais, ou seja, uma vez que à medida de acoplamento diminui, a coesão aumenta e vice-versa. Além disso, a modularização de recursos afeta um pouco as funções de coesão e sobrecarga de rede. Em relação a comparação com a busca aleatório, o *toMicroservice* obteve melhores resultados, alcançando uma maior diversidade de soluções não dominadas com diferentes compromissos entre os objetivos. Entretanto não foi especificado se qualquer aplicação monolítica pode fazer uso da ferramenta e se é possível controlar a granularidade das recomendações de microsserviços gerados.

(FILIPPONE et al., 2021). Este estudo propõe uma abordagem que visa a identificação dos microsserviços, extração de arquitetura e implementação de microsserviços de forma automatizada. Na fase 1 (identificação de microsserviços), é recebido como entrada o código fonte do sistema legado e após inspeção do código a nível de método, é gerado uma representação do sistema em forma de grafo. Na fase 2 (extração de arquitetura) é executado uma técnica de otimização combinatória no grafo gerado na etapa anterior, para obter maior coesão, menor acoplamento e mínima sobrecarga de comunicação entre os microsserviços identificados. E na

fase 3 (implementação de microsserviços) é executado o algoritmo de síntese, no resultado gerado pela fase 2, para gerar automaticamente o código fonte dos microsserviços. No entanto, não houve nenhuma avaliação referente a decomposição gerada, não foi mencionado se a ferramenta pode ser usada em qualquer aplicação monolítica ou se é possível ajustar a granularidade da recomendação sugerida.

(IVANOV; TASHEVA, 2021). Este artigo discorre sobre um procedimento de decomposição de aplicações monolíticas em microsserviços, baseado na estratégia de refatoração. Este procedimento consiste em oito passos para decompor aplicações monolíticas, sem causar inatividade. Os oito passos para decompor uma aplicação monolítica, sem causar inatividade são: (1) avaliação dos benefícios de migração do monolítico para o microsserviço, (2) definição dos limites de contexto, (3) separação das interfaces do usuário, (4) remoção da interface do usuário do monólito, (5) separação de um serviço, (6) sincronização dos dados, (7) redirecionamento do tráfico para o novo serviço e (8) repetição do processo para cada contexto. Para avaliar a ideia, as oito etapas foram aplicadas em um estudo de caso, porém, não foi implementado nenhuma ferramenta para automatizar o processo e o desenho da estrutura do microsserviço foi feito por um especialista, logo, não há como saber quais critérios ele utilizou para a concepção da arquitetura.

(KIRBY et al., 2021). Este artigo apresenta um estudo exploratório multi método, que usou a experiência de 10 profissionais experientes em extração de microsserviços, para tentar responder duas perguntas: (1) qual a aplicabilidade e utilidade de diferentes tipos de relacionamento durante o processo de extração? E (2) quais recursos beneficiariam a automação de ferramentas de extração que utilizam relacionamentos elemento a elemento? Uma ferramenta online que utiliza grafos foi criada e testada por 10 profissionais da área de extração de microsserviços. E o resultado levantado foi que os profissionais preferem uma ferramenta de análise que possa ajudar a examinar e experimentar diferentes relacionamentos e decomposições, pois eles consideram vários tipos de relacionamento durante o processo de extração. No entanto, o estudo não disse se a aplicação desenvolvida poderia ser aplicado a qualquer aplicação monolítica e não houve comparação dos resultados da abordagem proposta, com a abordagem criada por outro estudo.

(ZHAO; ZHAO, 2021). Este estudo propõe uma abordagem que extrai microsserviços do sistema monolítico orientado a objeto, através da combinação entre a identificação do domínio e a divisão lógica da aplicação alvo. A identificação de domínio considera as informações do banco de dados da aplicação e a divisão lógica pondera sobre o código fonte do sistema legado. As principais etapas do processo de extração de microsserviços são compostas pela (1) divisão de domínio, (2) divisão de camada, (3) divisão de negócios e (4) fusão de clusters. E para avaliar o processo, uma ferramenta foi criada e utilizada em dois sistemas. O primeiro sistema foi um ecommerce online chamado *ShopMaster*, um sistema monolítico em que sua estrutura é baseada em um *framework* de software. E o segundo foi um blog chamado Solo, um sistema construído no padrão MVC, mas sem ter a base de nenhum framework, ou seja, sua estrutura

não é clara. O resultado da extração de candidatos a microsserviços do *ShopMaster* mostrou que apenas um candidato não foi extraído. Já em relação ao *Solo*, o número de candidatos a microsserviços foi inconsistente em relação ao número de serviços. Além disso, este projeto não fez uso de nenhuma métrica para comprovar que a extração de microsserviços foi significativa e a ferramenta não é *open source*.

(SANTOS; PAULA, 2020). Este estudo propõe uma ferramenta que faz a análise do código fonte de uma aplicação monolítica e sugere decomposições em microsserviços. A ferramenta faz uso da combinação do algoritmo *Monobreak*, proposto pelo autor Rocha (2018), com a estratégia de acoplamento lógico dos autores Mazlami, Cito e Leitner (2017), para sugerir microsserviços. A decomposição de aplicações monolíticas em microsserviços é feita através da análise estática no código, considerando classes, métodos e histórico de alterações. Durante a análise estática no código, é realizado o agrupamento dos serviços considerando o quão similares eles são. A similaridade é calculada identificando os itens em comum entre os serviços. Esses itens são: as classes, os métodos e os arquivos fontes que foram alterados juntos no histórico de modificações do controle de versão. E para avaliar a qualidade das recomendações, a ferramenta foi aplicada em três sistemas monolíticos. As métricas utilizadas para a avaliação foram o coeficiente de silhueta e granularidade. Entretanto não foi citado se a abordagem pode ser aplicada em uma API monolítica e a abordagem proposta não foi comparada com a abordagem de outro estudo.

(SHEIKH; BS, 2020). Este estudo utiliza a abordagem orientada a dados, para identificar candidatos a microsserviços, com base em log de arquivos coletados durante o tempo de execução. Essa abordagem é composta por 6 etapas: (1) Análise de caminho para a execução. Nesta etapa é identificado através do log da aplicação, as classes e tabelas de banco de dados utilizados e gerado uma representação gráfica; (2) Frequência dos Módulos. Nesta fase é identificado quais formas de execução são utilizadas com mais frequência e quais módulos são ocasionalmente ou nunca são utilizados durante o tempo de execução; (3) Identificação e remoção da dependência circular. Nesta etapa é aplicado o algoritmo *Union Find* para identificar as dependências circulares e após, as dependências são removidas, gerando um gráfico acíclico; (4) Gráfico de pré-processamento. Nesta etapa o gráfico acíclico é processado; (5) Seleção da escolha de decomposição usando o gráficos. Neste etapa é aplicado um algoritmo baseado em hash para saber se o grafo do sistema possui alguma dependência. Caso sim, a aplicação monolítica não pode ser decomposta; (6) Seleção de uma solução que resolve o problema. Neste etapa é disponibilizado quais são os microsserviços sugeridos. A proposta foi avaliada em um sistema financeiro, no entanto, nenhuma ferramenta foi criada para automatizar o processo, não houve menção se qualquer sistema monolítico poderia fazer uso da abordagem e não foi dito como fica o resultado final da recomendação, e se é possível ajustar a granularidade da mesma.

(ROCHA, 2018). Este estudo apresenta a *Monólise*. A *Monólise* é uma técnica semiautomática que utiliza um algoritmo sensível à arquitetura da aplicação, para recomendar microsserviços. A *Monólise* é uma técnica agnóstica a linguagem de programação, que faz uso de

três parâmetros de entrada (configuração do sistema, rastro de execução da funcionalidade da aplicação monolítica e valor de similaridade), para que o algoritmo *MonoBreak* possa recomendar microsserviços. A recomendação da *Monólise* permite demonstrar a nível de código, quais as classes e métodos das funcionalidades, terão de ser migrados para novos microsserviços. 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. Entretanto, a ferramenta desenvolvida não é *open source*, impedindo a evolução do trabalho e a abordagem proposta não foi comparada com nenhuma abordagem de outro estudo.

3.2 Análise comparativa dos trabalhos relacionados

Critério de comparação. Foram definidos seis Critérios de Comparação (CC), para identificar similaridades e diferenças entre o trabalho proposto e os artigos selecionados. Está forma de comparação já foi validada em estudos anteriores (LAZZARI; FARIAS, 2022; RUBERT; FARIAS, 2022; FARIAS et al., 2019; URDANGARIN; FARIAS; BARBOSA, 2021b; BISCHOFF; FARIAS, 2020) e se mostrou efetiva para identificar oportunidades de pesquisa. Os critérios são descritos a seguir:

- **Métricas (CC1):** este critério verifica se o estudo utilizou alguma métrica para avaliar o resultado obtido;
- **Suporte a decomposição de APIs monolíticas (CC2):** este critério avalia se a abordagem proposta consegue decompor uma API monolítica em microsserviços;
- **Suporte de ferramenta (CC3):** este critério avalia se o estudo elaborou algum protótipo para automatizar (mesmo que de forma parcial) o processo de geração de microsserviços;
- **Ferramenta open source (CC4):** este critério avalia se o estudo elaborou alguma ferramenta *open source*, para que a comunidade consiga testar por conta própria o projeto elaborado e criar aprimoramentos com base no código fonte exposto.
- **Método de avaliação (CC5):** este critério avalia se foi utilizado um estudo de caso, onde a abordagem proposta pelo artigo, foi comparada com a abordagem de outro estudo.
- **Granularidade dos microsserviços (CC6):** este critério avalia se o estudo possui alguma opção que permite ajustar a granularidade das recomendações, para que se possa ter microsserviços compostos por muitos (granularidade grossa) ou poucos (granularidade fina) serviços.

Oportunidades de pesquisa. A Tabela 1 apresenta uma análise comparativa. Está tabela contrasta os trabalhos relacionados com o proposto, destacando o que é similar e diferente

Tabela 1 – Análise comparativa dos trabalhos relacionados selecionados

Trabalho Relacionado	Critério de Comparação					
	CC1	CC2	CC3	CC4	CC5	CC6
Trabalho Proposto	●	●	●	●	●	●
(ASSUNÇÃO et al., 2022)	●	○	●	○	○	○
(FILIPPONE et al., 2021)	○	○	○	○	○	○
(IVANOV; TASHEVA, 2021)	●	○	○	○	○	○
(KIRBY et al., 2021)	●	○	●	●	○	●
(ZHAO; ZHAO, 2021)	○	◐	●	○	○	●
(SANTOS; PAULA, 2020)	●	○	●	●	○	●
(SHEIKH; BS, 2020)	●	○	○	○	○	○
(ROCHA, 2018)	●	●	●	○	○	●

● Atende Completamente ◐ Atende Parcialmente ○ Não Atende

Fonte: Elaborado pelo autor.

entre eles. A lacuna observada é que dos 48 critérios de comparação, apenas 37.5% deles foram totalmente satisfeitos, resultando nas seguintes oportunidades de pesquisa: (1) nenhum trabalho relacionado tem como premissa decompor uma API monolítica (explorado na Seção 4), (2) nenhum trabalho comparou a abordagem proposta com a abordagem de outro estudo (explorado na Seção 5.4). E (3) apenas 2 estudos (KIRBY et al., 2021) e (SANTOS; PAULA, 2020) criaram uma ferramenta *open source*, onde se pode ver a implementação e possíveis melhorias (explorado na Seção 4.3).

4 ABORDAGEM PROPOSTA

Esta seção descreve a API *Slicer*, uma abordagem baseada em *features* para decomposição de APIs monolíticas. A Seção 4.1 descreve a visão do processo, a Seção 4.2 fala sobre a arquitetura proposta, a Seção 4.3 detalha os algoritmos propostos e a Seção 4.4 discorre sobre os aspectos de implementação.

4.1 Visão do processo

Está seção descreve as fases utilizadas para que a abordagem API *Slicer* consiga decompor uma API monolítica em microsserviços. O nome da abordagem vem da junção do termo API (a aplicação alvo dessa abordagem) + Slicer (propõe uma decomposição, fazendo com que a API se torne menor). A API *Slicer* visa apoiar a equipe de desenvolvimento, propondo sugestões de microsserviços, caso a equipe entenda que está na hora da API monolítica ser decomposta, por deter muitas funcionalidades entrelaçadas umas nas outras.

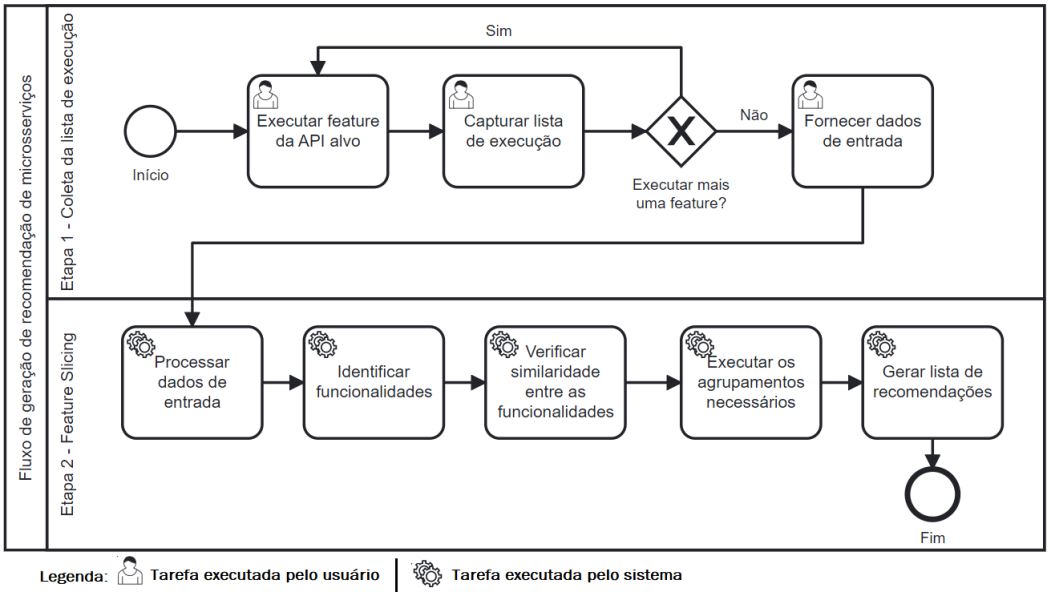
O fluxo completo do processo de extração dos microsserviços, pode ser visto na Figura 2. E abaixo, todas as etapas do processo são detalhadas:

- **Etapa 1 - Captura da lista de execução:** Nesta etapa o usuário irá executar todas as funcionalidades da API monolítica alvo, que ele deseja que a API *Slicer* avalie. A cada

execução de uma nova funcionalidade, o usuário capturará o *trace* de execução e salvará dentro de um arquivo .txt, que terá o nome da funcionalidade executada. O *trace* são linhas que mostram qual método e classe foram chamados, quando determinada funcionalidade foi executada. O fluxo de execução das funcionalidades, captura de *trace* e criação de um arquivo .txt contendo o *trace* de execução, se repetirá, até que o usuário tenha abordado todas as funcionalidades da API alvo.

- **Etapa 2 - Feature Slicing:** Etapa em que a API é avaliada e a API *Slicer* sugere decomposições em microsserviços. A API *Slicer* vai receber como parâmetros de entrada: (1) os *traces* de execução feitos na etapa 1, (2) os pacotes que deverão ser considerados para gerar a recomendação e (3) o nível de similaridade que deve ser aplicado na recomendação. Nível de similaridade é um valor de corte que diz quando duas ou mais funcionalidades devem estar presentes no mesmo microsserviço. Após o recebimento dos parâmetros de entrada, os arquivos são descompactados e convertidos em funcionalidades. Cada funcionalidade é composta por classes e seus respectivos métodos. E para identificar o quão similares cada funcionalidade é em relação a outra, é feito uma comparação entre o nome das classes de ambas as funcionalidades. Quanto mais classes iguais, maior o nível de similaridade. Após a identificação do nível de similaridade entre as funcionalidades, é gerado um agrupamento. Nesse agrupamento, todas as funcionalidades que alcançarem um nível de similaridade igual ou maior ao fornecido pelo usuário, estarão presentes no mesmo microsserviço. E por último, é impresso a lista de recomendação de microsserviços, composta por classes, métodos e quais funcionalidades devem estar presente no microsserviço.

Figura 2 – Diagrama geral do processo



Fonte: Elaborado pelo autor.

4.2 Visão de arquitetura

A Figura 3 apresenta uma arquitetura baseada em componentes. Cada componente possui um propósito implícito para implementar o processo proposta na Figura 2. A seguir, cada componente de arquitetura é descrito.

- **Interface do usuário:** Este componente é o terminal onde o usuário vai fornecer: (1) o valor do nível de similaridade, (2) os pacotes que devem ser considerados na recomendação e (3) o caminho da pasta zip que contém todos os arquivos de *traces* de execução capturados da API monolítica. O valor de similaridade é o valor de corte que dirá se duas ou mais funcionalidades ficarão no mesmo microsserviço, ou irão para microsserviços individuais. Exemplo: se o valor de similaridade for 67% e duas funcionalidades tiverem uma similaridade igual ou superior a 67%, ambas as funcionalidades ficarão no mesmo microsserviço. E os pacotes que devem ser considerados, são os pacotes que a API *Slicer* irá se basear, para dizer o quão similares as funcionalidades são.

- **Extração de dados:** Este componente é responsável por descompactar os arquivos da pasta zip fornecida pelo usuário, em um diretório que a aplicação tem acesso. E fazer o mapeamento desses arquivos descompactados em funcionalidades, para servirem de insumo para os demais componentes da arquitetura.

- **Identificação de similaridades:** Este componente é responsável por comparar o nome de cada classe da funcionalidade A, com as classes da funcionalidade B. E com base nisso, executar o cálculo de similaridade, para identificar o quão próximas as funcionalidades são. Esse processo de comparação é feito com todas as funcionalidades mapeadas pelo *trace* fornecido pelo usuário. E após a conclusão, o resultado das similaridades é armazenado em um dicionário.

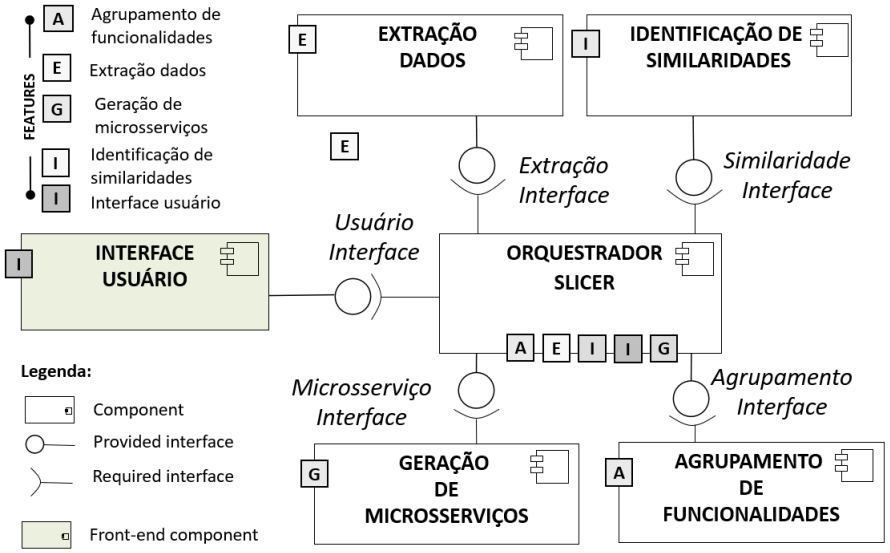
- **Agrupamento de funcionalidades:** Este componente é responsável por passar por todas as funcionalidades mapeadas e verificar se alguma funcionalidade do dicionário tem um nível de similaridade igual ou maior do que o fornecido pelo usuário. Caso a resposta seja positiva, as funcionalidades ficarão juntas no mesmo microsserviço. Do contrário, as funcionalidades irão para microsserviços diferentes.

- **Geração de microsserviços:** Este componente é responsável por montar a lista de recomendação de microsserviços, a lista de funcionalidades que devem permanecer na API e imprimir no terminal estas listas. Cada recomendação de microsserviço é composta por funcionalidades e cada funcionalidade possui suas respectivas classes e métodos.

- **Orquestrador Slicer:** Este componente é responsável por fazer a intermediação entre os componentes: (1) Interface do usuário, (2) Extração de dados, (3) Identificação de similaridades, (4) Agrupamento de funcionalidades e (5) Geração de microsserviços, para

conseguir sugerir recomendações de microsserviços e indicar quais funcionalidades devem permanecer na API monolítica.

Figura 3 – Diagrama de componentes



Fonte: Elaborado pelo autor.

4.3 Algoritmos

Esta seção descreve o algoritmo utilizado para fazer a recomendação de microsserviços (Algorithm 1). O algoritmo é composto por cinco subrotinas: `setInputData`, `getFileNames`, `convertFileToFeatures`, `createSimilarityMap` e `groupFeatures`. E tem como objetivo transformar a pasta zip com os *traces* de execução da API alvo, o nível de similaridade e pacotes providos pelo usuário, em recomendações de microsserviços. Abaixo cada uma das cinco subrotinas do algoritmo é descrita com maiores detalhes:

- **setInputData:** Algorithm 2 é a primeira sub-rotina a ser executada pelo algoritmo de recomendação de microsserviços e tem como objetivo capturar os dados de entrada fornecidos pelo usuário, via terminal. Os dados de entrada são: (1) caminho onde se localiza a pasta zip com todos os *traces* recolhidos da API alvo (linha 4); (2) os pacotes que devem ser considerados no momento de identificar o nível de similaridade entre as funcionalidades (linha 6) e o (3) valor de similaridade, que é o valor de corte que dirá se duas ou mais funcionalidades irão compartilhar o mesmo microsserviço (linha 8).

- **getFileName:** Algorithm 3 é a segunda sub-rotina executada pelo algoritmo de recomendação de microsserviços. Recebe como parâmetro o caminho onde está a pasta zip, que contém todos os *traces* de execução da API alvo e tem como objetivo descompactar a

pasta zip e informar a localização dos arquivos descompactados. O processo de descompactação inicia-se na linha 2, onde é setado o diretório onde os arquivos descompactados serão postos. Na linha 4 é executado a descompactação, passando uma cópia de todos os arquivos presentes dentro da pasta zip, para dentro do diretório fornecido na linha 2. E na linha 5 é concatenado o diretório com o nome dos arquivos, para que se tenha a localização completa dos arquivos descompactados.

- **convertFilesToFeatures:** Algorithm 4 é a terceira sub-rotina executada pelo algoritmo de recomendação de microsserviços. Essa subrotina recebe como parâmetros a lista contendo a localização dos arquivos descompactados e a lista de pacotes que serão priorizados para a recomendação dos microsserviços. Essa subrotina tem como objetivo converter os arquivos descompactados em funcionalidades que serão utilizados por subrotinas posteriores. As funcionalidades são compostas pelo nome da funcionalidade e lista de classses. E cada classe possui um pacote e seus respectivos métodos. O processo de conversão dos arquivos em funcionalidades do sistema inicia na linha 7, onde uma localização de arquivo por vez é convertida em um arquivo (linha 8), é identificado o nome da funcionalidade (linha 9), inserido o arquivo dentro de um buffer (linha 11). E para cada linha desse buffer (linha 13) é identificado o nome da classe (linha 15), o nome do pacote (linha 16) e se o pacote identificado for igual a algum dos pacotes recebidos por parâmetro (linha 17), é construído a classe com seu respectivo nome (linha 19), pacote (linha 20), métodos (linha 23) e adicionado dentro de um dicionário, onde a chave é o nome da funcionalidade (linha 37).

- **createSimilatiryMap:** Algorithm 5 é a quarta sub-rotina executada pelo algoritmo de recomendação de microsserviços. Essa subrotina recebe como parâmetros as funcionalidades mapeadas pelo sistema no passo anterior e tem como objetivo criar um dicionário onde a chave é o nome da funcionalidade e o valor são os níveis de similaridade das outras funcionalidades, em relação a essa. O processo de criação do dicionário de similaridades entre as funcionalidades inicia na linha 3 e 5, onde para cada funcionalidade, é pego o nome da funcionalidade e as classes que ela possui. E caso as funcionalidades não sejam iguais (linha 6), é buscado todas as classes que são comuns entre ambas as funcionalidades (linha 7), calculado a similaridade (linha 8) e criado a estrutura que possui o nome da funcionalidade e seu nível de similaridade em relação a outra funcionalidade (linha 9) e adicionado no dicionário (linha 12).

- **groupFeatures:** Algorithm 6 é a quinta sub-rotina executada pelo algoritmo de recomendação de microsserviços. Essa subrotina recebe como parâmetros (1) o dicionário contendo os níveis de similaridade das funcionalidades e (2) um dicionário com a estrutura das funcionalidades (nome, métodos, classes e pacote). Essa subrotina tem como objetivo gerar os microsserviços recomendados. O processo inicia na linha 3, onde para cada similaridade mapeada, é pego a linha (nome da funcionalidade) e as colunas (contém o nível

de similaridade das funcionalidades). Na linha 4 é recuperado o nome da funcionalidade. Na linha 5 é pego todas as funcionalidades que tiveram um nível de similariadade alto, em relação a funcionalidade obtida na linha 4. Caso a lista de microsserviços esteja vazio (linha 6), é gerado a estrutura de classes do microsserviço (linha 8), para a funcionalidade obtida linha 4. Após, é gerado a estrutura de classes (linha 9) das funcionalidades filtradas na linha 5 e gerado o microsserviço contendo essa estrutura de classes (linha 10). Porém, se a lista de microsserviços não estar vazia (linha 11), é verificado se a funcionalidade já está presente dentro do microsserviço (linha 12 e 13). Após isso é escolhido qual é a estratégia mais adequada (linha 14) para atualizar a estrutura de microsserviços (linha 15). E por último, é retornado todos os microsserviços identificados (linha 19).

Algorithm 1: Recomendação de microsserviços

```
1  function executeSlicerRecommendation()
2      consoleService.setInputData();
3      files ←fileService.getFileNames(consoleService.getReadDirectory());
4      packages
         ←Arrays.asList(consoleService.getImportantPackages().split(",","));
5      functionalities ←featureService.convertFilesToFeatures(files, packages);
6      similarityMap ←similarityService.createSimilarityMap(functionalities);
7      completeFuncionality ←featureService.convertFilesToFeatures(files);
8      similarity ←consoleService.getSimilarityValue();
9      return
         microservice.groupFeatures(similarityMap, completeFunctionalities, similarity);
10 end
```

Algorithm 2: Recomendação microsserviços - Sub-rotina1

```
1  function setInputData ()
2      input ←newScanner(System.in);
3      System.out.println(Zipfilepath :);
4      readDirectory ←input.nextLine();
5      System.out.println(Enterpackages :);
6      packages ←input.nextLine();;
7      System.out.println(Entersimilarity :);
8      similarity ←Integer.parseInt(input.nextLine());
9  end
```

4.4 Aspectos de implementação

Para implementar a abordagem API *Slicer*, foi criado uma ferramenta *open source*[3]. A ferramenta possui como tecnologias: (1) A linguagem Java 11, que é uma linguagem de progra-

[3]Repositório API *Slicer*: <https://github.com/CarlosFernandoXavier/API-Slicer>.

Algorithm 3: Recomendação microsserviços - Sub-rotina2

```
1 function getFileNames (readDirectory)
2     destinationDirectory ←src/main/resources/output;
3     fileNames ←list();
4     unzip(readDirectory, destinationDirectory, fileNames);
5     return getPathArquivos(destinationDirectory, fileNames);
6 end
```

mação orientada a objetos, usualmente utilizada em aplicações back-end; (2) biblioteca Jackson Databind, para converter objetos em estruturas json e com isso, apresentar recomendações de forma mais amigável ao usuário; (3) o gerenciador de dependências Maven, para gerenciar todas as bibliotecas inseridas no projeto. E como ferramentas de suporte foram utilizados: (1) a IDE Intellij, ambiente utilizado para fazer a implementação de código; (2) Github, repositório para armazenar o código fonte da aplicação e (3) Visual VM, ferramenta gráfica que foi utilizada para capturar os *traces* de execução das APIs alvo testadas.

5 AVALIAÇÃO

Esta seção descreve a avaliação para executar o estudo de caso, que é uma metodologia em engenharia de software, que se provou efetiva em pesquisas experimentais (**??**). A Seção 5.1 apresenta as descrições das aplicações alvo, as quais serão submetidas a este projeto, a Seção 5.2 apresenta as métricas utilizadas para avaliar o resultado, a Seção 5.3 descreve como será avaliado o resultado e a Seção 5.4 fala sobre os resultados obtidos.

5.1 Descrição da aplicação alvo

Está seção descreve as três aplicações alvo deste estudo, são elas:

- **Blog API**: É uma API que tem como principais funções fazer login, criar um post e comentários para o mesmo, utilizando como critério de seguraça, a autenticação JWT. Essa aplicação foi escolhida por possuir vários endpoints, fazer uso do springboot e ter uma boa documentação para reprodução na máquina local. A aplicação pode ser encontrado no GitHub[4]. As tecnologias aplicadas foram Springboot, Postgres, Java 11, Lombok, Maven, JJWT, JPA, SpringSecurity e ModelMapper.

- **Shopping cart API**: É uma API utilizada para fazer compras online. E Entre as funcionalidades estão: adicionar produto ao carrinho, buscar carrinho, buscar perfil do usuário, buscar todos os produtos e finalizar pedido, além de fazer uso autenticação JWT. A aplicação foi escolhida por possuir vários endpoints, ter uma documentação e fazer uso do springboot. As tecnologias utilizas foram: Springboot, Postgres, Java 11, Maven, JJWT,

[4]Repositório Blog-API: <https://github.com/RameshMF/springboot-blog-rest-api>.

Algorithm 4: Recomendação microsserviços - Sub-rotina3

```
1  function convertFilesToFeatures (functionalityFiles, packages)
2      functionalityMaps ←map();
3      class ←null;
4      classes ←null;
5      file ←null;
6      functionalityName ←null;
7      foreach fileName in functionalityFiles do
8          file ←File(fileName);
9          functionalityName
             ←file.getName().substring(0, file.getName().lastIndexOf("."));
10         classes ←list();
11         bufferedReader ←BufferedReader(FileReader(file));
12         auxiliary ←null;
13         while (auxiliary = bufferedReader.readLine()) ≠ null do
14             line ←auxiliary.split(",");
15             className ←line[0].substring(line[0].lastIndexOf(" : ") + 2);
16             packageName
                 ←className.substring(0, className.lastIndexOf("."));
17             if isAuthorizedPackage(packageName, packages) then
18                 class ←Class();
19                 classe.setClassName(className);
20                 classe.setPackageName(packageName);
21                 methodList ←list();
22                 methodList.add(line[1].substring(line[1].lastIndexOf(" : ") + 2));
23                 classe.setMethodName(methodList);
24                 classIndex ←getClassIndex(classes, class);
25                 if Objects.isNull(classIndex) then
26                     classes.add(classe);
27                 else
28                     classA ←classes.get(classIndex);
29                     classes.remove(classeA);
30                     classA.addMethodName(line[1].substring(line[1].lastIndexOf(" :
                       ") + 2));
31                     classes.add(classIndex, classA);
32                 end
33             end
34         end
35         functionalityMaps.put(functionalityName, classes);
36     end
37     return functionalityMaps;
38 end
```

Algorithm 5: Recomendação microsserviços - Sub-rotina4

```
1  function createSimilarityMap (functionalities)
2      similarityMap ←map();
3      foreach (functionality1, class1) in functionalities do
4          columns ←list();
5          foreach (functionality2, class2) in functionalities do
6              if !functionality1.equals(functionality2) then
7                  equalClasses ←intersection(classes1, classes2);
8                  similarity ←Double.valueOf(equalClasses.size() * (0.1 *
                     equalClasses.size()))/Double.valueOf(classes1.size() * (0.1 *
                     classes1.size())) * 100;
9                  columns.add(Column(functionality2, similarity));
10             end
11         end
12         similarityMap.put(functionality1, columns);
13     end
14     return similarityMap;
15 end
```

Algorithm 6: Recomendação microsserviços - Sub-rotina5

```
1  function groupFeatures(similarityMap, funcionalidadesMap)
2      microservices ←list();
3      foreach (row, columns) in similarityMap do
4          functionalities ←row;
5          filteredColumns ←filtredColumnsByThreshold(colmns);
6          if microservices.isEmpty() then
7              classResponses ←list();
8              generateMicroserviceClasses(functionalitiesMap, functionalities, classResponses)
9              functionalities ←filterColumn(functionalitiesMap);
10             microservices.add(generateMicroservie(functionalities, classResponses));
11         else
12             microserviceOneIndex
                 ←getMicroserviceIndex(microservices, functionalities);
13             microserviceTwoIndex
                 ←getMicroserviceIndex(microservices, filteredColumns);
14             microserviceFacade.getStrategyList().stream().filter(strategy− >
               strategy.isCompatible(microserviceOneIndex, microserviceTwoIndex))
15             .forEach(strategy− >
               strategy.genarateRecommentation(microservices, colunasFiltradas,
16             funcionalidadesMap, finalFunctionalities));
17         end
18     end
19     return microservices;
20 end
```

JPA e SpringSecurity. A aplicação está disponibilizada no GitHub[5]. Em função deste trabalho utilizar apenas APIs , a camada responsável pelo frontend foi totalmente removida.

- **Pedido API**: É um CRUD de pedidos que utiliza a arquitetura hexagonal, faz uso de tecnologias como Postgres, Java 11, SpringBoot, Gradle, Lombok e Swagger. E as principais funcionalidades comportadas são (1) buscar filial, (2) criar filial, (3) deletar filial, (4) adicionar itens a filial, (5) criar item, (6) consultar item, (7) adicionar itens a lista de preço, (8) deletar itens da lista de preço, (9) criar lista de preço, (10) buscar lista de preço, (11) gravar pagamento, (12) buscar pagamento, (13) remover pagamento, (14) atualizar pagamento e (15) geração do relatório de estoque por filial. Entretanto, para fins de testes apenas 4 funcionalidades foram utilizadas, são elas: buscar itens, buscar itens pelo id, buscar filiais e gerar relatório por filial. A aplicação foi escolhida por possuir vários endpoints, o autor ter domínio sobre seu funcionamento e por fazer uso do springboot. A aplicação está disponibilizada no GitHub[6].

5.2 Métricas

Para que se posso avaliar os resultados gerados pela abordagem API *Slicer* e *Monólise*, o presente trabalho fará uso das medidas de precisão e *recall*, para identificar a qualidade do que está sendo recomendado e a quantidade de recomendações corretas que está sendo feita, em relação ao resultado desejado.

Precisão. É a medida de eficácia que tem como objetivo excluir os elementos não relevantes do conjunto recuperado. E é visto como a proporção do número elementos relevantes, vindos dos elementos recuperados (BUCKLAND; GEY, 1994). Neste estudo, a precisão será utilizado para identificar (1) quantos microsserviços recomendados e (2) funcionalidades que devem permanecer na API, estão corretos, em relação ao resultado gerado. A seguir é apresentado a fórmula da precisão (ROELLEKE, 2013).

$$precision(q) = \frac{P(retrieved, relevant|q)}{P(retrieved|q)} \tag{1}$$

Recall. É uma medida de eficácia que inclui itens relevantes no conjunto recuperado e é visto como a proporção número de elementos relevantes recuperados, em relação ao total de itens relevantes (BUCKLAND; GEY, 1994). Neste trabalho, a revocação será utilizada para identificar (1) a quantidade de microsserviços recomendados e (2) funcionalidades que devem permanecer na API, estão corretos, em relação ao resultado esperado. A seguir é apresentado a fórmula *recall* (ROELLEKE, 2013).

$$recall(q) = \frac{P(retrieved, relevant|q)}{P(relevant|q)} \tag{2}$$

[5]Repositório do Shopping cart: <https://github.com/zhulinn/SpringBoot-Angular7-Online-Shopping-Store>.
[6]Repositório do Peido API: <https://github.com/CarlosFernandoXavier/hexagonal-architecture>.

5.3 Processo experimental

A Figura 4 mostra todos os passos seguidos para executar a análise e comparação dos resultados obtidos pela abordagem API *Slicer* e a abordagem *Monólise*, quando foram submetidas a três aplicações alvo diferentes. A forma de demonstração do processo experimental foi baseado no trabalho dos autores Farias et al. (2012). As etapas do processo experimental são apresentadas abaixo:

- **Fase 1: Escolha da aplicação alvo**. Este estudo utilizou três aplicações alvo para avaliar as abordagens API *Slicer* e *Monólise*, são elas pedido-api, shopping-cart-api e blog-api (maiores detalhes podem ser encontradas na Seção 5.1). O requisito utilizado para a escolha das APIs foi: (1) as APIs estarem em algum repositório público, (2) ser possível executar a API localmente, (3) as APIs terem mais de duas funcionalidades e (4) as APIs terem sido desenvolvidas em Java, utilizando o framework Springboot[7].

- **Fase 2: Decomposição manual**. Este estudo utilizou três aplicações alvo, são elas pedido-api, shopping-cart-api e blog-api (maiores detalhes podem ser encontradas na Seção 5.1). As aplicações alvo, uma por vez, foram avaliadas em relação as suas funcionalidades e código. Após a avaliação, o autor sugeriu a recomendação de qual feature deveria se tornar um microsserviço e qual feature deveria permanecer na API monolítica. Com base nessa recomendações, se teve uma base de qual resultado deveria ser o esperado, após executar as abordagens.

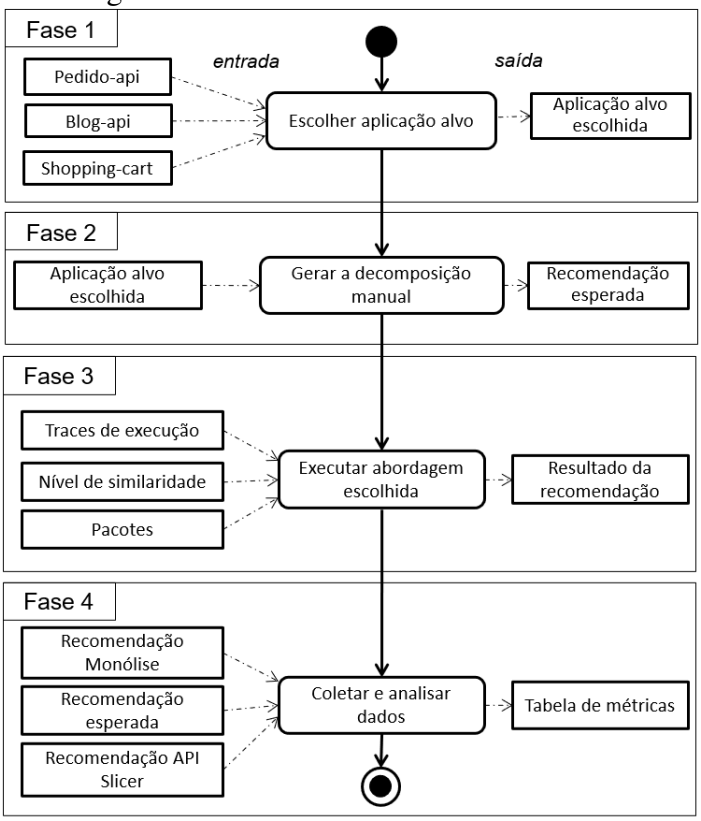
- **Fase 3: Executar a implementação da abordagem escolhida**. Para a API *Slicer*, foi fornecido como dados de entrada: (1) os *traces* de execução das funcionalidades que se quer gerar a recomendação, (2) o nível de similaridade aplicado (valor de corte que diz quando os serviços devem ficar no mesmo microsserviço) e (3) quais pacotes devem ser considerados durante a recomendação dos microsserviços. Já para a abordagem *Monólise* do autor Rocha (2018), utilizou-se como dados de entrada: 1) os *traces* de execução das funcionalidades que se quer gerar a recomendação, (2) o nível de similaridade aplicado (valor de corte que diz quando os serviços devem ficar no mesmo microsserviço) e (3) o arquivo de configuração, onde os pacotes model, DAO, service, repository e controller ganharam pesos que podem variar de 0.1 a 1.

- **Fase 4: Coletar e analisar dados**. Os resultados fornecidos por ambas as abordagens foi coletado e depois estruturado em uma planilha. Nesta planilha foram postos os dados da recomendação esperada (elaborado pelo autor) e das recomendações sugeridas pelo API *Slicer* e *Monólise*. Ambas as abordagens API *Slicer* e *Monólise*, utilizaram um nível de similaridade que iniciou em 10% e foi incrementando de 10 em 10, até chegar a 90%. E a cada variação do nível de similaridade, as métricas *precision* e *recall* foram

[7]Para mais informações sobre o Springboot: <https://spring.io/projects/spring-boot>

medidas. As medidas *precision* e *recall*, foram as utilizados para identificar qual abordagem conseguia chegar mais rápido na recomendação esperada, usando o menor nível de similaridade. Na Figura 5, Figura 6 e Figura 7 pode ser visto os cenários onde ao menos uma das abordagens conseguiu chegar mais rápido ao resultado esperado. Uma ressalva a ser feita, é que a *Monólise* não diz quais funcionalidades devem permanecer na API monolítica, então para cobrir essa brecha, o microsserviço que teve o maior número de microsserviços, era classificado como os serviços que deveriam permanecer na API monolítica.

Figura 4 – Fluxo do estudo de caso



Fonte: Elaborado pelo autor.

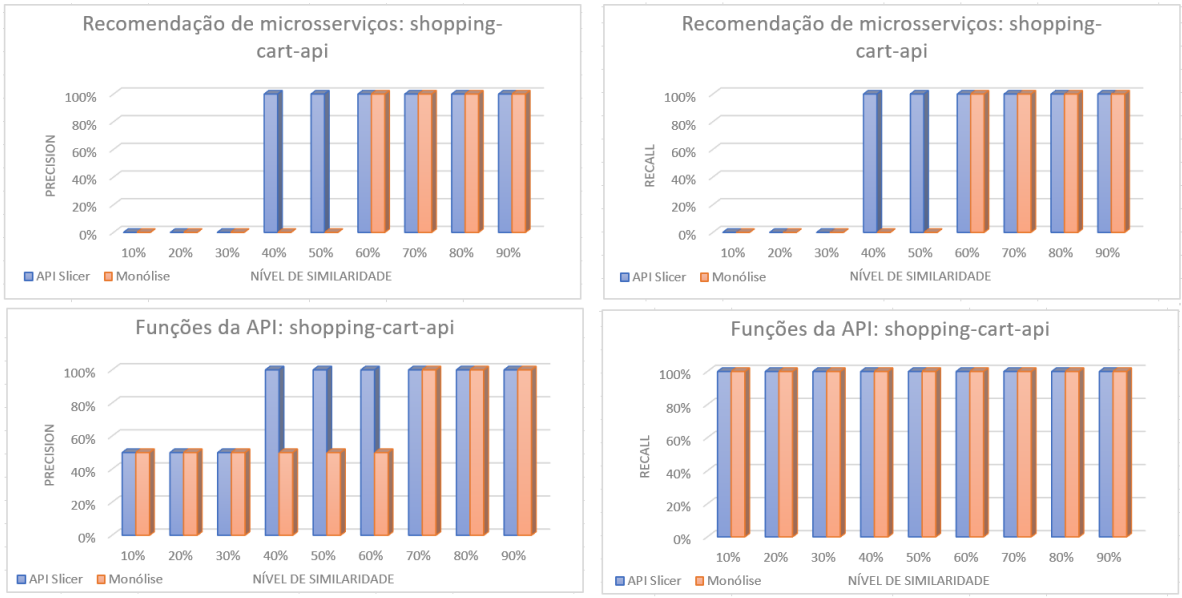
5.4 Resultados

Esta seção apresenta os resultados coletados após a execução dos passos descritos na Seção 5.3. A Figura 5, Figura 6 e Figura 7, apresentam o resultados obtidos, aplicando as abordagens API *Slicer* e *Monólise*, nas aplicações alvo blog-api, shopping-cart-api e pedido-api.

Case 1: aplicação alvo shopping-cart-api. Na aplicação alvo shopping-cart-api, a abordagem API *Slicer* teve melhores resultados tanto para recomendar microsserviços, quanto para dizer quais funcionalidades devem permanecer na API monolítica. Na recomendação de microsserviços, a partir de 40% de nível de similaridade, a API *Slicer* começa a recomendar mi-

crosserviços com 100% de *precision* e *recall*, enquanto que na *Monólise*, esse resultado só foi alcançado a partir de 60%. Em relação as funcionalidades que devem se manter na API monolítica, embora a API *Slicer* e *Monólise* tenham atingido 100% na métrica de *recall*, a API *Slicer* conseguiu atingir o nível máximo de precisão a partir de 40% de nível de similaridade. Já a *Monólise* alcançou este feito apenas quando o nível de similaridade chegou a 70%.

Figura 5 – Case 1: shopping-cart-api

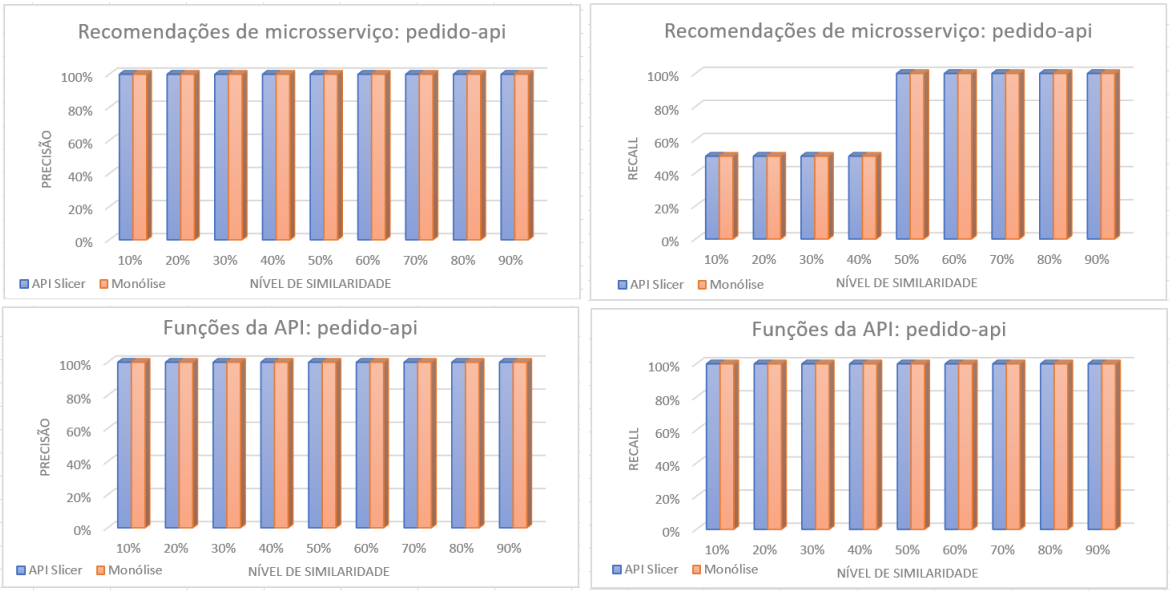


Fonte: Elaborado pelo autor.

Case 2: aplicação alvo pedido-api. Na aplicação alvo pedido-api, a abordagem API *Slicer* e *Monólise* tiverem um empate. Na recomendação de microsserviços, ambas as abordagens alcançaram as mesmas métricas de *precision* e *recall*, para os diferentes níveis de similaridade. Em relação as funcionalidades que devem se manter na API monolítica, tanto a API *Slicer* e *Monólise* atingiram os mesmos resultados de *precision* e *recall*, para todos os níveis de similaridade aplicados.

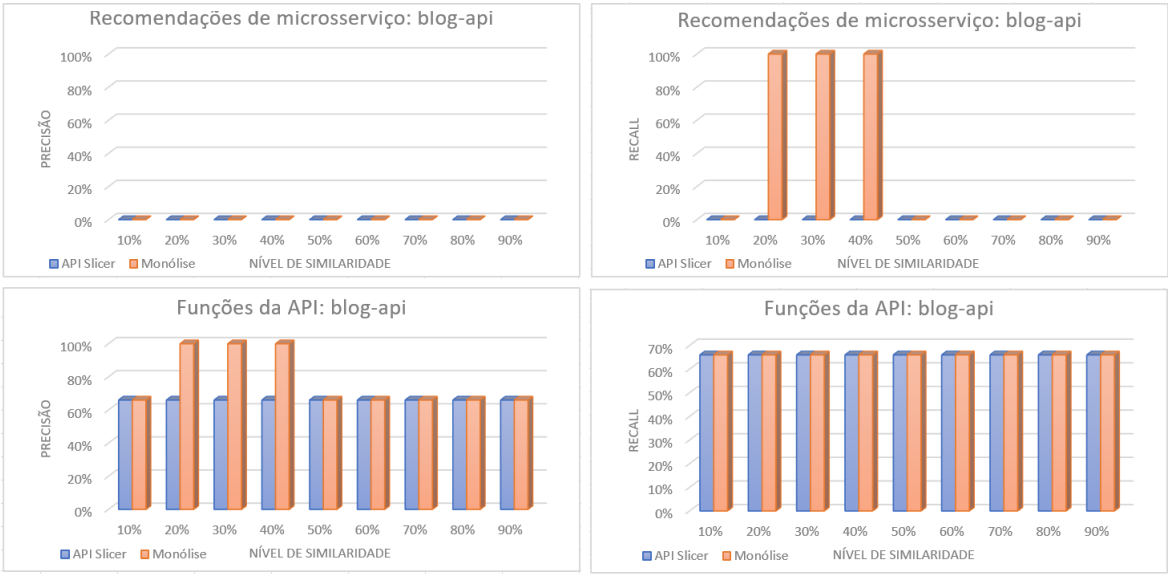
Case 3: aplicação alvo blog-api. Na aplicação alvo blog-api, a abordagem *Monólise* teve melhores resultados tanto para recomendar microsserviços, quanto para dizer quais funcionalidades devem permanecer na API monolítica. Na recomendação de microsserviços tanto a API *Slicer* quanto a *Monólise* tiverem uma baixo desempenho, porém em relação a métrica de *recall*, a *Monólise* conseguiu atingir 100%. Já nas funcionalidades que devem permanecer na API monolítica, embora ambas as abordagens tenham atingido os mesmos resultados para *recall*. Na métrica de precisão, na faixa de 20% a 40% de similaridade, a *Monólise* atingiu uma precisão de 100%.

Figura 6 – Resultados pedido-api



Fonte: Elaborado pelo autor.

Figura 7 – Resultados blog-api



Fonte: Elaborado pelo autor.

5.5 Discussão dos resultados

Baseado nas observações dos resultados obtidos na construção deste projeto e na comparação de ambas as abordagens, foram identificados os seguintes assuntos para discussão:

Melhor desempenho. Utilizando como base os resultados obtidos para cada uma das três aplicações alvo, pode-se dizer que a abordagem API *Slicer* obteve melhor desempenho na recomendação de microsserviços. Pois alcançou uma precisão e revocação de 100%, em duas das

três aplicações alvo testadas (Seção 5.4). Isso se deve a flexibilidade do usuário escolher quais pacotes ele quer que seja considerado, no momento de gerar as recomendações. Uma segunda consequência dessa escolha de pacotes, é que classes transversais tendem a não atrapalhar o resultado da recomendação, como foi visto no trabalho da *Monólise*

Recomendação incorreta. Pode-se notar que a abordagem API *Slicer* não teve um resultado satisfatório, quando gerou as recomendações para a aplicação alvo blog-api. Isso aconteceu porque os pacotes que geravam a diferenciação da funcionalidade login, também eram utilizados por outros serviços, gerando a impossibilidade de se chegar ao resultado desejado. E isso leva a especulação de que quando se tem serviços muito acoplados, talvez a abordagem API *Slicer* não seja a melhor escolha. Entretanto para confirmar essa suposição, novas comparações teriam de ser feitas com ferramentas que utilizassem diferentes abordagens.

Trace de execução. O *trace* de execução é um arquivo que diz quais classes e métodos foram executados quando determinada funcionalidade foi ativada. Neste projeto ele serve como entrada para a API *Slicer* entender a aplicação alvo e poder fazer alguma recomendação. Entretanto essa é uma das etapas mais custosas do processo de recomendação, pois além de ser uma etapa manual feita apenas pelo usuário, as ferramentas de instrumentação de código Java utilizadas para auxiliar neste processo, ou não conseguem entregar o *trace* de forma simples, ou possuem limitações, como pode ser visto no trabalho da Santos e Paula (2020).

Ferramentas open source. Há vários trabalhos relacionados a decomposição de aplicações monolíticas, utilizando as mais diversas abordagens, como grafo (KIRBY et al., 2021), junção de código com banco de dados (ZHAO; ZHAO, 2021), analisando o código de forma sintática (SANTOS; PAULA, 2020), entre outros. No entanto, na maioria dos casos, o código fonte das ferramentas não é disponibilizado em um repositório público, impossibilitando uma série de melhorias que poderiam surgir através da comparação de abordagens e respectivas evoluções. A constatação de que mais trabalhos deveriam disponibilizar seu código fonte, para evolução não só da ferramenta, como também da pesquisa, corrobora com o trabalho do Coppola e Neelley (2004), que diz o seguinte: "universidades possuem sinergia com programas *open source*, pois criar e compartilhar conhecimento para o bem público, é parte fundamental da missão das universidades".

5.6 Desafio e Implicações

Este seção apresenta os desafios e implicações que foram derivados a partir da análise dos resultados dos dados obtidos.

Desafio 1: Abordagem proposta. Embora a API *Slicer* tenha tido melhores resultados na recomendação de microsserviços, em relação a métrica de precisão. Para pesquisas futuras seria interessante ser criado uma abordagem que funcione em qualquer cenário e que tenha um certo contexto de inteligência por trás, para que o resultado da recomendação de microsserviços não seja fortemente influenciado pelas escolhas do usuário. E sim em métricas e conhecimentos já

validados. No caso do API *Slicer*, pode ser possível alterar o resultado da recomendação de microsserviços através do *trace* de execução fornecido, do valor de similaridade ou através do nome dos pacotes que devem ser considerados no momento da recomendação. E isso pode ser visto como um problema, pois a recomendação será enviesada em algo que o usuário já quer como resultado. E nem sempre o que o usuário almeja como resultado, é o melhor.

Implicação 1: trace de execução. O *trace* de execução é um arquivo que contém todas as classes e métodos que foram utilizados, quando uma funcionalidade foi executada. O *trace* de execução pode ser obtido através de ferramentas de instrumentação ou aplicação da programação orientada a aspecto, no projeto. Após análise, se identificou que fazer uso do *trace* de execução para mapear funcionalidades que estão dentro da aplicação, é extremamente útil e recomendado para qualquer abordagem que seja necessário compreender quais classes e métodos que determinada funcionalidade é composta.

5.7 Limitações

O estudo de caso tratado neste trabalho é um estudo inicial que explora a decomposição de APIs monolíticas, um tema pouco explorado na literatura. Devido a isso, ele possui algumas limitações que precisam ser consideradas, como por exemplo: embora não há nada que impeça o uso deste projeto para outras aplicações, o foco dele engloba apenas APIs . Logo os testes de efetividade das recomendações, se restringe apenas a este escopo. Por a ferramenta criada pela abordagem *Monólise* não estar disponível em um repositório público e nem ter havido acesso a ela, o protótipo foi recriado com base nas informações do artigo da *Monólise*. Logo os resultados obtidos com a ferramenta recriada, podem destoar um pouco da implementação original (seja de forma positiva ou negativa). A qualidade das recomendações possui uma forte dependência do *trace* capturado pelo usuário. E visto que essa captura é de total responsabilidade dele, pois a ferramenta não possui alguma funcionalidade que faça captura do rastro de execução, este ponto é visto como um limitador.

6 CONCLUSÕES E TRABALHOS FUTUROS

O presente trabalho apresentou a API *Slicer*, uma abordagem baseada no nível de similaridade entre *features*, que tem como objetivo ler uma API monolítica e gerar sugestões de microsserviços. A abordagem foi avaliada por meio de um estudo de caso, onde a abordagem proposta foi comparada com a abordagem *Monólise* do autor Rocha (2018), para testar a efetividade da recomendação de microsserviços. As métricas *precision* e *recall* foram utilizadas para comparar o resultados gerados de ambas as abordagens, com o resultado desejado.

Os resultados indicaram que a abordagem API *Slicer* teve melhores resultados na precisão de recomendação de microsserviços, visto que a *Monólise* obteve empate na aplicação alvo pedido-api, mas não teve nenhuma vitória. Já em relação a métrica *recall* para a recomendação

de microsserviços houve um empate técnico, pois dos três cenários, em um a API *Slicer* ganhou, em outro um a *Monólise* e no último, houve um empate.

Sobre os trabalhos futuros, foi verificado três pontos importantes: (1) fazer o uso de aprendizagem de máquina para aprimorar os resultados da recomendação, (2) implementar uma funcionalidade que capture o *trace* de execução da API alvo e (3) executar a ferramenta em mais aplicações, para identificar se as recomendações ainda continuam sendo assertivas.

Referências

APEL, S.; KÄSTNER, C. An overview of feature-oriented software development. **J. Object Technol.**, v. 8, n. 5, p. 49–84, 2009.

ASSUNÇÃO, W. et al. Analysis of a many-objective optimization approach for identifying microservices from legacy systems. **Empirical Software Engineering**, v. 27, 03 2022.

BISCHOFF, V.; FARIAS, K. Vitforecast: An iot approach to predict diseases in vineyard. In: **XVI Brazilian Symposium on Information Systems**. [S.l.: s.n.], 2020. p. 1–8.

BUCKLAND, M.; GEY, F. The relationship between recall and precision. **Journal of the American society for information science**, Wiley Online Library, v. 45, n. 1, p. 12–19, 1994.

COPPOLA, C.; NEELLEY, E. Open source-opens learning: Why open source makes sense for education. 2004.

FARIAS, K. et al. Evaluating the effort of composing design models: A controlled experiment. In: . [S.l.: s.n.], 2012. v. 7590, p. 676–691. ISBN 978-3-642-33665-2.

FARIAS, K. et al. Uml2merge: a uml extension for model merging. **IET Software**, IET, v. 13, n. 6, p. 575–586, 2019.

FILIPPONE, G. et al. Migration of monoliths through the synthesis of microservices using combinatorial optimization. In: **2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. [S.l.: s.n.], 2021. p. 144–147.

GUPTA, a. A first look at microservice. 2015.

IVANOV, N.; TASHEVA, A. A hot decomposition procedure: Operational monolith system to microservices. In: **2021 International Conference Automatics and Informatics (ICAI)**. [S.l.: s.n.], 2021. p. 182–187.

JÚNIOR, O. A. Arquitetura de micro serviços: uma comparação com sistemas monolíticos. Universidade Federal da Paraíba, 2017.

KIRBY, L. J. et al. Weighing the evidence: On relationship types in microservice extraction. In: **2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2021. p. 358–368.

LAZZARI, L.; FARIAS, K. Event-driven architecture and rest architectural style: An exploratory study on modularity. **Journal of applied research and technology**, Universidad Nacional Autónoma de México, Instituto de Ciencias Aplicadas y . . . , v. 1, n. 1, p. 1–2, 2022.

LUCIO, J. P. D. et al. Análise comparativa entre arquitetura monolítica e de microsserviços. Florianópolis, SC, 2017.

MAZLAMI, G.; CITO, J.; LEITNER, P. Extraction of microservices from monolithic software architectures. In: **2017 IEEE International Conference on Web Services (ICWS)**. [S.l.: s.n.], 2017. p. 524–531.

NEWMAN, S. **Building microservices**. [S.l.]: "O'Reilly Media, Inc.", 2021.

ROCHA, D. Pereira da. **Monólise: Uma Técnica para Decomposição de Aplicações Monolíticas em Microsserviços**. Tese (Doutorado) — Universidade do Vale do Rio dos Sinos, 2018.

ROELLEKE, T. Information retrieval models: Foundations and relationships. **Synthesis Lectures on Information Concepts, Retrieval, and Services**, Morgan & Claypool Publishers, v. 5, n. 3, p. 1–163, 2013.

RUBERT, M.; FARIAS, K. On the effects of continuous delivery on code quality: A case study in industry. **Computer Standards & Interfaces**, Elsevier, v. 81, p. 103588, 2022.

SANTOS, A. P. dos; PAULA, H. B. D. Implementação e avaliação de uma ferramenta de sugestões para decomposição de aplicação monolítica em microsserviços. 2020.

SHEIKH, A.; BS, A. Decomposing monolithic systems to microservices. In: **2020 3rd International Conference on Computer and Informatics Engineering (IC2IE)**. [S.l.: s.n.], 2020. p. 478–481.

THüM, T. et al. Featureide: An extensible framework for feature-oriented software development. **Science of Computer Programming**, v. 79, p. 70–85, 2014. ISSN 0167-6423. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642312001128>.

URDANGARIN, R. G.; FARIAS, K.; BARBOSA, J. Mon4aware: A multi-objective and context-aware approach to decompose monolithic applications. In: **XVII Brazilian Symposium on Information Systems**. New York, NY, USA: Association for Computing Machinery, 2021. (SBSI 2021). ISBN 9781450384919. Disponível em: <https://doi.org/10.1145/3466933.3466949>.

URDANGARIN, R. G.; FARIAS, K.; BARBOSA, J. Mon4aware: A multi-objective and context-aware approach to decompose monolithic applications. In: **XVII Brazilian Symposium on Information Systems**. [S.l.: s.n.], 2021. p. 1–9.

WAHLSTRÖM, S. **Comparing Scaling Benefits of Monolithic and Microservice Architectures Implemented in Java and Go**. 2019.

ZHAO, J.; ZHAO, K. Applying microservice refactoring to object-2riented legacy system. In: **2021 8th International Conference on Dependable Systems and Their Applications (DSA)**. [S.l.: s.n.], 2021. p. 467–473.