MIND OVERFLOW

Tcharles Pereira da Silva[*]

Kleinner Silva Farias de Oliveira[**]

Resumo: Alterações constantes feitas por perfis diferentes de desenvolvedores acabam transformando aplicações legadas em monolíticas. Embora seja uma problemática conhecida, pouco tem sido feito para mitigá-la. Este trabalho propõe o Mind Overflow, um processo para guiar a decomposição de uma aplicação monolítica para uma arquitetura de microsserviços. Com o Mind Overflow, pesquisadores e desenvolvedores se beneficiam do uso de padrões de design, arquiteturais e tecnologias consolidadas através de um workflow compreensível e consistente de decomposição. O estudo de caso realizado mostrou resultados promissores, ao indicar que o Mind Overflow é factível para decompor arquiteturas monolíticas para baseadas em microsserviços, inclusive reduzindo a complexidade ciclomática e produzindo microsserviços altamente coesos.

Palavras-chave: Microsserviços. Arquitetura de software. Modelagem de software. Desenvolvimento de software.

1 INTRODUÇÃO

Aplicações legadas passam por várias alterações ao longo do seu ciclo de vida, as quais são realizadas geralmente por diferentes desenvolvedores. Essas alterações constantes e realizadas por perfis diferentes de desenvolvedores acabam transformando aplicações legadas em monolíticas. Embora existam diversas técnicas que ajudam o código de aplicações monolíticas a não apresentarem erros (VERNON, 2013) (BENNET; RAJLICH, 2000), isso não é sinônimo de ter aplicações de fácil manutenção. Tipicamente, ao satisfazer as necessidades de negócio dos clientes sem apresentar erros, as aplicações são entregues, finalizando, por sua vez, o ciclo de desenvolvimento.

Uma aplicação considerada legada em uma companhia, normalmente padece de responsáveis técnicos e mantenedores, pois a mesma foi desenvolvida utilizando técnicas que não estão sendo utilizadas pelos desenvolvedores atuais da companhia; entretanto, tal solução ainda é vital e valiosa para o negócio de quem a utiliza (BENNET, 1995). Embora essa problemática de manter aplicações legadas

[*] Experiência de mais de dez anos na área de desenvolvimento de software, atuando hoje em projetos internacionais. E-mail: tcharlezin@gmail.com

[*] [*] Doutor e pesquisador. tem experiência na área de Ciência da Computação com ênfase em Engenharia de Software. E-mail: kleinnerfarias@unisinos.br

seja bastante conhecida, pouco tem sido feito para mitigá-la. O uso de arquiteturas baseadas em microsserviços tem se mostrado promissor neste sentido. Microsserviço pode ser definido como uma aplicação composta por pequenas funcionalidades (ou serviços) que levam em consideração o contexto do negócio, que sejam autônomos e independentes na sua execução e na sua publicação, fazendo isso de forma automatizada e que possam se comunicar entre si, além de fornecer *endpoints* para o meio externo de seus limites (FOWLER; LEWIS, 2014).

Este trabalho, portanto, propõe o Mind Overflow, um proposta de processo agnóstico a tecnologia e a framework, o qual tem por objetivo guiar a decomposição de uma aplicação monolítica para uma arquitetura de microsserviços através de uma sequência sistemática de pequenas decomposições. Ao utilizar o Mind Overflow, pesquisadores e desenvolvedores se beneficiam do uso de padrões de design, arquiteturais e tecnologias bem maduradas através de um *workflow* compreensível e consistente. Os diferenciais do Mind Overflow são os seguintes: (1) definição de um processo de reengenharia de software baseado em iterações que possam evoluir de maneira incremental; e (2) introdução de um *workflow* de decomposição de arquiteturas monolíticas, buscando modularizar as principais responsabilidades da aplicação em microsserviços.

O estudo de caso realizado mostrou resultados promissores, ao indicar que o Mind Overflow é factível para decompor arquiteturas monolíticas para baseadas em microsserviços, inclusive reduzindo a complexidade ciclomática e produzindo microsserviços altamente coesos.

Este trabalho está organizado como segue: A seção 2 apresenta os conceitos fundamentais para entendimento do trabalho. A seção 3 aborda os trabalhos relacionados. A seção 4 apresenta o trabalho proposto. A seção 5 descreve a avaliação e como foi feita a análise dos resultados. A seção 6 aborda as conclusões e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir, são apresentados os conceitos-chave adotados neste trabalho.

2.1 Arquitetura de Software

Os estilos e padrões de arquitetura utilizados nos sistemas devem ser orientados pelas demandas reais e de qualidades específicas, satisfazendo e excedendo as expectativas de seu uso (VERNON, 2013). Evitar a utilização de padrões e estilos de maneira excessiva torna-se tão fundamental quanto aplicar a arquitetura de maneira correta. Uma boa base para guiar as escolhas destes estilos e padrões é saber justificar sua finalidade para a solução baseando-se nos requisitos funcionais, do contrário, eles devem ser eliminados do sistema.

2.1.1 Camadas arquiteturais

Este padrão de arquitetura suporta N camadas e é comumente utilizado em aplicações Web, corporativas e desktop. O modo básico de utilização deste padrão são três camadas: interface com o usuário, camada de aplicação e camada de infraestrutura. Sendo uma premissa de boa prática deste padrão, uma camada deve apenas interagir com a sua camada ou camadas abaixo de si, evitando assim, chamadas para camadas superiores (VERNON, 2013). O processo desenvolvido neste trabalho tem como premissa que os sistemas gerados utilizem uma arquitetura em camadas.

2.1.2 Arquitetura Orientada a Serviço

Uma proposta para a implementação de SOA é a utilização de uma arquitetura hexagonal, a qual possui *adapters* para entrada de dados, estes podendo ser REST, SOAP e mensagens através de serviços de mensageria. Com os pontos de entrada podendo ser distintos, todos eles direcionam as requisições para um ponto comum que seria a camada de aplicação, que por subsequente, direciona para o modelo de domínio a fim de realizar um processamento baseado no negócio e na finalidade para tal requisição (VERNON, 2013).

2.1.3 Arquitetura Orientada a Eventos

Segundo Vernon (VERNON, 2013), é um padrão capaz de promover a produção, detecção, consumo e reação a eventos. Estes eventos não tem a

finalidade de simplesmente ser notificações técnicas decorativas, mas sim, representam as ocorrências de atividades do processo de negócio que serão encaminhadas para todos os pontos de recepção destas atividades, que normalmente, realizarão outras atividades decorrente da atividade inicial identificada. Para receber esta notificação, podendo ser ela síncrona ou assíncrona, é utilizado o padrão de produtor e consumidor, sendo possível que aplicações distintas se comuniquem entre si a partir do encadeamento de eventos. O processo desenvolvido neste trabalho, utiliza o padrão produtor e consumidor para a comunicação entre os microsserviços, onde cada aplicação atua com uma responsabilidade de domínio diferente.

2.2 Domain-Driven Design

A abordagem de desenvolvimento de software *Domain-Driven Design* (DDD), criada por Eric Evans (EVANS, 2003) em seu famoso livro azul, vem para auxiliar na maneira como o software expressa a solução para o negócio, de forma que os modelos gerados demonstrem explicitamente o objetivo pretendido do negócio. O DDD coloca em condições de igualdade especialistas de domínio e desenvolvedores na forma de uma equipe coesa e unida, fazendo com que o software que é desenvolvido, faça sentido para o negócio e não apenas para os codificadores, ficando o conhecimento de negócio centralizado a nível de código fonte e não mais em seletos grupos de pessoas, que normalmente são os desenvolvedores. Objetivando que o código seja o projeto, e o projeto seja o código. Outra característica da consequência da utilização do DDD, é a inexistência de traduções entre os especialistas de negócio e os desenvolvedores, pois a equipe desenvolve uma linguagem única pertinente ao domínio trabalhado e todos desenvolvem o mesmo entendimento do significado de qualquer termo em relação ao contexto do domínio trabalhado.

2.2.1 Linguagem Ubíqua

De acordo com Evans (EVANS, 2003), um dos maiores problemas na construção de software é a baixa compreensão dos objetivos de negócio motivado por uma comunicação imprópria entre os especialistas de negócio e os

desenvolvedores. Para os especialistas de domínio, termos técnicos de implementação e de design não são facilmente entendíveis, pois a visão que eles têm do negócio e de como funciona, utiliza normalmente jargões próprios do domínio. Já os desenvolvedores que dominam os termos técnicos e termos de design, possuem muita dificuldade na compreensão dos objetivos e no entendimento do que está sendo tratado pelos especialistas, dado que normalmente um sistema é discutido e visto apenas a nível de implementação pelos desenvolvedores, isso sem falar da enorme quantidade de traduções aplicadas para a linguagem dos especialistas virar a linguagem do desenvolvedor. Desenvolvedores que atuam em diversas partes do negócio, normalmente criam seus próprios conceitos sobre o significado de termos, muitas vezes não fazendo sentido ao negócio em si. Para esse problema conhecido é utilizada a linguagem ubíqua, que nada mais é do que a linguagem do domínio, onde há o alinhamento dos termos entre todos os membros da equipe, zerando simplesmente as traduções de significados entre especialistas de negócio e desenvolvedores, exigindo a compreensão igualitária por todos os membros da equipe junto ao domínio trabalhado.

2.2.2 Contexto Delimitado

Também conhecido como *bounded context*, delimita o escopo de entendimento e implementação de um determinado modelo em relação a sua participação para com o negócio. Para cada contexto delimitado é definida uma linguagem ubíqua que deve ser conhecida por todos os membros da equipe (EVANS, 2003). Todas as responsabilidades do modelo devem ser bem definidas e o entendimento deste para com os demais contextos deve ser definido, embora o conhecimento de traduções e integrações entre os contextos delimitados seja algo a parte. Para as implementações deste trabalho, um microsserviço será tratado sempre na forma de contexto delimitado.

2.2.3 Mapa de contextos

De acordo com Vernon (VERNON, 2013), um mapa de contextos define como os contextos delimitados vão se comunicar uns com os outros, dando um contexto global sobre a solução do negócio. As relações existentes entre os contextos

delimitados no mapa de contexto, podem ser classificadas e traduzidas de acordo com o alinhamento entre as equipes. Para este trabalho, o mapa de contextos será gerado a partir do *Event Storming*, uma técnica criada por Brandoline (BRANDOLINE, 2013), que é definida como primeira parte do processo proposto, tendo as comunicações identificadas na forma de respostas a eventos ocorridos na sequência das atividades do negócio.

2.2.4 Eventos de domínio

O evento de domínio é definido através da publicação de atividades na forma de notificações pelos publicadores para seus assinantes, podendo estes, ser da própria aplicação ou de outras aplicações que estão assinadas para receber as notificações (VERNON, 2013). A proposta de processo deste trabalho considera eventos de domínio como parte da etapa e seu entendimento é fundamental para a elaboração das estórias de usuários a partir de *Event Storming*.

3 TRABALHOS RELACIONADOS

Neste capítulo será disponibilizada uma análise comparativa dos trabalhos relacionados. A análise tem por objetivo o enriquecimento do processo que será elaborado com base nos estudos já realizados sobre a decomposição de aplicações monolíticas em microsserviços. O capítulo está dividido da seguinte forma: na seção 3.1, será descrita a forma do levantamento dos trabalhos relacionados; na seção 3.2, será realizada a análise de cinco artigos que satisfazem os critérios de seleção; na seção 3.3, será realizada a comparação dos trabalhos, mediante critérios definidos; na seção 3.4, as oportunidades de pesquisa são identificadas.

3.1 Critérios de seleção dos trabalhos

Este trabalho utilizou como base de dados o Google Scholar, pesquisando a temática de decomposição de sistemas monolíticos para uma arquitetura de microsserviços através das palavras-chave “microservice” e “monolith”. Feito o primeiro levantamento sobre o tema de decomposição para microsserviços, foram filtrados somente trabalhos que tinham como objetivo a abordagem ou da

construção ou da reestruturação de aplicações para microsserviços, utilizando algum tipo de técnica, processo ou metodologia. Um segundo filtro utilizado, foi a consideração de artigos publicados entre os anos de 2015 a 2019, devido a microsserviços ser uma temática muito recente.

3.2 Análise dos trabalhos

Nesta seção, será realizada uma análise comparativa de cinco trabalhos que exploram a temática de decomposições de sistemas monolíticos para microsserviços, seja por meio de técnica, processo ou metodologia, com o objetivo de identificar quais são os critérios comuns entre os trabalhos.

3.2.1 Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems

No trabalho de LEVCOVITZ et al. (2015), foi criada uma técnica para decomposição de sistemas monolíticos em pequenos serviços coesos e independentes. A técnica foi aplicada em um sistema bancário que possuía em torno de 750 mil linhas de código-fonte, 2 milhões de contas bancárias, operando aproximadamente 2 milhões de autorizações diárias. Como premissa para o trabalho, o fluxo de uma iteração simples no sistema que será decomposto, parte da interface com o usuário, chegando a uma *facade* que é responsável pela designação da requisição para as funções de negócio da aplicação. A partir da função de negócio, uma ou mais funções de negócio podem ser chamadas de forma encadeada, podendo utilizar uma ou mais tabelas de banco dados, sendo cada função de negócio classificada em uma área, a qual o artigo trata como unidade organizacional. A técnica é constituída de seis passos: o primeiro passo é identificar os subsistemas de acordo com as áreas de negócio e as tabelas do banco de dados; o segundo passo cria um grafo de dependências entre as *facades*, uma ou mais funções de negócio e tabelas de banco de dados; no terceiro passo, com base no grafo de dependência gerado são mapeados os vínculos entre *facade* e tabela da base de dados; no quarto passo, para cada subsistema identificado no primeiro passo, são mapeadas as *facades* que possuem relação com as tabelas de banco de dados pertencentes ao subsistema; o quinto passo é responsável pela identificação

e avaliação de candidatos a seguirem na decomposição. No último passo, é criado um *API Gateway* que funciona como um orquestrador das requisições para os microsserviços, viabilizando o acesso do cliente ao servidor, com o objetivo de garantir a transparência para os utilizadores. Após executada a técnica em um ambiente real, segundos os autores, foi possível mapear todos os subsistemas da aplicação bancária. Também com base na técnica proposta, foi possível desenvolver os microsserviços de maneira incremental, sem ter que migrar a aplicação inteira para microsserviços de uma vez, permitindo que a aplicação monolítica e os microsserviços coexistam.

3.2.2 Using Microservices for Legacy Software Modernization

KNOCHE; HASSELBRING. (2018), apresentam uma técnica de modernização de aplicações, baseada na decomposição de sistemas legados para uma arquitetura de microsserviços. O sistema legado utilizado para a fundamentação e aplicação da técnica, é um sistema de gerenciamento de clientes desenvolvido em Cobol com mais de um milhão de linhas de código, iniciado nos anos de 1970 e 1980. A técnica de modernização proposta é dividida em cinco passos, que são: o primeiro passo, realiza a identificação dos serviços de domínio que irão prover as funcionalidades esperadas pelos clientes, após isso, é realizada uma análise para a identificação dos *entry points* da aplicação, sendo eles qualquer forma de acesso utilizado por outros aplicações; o segundo passo, realiza a implementação das *facades* na forma de *adapters* para o recebimento das requisições, sempre considerando que essas implementações devem garantir o mesmo comportamento das operações já realizadas pelo sistema, e para haver essa garantia, a equipe utilizou técnicas de teste de software, gerando confiabilidade no processamento esperado nas implementações; para o terceiro passo é realizado a migração das aplicações clientes para as novas *facades* criadas. Dessa forma, não mais acessando a aplicação legada diretamente, limitando os pontos de entrada através das *facades*; o quarto passo estabelece as *facades* internas da aplicação, com o objetivo de organizar as solicitações das funcionalidades para *facades* internas, e não mais para outras funcionalidades, podendo isso ser feito em paralelo, além deste passo demarcar quais funcionalidades estão obsoletas e que podem ser removidas; no último passo é realizado a troca das *facades* por microsserviços de

forma transparente, dado que toda a estrutura e funcionalidades envolvidas já estão encapsuladas dentro das *facades*, a transformação destas em microsserviços já possui todo o cenário garantido pelos passos anteriores, principalmente pelos testes. Como conclusão, os autores descrevem que a modernização do sistema legado aconteceu por quatro anos. Apesar de todo o esforço envolvido, algumas operações ainda dependem da aplicação legada, pelo motivo de que uma das principais dificuldades e desafios, sendo os microsserviços distribuídos, é o controle de transações que garantem a integridade dos dados, sendo isso, trivial em aplicações monolíticas.

3.2.3 Service Cutter: A Systematic Approach to Service Decomposition

O trabalho de GYSEL et al. (2016), propõe uma ferramenta para decomposição de aplicações monolíticas em microsserviços, baseada nas experiências de vários arquitetos de software. Com um catálogo contendo 16 critérios de decomposição, classificados em quatro categoria, a ferramenta trabalha sobre uma entrada fornecida pelo usuário no formato *JSON*, que pode ser: um modelo de entidade de relacionamento baseado na modelagem de banco de dados, entidades e agregados do padrão *Domain-Driven Design* ou um modelo de caso de uso. As quatro categorias para os critérios são: coesão, compatibilidade, restrições e comunicação. Cada critério é formatado com um template bem parecido ao que é utilizado nas práticas agile, contendo informações como: identificação e nome; descrição; artefatos de especificação no sistema; referências da literatura, categoria e características. Além de também possuírem um escore pré-definido e o valor de prioridade passado pelo usuário após a importação. A ferramenta realiza o processamento da entrada fornecida pelo usuário, gerando um grafo da representação do acoplamento entre as nano entidades geradas pela aplicação. Com o grafo formado, é realizada a classificação e o agrupamento através da clusterização, gerando assim, as indicações dos serviços candidatos para avaliação do usuário. Esta clusterização pode ser adaptada, pois a ferramenta não se limita a um algoritmo específico, neste trabalho, os autores utilizaram os algoritmos Girvan-Newman (determinístico) e *Epidemic Label Propagation* (também chamado de ELP, sendo ele não determinístico). A avaliação foi realizada por 20 engenheiros e arquitetos de software com experiência em design em SOA. Os avaliadores

apreciaram a ferramenta e consideraram que é promissora. Além de relatar que seu uso não se restringe apenas para o contexto de SOA, mas também, para a decomposição de módulos sem a necessidade de interfaces remotas. Como limitação, há um alto esforço para a geração dos dados de entrada, sendo um possível trabalho futuro, a integração com ferramentas de modelagem UML, para extrair dados de forma automática.

3.2.4 Monólise: Uma Técnica para Decomposição de Aplicações Monolíticas em Microsserviços

No trabalho de Rocha (ROCHA, 2018), é proposta uma técnica para decomposição de aplicações monolíticas em microsserviços foi proposta, a qual foi nomeada de Monólise. Essa técnica possui o algoritmo MonoBreak, o qual permite que seus utilizadores simulem vários cenários de decomposição, dessa forma, encontrando o melhor grau de granularidade entendido como a decomposição ideal para o microsserviço. A técnica é composta por quatro etapas, sendo elas: coleta de dados, processamento de dados, disponibilização dos resultados e implementação dos resultados. Na primeira etapa, o objetivo é coletar os parâmetros para a execução do algoritmo, definindo uma aplicação alvo que tenha sido desenvolvida em uma linguagem orientada a objetos e que utilize a arquitetura em camadas MVC. Para a identificação dos fluxos e sequências de chamadas entre classes, esta técnica sugere a utilização de ferramentas de instrumentação de código. Com os dados das chamadas para classes e métodos definidas por funcionalidades, cabe ao utilizador definir quais funcionalidades serão decompostas para microsserviços. Também é aplicado ao utilizar, um questionário de configuração, para que o algoritmo possa entender como a aplicação foi projetada. A segunda etapa de processamento de dados é a execução do algoritmo MonoBreak. O algoritmo é dividido em seis sub-rotinas, sendo cinco de processamento de dados e a última de apresentação dos resultados. Nas duas primeiras etapas, o algoritmo transforma os arquivos de rastro de execução de funcionalidades e o arquivo de configuração para uma estrutura de dados que o algoritmo manipula. A terceira sub-rotina classifica as classes dos rastros de execução das funcionalidades com base nas nas camadas definidas no arquivo de configuração. A quarta sub-rotina gera uma tabela de similaridade entre todas as funcionalidades, aplicando sobre cada combinação um

cálculo que leva em consideração os pesos definidos no arquivo de configuração e o quanto as classes entre as funcionalidades são iguais. A quinta etapa utiliza a matriz gerada pela etapa quatro e é responsável pela identificação de quais funcionalidades apresentam percentual de similaridade maior do que o limite de percentual de decomposição informado no arquivo de configuração. A terceira etapa do processo é a avaliação do resultado gerado pelo MonoBreak, a partir da lista de funcionalidades gerada, o utilizador poderá iniciar o processo de migração das suas funcionalidades para o microsserviço. Como última etapa do processo, o utilizador pode iniciar o desenvolvimento das funcionalidades para o microsserviço, levando em consideração boas práticas a nível de padrões de projeto e análise de quais tecnologias serão melhor empregadas para o desenvolvimento dos microsserviços. A técnica foi avaliada através de um estudo de caso, onde foi comparada a decomposição realizada pela técnica com a decomposição executada por um especialista em um sistema de mapeamento de competência (SMC). O sistema tem como finalidade permitir a avaliação de usuários na participação de projetos. Como resultado da avaliação, é possível concluir que a técnica tem potencial para realizar a decomposição dos microsserviços de forma semiautomática.

3.2.5 Towards the Understanding and Evolution of Monolithic Applications as Microservices

O trabalho de ESCOBAR et al. (2016), propõe um processo que auxilia o desenvolvedor a decompor uma aplicação monolítica em microsserviços, através de diagramas resultantes da análise da camada dos dados pertencentes a cada *Enterprise Java Beans* (EJB) através de clusterização. O processo está dividido em três etapas: a primeira camada denominada injeção de dados, que trata do processamento do código fonte da aplicação, gerando como resultado um metamodelo no padrão KDM que é usado como *input* para a próxima etapa; para a segunda etapa, são identificadas as classes, interfaces e métodos presentes no metamodelo gerado pela primeira etapa, sendo estes classificados em dois possíveis clusters que são cluster de EJB e cluster de microsserviço. No cluster de EBJ, estarão presentes todas as entidades que possuem algum tipo de relacionamento e os EJBs presentes na aplicação monolítica. Com esse agrupamento definido, é gerado um grafo de dependência que será processado por

um algoritmo de clusterização, que tem como objetivo gerar dois grupos denominados esquerdo e direito, calculando um percentual de relacionamento entre os grupos; para a terceira e última fase, são gerados os diagramas de representação. Para o primeiro diagrama gerado, é apresentado o relacionamento entre os EJBs. O segundo, disponibiliza a visualização das classes utilizadas por cada grupo de EJB. O terceiro diagrama traz os microsserviços e seus grupos de EJBs, e por fim, o último diagrama apresenta o relacionamento entre os microsserviços gerados. Como forma de avaliação do trabalho, o processo foi aplicado sobre duas aplicações fictícias do tipo JavaEE, sendo uma um e-commerce e outra uma aplicação de gerenciamento de processos acadêmicos. Segundo os autores, os resultados obtidos foram satisfatórios devido ao sucesso na decomposição das aplicações através do auxílio dos diagramas gerados. Como limitações, é identificado a limitação da aplicação do processo apenas para aplicações criadas utilizando Java, além de ser um processo aplicado e baseado na análise estática do código fonte, de maneira que todos os relacionamentos formados em tempo de execução são ignorados para o processo.

3.3 Análise comparativa dos trabalhos

Nesta seção será realizada a comparação dos trabalhos apresentados no item 3.2. Foram elaborados alguns critérios para fins de comparação: principal contribuição; método de avaliação; contexto de avaliação; tipo de técnica; granularidade dos microsserviços e ferramenta.

Na Tabela 1 é apresentado uma visão geral sobre a relação entre os trabalhos relacionados e os critérios comparativos.

Tabela 1 - Visão geral dos trabalhos relacionados e critérios comparativos

Critérios		Artigos				
		GYSEL *et al.* 2016	LEVCOVITZ *et al.*, 2015	KNOCHE; HASSELBRING, 2018	ROCHA, 2018	ESCOBAR et al., 2016
Principal contribuição	Algoritmo	-	-	-	+	-
	Arquitetura	-	-	-	-	-
	Ferramenta	+	-	-	-	-
	Processo	-	+	+	-	+
Método de avaliação	Estudo de caso	+	+	+	+	+
	Experimento controlado	-	-	-	-	-
	Pesquisa	+	-	-	-	-
Contexto de avaliação	Indústria	+	+	+	+	+
	Academia	+	-	-	-	+
Tipo da técnica	Clusterização	+	-	-	-	+
Granularidade dos microserviços	É configurável	-	-	-	+	+
Ferramenta	IDE Plugin	-	-	-	-	-
	Protótipo	-	-	-	-	+
	Sistema	+	-	-	-	-

Fonte: Elaborado pelo autor.

Sobre as principais contribuições, apenas o trabalho de GYSEL et al (2016) teve como proposta uma ferramenta para decomposição de sistemas monolíticos em microsserviços. Os trabalhos de LEVCOVITZ et al. (2015), KNOCHE; HASSELBRING (2018) e ESCOBAR et al (2016) possuem como contribuição a criação de um processo para realizar a decomposição. O trabalho de ROCHA (2018), propõe a utilização de um algoritmo para realizar a decomposição.

Para os métodos de avaliação dos trabalhos selecionados, todos os trabalhos apresentaram um estudo de caso de suas propostas. No trabalho de GYSEL et al (2016), além do estudo de caso apresentado, também foi executada uma pesquisa para averiguar a qualidade resultante da ferramenta proposta. O ambiente dos trabalhos de GYSEL et al (2016) e ESCOBAR et al (2016) foram aplicados na academia e na indústria, já os trabalhos de LEVCOVITZ et al. (2015), KNOCHE; HASSELBRING (2018) e ROCHA (2018) foram aplicados exclusivamente na indústria.

Dos artigos selecionados, apenas os trabalho de GYSEL et al (2016) e ESCOBAR et al (2016) utilizaram formas de clusterização no desenvolvimento da ferramenta. Apenas os trabalhos de ROCHA (2018) e ESCOBAR et al (2016), permitem configuração sobre a granularidade em que os microsserviços serão gerados.

Apenas o trabalho de GYSEL et al (2016) apresentou um sistema para a decomposição de sistemas monolíticos para microsserviços, já ESCOBAR et al (2016) trouxe um protótipo.

3.4 Oportunidades de pesquisa

Identificadas através da análise dos artigos da seção 3.2, as oportunidades de pesquisa levantadas por este trabalho são: (1) a criação de mais processos e algoritmos que guiem os desenvolvedores e arquitetos de software na etapa de decomposição dos microsserviços; (2) a aplicação da decomposição independente da contribuição do trabalho, para que este seja avaliado pela academia e pela indústria, tendo assim, uma paridade de resultados que possam ser avaliados; (3) prototipagem de aplicações modelos, representando padrões de codificação, arquiteturas utilizadas e tecnologias que contribuem para o desenvolvimento e avaliação dos resultados.

Destas oportunidades, este trabalho desenvolve um processo como guia para o pesquisador e desenvolvedor na decomposição de sua aplicação alvo, a avaliação será através de um estudo de caso descrito na seção 5.1.

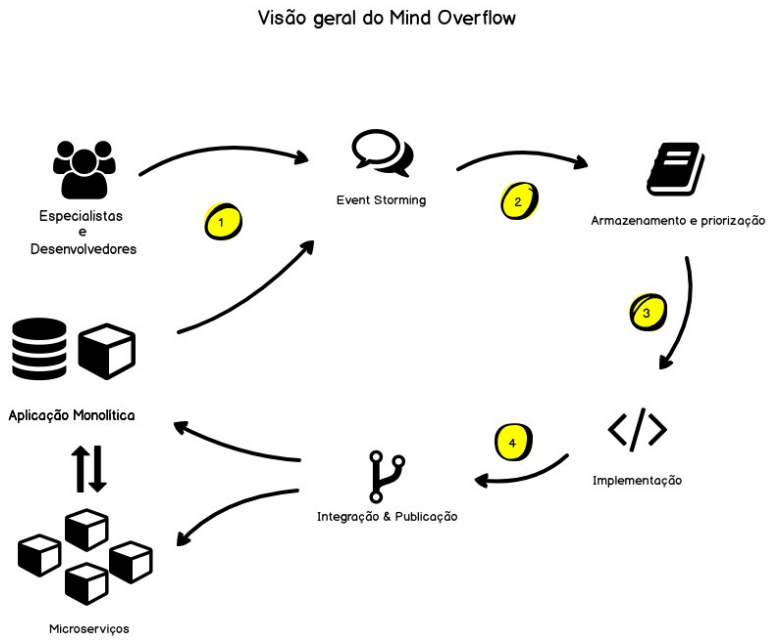
4 ABORDAGEM PROPOSTA

Esta seção apresenta o Mind Overflow, um processo agnóstico a tecnologia e a framework, que tem por objetivo guiar a decomposição de uma aplicação monolítica para uma arquitetura de microsserviços através de uma sequência de sistemática de decomposição.

4.1 Visão geral

A Figura 1 apresenta uma visão geral da abordagem proposta, a qual é formada por quatro etapas: (1) realização do *Event Storming*; (2) armazenamento e priorização das estórias; (3) implementação do microsserviço; e (4) integração com o monolítico. Essa abordagem utiliza técnicas e metodologias emergentes, incluindo *Domain-Driven Design*, métodos ágeis e *Event Storming*. Cada etapa é descrita a seguir.

Figura 1 - Visão geral da abordagem proposta



Fonte: Elaborado pelo autor.

4.1.1 Etapa 1: Realização do Event Storming

Essa etapa foca no levantamento dos requisitos e suas especificações. A técnica de *Event Storming* foi utilizada para este propósito. Criada por Brandoline (BRANDOLINE, 2013), com o objetivo de engajar a equipe técnica e especialistas de domínio, a técnica se caracteriza pela utilização de *stickers* coloridos em um espaço de discussão organizado de forma cronológica e de maneira linear, onde a finalidade é o entendimento dos processos de negócio.

Normalmente esta atividade de *storming* é dividida não apenas em um dia, mas sim, em alguns dias para que o que está sendo proposto seja repensado pelos

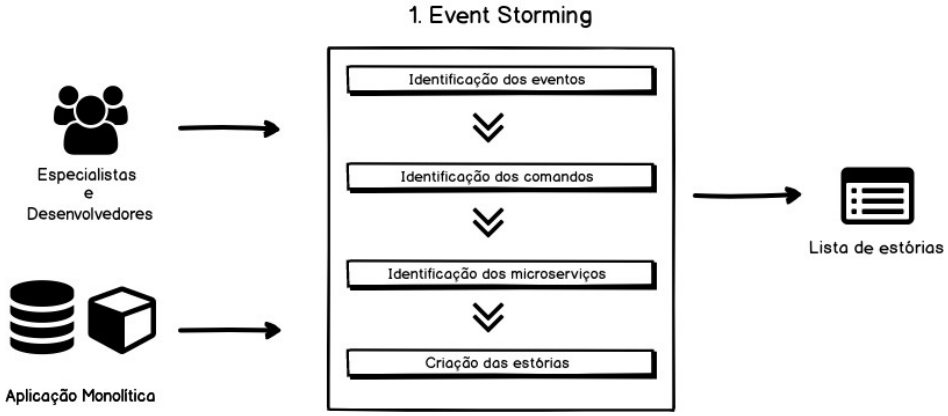
participantes, objetivando a maturidade e a eficácia das ideias para o negócio da organização.

Para o processo proposto, nesta etapa devem ser alcançados os seguintes objetivos através do *Event Storming*:

1. **Identificação dos eventos:** são acontecimentos que devem ser nomeados no passado, por exemplo, quando um produto é criado na aplicação, este acontecimento deve ser nomeado como, por exemplo, *ProductCreated*;
2. **Identificação dos comandos:** são as execuções antecedentes aos eventos, que fazem com que os mesmos sejam disparados. Seguindo o exemplo, um comando para o evento *ProductCreated* poderia ser o comando *CreateProduct*;
3. **Identificação dos microsserviços:** o objetivo de cada microsserviço resultante da aplicação desse processo é que cada domínio da aplicação, juntamente com seus sub-domínios, esteja presente em um único lugar, de forma que a linguagem utilizada no domínio seja compreendida por todos os membros da equipe sem ubiquidade. Lembrando que pode existir comunicação entre os domínios, isso será visto na etapa de implementação, quando for explicado a parte de disparo de eventos;
4. **Criação das estórias:** já com os eventos definidos e os microsserviços identificados, é necessário identificar o comportamento e fluxos que o negócio tem. Para cada estória, são executados comandos e são disparados eventos que terão uma finalidade, sendo estes passos documentados, é a definição de uma estória.

Dependendo do tamanho e abrangência da aplicação monolítica, podem ser identificadas centenas de estórias, o que pode inviabilizar o mapeamento do negócio como um todo. O processo proposto tem por objetivo ser um facilitador. Dessa forma, é aconselhável a priorização da migração do core do negócio em primeira instância, juntamente com as estórias que possuem maior relevância na migração para microsserviços. Posteriormente, o mesmo processo pode ser aplicado novamente para as partes que faltarem, sem impacto no que já foi trabalhado. A Figura 2 descreve a primeira etapa do Mind Overflow. Com esses objetivos alcançados, a Etapa 2 será executada.

Figura 2 - Etapa 1 do Mind Overflow



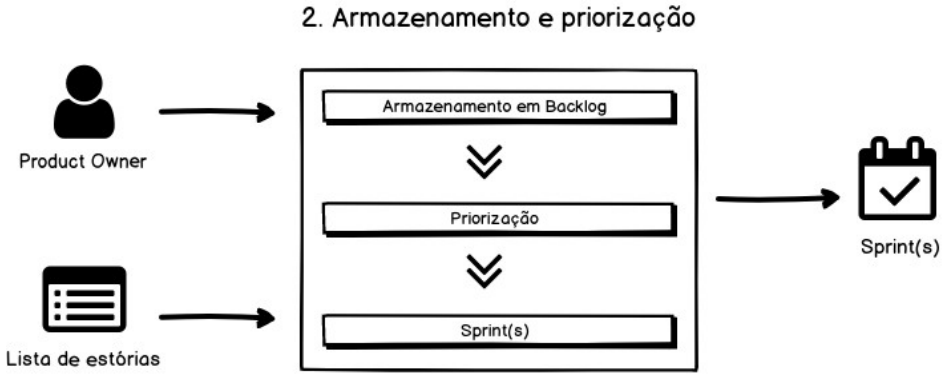
Fonte: Elaborado pelo autor.

4.1.2 Etapa 2: Armazenamento e priorização das estórias

Após identificadas as estórias, elas devem ser armazenadas em um repositório, também conhecido no agile como *Backlog*. Este repositório pode ser desde uma planilha de excel até uma aplicação de gestão de tarefas. O objetivo do *Backlog* é armazenar todas as tarefas pendentes que serão executadas nas próximas *Sprints*. O *Backlog* tende a aumentar conforme o tempo de vida do software. Dessa forma, as atividades pertencentes a ele, devem ser sempre priorizadas pelo quanto que irão agregar ao negócio da organização, também considerando o custo para o desenvolvimento da mesma.

Se a organização não tiver um método para classificação das atividades, este trabalho sugere a ponderação dessas atividades de acordo com a percepção da equipe ou do *Product Owner*. Para isso, podem ser considerados os valores 1, 2, 3, 5, 8, 13 e 21, algo similar ao *Planning Poker*. As atividades que possuem o maior valor agregado, devem ser priorizadas. A priorização das atividades não é a proposta do processo, ou seja, a organização fica livre para classificar como quiser as atividades que serão consideradas para a próxima etapa. A Figura 3 descreve a segunda etapa do Mind Overflow. Com as estórias classificadas, descritas e definidas na forma de Sprint, a Etapa 3 é executada.

Figura 3 - Etapa 2 do Mind Overflow



Fonte: Elaborado pelo autor.

4.1.3 Etapa 3: Implementação

Para a etapa de implementação, há algumas premissas que devem ser realizadas para a continuidade do projeto da construção do microsserviço. É fundamental que a equipe do microsserviço esteja alinhada de acordo com a linguagem onipresente utilizada no domínio da aplicação, e que tenha pré-entendido o objetivo para qual o microsserviço está sendo criado. Isso pode ser obtido nas reuniões de planejamento, caso a empresa siga as metodologias do Scrum.

O design da solução deve obrigatoriamente seguir um modelo de arquitetura em camadas MVC. Outra abordagem altamente recomendada, é que o microsserviço seja construído para disponibilizar API’s, ou seja, que tenha recursos disponibilizados através de requisições HTTP, sejam as requisições autorizadas ou não, dependendo das regras de cada microsserviço. Outra funcionalidade sugerida, é a execução de comandos periódicos programados pela própria aplicação, dessa forma, a aplicação fica independente de configuração do sistema operacional para executar suas crons, rotinas executadas repetidamente em um determinado intervalo de tempo. A Figura 4 exibe a estrutura do código de uma *feature* desenvolvida com base no Mind Overflow.

Figura 4 - Exemplo de Feature implementada através do Mind Overflow

```php
class UpdateProductFeature implements Feature
{
    private $slug;
    private $data;

    public function __construct($slug, $data)
    {
        $this->slug = $slug;
        $this->data = $data;
    }

    public function run()
    {
        $product = (new UpdateProductJob($this->slug, $this->data))->run();
        (new ProductUpdateEvent($product))->run();

        return $product;
    }
}
```

Fonte: Elaborado pelo autor.

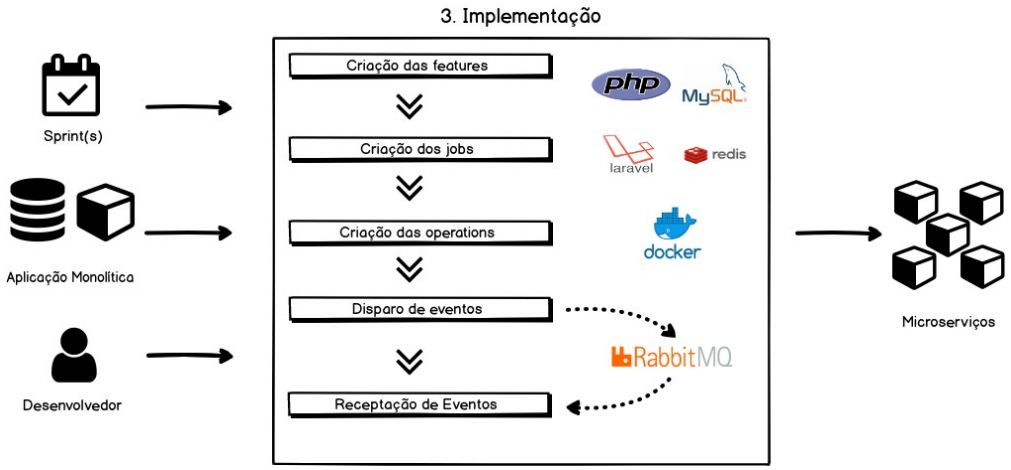
Com essas premissas satisfeitas, o desenvolvimento está dividido em cinco fases exibidos na Figura 5 e descritos a seguir:

1. **Criação das features:** a ideia é que cada estória de usuário vire uma feature na forma de classe para o microsserviço, mantendo de forma centralizada e representando todas as responsabilidades da estória através da execução de *jobs* e operações. O ideal é que haja apenas um método público na *feature* que será responsável pela chamada das demais funcionalidades, dessa forma, com uma nomenclatura que deixe expressivo a finalidade desta *feature*, é possível que qualquer desenvolvedor entenda facilmente quais são os impactos na alteração de um determinado fluxo e o que é contido para essa pequena abordagem de negócio, tornando-se assim, um código que ensina para quem o lê, quais são as características do domínio, sem a necessidade de um profundo *debug* dentre as funcionalidades. Uma *feature* também pode disparar eventos e ser chamada através da recepção de eventos e através dos controladores (que poderão ser acionados através de requisições REST e crons na forma de tarefas periódicas);
2. **Criação dos jobs:** o principal objetivo do *job* é ser responsável por uma restrita execução e que tenha apenas uma responsabilidade no fluxo do sistema. Aqui deve ser considerado o princípio de *Single Responsibility*, ou seja, este job deve ter apenas uma razão para existir e também o padrão *Open-Closed*, que é quando uma classe deve ser fechada para

alteração. Dessa forma o código fica altamente desacoplado, sendo possível a utilização de *mocks* para os testes unitários, que devem ser criados a nível de *job*. Como sugestão, um *job* não deve chamar outros *jobs*, nem invocar eventos, pelo motivo de rastreabilidade e padronização de código;

3. **Criação das operations:** para que não haja duplicação de código onde *features* normalmente chamam os mesmos *jobs*, é criado o conceito de *operation*. Uma *operation* deve conter apenas as chamadas para *jobs*, lembrando que estas chamadas, se sofrerem alterações, terão impacto nas demais *features* que utilizam essa *operation*. Como sugestão, uma *operation* não deve disparar eventos e nem disparar outras *operations* pelo motivo de rastreabilidade e padronização de código;
4. **Disparo de eventos:** como este é um processo agnóstico a framework e linguagem de programação, fica a critério do utilizador do processo escolher uma tecnologia que possua a possibilidade de disparo de eventos através de serviços de mensageria, como por exemplo o RabbitMQ. Os eventos aqui disparados, são os mesmos levantados pelo *Event Storming*, no início do processo. O objetivo com o disparo de eventos pela aplicação, é que seja possível particionar e reconhecer fluxos subjacentes tanto pela própria aplicação quanto em outras aplicações que estarão escutando a indicação de determinado evento, para então tomar alguma atitude sistemática, tornando-se assim, um sistema caracterizado reativo a eventos;
5. **Recepção de eventos:** como no disparo de eventos, a tecnologia utilizada por quem está implementando o processo, deve ser capaz de comunicar-se através de serviços de mensageria, como por exemplo o RabbitMQ. Quando o microsserviço identificar um evento ao qual ele é inscrito, a aplicação deve disparar uma *feature* implementada para este comportamento. Conforme descrito anteriormente, essa nova *feature* pode lançar um novo evento, formando assim, um encadeamento de execuções que irão satisfazer alguma característica do negócio.

Figura 5 - Etapa 3 do Mind Overflow

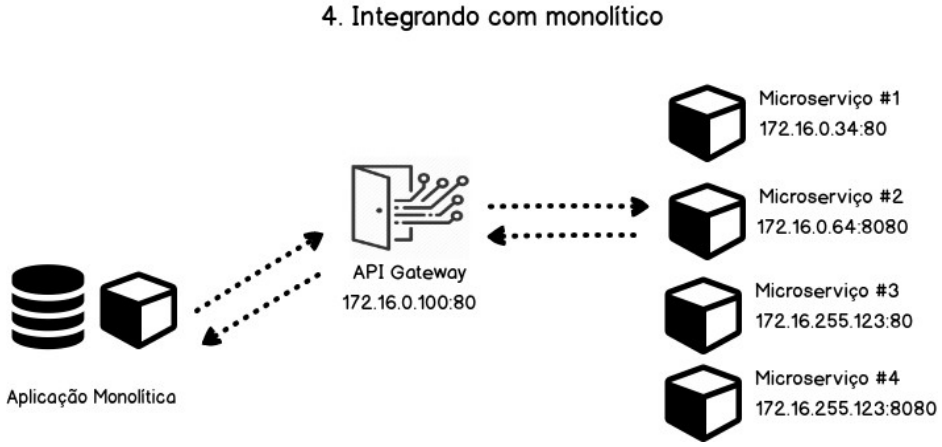


Fonte: Elaborado pelo autor.

4.1.4 Etapa 4: Integração com o monolítico

Para realizar a integração com o sistema monolítico de maneira incremental e transparente, será necessário o desenvolvimento de um *API Gateway*, que será responsável por receber todas as requisições da aplicação monolítica, e então, direcionar para o determinado microsserviço de destino. Para esta primeira abordagem, será utilizada a arquitetura REST para a realização das requisições por parte do sistema monolítico utilizando o protocolo HTTP. Definidos os *endpoints* para o *API Gateway*, e sendo estes *endpoints* direcionados para os microsserviços já implementados, é iniciada a adaptação no sistema monolítico para que as requisições possam ser realizadas de maneira transparente, não mais dependendo do legado, mas sim, dos novos microsserviços implementados. A Figura 6 exibe um exemplo de integração com *API Gateway*.

Figura 6 - Etapa 4 do Mind Overflow



Fonte: Elaborado pelo autor.

4.2 Principais características e abordagem

A abordagem desenvolvido neste trabalho possui as seguintes características: (1) ser agnóstico a framework e linguagem de programação; (2) visibilidade do negócio através do código fonte; (3) basear-se nas boas práticas de desenvolvimento de software, que são muito visadas hoje pelas organizações, como *Clean Code* (MARTIN, 2008) e SOLID (MARTIN, 2017); (4) manutenção facilitada decorrente das separações de responsabilidades e rápida identificação de fluxo de execução; (5) entrega de maneira natural uma estrutura quase pronta caso a organização deseja minimizar ainda mais o microsserviço, levando o mesmo ao nível de funcionalidade, podendo ser ela uma *feature*, um *job* ou uma *operation*; (6) viabilidade de escala horizontal dos microsserviços utilizando *load balancer* na infraestrutura; (7) domínio dos termos e da linguagem do domínio por todos os membros da equipe, umas vez que não haja traduções da equipe de negócio para a equipe de desenvolvimento; (8) permitir a utilização de mais de uma linguagem de programação para melhor resolver a demanda do negócio pelos microsserviços; (9) possibilidade de otimização e alterações no processo de negócio devido à rápida análise a partir dos impactos que as mesmas iriam gerar; e, por fim, (10) todos os *jobs* possuem testes unitários, sendo eles altamente desacoplados, porém, coesos ao microsserviço.

4.3 Premissas para utilização

Para que seja aplicado com êxito, algumas premissas devem ser satisfeitas: (1) utilizar uma linguagem no paradigma da orientação a objetos para a criação dos microsserviços; (2) possibilidade de alteração no código da aplicação monolítica, para proporcionar avanço de maneira incremental e contínua junto aos microsserviços criados; (3) disponibilidade e acessibilidade aos especialistas de negócio, para auxiliar nas definições dos processos de negócio; (4) definição de uma linguagem onipresente de acordo com os contextos e escopos dos microsserviços, para que não haja traduções da equipe de negócio para a equipe de desenvolvimento; (5) desenvolvedores com conhecimento moderado a respeito dos princípios SOLID, principalmente em relação à responsabilidade única e *open-closed* (MARTIN, 2017); e (6) utilização ou desenvolvimento de framework para disparo e recepção de eventos para comunicação entre aplicações, preferencialmente, adotando algum software de mensageria como base, por exemplo RabbitMQ.

5 AVALIAÇÃO

Para a avaliação deste trabalho, foi utilizada a metodologia de estudo de caso através do desenvolvimento de uma aplicação monolítica que foi decomposta através da aplicação do processo Mind Overflow.

5.1 Aplicação Escolhida

Para o estudo de caso foi desenvolvida uma aplicação monolítica de e-commerce criada pelo autor. A Tabela 2 apresenta a lista de funcionalidades que fazem parte da aplicação monolítica e que serão decompostas com o processo do Mind Overflow.

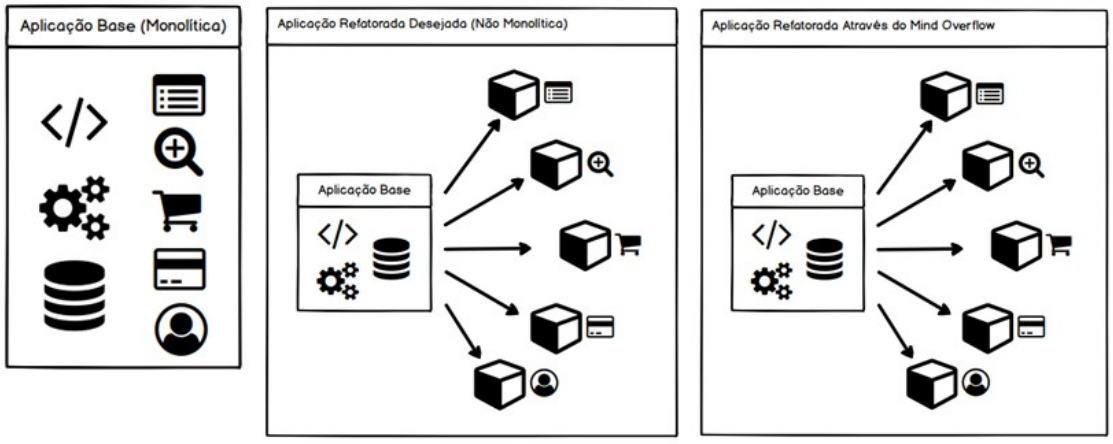
Tabela 2 - Funcionalidades da aplicação utilizada no estudo de caso

Nome	Descrição
Página inicial	Página inicial do ecommerce, mostrando últimos produtos cadastrados e informações gerais.
Autenticação de usuário	Possibilidade de registro e autenticação na aplicação para usuários do ecommerce.
Recuperação de carrinho	Baseado na última autenticação do usuário, caso um pedido não tenha sido finalizado, os itens do último carrinho são recarregados para o usuário continuar comprando.
Listar categorias	Aplicação disponibiliza listagem de categoria e subcategorias cadastradas pelo administrador.
Página de categorias	Visualizar todos os produtos de uma categoria, ou todas as subcategorias desta categoria.
Página de produto	Visualizar todas as informações do produto.
Adicionar item ao carrinho	Usuário adiciona item no carrinho de compras.
Remover item do carrinho	Usuário remove item do carrinho de compras.
Atualizar quantidade para item no carrinho	Usuário atualiza a quantidade de itens de um determinado produto no carrinho.
Aplicar cupom de desconto	Usuário pode aplicar vários cupons de desconto enquanto estiver comprando, sistema identifica qual o cupom que possui maior desconto, e este é considerado.
Visualizar itens no carrinho	Tela para conferência de itens, quantidades e valores do carrinho.
Finalizar pedido	Usuário finaliza a compra na aplicação.
Visualizar pedidos anteriores	Listagem de pedidos do usuário autenticado.
Cadastro de categoria	Administrador cadastra categoria ou subcategoria na aplicação.
Cadastro de produto	Administrador cadastra produto na aplicação.
Cadastro de cupom de desconto	Administrador cadastra cupom de desconto na aplicação.
Gerenciamento de pedido	Administrador atualiza informações do pedido, podendo mudar status.
Gerenciamento de usuário	Administrador gerencia informações dos usuários cadastrados.

Fonte: Elaborada pelo autor.

Para a obtenção do modelo de decomposição desejada, a aplicação monolítica foi analisada e, através de boas práticas de design e implementação, uma nova arquitetura envolvendo microsserviços foi implementada. Essa arquitetura, denominada de “Aplicação Refatorada Desejada”, foi comparada com o resultado da aplicação do processo Mind Overflow. Isto é, a aplicação refatorada desejada será comparada com a aplicação refatorada utilizando o Mind Overflow. A Figura 7 mostra como foram organizados os módulos e a utilização de microsserviços para cada aplicação.

Figura 7 - Aplicações desenvolvidas neste trabalho



Fonte: Elaborado pelo autor.

5.2 Tecnologías Utilizadas

As tecnologias utilizadas para as três aplicações criadas, foram selecionadas pelo autor com base na experiência de desenvolvimento adquirida após anos de trabalho na área. Pelo processo abordado como foco de estudo ser agnóstico a tecnologias específicas, as ferramentas aqui mencionadas não são restritivas para a aplicação do processo e análise:

- **Monolítico.** A aplicação monolítica foi desenvolvida utilizando PHP 7.2, Laravel 5.8, Mysql 5.7, Docker 19.03.
- **Aplicação Refatorada Desejada.** A aplicação refatorada desejada foi desenvolvida utilizando PHP 7.2, Laravel 5.8 para a aplicação frontend, Lumen 6.0 para os microsserviços, php-amqplib 2.10, Mysql 5.7, Docker 19.03 e RabbitMQ 3.7.
- **Aplicação Refatorada Através do Mind Overflow.** A aplicação refatorada através do Mind Overflow foi desenvolvida utilizando PHP 7.2, Laravel 5.8 para a aplicação frontend, Lumen 6.0 para os microsserviços, php-amqplib 2.10, Mysql 5.7, Docker 19.03 e RabbitMQ 3.7.

5.3 Métricas

Oito métricas foram selecionadas para permitir a avaliação de diferentes facetas das aplicações, incluindo linhas de código, conceitos de programação orientada a objetos (POO), acoplamento, complexidade e bugs. Tabela 3 apresenta as métricas selecionadas. Essas métricas foram contabilizadas utilizando a ferramenta phpmetrics, a qual trata-se de uma projeto *open-source* (https://phpmetrics.org). Para a realização da análise comparativa entre a aplicação desejada e a aplicação gerada, foram utilizados cálculos de distância (KELLY, 2006) entre a aplicação desejada e a aplicação gerada pelo Mind Overflow. A distância representa a diferença entre os valores apresentados pelas métricas para a aplicação refatorada gerada e aplicação refatorada através do Mind Overflow.

Tabela 3 - Métricas de avaliação

Grupo	Métrica	Descrição
LOC	**Linhas de código**	Medir o tamanho de um programa de computador, contando o número de linhas no texto do código-fonte do programa.
OOP	**Classes**	Número de classes de um programa de computador, utilizando programação orientada a objetos.
	Métodos	Número de métodos de um programa de computador, utilizando programação orientada a objetos.
Acoplamento	**Média de acoplamento aferente**	Representa a contagem de quantas classes diferentes referem-se à classe atual, por meio de campos ou parâmetros. Se esse número for alto, essa classe tem a alta chance de ser estável, diminuindo o risco do acoplamento.
	Média de acoplamento eferente	Representa a contagem de quantas classes diferentes a classe atual faz referência, por meio de campos ou parâmetros. Se uma classe possui acoplamento eferente alto, isso significa que ela depende de muitas classes.
Complexidade	**Média de complexidade ciclomática por classe**	Contagem do número de caminhos independentes que podem ser executados para cada método. O resultado da complexidade ciclomática indica quantos testes precisam ser executados para que se verifique todos os fluxos possíveis que o código pode tomar, a fim de garantir uma completa cobertura de testes.
	Média de complexidade relativa do sistema	Métrica composta da complexidade dentro de funções e entre elas. Ele mede a complexidade de um projeto de sistema em termos de chamadas de funções, passagem de parâmetros e uso de dados.
Bugs	**Média de bugs por classe**	Com base no número de operadores (nomes de métodos, operadores aritméticos) e no número de operandos (variáveis, constantes numéricas e de string). As métricas de Halstead dão uma idéia de quão complexas são as linhas de código (ou instruções) individuais. O Halstead Bugs tenta estimar o número de bugs que podem estar em um trecho de código específico.

Fonte: Elaborado pelo autor.

5.4 Resultados

Tabela 4 apresenta os resultados obtidos, considerando dois microsserviços Catálogo e Vendas. Esses microsserviços foram escolhidos, pois eles são representativos em relação aos demais. Os resultados serão apresentados seguindo a ordem de apresentação dos grupos de métricas.

LOC. O uso da abordagem proposta ajudou a produzir as funcionalidades desejadas, ao obter uma distância menor que 50% em ambos microsserviços, bem como permitiu chegar a funcionalidade desejada. No microsserviço Catálogo, a distância representou 38,68% (294/760) do código desejado. No microsserviço Vendas, a distância apresentou um valor maior de 49,27% (643/1305). Remoções de linhas de código podem ser feitas, visando obter um código mais próximo possível da aplicação desejada. O Mind Overflow foi efetivo em termos de linhas de código.

POO. Através da proposta, o valor das métricas de classes e de métodos aumentaram em relação a aplicação desejada. Considerando a métrica número de classes, para o microsserviço de Catálogo, a distância representou 88,89% (24/27). No microsserviço Vendas, a distância obtida foi de 108,70% (50/46). Na métrica de número de métodos, para o microsserviço de Catálogo, a distância representou 36,25% (29/80). Para o microsserviço de Vendas a distância representou 50% (65/130). Uma possível justificativa para que ambas as métricas tenham aumentado,

é devido a utilização dos princípios de responsabilidade única e *open-close* para a criação das *features* e dos *jobs*, sendo estes de responsabilidade específica, criando uma classe para cada responsabilidade, e consequentemente, método(s) para estas classes. O benefício da utilização deste padrão, será visto nas métricas de complexidade.

Acoplamento. Ambos microsserviços avaliados obtiveram um aumento nos acoplamentos aferente e eferente em relação a aplicação desejada. Para a métrica da média de acoplamento aferente, no microsserviço de Catálogo, a distância representou 120,83% (0,58/0,48). No microsserviço Vendas, a distância calculada foi de 113,43% (0,76/0,67). Na métrica referente a media de acoplamento eferente, para o microsserviço Catálogo, a distância representou 21,62% (0,4/1,85). Para o microsserviço de Vendas, a distância representou 24,88% (0,53/2,13). O aumento do valor médio do acoplamento aferente, resulta no aumento das chances da classe ser estável, diminuindo os riscos de acoplamento. Já o aumento do acoplamento eferente, significa que uma classe passa a depender de um maior número de classes, sendo este valor para ambos os microsserviços inferior a 50%, o Mind Overflow é considerado efetivo em termos de acoplamento.

Complexidade. Através do processo do Mind Overflow, todas as métricas avaliadas de complexidade alcançaram a redução de suas médias. Para a média da complexidade ciclomática por classe, no microsserviço de Catálogo, a distância representou 13,89% (0,2/1,44). No microsserviço de Vendas, a distância calculada foi de 21,02% (0,37/1,76). Na métrica média de complexidade relativa do sistema, o microsserviço de Catálogo apresentou a distância de 50,25% (16,27/32,38). Para o microsserviço de Vendas, a distância calculada foi de 70,67% (30,58/43,27). Reduzindo a complexidade, a facilitação da manutenibilidade é alcançada, dessa forma, o Mind Overflow é considerado efetivo em termos de complexidade.

Bugs. O valor da métrica de média de bugs por classe reduziu para ambos os microsserviços. Para o microsserviço de Catálogo e Vendas, a distância representou 50% (0,01/0,02 e 0,02/0,04). Com a redução da métrica, o Mind Overflow é considerado efetivo em termos de bugs.

Tabela 4 - Resultados obtidos

Grupo	Métrica	Catálogo			Vendas		
		Desejada	Gerada	Dist.	Desejada	Gerada	Dist.
LOC	Linhas de código	760	1054	294	1305	1948	643
OOP	Classes	27	51	24	46	96	50
	Métodos	80	109	29	130	195	65
Acoplamento	Média de acoplamento aferente	0,48	1,06	0,58	0,67	1,43	0,76
	Média de acoplamento eferente	1,85	2,25	0,4	2,13	2,66	0,53
Complexidade	Média de complexidade ciclomática por classe	1,44	1,24	0,2	1,76	1,39	0,37
	Média de complexidade relativa do sistema	32,38	16,11	16,27	43,27	12,69	30,58
Bugs	Média de bugs por classe	0,02	0,01	0,01	0,04	0,02	0,02

Fonte: Elaborado pelo autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou o Mind Overflow, o qual trata-se de uma abordagem suportada por um processo agnóstico a tecnologia e a framework para guiar a decomposição de uma aplicação monolítica para uma arquitetura de microsserviços através de uma sequência de sistemática de decomposição. Pesquisadores e desenvolvedores podem se beneficiar da abordagem proposta tanto para utilizá-la como base para novas abordagem de decomposição, bem como técnica para decompor aplicações legadas, respectivamente.

A avaliação inicial da abordagem, feita através de um estudo de caso, mostrou resultados promissores, em particular ela destacou que a abordagem é factível para decompor arquiteturas monolíticas para baseadas em microsserviços, inclusive reduzindo a complexidade ciclomática e produzindo microsserviços altamente coesos. Como trabalho futuro, pretende-se realizar novos estudos de casos, visando melhorar a avaliação e entender os pontos fortes e de melhorias da abordagem. Por fim, espera-se que o Mind Overflow sirva como base para que novos trabalhos surjam a partir dele, servindo como um ponto de partida.

REFERÊNCIAS

BENNET, K. **Legacy systems: Coping with success**. IEEE software, IEEE, v. 12, n. 1, p. 19–23, 1995.

BENNETT, K. H.; RAJLICH, V. T. **Software maintenance and evolution: a roadmap**. In: ACM. Proceedings of the Conference on the Future of Software Engineering. [S.l.], 2000. p. 73–87.

BRANDOLINE, A. “**Introducing Event Storming**,” blog, Ziobrando’s Lair, 18 Nov. 2013; http://ziobrando.blogspot.de/2013/11/introducing-event-storming.html.

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. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [S.l.]: Addison-Wesley Professional, 2003. ISBN 0321125215.

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

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.

KELLY, D. **A Study of Design Characteristics in Evolving Software Using Stability as a Criterion**. IEEE Transactions on Software Engineering 32(5), 315-329. IEEE Trans. Software Eng.. 32. 315-329. 10.1109/TSE.2006.42. 2006.

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.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. [S.l.]: Prentice Hall, 2008. ISBN 0132350882.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [S.l.]: Prentice Hall, 2017. ISBN 0134494164.

ROCHA, D. P. **Monólise: Uma Técnica para Decomposição de Aplicações Monolíticas em Microsserviços**. 2018. 171f. Dissertação (mestrado) - Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, São Leopoldo, 2018.

VERNON, V. **Implementing Domain-Driven Design**. [S.l.]: Pearson Education, Inc., 2013, ISBN 9780321834577.