Aplicando Desenvolvimento Guiado por Testes para Auxiliar na Evolução de Arquiteturas Monolíticas para Microsserviços

Daniel Pinheiro Wagner[1], **Felipe Silva Fantoni**[1]

[1]Bacharelado em Engenharia de Software
Instituto de Ciências Exatas e Informática – PUC Minas
Rua Cláudio Manoel, 1162 – Belo Horizonte – MG – Brasil

{felipe.fantoni, daniel.wagner}@sga.pucminas.br

Abstract. *The many advantages of the microservice architecture make developers choose it over the monolithic architecture. So, companies submit their old systems to an evolutionary migration to take advantage of the advantages of this architecture. However, this migration can present challenges regarding its duration, its complexity and its maintainability. It is proposed in this work an evaluation of an architecture migration carried out using a Test-driven Development technique. Through an experimental and quantitative research, the objective is to evaluate the effectiveness of the Test-driven Development technique during the evolutionary migration from a monolithic architecture to microservices. To this end, 20 migrations are analyzed. Migrations are evaluated considering metrics associated with the time of development, the quality of the source code, and the quality of the test cases. Although the results do not point to a positive impact on migration time when Test-driven Development is used, they demonstrate an improvement in the overall quality of the system, by pointing out reductions in the overall complexity of the service and an increase in its maintainability.*
Keywords: *microservices, migration, monolith architecture, TDD, software modernization*

Resumo. *As diversas vantagens da arquitetura de microsserviços faz com que os desenvolvedores optem por ela sobre a arquitetura monolítica. Com isso, as empresas submetem seus antigos sistemas a uma migração evolutiva para usufruírem das vantagens dessa arquitetura. Contudo, essa migração pode apresentar desafios quanto a sua duração, sua complexidade e sua manutenibilidade. Propõe-se neste trabalho uma avaliação de uma migração de arquitetura realizada utilizando a técnica de Desenvolvimento Guiado por Testes. Por meio de uma pesquisa experimental e quantitativa objetiva-se avaliar a efetividade da técnica de Desenvolvimento Guiado por Testes durante a migração evolutiva de uma arquitetura monolítica para microsserviços. Para tal, são analisadas 20 migrações. As migrações são avaliadas considerando métricas associadas ao tempo de desenvolvimento, à qualidade do código fonte, e à qualidade dos casos de teste. Apesar dos resultados não apontarem um impacto positivo no tempo de migração quando o Desenvolvimento Guiado por Testes é utilizado, esses demonstram uma melhora da qualidade geral do sistema, ao apontarem reduções na complexidade geral do serviço e aumento de sua manutenibilidade.*
Palavras-chave: *microsserviços, migração, arquitetura monolítica, TDD, desenvolvimento guiado por testes, modernização de software*

Bacharelado em Engenharia de Software - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): Laerte Xavier - laertexavier@gmail.com
Orientador acadêmico (TCC I): Lesandro Ponciano - lesandrop@pucminas.br
Orientador do TCC II: Lesandro Ponciano - lesandrop@pucminas.br

Belo Horizonte, 14 de maio de 2021.

1. Introdução

Atualmente, existe uma grande tendência de migração de sistemas monolíticos para estruturas de microsserviços [De Lauretis 2019]. O número de aplicações construídas com base na arquitetura de microsserviços aumenta à medida que suas vantagens vão sendo notadas por desenvolvedores e empresas [Abdel Khaleq and Ra 2019]. Nesse sentido, as principais vantagens obtidas ao optar por essa arquitetura são a garantia de melhor manutenibilidade, escalabilidade, reusabilidade e disponibilidade [Al-Debagy and Martinek 2018]. Sendo assim, os sistemas construídos de forma monolítica, que possuem diferentes componentes combinados em um único programa a partir de uma única plataforma, deixam de ser a primeira opção em razão do seu alto acoplamento e a sua maior possibilidade de gerar uma falha geral do sistema [Gos and Zabierowski 2020].

Entretanto, essa migração arquitetural pode demandar um longo tempo de duração e gerar uma baixa qualidade do sistema resultante [Fritzsch et al. 2019]. Devido ao aumento na comunicação de dados necessária entre os vários serviços, e as diversas refatorações necessárias durante a migração, a complexidade pode gerar baixa qualidade e mais tempo gasto durante a evolução [Sotomayor et al. 2019]. Nesse contexto, o problema a ser tratado por este trabalho é o **longo tempo de duração e a baixa qualidade dos sistemas gerados em uma migração de uma arquitetura monolítica para microsserviços**.

Existem diversas maneiras de realizar essa evolução de arquitetura, podendo ser com ou sem auxílio de ferramentas. Neste trabalho, é observada uma migração sem auxílio de ferramentas. Utiliza-se o Desenvolvimento Guiado por Testes (TDD, do inglês *Test-driven Development*) como técnica para realizar a evolução arquitetural, a fim de explorar, delinear e analisar os problemas supracitados. Essa técnica de programação busca maior conhecimento por parte dos programadores sobre os casos de testes, auxilia na entrega de produtos em um curto período de tempo e garante maior qualidade de software [Rocha et al. 2019]. Ao utilizá-la, o ciclo do desenvolvimento se inicia com a escrita dos testes, seguida pelo desenvolvimento do código e, por fim, são feitas refatorações de melhoria e correção [Fontela and Garrido 2013].

Desta forma, o objetivo geral do presente trabalho é **avaliar a efetividade da técnica TDD para realizar uma evolução de arquitetura de modo a se obter maior qualidade de software e menor tempo de migração.** No intuito de atingir o objetivo geral, os objetivos específicos são: 1) avaliar as suítes de teste geradas nas migrações com e sem o uso do TDD; 2) comparar o tempo de migração nos dois cenários; e 3) utilizar métricas para comparar a qualidade dos microsserviços gerados.

A fim de atingir os objetivos propostos, são analisados microsserviços resultan-

tes de migrações arquiteturais que têm como base um sistema monolítico desenvolvido na linguagem Python que contém os desafios típicos de um sistema monolítico a ser migrado. A partir de métricas de complexidade, qualidade e tempo são analisadas evoluções realizadas por 20 participantes no contexto com e sem a aplicação de TDD. Como resultado do estudo, é identificado que os sistemas evoluídos com a utilização dessa técnica possuem índices de qualidade superiores aos que não aplicam tal abordagem. A Complexidade Ciclomática e a Dificuldade Halstead atingem valores, aproximadamente, 38% e 46% menores, respectivamente. Além disso, é possível verificar que o tempo médio da evolução arquitetural foi semelhante em ambos contextos, possuindo aproximadamente 90 minutos.

Com este estudo, espera-se avaliar a efetividade da técnica de TDD durante a migração de arquitetura, a fim de atingir uma redução do tempo e dos custos dessa mudança. Espera-se também que essas reduções ocorram devido ao menor tempo de desenvolvimento, a maior facilidade em definir os casos de testes, e ao aumento da qualidade de software ao diminuir a quantidade de falhas sistêmicas e refatorações após o desenvolvimento.

Este trabalho está organizado em outras 6 seções. A Seção 2 aborda a fundamentação teórica, aprofundando-se nos conceitos e teorias utilizadas neste estudo. A Seção 3 apresenta os artigos que possuem relação com o presente trabalho. A Seção 4 contempla a metodologia proposta no experimento. A Seção 5 apresenta os resultados que foram obtidos no estudo. A Seção 6 demonstra discussões sobre os resultados obtidos, e limitações desta pesquisa. Por fim, a Seção 7 apresenta as conclusões e os trabalhos futuros.

2. Fundamentação Teórica

Nesta seção são detalhados os principais conceitos e técnicas que estão envolvidos na solução do problema apresentado. São eles: arquiteturas monolíticas, microsserviços, TDD e modernização de software.

2.1. Arquitetura Monolítica

A maioria dos sistemas tradicionais de empresas são projetados como aplicativos monolíticos. Esses são difíceis de escalar, corrigir e manter [Eski and Buzluca 2018]. Essa arquitetura é considerada uma forma simples de desenvolver e implementar um sistema, onde todos os componentes da aplicação são combinados dentro de uma única parte do software, formando um único bloco chamado de monólito [Goli et al. 2020]. Essa abordagem arquitetural normalmente é implantada em um ambiente com um único servidor, onde sua escalabilidade e disponibilidade é limitada a capacidade do mesmo [Villamizar et al. 2016].

Quando a aplicação atinge um tamanho considerável, as desvantagens dessa arquitetura começam a sobrepor as vantagens. Por exemplo, a base do código se torna extremamente complexa e incompreensível, e o tamanho do monólito começa a retardar o desenvolvimento e as correções de erros, gerando um obstáculo para a implantação contínua [Chen et al. 2017]. Esses são os principais motivos que ocasionam a evolução da arquitetura monolítica para microsserviços [De Lauretis 2019].

2.2. Microsserviços

Microsserviço é uma abordagem arquitetural no qual o software é dividido em pequenos serviços independentes. Cada serviço é mais refinado e autônomo, isolando funcionalidades de negócio e interagindo por meio de interfaces padronizadas. Os aplicativos compreendem um conjunto de pequenos serviços implantados de forma independente, cada um executando em seu próprio processo ou máquina virtual e, normalmente, interagindo via Interface de Programação de Aplicações (API, do inglês *Application Programming Interface*), Protocolo de Transferência de Hipertexto (HTTP, do inglês *Hypertext Transfer Protocol*) ou mensagens assíncronas [Merson and Yoder 2020]. Devido às necessidades de escalabilidade, a aplicação de microsserviços vem aumentando cada vez mais. Muitos sistemas distribuídos e baseados em nuvem evoluíram de arquiteturas monolíticas para arquiteturas de microsserviços [Hassan et al. 2017].

Além disso, o uso de microsserviços gera uma interface de componente explícita. A maioria das linguagens não possui um bom mecanismo para definir uma interface de publicação explícita. Normalmente, para impedir que ocorra a quebra do encapsulamento de componentes, são definidas normas e documentações, resultando em um acoplamento alto entre os componentes. Usando um mecanismo explícito de chamada remota, os microsserviços podem evitar mais facilmente essa situação [Fowler and Lewis 2014].

A Figura 1 apresenta uma comparação entre ambas as abordagens. Nela, observa-se que sistemas de arquitetura monolítica possuem um único bloco composto por: interface do usuário; camada de lógica e de regras de negócio; e camada de acesso ao banco de dados. Por outro lado, os sistemas de microsserviços possuem seus componentes lógicos e de acesso ao banco espalhados em diversos serviços diferentes e independentes.

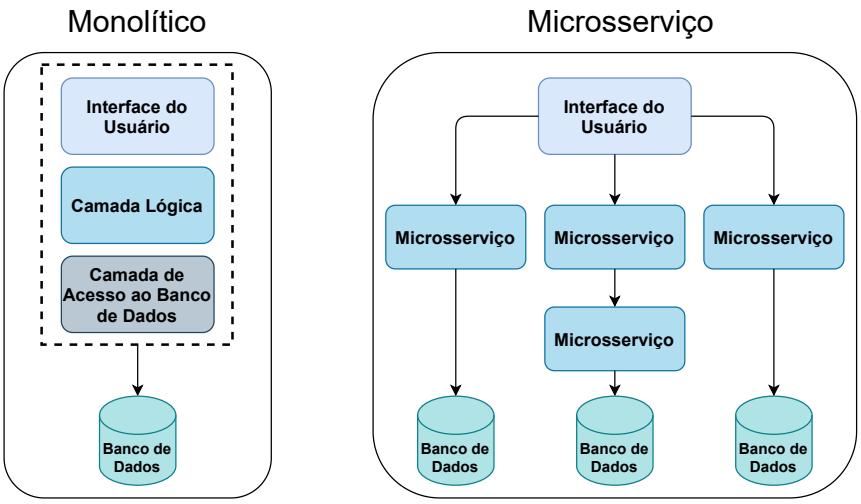

Figura 1: Ilustração arquitetural monolítica e de microsserviços

2.3. TDD

O Desenvolvimento Guiado por Testes foi apresentado por Kent Beck como parte do processo *Extreme Programming* (XP). Esse processo foi proposto como uma solução necessária para realizar análises e testes que possibilitem a refatoração e a integração

contínua [Bissi et al. 2016]. Contudo, o TDD ganhou popularidade para além do processo XP devido a suas vantagens de aumento da qualidade do software, diminuição do retrabalho e menor tempo total de desenvolvimento [Fontela and Garrido 2013].

Conforme indicado na Figura 2, o ciclo de TDD é composto por três passos: a) escrita de um teste para uma funcionalidade ainda não implementada, para garantir que o mesmo esteja funcionando e capturando as exceções corretamente; b) escrita de código que implemente uma solução funcional para o erro capturado; e c) evolução do código escrito através de refatorações para remoção de redundâncias [Rocha et al. 2019].

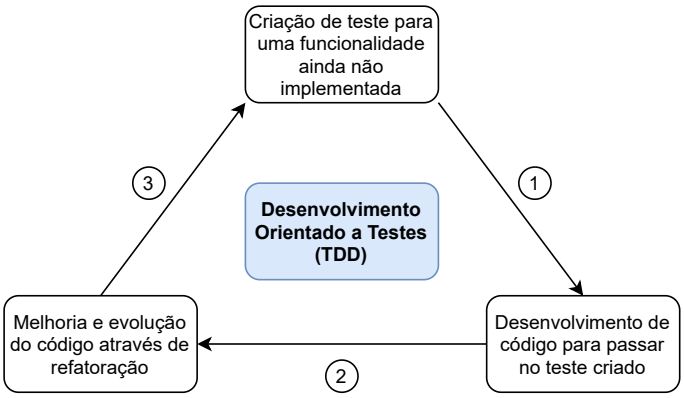

Figura 2: Ciclo de Vida do TDD

O ciclo de vida do TDD foi pensado desta forma a fim de forçar os desenvolvedores a pensarem sobre diversos aspectos de uma funcionalidade antes de iniciarem a codificação. Para cada pequeno pedaço de funcionalidade codificada, devem ser primeiro escritos testes de unidade e códigos que fazem estes testes passarem, ou seja, o desenvolvedor deve pensar em como testar uma funcionalidade que ainda não existe. Essa técnica de desenvolvimento também é capaz de fornecer diversos testes que os programadores podem executar a cada modificação no código, garantindo que o código refatorado, ou o código novo, não interrompa uma funcionalidade já em funcionamento [Crispin 2006].

2.4. Modernização de software

A modernização de software é o processo de evolução dos sistemas de software existentes, substituindo, re-desenvolvendo, reutilizando ou migrando os componentes e plataformas de software, quando as práticas de manutenção tradicionais não podem mais atingir as propriedades de sistema desejadas [Khadka et al. 2014]. A modernização de software envolve transformações de código complexas que convertem o código legado em novas arquiteturas ou plataformas, enquanto preservam a semântica dos programas originais [Iosif-Lazăr et al. 2015]. Isso acontece devido às rápidas mudanças no cenário tecnológico que levam as empresas a evoluir seus sistemas antes que se tornem obsoletos e problemáticos [Bruneliere et al. 2015]. Nesse contexto, a evolução de uma arquitetura monolítica para microsserviços pode ser vista como um caso de modernização.

3. Trabalhos Relacionados

Nesta seção são apresentados os trabalhos relacionados à evolução arquitetural, onde são explicados seus desafios e propostas diferentes abordagens para mitigar os problemas

nesse processo. Ainda que nem todos tratem exclusivamente desse assunto, apresentam abordagens relevantes ao presente estudo.

Chen et al. (2017) propõem uma abordagem de análise *top-down* baseada em fluxo de dados para encontrarem no software candidatos a migração para microsserviços [Chen et al. 2017]. Nesse trabalho, é criado um diagrama de fluxo de dados a partir da lógica de negócio com base na análise de requisitos. Em seguida, esse diagrama é condensado em um diagrama virtual e decomposto, onde as mesmas operações com os mesmo dados de saída são propostas para ambos. Por fim, ocorre a extração dos módulos desse fluxo abstrato, que representam os candidatos à migração. A comparação e avaliação mostram que a abordagem proposta identifica candidatos a migração de arquitetura por meio de um procedimento de implementação rigoroso e prático.

Eski e Bazluca (2018) desenvolveram uma técnica orientada ao agrupamento de microsserviços baseada em gráficos, utilizando acoplamento estático e evolutivo entre classes de software, para transformar as aplicações já existente em microsserviços [Eski and Buzluca 2018]. Inicialmente, o sistema é representado com um gráfico que demonstra as diferentes relações e acoplamentos entre as classes existentes. Em sequência, o gráfico gerado é submetido a uma técnica de agrupamento para, por fim, identificar os candidatos a microsserviços. Em conclusão, os autores afirmam que sua técnica é capaz de atingir um percentual de 89% de sucesso, comparando microsserviços extraídos com os resultados reais.

Zhang et al. (2020) percebem que é um grande desafio particionar adequadamente uma arquitetura monolítica em microsserviços [Zhang et al. 2020]. Com isso, é proposta uma abordagem de identificação automatizada de microsserviço que os extrai a partir de *logs* de execução e desempenho. O trabalho inicialmente coleta *logs* do sistema monolítico. Em seguida, os objetos do controlador (OB) são identificados como os objetos-chave para convergir os objetos subordinados (OS) fortemente relacionados. Posteriormente, a relação entre cada par de OB e OS é avaliada por uma matriz de relação tanto do ponto de vista funcional quanto do não funcional. Por fim, as classes são agrupadas em microsserviços, respeitando os conceitos de alta coesão e baixo acoplamento e equilíbrio de carga.

Amiri (2018) observou que um dos problemas atuais no projeto de arquiteturas de microsserviço é decompor um sistema em microsserviços coesos, fracamente acoplados e de granularidade fina [Amiri 2018]. Quando as funcionalidades de um sistema estão altamente interconectadas, é um desafio decompor o sistema em microsserviços apropriados. Com isso, é apresentado um método de identificação de microsserviço que decompõe um sistema usando a técnica de agrupamento. Para tal, é modelado um sistema como um conjunto de processos de negócios, levando em consideração dois aspectos da dependência estrutural e da dependência do objeto de dados das funcionalidades. O método proposto é capaz de identificar microsserviços coesos, fracamente acoplados e de granularidade fina a partir de um único processo de negócios, ou um conjunto de processos.

Os artigos supracitados demonstram soluções para se realizar a evolução de arquitetura de formas mais eficientes. São propostas novas técnicas e abordagens de extração de possíveis candidatos a migração e de ferramentas que fazem a evolução automaticamente. Da mesma forma, este trabalho também tem como objetivo estudar essa migração

evolutiva e os fatores que podem aumentar a qualidade dos sistemas gerados. Contudo, não são utilizadas novas ferramentas ou técnicas para realizar a evolução do sistema, a fim de analisar melhor os problemas inerentes a esse tipo de evolução. Dessa forma, para buscar um aumento na qualidade dos microsserviços gerados, é aplicada a técnica de TDD durante a evolução para complementar ao que os trabalhos discutem.

Khadka et al. (2014) realizam um estudo exploratório para identificar os principais motivadores em uma modernização de um sistema legado [Khadka et al. 2014]. Foram projetadas e analisadas entrevistas semiestruturadas com 198 profissionais, em uma abordagem de teoria fundamentada. Dessa forma, os principais motivos constatados para realizar uma modernização de sistema foram os altos custos de manutenção dos sistemas legados, e a obtenção de flexibilidade. Além disso, os principais desafios constatados pelos profissionais foram técnicos e organizacionais. Nesse contexto, o presente artigo também retrata a modernização de software. Contudo, o presente estudo analisa uma evolução de sistema executada na prática, enquanto os autores anteriores realizaram uma pesquisa para descobrirem os principais motivadores em uma migração evolutiva e os principais desafios dos profissionais.

Kerthyayana Manuaba (2019) implementa um método para aprimorar os testes realizados no *backend* do sistema através da combinação das técnicas de Desenvolvimento Guiado por Testes e Desenvolvimento Orientado por Comportamento [Kerthyayana Manuaba 2019]. Os resultados apontam que a combinação das técnicas é capaz de gerar testes de software de maneira superior em comparação com a utilização apenas da técnica de TDD, através do aumento da porcentagem de cobertura de testes. O presente trabalho também aplica o TDD a fim de observar seus impactos em um desenvolvimento. Contudo, o trabalho supracitado aplica o TDD juntamente com uma outra técnica de desenvolvimento, para aumentar a qualidade dos testes, enquanto o presente estudo aplica o TDD para aumentar a qualidade do software de um sistema gerado a partir de uma evolução de arquitetura.

4. Materiais e Métodos

O estudo apresentado neste trabalho trata-se de uma pesquisa experimental e quantitativa. O experimento possui a evolução de um sistema monolítico para uma arquitetura de microsserviços como objeto de estudo. Busca-se identificar o benefício da aplicação do TDD nesse processo, através da medição e comparação de métricas quantitativas. Essa seção apresenta as etapas necessárias para o alcance do objetivo do estudo.

4.1. Procedimentos

Para a realização do experimento e identificação dos pontos de avaliação, a migração é realizada em dois contextos diferentes. No primeiro contexto os desenvolvedores realizam a evolução sem a utilização do TDD, e no segundo com a aplicação de TDD. Os dois cenários tratam da migração do mesmo sistema monolítico desenvolvido utilizando a linguagem de programação Python. Esse sistema utiliza API de Transferência de Estado Representacional (REST, do inglês *Representational State Transfer*), tornando possível realizar o acesso de todas as suas interfaces através do protocolo HTTP. O software possui duas entidades principais, usuários e livros, e 13 funcionalidades. Ele se encontra

disponível no repositório[1] deste trabalho na plataforma GitHub.

Das funcionalidades do sistema a ser migrado, 10 são de gerenciamento e leitura das informações das entidades principais e são consideradas de baixa complexidade. As outras três funcionalidades são referentes a criação, exclusão e modificação de relacionamentos entre as duas entidades principais. Ou seja, são funcionalidades de cadastro de livro para um usuário, devolução do livro, e inclusão do usuário a uma fila de espera quando um livro não estiver disponível. Além disso, o sistema possui um arquivo de Notação de Objeto JavaScript (JSON, do inglês *JavaScript Object Notation*) que é responsável por salvar as informações do sistema, simulando um banco de dados.

Dessa forma, o sistema monolítico escolhido para ser migrado se aproxima de uma aplicação utilizada no mundo real, mas possui complexidade e tamanho reduzidos. Contudo, suas funcionalidades são pensadas para gerar desafios típicos em uma migração evolutiva desse tipo. Após a migração, o sistema monolítico se transforma em dois microsserviços, um para cada entidade do sistema. Os desafios de evolução presentes no atual experimento são:

1. Separação da lógica de negócio: cada microsserviço deve contemplar apenas uma única responsabilidade do sistema, sendo pouco abrangente.
2. Separação do banco de dados: com a criação de dois novos microsserviços, é necessário realizar a separação do banco de dados, visto que dois microsserviços distintos não devem acessar o mesmo banco de dados.
3. Criação de novas interfaces de acesso: como os dados não são acessados da mesma forma que no sistema monolítico, é necessário criar novas interfaces de acesso das APIs para que se consiga atingir o mesmo objetivo de negócio.
4. Mudança na lógica da aplicação: possuindo novas interfaces, a lógica de implementação do sistema também se altera.

Para este estudo foram convidadas 41 pessoas, estudantes e profissionais, para performarem a migração evolutiva. Dessas, 21 foram convidadas a realizarem a migração sem a utilização de TDD, e 20 para realizarem a migração com a utilização dessa técnica de desenvolvimento. Desconsiderando aqueles que recusaram o convite, aqueles que aceitaram o convite, mas não realizam o experimento, e aqueles cujo as migrações não atendiam os requisitos mínimos de participação[2], foram analisadas 10 migrações no contexto sem utilização do TDD, e 10 no contexto com a utilização. A separação dos participantes entre os contextos de migração foi feita de forma aleatória. O contato com os participantes e as migrações foram realizadas entre o dia 13 de abril de 2021 e o dia 27 de abril de 2021. As medições e coletas das métricas são feitas da mesma maneira para ambos os contextos, e os resultados gerados são comparados.

Aos participantes é enviado um link de acesso a um ambiente privado na plataforma GitHub Classroom, onde constam o sistema monolítico e as instruções para realização do experimento. Essa plataforma foi escolhida, pois garante um gerenciamento

[1] O repositório está na seguinte URL: `https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2021-1-tcc2-5732100-felipe-e-daniel`, sendo esse o repositório na organização do curso.

[2] *Checklist* de verificação do cumprimento dos requisitos mínimos de participação, disponível na URL: `https://docs.google.com/document/d/1QnRbrB2EyZWWaS0mYGpNl-KOVpD8lRsjxo0xw2BHOsQ/edit?usp=sharing`

dos microsserviços gerados por todos os participantes, além da extração das métricas de estudo deste trabalho. Além disso, a ferramenta garante que os códigos gerados pelos participantes são privados e que os desenvolvedores não conseguem visualizar os repositórios dos demais.

Ao acessar os ambientes de realização do experimento, antes de realizar a migração, os desenvolvedores respondem a questionários sobre seus conhecimentos[3]. A ferramenta cria automaticamente o *commit* inicial quando os desenvolvedores acessam o link com as especificações do trabalho pela primeira vez. Os *commits* seguintes são realizados pelos participantes à medida que os microsserviços são criados. Ao iniciar a migração, os desenvolvedores realizam todo o experimento sem realizar longas pausas ou intervalos. Durante o experimento, é utilizado o padrão *Strangler application*, onde há a modernização da aplicação de forma incremental em torno do sistema legado [Richardson 2019]. Nesse cenário, o microsserviço é gerado de forma gradativa e modular.

Com a definição dos microsserviços, é iniciada a migração com as especificidades de cada contexto. Contudo, independente do contexto, os desenvolvedores começam a evolução da entidade usuários, para depois realizarem a de livros. Os participantes encarregados do cenário em que se utiliza o TDD, iniciam escrevendo casos de teste para cada módulo e função que é reescrita para se adequar aos microsserviços. Em seguida, é realizada a implementação do código para satisfazer os casos de testes criados e, por fim, ocorre a refatoração do código visando a melhoria e evolução do mesmo. No outro cenário, os desenvolvedores iniciam implementando os microsserviços definidos nas análises das funções e módulos, sem necessariamente possuir a definição dos casos de teste. Após a implementação das funcionalidades, são realizados os testes para validar o que foi produzido. Em seguida, é realizada a correção dos problemas identificados no sistema.

4.2. Métricas e Avaliações

Este trabalho se baseia em métricas associadas ao tempo e à qualidade do software e da suíte de testes. A medição da qualidade é realizada seguindo a norma ISO/IEC 25010, que define em um modelo quais características de qualidade são levadas em consideração ao avaliar as propriedades de um produto de software [ISO 2017]. A manutenibilidade se refere a facilidade para realizar alterações de melhorias, extensões de funcionalidade e correções de defeitos, erros e falhas. E a complexidade do sistema está relacionada a uma medida interna da qualidade do software [ISO 2017].

As métricas consideradas são:

- **Tempo de duração da migração** de monolítico para microsserviço. A migração é considerada finalizada quando os microsserviços executam de forma bem sucedida. Quanto maior a duração, maior o tempo gasto para realizar todas as atividades da migração. Quanto menor o tempo, menor é a duração de tais atividades. Busca-se verificar como as atividades de TDD afetam esse tempo.
- **Índice de Manutenibilidade Geral**. Descreve a manutenibilidade do código, ou seja, o nível de dificuldade em realizar correções e melhorias. Os resultados são

[3]Questionário sobre o conhecimento dos participantes que não usam TDD, disponível na URL: `https://forms.gle/2gTcB2AjuUuKvtC58` Questionário sobre o conhecimento dos participantes que usam TDD, disponível na URL: `https://forms.gle/NN4rqBhpaMs28obAA`

representados em uma escala de 0 a 100. Quanto maior o valor, maior a facilidade em realizar manutenções no código. Quanto menor o valor, mais complexa a manutenção. [Martins et al. 2019].

- **Complexidade Ciclomática**. Representa o número de caminhos linearmente independentes no código. Se o código não contém nenhuma instrução de fluxo de controle, o código contém um único caminho e sua complexidade ciclomática é 1. Da mesma forma, se o código possui uma estrutura condicional, a complexidade ciclomática é 2, porque existem dois caminhos, um para verdadeiro e outro para falso. Quanto menor o valor, menor a complexidade do software. Quanto maior o valor, maior a complexidade do sistema. [Ståhl et al. 2019].
- **Número de *Bugs* Entregues Halstead**. Estima o número de defeitos na implementação. O número de *bugs* entregue por arquivo deve ser menor que 2 para que o sistema seja considerado minimamente utilizável. O objetivo da métrica é estimar o número de defeitos presente em cada módulo. Quanto maior o valor, maior o número de defeitos presentes no sistema. Quanto menor o valor, menor o número de defeitos encontrados no sistema [Hamer and Frewin 1982].
- **Dificuldade Halstead**. Demonstra o quão difícil é lidar e compreender um código. A dificuldade é afetada pela estrutura do programa. O uso de operandos redundantes e a falta de estrutura de controle de alto nível aumentam a dificuldade de compreensão. Quanto maior o valor, maior a dificuldade. Quanto menor o valor, menor a dificuldade [Naib 1982].
- **Cobertura de *Branch***. Aponta quais resultados de decisão foram testados. A cobertura de um arquivo é determinada pelas execuções reais divididas pelas oportunidades de execução. Cada linha no arquivo é uma oportunidade de execução, assim como cada destino de ramificação. Quanto maior o valor, maior a cobertura. Quanto menor o valor, menor a cobertura [Shams and Edwards 2015].

A demonstração dos cálculos das métricas está disponível no Apêndice A.

Para realizar a medição do tempo de migração, são coletados os horários de *commits* realizados nos repositórios privados dos desenvolvedores. Para realizar a comparação, é calculada a diferença de tempo entre o último *commit* do desenvolvedor e o primeiro *commit* automático feito quando o participante começa a realizar a migração. Para realizar a medição das métricas de qualidade definidas, ao final do desenvolvimento, os sistemas migrados pelos desenvolvedores são submetidos a ferramenta Radon[4] que faz os devidos cálculos. O cálculo da cobertura de *branches* é feito por meio da ferramenta Coverage.py [5].

A fim de identificar a correlação entre as métricas do estudo, é utilizado o Coeficiente de Correlação de Spearman. Esse é um índice não paramétrico usado para medir a correlação de duas variáveis. Observa-se que se o coeficiente de correlação é positivo, um aumento na variável independente ocasiona um aumento na variável dependente. Assim como, se o coeficiente de correlação é negativo, um aumento na variável independente ocasiona uma diminuição na variável dependente. Os valores de correlação ficam entre -1 e 1, sendo valores negativos a indicação de correlação negativa; valores positivos

[4]Documentação da ferramenta utilizada, disponível na URL: `https://radon.readthedocs.io/en/latest/index.html`

[5]Documentação da ferramenta utilizada, disponível na URL: `https://coverage.readthedocs.io/en/coverage-5.5/branch.html`

indicação de correlação positiva; e 0 indicação de ausência de correlação. [Hernández-Aguirre et al. 2008].

Todos os repositórios considerados válidos para o estudo são submetido a um *script* Python desenvolvido pelos realizadores da pesquisa, para que as métricas de tempo, qualidade, conhecimento e cobertura sejam coletadas e agrupadas em um único arquivo de *Comma-separated values* (CSV) integrado. Em sequência, a linguagem R é utilizada para plotar os gráficos com os resultados obtidos. Todas as barras de erro apresentadas nos resultados são para um erro estatístico obtido pela distribuição t-student considerando um nível de confiança de 95%. No caso do cenário monólito, apresentado para fins comparativos, tem-se apenas uma instância e, portanto, não há erro estatístico associado.

5. Resultados

Ao longo desta seção são apresentados os resultados obtidos pela pesquisa. Primeiro são apresentados os conhecimentos dos participantes do estudo. Após isso, analisa-se as principais métricas de qualidade dos projetos, a cobertura de *branches* e o tempo médio utilizado para realizar a migração do software.

5.1. Características dos Participantes

A Tabela 1 mostra o conhecimento médio dos participantes a respeito das tecnologias e abordagens utilizadas no projeto. Observa-se que nenhuma média de conhecimento foi inferior a 3. Contudo, a utilização do Framework Flask obteve os menores valores em ambos os contextos. Por outro lado, a utilização do Postman e do Github obtiveram as maiores médias. Os participantes que não utilizam TDD não respondem as perguntas sobre TDD e Mock em Testes, pois não se aplicam a seu contexto de evolução do sistema.

Tabela 1: Distribuição do Conhecimento Médio dos Participantes em uma Escala de 0 (Nenhum Conhecimento) a 5 (Muito Conhecimento).

	Conhecimento Médio por Participantes			
	Não utilizaram TDD		Utilizaram TDD	
Ferramenta / Linguagem	Valor médio	Erro	Valor Médio	Erro
Python	3.3	0.48	3.7	0.46
Github	3.8	0.74	4.4	0.35
Microsserviços	3.2	0.56	3.2	0.43
Flask	3.1	0.71	3.2	0.70
Postman	3.8	0.88	4.3	0.56
TDD	-	-	3.8	0.43
Mock em Testes	-	-	3.7	0.33

5.2. Média de Dificuldade Halstead por Cenário

Na Figura 3, no eixo X, estão os 3 tipos de sistema analisados neste trabalho: monolítico; microsserviço gerado sem TDD e; microsserviço gerado com TDD. No eixo Y estão os valores médios da Dificuldade Halstead. Observa-se que essa métrica foi menor na migração com TDD em relação a migração sem TDD. Além disso, a migração com TDD também obteve uma média menor que o sistema monolítico inicial.

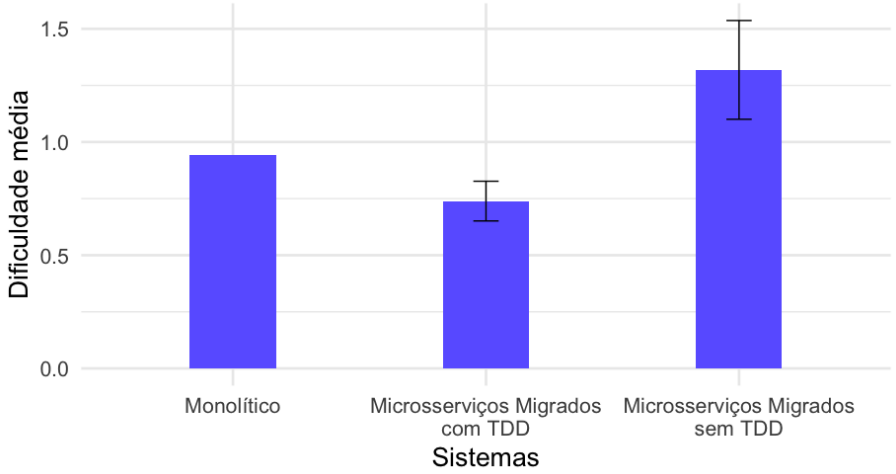

Figura 3: Dificuldade média Halstead

5.3. Média de *Bugs* Entregues Halstead por Cenário

A métrica de *bugs* entregues representa a estimativa de defeitos esperados nos módulos do sistema. Como ilustrado na Figura 4, o sistema monolítico inicial possui uma probabilidade estimada de 0.02 de possuir defeitos entregues em sua versão final. O microsserviço criado a partir do contexto da utilização do TDD reduziu esse valor para 0.01, enquanto o microsserviço do contexto sem a utilização do TDD aumentou o valor estimado da métrica para 0.03.

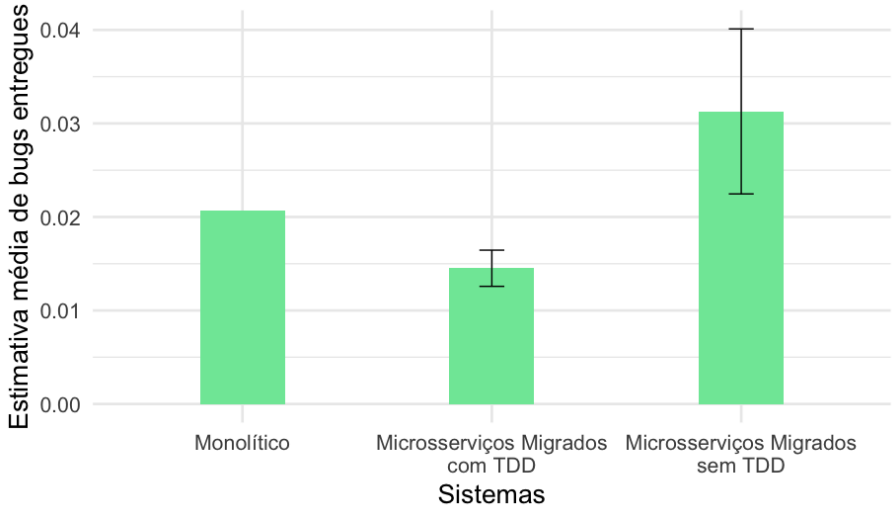

Figura 4: Média de *bugs* entregues por sistema

5.4. Análise da Complexidade Ciclomática

Observa-se na Figura 5 a comparação entre a complexidade ciclomática média dos 3 sistemas. A complexidade ciclomática geral se mantém baixa independentemente do sistema, devido ao tamanho reduzido das aplicações. Contudo, se observa um crescimento da métrica apenas no microsserviço migrado sem TDD. O valor de 1.86 no sistema monolítico se elevou para 2.44 nesse contexto, enquanto no contexto com a utilização de TDD o valor reduziu para 1.50.

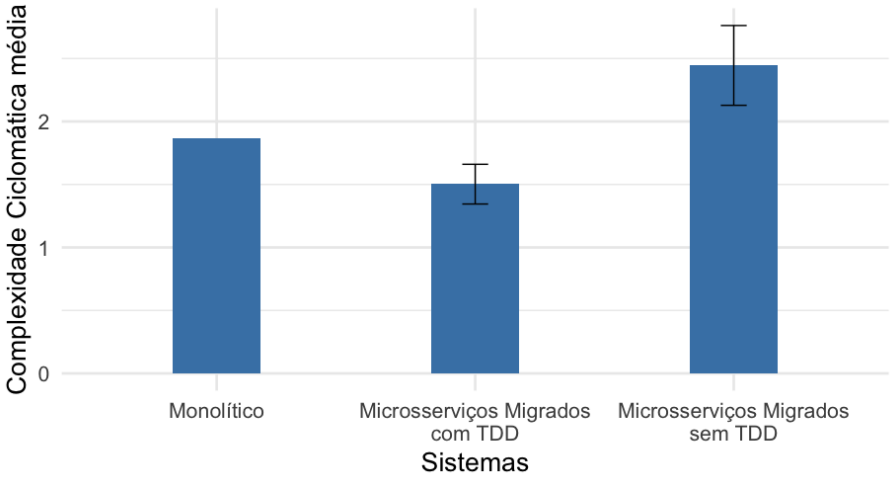

Figura 5: Complexidade Ciclomática Média

5.5. Cobertura Média de *Branches*

Ao realizar a execução de testes de *branches*, observa-se os destinos que são visitados nos softwares gerados com a utilização de TDD e também nos que não utilizam essa técnica de desenvolvimento. Como mostrado na Figura 6, os softwares gerados com a técnica de TDD têm uma maior cobertura de *branches* comparado aos que não contam com essa abordagem.

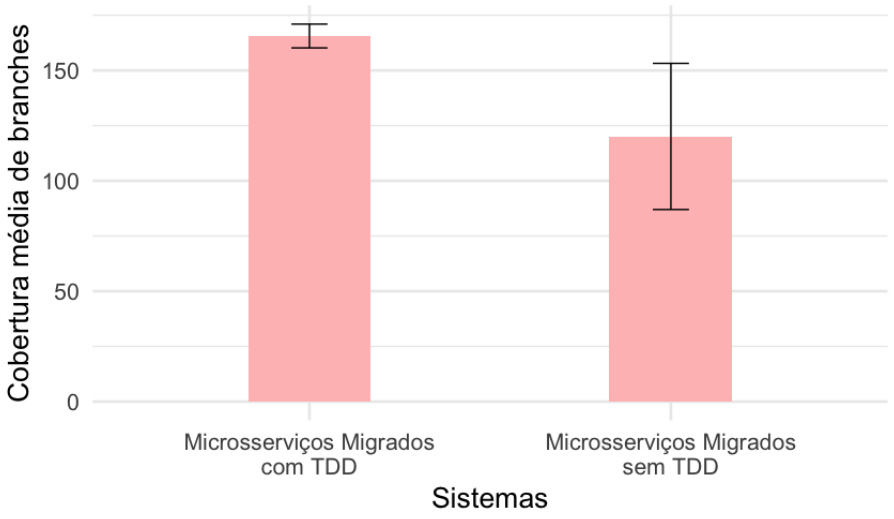

Figura 6: Cobertura Média de *Branches*

5.6. Índice de Manutenibilidade Geral Média dos Sistemas por Cenário

A Figura 7 apresenta o índice de manutenibilidade média dos softwares gerados, comparando também com o software monolítico base para a realização dos experimentos. O índice de manutenibilidade aumenta nos softwares migrados utilizando a técnica de desenvolvimento guiada por testes, enquanto diminui nos sistemas em que não foi empregada tal técnica.

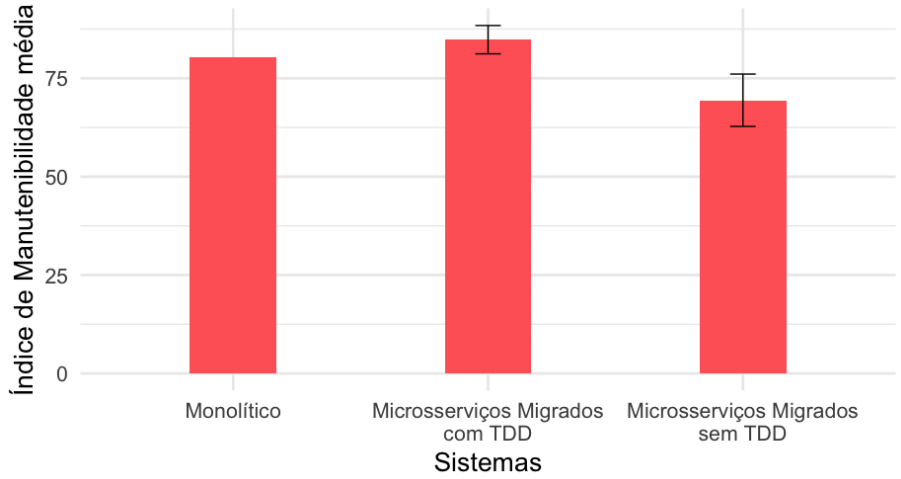

Figura 7: Índice de Manutenibilidade Média por contexto do sistema

5.7. Tempo Médio Utilizado para Realizar a Evolução Arquitetural

O tempo médio gasto para realizar a evolução arquitetural foi semelhante entre as duas abordagens utilizadas nos experimentos. Os diferentes contextos de realização do experimento obtiveram um tempo médio de migração de aproximadamente 90 minutos. Contudo, observa-se na Figura 8 que a evolução da primeira entidade do sistema monolítico demanda mais tempo no contexto em que se aplica o TDD. Por outro lado, nesse mesmo contexto, a evolução da segunda entidade demanda menos tempo.

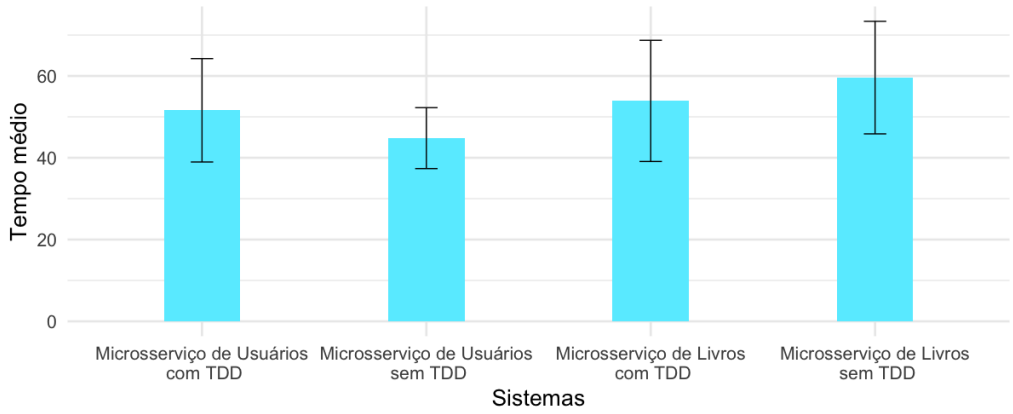

Figura 8: Tempo médio gasto em minutos por evolução de entidade do sistema

5.8. Correlação entre as métricas

Observa-se uma correlação entre as métricas de complexidade e qualidade analisadas. Nesse contexto, a Tabela 2 expõe uma correlação positiva entre as métricas utilizadas neste estudo para ambos os contextos de migração. Percebe-se que a maior correlação é de 0.88 entre a métrica de Dificuldade Halstead e a de Probabilidade de *bugs* entregues. Também observa-se que quanto maior a Complexidade Ciclomática, maior a dificuldade de lidar com o sistema. Dessa forma, como a métrica de dificuldade se mostrou maior nos microsserviços gerados sem aplicação do TDD, a probabilidade de *bugs* serem entregues nesses sistemas é maior, assim como sua Complexidade Ciclomática.

Tabela 2: Correlação entre as métricas

Métricas correlacionadas	Complexidade Ciclomática	Dificuldade Halstead	Número de Bugs Entregues Halstead
Complexidade Ciclomática	1.00	0.79	0.87
Dificuldade Halstead	0.79	1.00	0.88
Número de Bugs Entregues Halstead	0.87	0.88	1.00

6. Discussões e Limitações

Com o intuito de responder a questão de pesquisa proposta, este trabalho verificou o impacto que a utilização da técnica TDD causa em uma migração arquitetural evolutiva. Os resultados obtidos através da coleta das métricas demonstram que a qualidade geral dos microsserviços que são migrados utilizando TDD é maior que a dos microsserviços que não utilizam essa técnica. A queda nos indicadores de Complexidade Ciclomática, Dificuldade Halstead e Número de *bugs* entregues Halstead, pode indicar que o TDD é capaz de auxiliar a reduzir o problema de alta complexidade e de baixa qualidade dos microsserviços que são gerados durante uma migração evolutiva. Além disso, os sistemas gerados no contexto do uso do desenvolvimento guiado por testes também garantem uma melhora nas métricas em relação ao sistema monolítico inicial.

O índice médio de manutenibilidade dos microsserviços gerados com a utilização da técnica de TDD apresenta um aumento. Isso se dá pela geração de códigos mais limpos e simples devido aos pequenos passos que o processo dessa abordagem exige, eliminando códigos desnecessários. Além disso, a confiabilidade do código é maior, pois a prática do TDD faz com que o projeto passe por testes de validação de forma frequente, impactando na manutenibilidade do sistema. Em contramão, os microsserviços gerados sem a utilização do TDD, obtiveram uma redução do índice de manutenibilidade em relação ao sistema monolítico. Devido à piora da qualidade geral do sistema observada pelas métricas de números de defeitos entregues, de dificuldade, e de complexidade ciclomática, as manutenções se tornam tarefas mais complexas.

Ao observar as métricas obtidas, pode-se destacar que não é observado um menor tempo de migração no contexto em que se utiliza o TDD. Dentre as definições de pesquisa deste trabalho, é esperado observar um menor tempo de realização das migrações que utilizam essa abordagem, devido a menor quantidade de correções de erros que são realizadas nesse contexto. Contudo, os resultados apontam que os valores são semelhantes nos dois cenários. Apesar disso, uma análise mais detalhada dos horários de *commit* aponta uma diferença no tempo gasto em cada entidade do sistema. Os desenvolvedores que utilizam TDD gastam mais tempo na migração da primeira entidade, devido aos casos de teste que constroem antes do desenvolvimento. Porém, esses gastam menos tempo na segunda evolução, pois realizam menos correções e adequações no código.

É observado que a média da cobertura de *branches* é maior no cenário em que foi utilizado o TDD. Nesse contexto, a maioria dos destinos percorridos na execução do código conseguem ser atingidos pelos testes. Isso se dá pela garantia que o processo de desenvolvimento orientado a testes propicia através de testes desenhados antes que cada funcionalidade seja desenvolvida. Percebe-se que, em um cenário que não é utilizado TDD, mesmo realizando testes após o desenvolvimento das funcionalidades, não

é atingida a mesma cobertura de caminhos, fazendo com que *bugs* possam passar despercebidos, aumentando o índice de *bugs* entregues. Além disso, o erro amostral é mais elevado no cenário em que não foi aplicado o TDD, tendo em vista que o número de testes realizado nesse contexto é menor comparado ao contexto em que foi aplicado o TDD.

Os resultados deste trabalho são extraídos de sistemas evoluídos sem o auxílio de ferramentas. Em contramão, as pesquisas dos trabalhos relacionados propõem ferramentas para realizar evoluções arquiteturais mais eficientes. Nesse contexto, os resultados obtidos no presente trabalho também se mostram relevantes em cenários onde a evolução arquitetural utiliza ferramentas, sendo capaz de auxiliar na realização de uma evolução arquitetural mais eficiente e com a geração de microsserviços de maior qualidade.

O trabalho desenvolvido possui limitações, assim como todo trabalho científico dessa natureza. A quantidade de desenvolvedores para participar do experimento é limitada, devido à especificidade das tecnologias que são utilizadas no estudo. Realizar o experimento com uma quantidade maior de participantes pode aumentar a significância estatística dos resultados adquiridos sobre os cenários. Além disso, devido ao tempo disponível para a realização do estudo, o sistema monolítico inicial foi desenvolvido para possuir menor tamanho e complexidade. Realizar o experimento com um sistema inicial maior e mais complexo pode auxiliar na observação da efetividade da aplicação do TDD.

7. Conclusão e Trabalhos Futuros

Este trabalho teve como objetivo avaliar a aplicação da técnica TDD em uma migração arquitetural evolutiva de um sistema monolítico para microsserviços. Para tal, foram analisadas e comparadas evoluções, de um mesmo sistema monolítico, em dois contextos: com a utilização do TDD e sem a aplicação dessa técnica de desenvolvimento. Para obter os dados, as ferramentas Radon e Coverage.py foram utilizadas para coletar as métricas de qualidade dos repositórios do Github Classroom dos participantes. Além disso, também foi desenvolvido um *script* em Python para coletar e agrupar as métricas de tempo de evolução e conhecimento dos participantes.

Os resultados apontaram que a aplicação do TDD na migração evolutiva possibilita uma melhora geral da qualidade dos microsserviços gerados. Foram observadas reduções nas complexidades ciclomática, de dificuldade e entrega de *bugs* do Halstead. Além disso, notou-se uma melhora na manutenibilidade geral dos microsserviços e na compreensão de seus códigos fonte, devido a queda de acoplamento. Contudo, não foi possível observar uma redução no tempo total de migração com a aplicação de TDD.

Como trabalhos futuros pretende-se realizar a evolução arquitetural em sistemas maiores e com uma complexidade mais elevada. Além disso, almeja-se praticar o experimento com um número maior de participantes. Dessa forma, seria possível verificar o comportamento das métricas de qualidade e de tempo em um contexto onde o sistema demanda maior conhecimento e mais tempo disponível para realizar as evoluções. Adicionalmente, são sugeridas as verificações de outras métricas de qualidade do sistema e da qualidade das suítes de testes geradas.

Referências

Abdel Khaleq, A. and Ra, I. (2019). Agnostic approach for microservices autoscaling in

cloud applications. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1411–1415.

Al-Debagy, O. and Martinek, P. (2018). A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154.

Amiri, M. J. (2018). Object-aware identification of microservices. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 253–256.

Bissi, W., Serra Seca Neto, A. G., and Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, 74:45 – 54.

Bruneliere, H., Cabot, J., Canovas Izquierdo, J. L., Arrieta, L. O., Strauss, O., and Wimmer, M. (2015). Software modernization revisited: Challenges and prospects. *Computer*, 48(8):76–80.

Chen, R., Li, S., and Li, Z. (2017). From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475.

Crispin, L. (2006). Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23(6):70–71.

De Lauretis, L. (2019). From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96.

Eski, S. and Buzluca, F. (2018). An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, XP '18, New York, NY, USA. Association for Computing Machinery.

Fontela, C. and Garrido, A. (2013). Connection between safe refactorings and acceptance test driven development. *IEEE Latin America Transactions*, 11(5):1238–1244.

Fowler, M. and Lewis, J. (2014). Definition of microservices. Disponível em: `https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture`. (Acesso em: 10 dez. 2020.

Fritzsch, J., Bogner, J., Wagner, S., and Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490.

Goli, A., Hajihassani, O., Khazaei, H., Ardakanian, O., Rashidi, M., and Dauphinee, T. (2020). Migrating from monolithic to serverless: A fintech case study. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 20–25, New York, NY, USA. Association for Computing Machinery.

Gos, K. and Zabierowski, W. (2020). The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153.

Hamer, P. G. and Frewin, G. D. (1982). M.h. halstead's software science - a critical examination. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, page 197–206, Washington, DC, USA. IEEE Computer Society Press.

Hassan, S., Ali, N., and Bahsoon, R. (2017). Microservice ambients: An architectural meta-modelling approach for microservice granularity. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 1–10.

Hernández-Aguirre, A., Villa-Diharce, E., and Barba-Moreno, S. (2008). An estimation distribution algorithm with the spearman's rank correlation index. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, page 469–470, New York, NY, USA. Association for Computing Machinery.

Iosif-Lazăr, A. F., Al-Sibahi, A. S., Dimovski, A. S., Savolainen, J. E., Sierszecki, K., and Wasowski, A. (2015). Experiences from designing and validating a software modernization transformation. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 597–607. IEEE Press.

ISO, I. (2017). iec/ieee international standard-systems and software engineering–vocabulary. Technical report, Technical report, ISO/IEC/IEEE 24765: 2017 (E).

Kerthyayana Manuaba, I. B. (2019). Combination of test-driven development and behavior-driven development for improving backend testing performance. *Procedia Computer Science*, 157:79–86. The 4th International Conference on Computer Science and Computational Intelligence (ICCSCI 2019) : Enabling Collaboration to Escalate Impact of Research Results for Society.

Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S., and Hage, J. (2014). How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 36–47, New York, NY, USA. Association for Computing Machinery.

Martins, J., Bezerra, C. I. M., and Uchôa, A. (2019). Analyzing the impact of inter-smell relations on software maintainability: An empirical study with software product lines. In *Proceedings of the XV Brazilian Symposium on Information Systems*, SBSI'19, New York, NY, USA. Association for Computing Machinery.

Merson, P. and Yoder, J. (2020). Modeling microservices with ddd. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 7–8.

Naib, F. A. (1982). An application of software science to the quantitative measurement of code quality. *SIGMETRICS Perform. Eval. Rev.*, 11(3):101–128.

Richardson, C. (2019). *Microservices patterns : with examples in Java*. Manning Publications Co., Shelter Island, NY 11964.

Rocha, F. G., Souza, L. S., Silva, T. S. C., and Rodríguez, G. (2019). Agile teaching practices: Using tdd and bdd in software development teaching. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, page 279–288, New York, NY, USA. Association for Computing Machinery.

Shams, Z. and Edwards, S. H. (2015). Checked coverage and object branch coverage: New alternatives for assessing student-written tests. In *Proceedings of the 46th ACM*

Technical Symposium on Computer Science Education, SIGCSE '15, page 534–539, New York, NY, USA. Association for Computing Machinery.

Sotomayor, J. P., Allala, S. C., Alt, P., Phillips, J., King, T. M., and Clarke, P. J. (2019). Comparison of runtime testing tools for microservices. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 356–361.

Ståhl, D., Martini, A., and Mårtensson, T. (2019). Big bangs and small pops: On critical cyclomatic complexity and developer integration behavior. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, page 81–90. IEEE Press.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., and Lang, M. (2016). Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '16, page 179–182. IEEE Press.

Zhang, Y., Liu, B., Dai, L., Chen, K., and Cao, X. (2020). Automated microservice identification in legacy systems with functional and non-functional metrics. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 135–145.

A. Apêndice A. Métricas de qualidade do código e da suíte de testes

Neste apêndice são apresentadas as equações de cálculo das métricas: Dificuldade Halstead; Número de *Bugs* Entregues Halstead; Complexidade Ciclomática; Índice de Manutenibilidade Geral; e Cobertura de *Branches*. A Tabela 3 apresenta um resumo da notação utilizada nas equações das métricas.

A Equação 1 demonstra o cálculo da Dificuldade Halstead. Essa métrica demonstra o quão difícil é lidar e compreender um código. A dificuldade é afetada pela estrutura do programa. O uso de operandos redundantes e a falta de estrutura de controle de alto nível aumentam a Dificuldade Halstead [Naib 1982]. A métrica de Número de *Bugs* Entregues Halstead, calculada pela Equação 2, estima o número de defeitos na implementação. O número de *bugs* entregue por arquivo deve ser menor que 2 para que o sistema seja considerado minimamente utilizável. O objetivo da métrica é estimar o número de defeitos presente em cada módulo [Hamer and Frewin 1982].

$$\frac{\eta_1 + \eta_2}{2} * \frac{N_2}{\eta_2} \tag{1}$$

$$N_1 + N_2 \log_2 \eta_1 + \eta_2 \tag{2}$$

Calculada pela Equação 3, a Complexidade Ciclomática representa o número de caminhos linearmente independentes no código. Quanto menor o valor, menor a complexidade [Ståhl et al. 2019]. O Índice de Manutenibilidade Geral do Sistema descreve a manutenibilidade do código, ou seja, o nível de dificuldade em realizar correções e melhorias. Os resultados são representados em uma escala de zero a cem, onde os valores

mais altos representam uma maior facilidade em realizar manutenções no código [Martins et al. 2019]. Seu cálculo é realizado pela Equação 4.

$$\delta + 1 \tag{3}$$

$$max[0.1 * \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin(\sqrt{2.4C}))}{171}] \tag{4}$$

A Equação 5 é utilizada para calcular a Cobertura de *Branches*. Seu resultado aponta quais resultados de decisão foram testados. A cobertura de um arquivo é determinada pelas execuções reais divididas pelas oportunidades de execução. Cada linha no arquivo é uma oportunidade de execução, assim como cada destino de ramificação [Shams and Edwards 2015].

$$\frac{B_t}{B_e} \tag{5}$$

Tabela 3: Descrição das variáveis utilizadas

Símbolo	Descrição
η_1	Número de operadores distintos
η_2	Número de operandos distintos
N_1	Número total de operadores
N_2	Número total de operandos
δ	Número de decisões que um bloco de código contém
V	Volume conforme Halstead
G	Complexidade Ciclomática total
L	Número de linhas do código fonte
C	Porcentagem de linhas comentadas (convertido para radiano)
B_e	Oportunidades totais de execução de um comandos em um arquivo
B_t	Execuções reais de comandos em um arquivo