UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS

UNIDADE ACADÊMICA DE GRADUAÇÃO

CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALEXANDRE BRENTANO COLOMBO

MIGRAÇÃO PARA MICROSERVIÇOS:

Uma Metodologia de Anteposição de Serviços Com Indicadores de Qualidade de Software

SÃO LEOPOLDO

2018

ALEXANDRE BRENTANO COLOMBO

MIGRAÇÃO PARA MICROSERVIÇOS:

Uma Metodologia de Anteposição de Serviços Com Indicadores de Qualidade de Software

Artigo apresentado como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, pelo Curso de Ciência da Computação da Universidade do Vale do Rio dos Sinos - UNISINOS

Orientador: Prof. Dr. Mateus Raeder

SÃO LEOPOLDO

2018

MIGRAÇÃO PARA MICROSERVIÇOS

Alexandre B. Colombo*

Mateus Raeder**

Resumo: Tendo em vista o rápido crescimento tecnológico e a alta demanda por informações e poder de decisão no marcado atual, a transformação das empresas e seus modelos de negócio é essencial para a adaptabilidade da organização. As aplicações corporativas são uma das bases que suportam o funcionamento do negócio e permitem o crescimento das empresas. Uma abordagem comumente empregada para a arquitetura dessas aplicações é o padrão monolítico, por sua simplicidade, em linhas gerais, quando em sistemas pequenos. Todavia, esse padrão vem acompanhado de grandes reveses quando a aplicação se torna maior, aumentado a necessidade de escalabilidade e entregas frequentes. Dessa forma, a arquitetura de microserviços mostra-se como uma solução para diversas complicações oriundas de uma arquitetura monolítica em aplicações mais complexas. Fundamentado nesse novo conceito de arquitetura, esse trabalho levanta as principais considerações envolvendo microserviços e propõe uma metodologia de priorização de serviços que visa auxiliar a equipe no processo de migração de arquitetura. Os resultados obtidos com a aplicação dessa metodologia, mostram a evidência das características do método desenvolvido para a priorização dos serviços.

Palavras-chave**:** Monolítico. Microserviços. Sistemas distribuídos. Migração.

1. INTRODUÇÃO

Tendo em vista o rápido crescimento tecnológico e a velocidade com a qual necessita-se adaptar, a transformação das empresas e seus modelos de negócio não é mais uma questão de “se acontecer”, mas uma questão de quando irá acontecer (Borghezan, Brunna R. 2017). Toda a comunicação voltada ao universo digital, bem como os princípios da indústria 4.0, são os primeiros passos para perceber que a tecnologia não é mais apenas a forma como uma empresa se relaciona com o cliente, mas se tornou a maneira como a corporação se organiza e opera (Conway, M. E. 1968).

Com a transformação digital, os processos dentro das empresas convergem do formato analógico para o formato digital, transformando as aplicações corporativas em sistemas cada vez maiores e mais complexos, pois além de armazenarem dados e processarem lógicas transacionais, são responsáveis por disponibilizar essas

* Estudante de Ciência da Computação da UNISINOS. Email: bcolomboalexandre@gmail.com

** Doutor em Ciência da Computação. Professor e coordenador do curso de Ciência da Computação da UNISINOS. Email: mraeder@unisinos.br

informações em diferentes meios e dispositivos a fim de aumentar a agilidade nos negócios. Com a popularização desse conceito e a crescente demanda do mercado por agilidade na tomada de decisão, resiliência dos modelos de negócio e tecnologias disruptivas, abre-se espaço e investimentos para novas formas de aumento de performance e transformação dos processos operacionais.

A arquitetura da aplicação é um fator determinante de seu comportamento e limitações. Um grande número de empresas ainda possui como estrutura sistemas monolíticos que já extrapolaram os limites de manutenibilidade e boas práticas, tornando-os um impeditivo para modernização da organização. Essas aplicações monolíticas possuem responsabilidades centralizadas, são tipicamente difíceis de implementar, atualizar, manter e limitam-se a serem escalonáveis apenas por replicação. Considerando esses aspectos, os sistemas se tornam cada vez mais complexos e engessados com o tempo, reduzindo significativamente suas capacidades de disponibilidade, modernização e simplicidade de uso.

Para resolver as limitações impostas por sistemas monolíticos, surge a arquitetura de microserviços, dentro do domínio arquitetônico da computação distribuída com software orientado a serviço. Microserviços coloca ênfase na divisão dos sistemas em pequenos serviços construídos com o proposito único de executar uma função de negócio de forma coesa e independente, atendendo, assim, a crescente demanda por recursos descentralizados. Essa abordagem visa o desenvolvimento de aplicações como um conglomerado de serviços especifico, que funcionam, comunicam-se e se atualizam de forma independente, porém construídos em volta de um modelo negócio atendendo aos casos de uso de forma conjunta.

O processo de migração para microserviços é um processo ímprobo, uma vez que requer a adequação de funcionalidades e relacionamentos dentro do sistema. Pode-se argumentar que o tamanho reduzido do serviço irá aumentar o entendimento sobre o mesmo. Fato verídico, porém, aumenta-se também a complexidade na intercomunicação da arquitetura e a suscetibilidade a inconsistências na lógica de negócio, devido a dados não atualizados de forma coerente, por exemplo. A fim de mitigar problemas provenientes do processo de extração, através do ganho de experiência da equipe, esse trabalho elabora uma metodologia de priorização de serviços candidatos a extração, tendo como parâmetros quatro indicadores de qualidade de software.

1.1. Objetivos

O objetivo geral do trabalho é criar um método que auxilie na mitigação de problemas relacionados a migração de arquitetura através da priorização de serviços a serem extraídos do sistema monolítico.

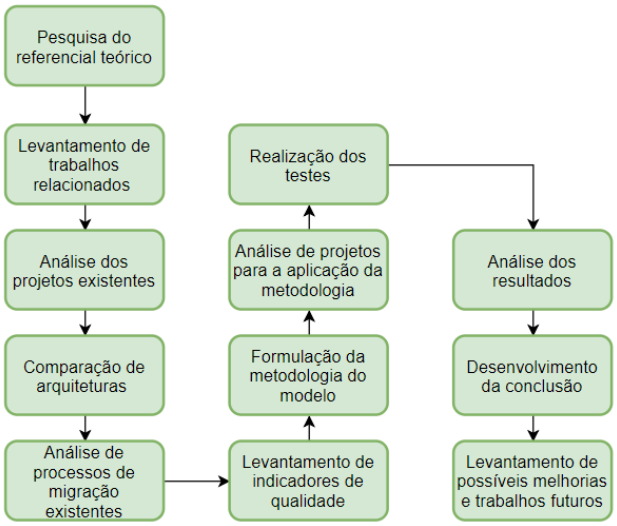
Por sua vez, os objetivos específicos consistem em:

- Diferenciar o estilo monolítico da arquitetura orientada à serviços;
- Entender as implicações do processo de migração;
- Identificar métricas de qualidade de software adequadas à proposta;
- Desenvolver e testar a metodologia;
- Analisar os resultados obtidos.

1.2. Metodologia de Pesquisa

A Figura 1 representa a metodologia utilizada no desenvolvimento desse trabalho.

Figura 1. Metodologia de Pesquisa



Fonte: Desenvolvido pelo autor.

1.3. Estrutura do Trabalho

Esse artigo é composto por 6 seções: introdução, fundamentação teórica, trabalhos relacionados, trabalho realizado, conclusões e trabalhos futuros.

A Seção 1 é a explanação do contexto no qual as organizações se encontram no mercado e as características que isso demanda de suas aplicações corporativas. Na segunda Seção, tem-se a fundamentação teórica, que apresenta uma visão geral

sobre a arquitetura monolítica e as características de microserviços. Na terceira Seção, apresenta-se os trabalhos relacionados a arquitetura de microserviços. A quarta Seção refere-se ao trabalho desenvolvido, desde a escolha das métricas até a aplicação do método desenvolvido e os resultados. Na quinta Seção expõe-se as considerações e conclusões sobre esse artigo. Na sexta e última Seção presenta-se os trabalhos futuros.

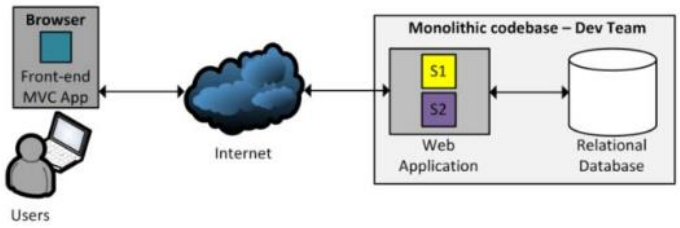
2. FUNDAMENTAÇÃO TEÓRICA

Esse capítulo aborda os problemas relacionados a arquiteturas monolíticas e os principais conceitos e aspectos relacionados a arquitetura de microserviços.

2.1. Sistemas Monolíticos

O design monolítico já existe ha muitos anos e preza a organização da estrutura do código-fonte da aplicação em uma grande e única estrutura, onde o core do sistema, bibliotecas, drivers estão condicionados a um único processo, com uma única tecnologia e um único design. A Figura 2 representa uma estrutura monolítica de uma aplicação web.

Figura 2. Arquitetura monolítica



Fonte: (Villamizar, Mario, et al. 2015).

Geralmente, sistemas de grandes corporações utilizam sistemas monolíticos estruturados em um banco de dados relacional, o qual armazena os dados em tuplas, formando um conjunto ordenado de atributos. O lado do cliente, que se refere a interface do usuário, constituído de páginas HTML e Javascript rodando no browser do cliente que faz as requisições de conteúdo e serviços ao servidor, que lida com as requisições HTML, recupera e atualiza informações no banco de dados e prove o conteúdo para o browser cliente.

Uma aplicação monolítica tem diversas funções e muitas responsabilidades, o que torna essa arquitetura tipicamente difícil de implementar, atualizar, manter e difícil

de entender com o tempo (Escobar, Daniel, et al. 2016), pois possuem uma baixa compreensão do código-fonte e as atualizações englobam um impacto geral no sistema. Portanto, qualquer mudança nesse sistema requer também uma atualização no lado do servidor e banco de dados. Dessa forma, uma pequena alteração pode requerer que diversas partes da estrutura monolítica sejam reimplementadas, o que tende a alcançar um ponto mais crítico com as rápidas mudanças tecnológicas uma vez que a estrutura da aplicação está intimamente interligada e dependente.

2.2. Arquitetura de Microserviços

Microserviços é um estilo de arquitetura no qual um grande sistema é composto por vários pequenos e específicos serviços que funcionam, comunicam-se e se atualizam de forma independente, porém construídos em volta de um modelo negócio atendendo aos casos de uso de forma conjunta. Cada microserviço é construído com um foco especifico em uma funcionalidade, dentro do princípio de *Bounded Context*, ou seja, um microserviço não necessita ter informações de implementação de outros microserviços, e se comunicam através de *Application Programming Interfaces* (APIs) como *Representational State Transfer* (REST) ou outros métodos de comunicação através de interfaces.

De acordo com Newman(2015), microserviços também possuem uma relação com a lei de Conway. Essa lei refere que qualquer empresa que projeta um sistema, inevitavelmente produz um projeto cuja estrutura é uma cópia da estrutura de comunicação da organização - Melvyn Conway, 1967.

Pode-se elencar de forma mais clara um comparativo de estruturas monolíticas com microserviços. A Tabela 1 compara as duas arquiteturas em relação a 5 categorias.

Tabela1: Diferenças entre microserviços e monolítico

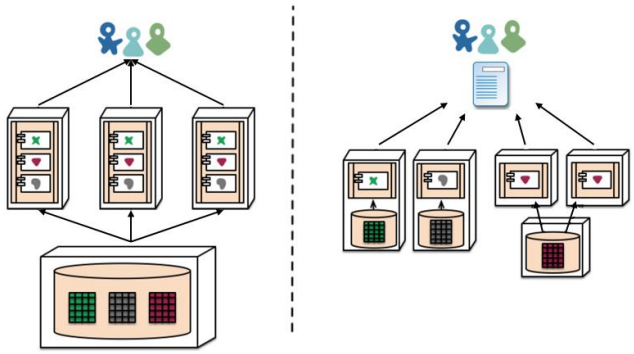
Categoria	Microserviço	Monolítico
Código	Cada microserviço possui a própria base de códigos.	Apenas uma base de código para toda aplicação.
Linguagem de programação	Cada microserviço pode ser desenvolvido em uma linguagem diferente.	Comumente a aplicação é desenvolvida na mesma linguagem.
Facilidade de compreensão	Mais legível, pois cada serviço tem sua base de	Geralmente mais confuso conforme a aplicação cresce e a

	código e tecnologia, tornando o desenvolvedor do serviço um especialista naquela funcionalidade.	necessidade de atualizações frequentes. Diferentes times e funcionalidades necessitam ser envolvidos em uma atualização pequena.
Implementação	Mais simples, pois cada microserviço pode ser implementado, atualizado de forma individual.	Mais complexo, pois requer uma janela de manutenção para todo o sistema.
Escalabilidade	Serviços chave podem ser escalados individualmente, o que gera economia de recursos e uso de memoria	Necessita que toda a aplicação seja escalonada, até mesmo os serviços que não são gargalos.

Fonte: Adaptado de (Daya, S, et al. 2016)

As categorias apresentadas na Tabela 1, são representadas pela Figura 3. No lado direito, a representação em microserviços de uma aplicação cujas funcionalidades foram decompostas em pequenas estruturas independentes. Essa arquitetura permite que as estruturas possam ser escalonadas de forma independente. No lado esquerdo, tem-se a ilustração de uma arquitetura monolítica, escalonada por replicação.

Figura 3: Comparação entre arquitetura monolítica e microserviços



Fonte: Adaptado de (Fowler, Martin. 2014).

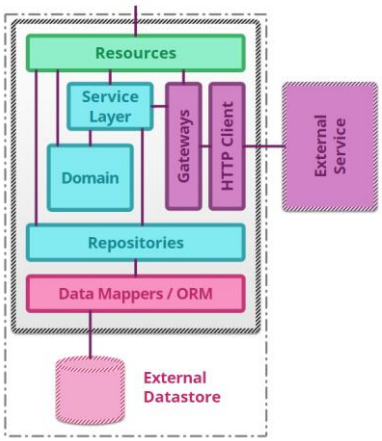
De acordo com Heydari, Babak, e Kia Dalili (2012) a complexidade e modularidade estão fortemente relacionadas. Essa teoria refere que a modularidade surge como uma resposta de um sistema complexo para maximizar a sua adaptabilidade a um ambiente de rápidas mudanças e manter áreas de estabilidade. A integração dos serviços é um fator importante para o aumento de complexidade. Em

sistemas monolíticos, os erros ocorrem em tempo de compilação, pois os serviços estão na mesma estrutura, enquanto que em microserviços os erros de comunicação acontecem em tempo de execução, dificultando a identificação. Junto a isso, o monitoramento do fluxo de dados é um desafio inerente de uma arquitetura distribuída.

2.2.1. Anatomia de um Microserviço

Um sistema monolítico, quando decomposto em microserviços, de forma geral, é desmembrado em pequenas unidades independentes consistentes de algumas ou todas as camadas ilustradas na Figura 4.

Figura 4: Unidades de um microserviço



Fonte: Adaptado de (Fowler, Martin 2014).

A camada de recursos (*Resources*) é responsável por mapear os protocolos de comunicação expostos pelo serviço para os objetos internos desse serviço. A camada de serviço (*Service Layer*), o domínio (*Domain*) e os repositórios (*Repositories*) são os componentes responsáveis por armazenar toda a lógica de negócio pela qual o serviço é responsável. A *Service Layer* define os limites daquele serviço através de um conjunto de operações, disponíveis para a interface do cliente, e coordenando a forma como a aplicação responderá a cada requisição com base nas operações implementadas, enquanto que os repositórios são responsáveis por encapsular um conjunto de objetos persistidos no banco de dados de forma a agir como uma coleção de objetos em memória (*in-memory*) que interage entre o domínio e a camada de ORM (*Object Relational Mapping*), responsável pela lógica de persistência de dados. O *HTTP Client* e *Gateway* são responsáveis pela lógica de comunicação com outros recursos ou sistemas. O *Gateway* encapsula todas as APIs necessárias para comunicação, JDBC, SQL, W3C, entre outras, em uma classe, cuja interface se

assemelha a um objeto, utilizado por outros objetos para comunicação com esse serviço.

2.2.2. Comunicação Entre Serviços

Uma das amplas características da arquitetura em microserviços é a interface de comunicação entre os serviços. Em um sistema monolítico, os componentes comunicam-se através de chamadas de métodos em um processo, porém na estrutura de microserviços, utiliza-se um mecanismo de comunicação entre processos (IPC – inter-process communication) que pode seguir uma abordagem síncrona ou assíncrona. Esses dois diferentes modos de comunicação possibilitam duas formas de colaboração: *request/response* ou event-*based* (Newman 2015). Em uma aplicação síncrona, o cliente e o servidor devem estar disponíveis de forma síncrona, o que não é sempre viável ou simples dado que os serviços podem ser auto escaláveis em um ambiente cloud (Daya, S. et al. 2016). A comunicação assíncrona, segundo Newman (2015) é eficiente em funcionalidades de baixa latência e em casos nos quais a conexão necessita sem manter aberta entre cliente e servidor.

2.3. Princípios dos Microserviços

Newman (2015) elenca os principais fatores para o desenvolvimento de software baseado em microserviços. Esses princípios visam a construção de serviços com baixo acoplamento, independentes, que permitam mudanças rápidas e reduza a dependência entre as equipes de desenvolvimento. Os princípios são:

- Modelagem em torno do modelo de negócio: Quando uma arquitetura de microserviços de um sistema é analisada, necessita-se ter uma ideia do domínio no qual cada serviço opera. Serviços modelados em torno de um modelo de negócio são mais estáveis, o que significa que as APIs em si não sofrem grandes mudanças com muita frequência;
- Cultura de automação: Recomenda-se a máxima automatização possível, sem provisionamento manual de máquinas ou implementações manuais, seguindo uma abordagem de *Continuous Delivery*;
- Abstrair detalhes de implementação: Cada serviço possui sua própria implementação, sem que outros serviços tenham conhecimento;

- Governaça descentralizada: Microserviços são otimizados com autonomia. Cada time é responsável por um serviço especifico, tendo a liberdade na escolha da tecnologia, implementação e manutenção;
- Deploy independente: A equipe deve ser capaz de implementar uma mudança ou fazer manutenção em um serviço, em produção, de forma isolada, sem afetar demais funcionalidades da aplicação;
- Isolar as falhas: É importante que a aplicação seja desenvolvida de forma que possa tolerar e lidar com falhas nos serviços. Newman sugere a implementação de *Circuit Breakers,* de tal forma que o fluxo de solicitações a um serviço seja bloqueado ao atingir um determinado número de erros;
- Consumidor em primeiro lugar: Uma abordagem simples para esse princípio é ter uma boa documentação para as APIs dos serviços, de tal forma que, apesar de esconder detalhes de implementação, desenvolvedores de outros serviços saibam como se comunicar e consumir os diferentes serviços;
- Alta capacidade de monitoramento: Para que se tenha uma visão holística do comportamento da aplicação, deve-se implementar uma funcionalidade de monitoramento para geração de logs que fiquem armazenados em um único lugar, a fim de prover uma melhor visualização dos acontecimentos.

Como base para o desenvolvimento em microserviços, os princípios citados refletem boas práticas na criação da arquitetura visando a diminuição da complexidade do sistema, que afeta em grande parte a integração e comunicação entre os serviços.

2.4. Testes em Microserviços

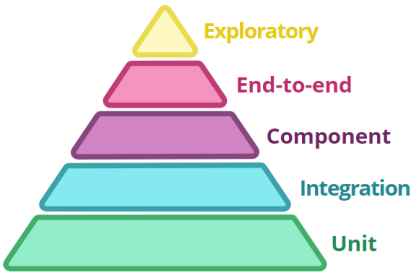
Pelo fato de os serviços serem compostos de diferentes elementos, com características peculiares, um ponto importante da implementação dessa arquitetura é a parte de testes.

2.4.1. Estratégias de Testes em uma Arquitetura de Microserviços

No seu livro, *Succeeding with Agile: Software Development Using Scrum*, Cohn, Mike (2009) desenvolve o conceito de teste pirâmide. Esse conceito define uma linguagem comum para a equipe de testes e em qual situação utilizada cada tipo de teste devera se executado. No contexto de microserviços, ao desmembrar-se uma aplicação monolítica, expõe-se interfaces que antes estavam abstraídas, logo as

oportunidades e flexibilidade dos tipos e níveis de testes aumentam. Fowler, Martin (2014) e Clemson, Toby (2014) estruturam uma pirâmide de testes, com base na anatomia de microserviço, apresentado na Figura 5.

Figura 5: Teste Pirâmide para testes em microserviços



Fonte: (Fowler, Martin 2014).

No topo da pirâmide, tem-se os testes exploratórios, não implementados em scripts. Os testes *end-to-end* tem como objetivo verificar se o sistema como um todo atende aos requisitos de negócio, independentemente da arquitetura e tecnologia utilizadas, e facilitam a implementação ou refatoração da aplicação. Um teste do tipo *Component* é responsável por testar componentes específicos de uma aplicação, como um serviço, de forma isolada. Os testes de integração validam a comunicação e verificam se os componentes sabem como se comunicar com a contraparte. Em microserviços, esse tipo de teste é aplicado na integração do componente de *Data Mappers/ORM* com um banco de dados externo e na comunicação do *Gateway* com recursos externos ao serviço. Na base da pirâmide, tem-se os testes unitários. Esse tipo de abordagem testa unidades individuais do código-fonte, como classes, métodos ou módulos, de forma a garantir que cada unidade atenda corretamente a sua especificação.

2.4.2. Formas de Decomposição do Sistema Monolítico

O processo de transformação de uma aplicação monolítica em microserviços deve ser gradual. Existe a estratégia do Big Bank *rewrite*, quando a organização foca todos os seus recursos de desenvolvimento em reescrever o sistema todo em forma de microserviços. No entanto, é uma técnica arriscada e propensa a falha. Martin Fowler descreve a técnica de *StranglerApplication*, referindo que se deve gradualmente desenvolver os microserviços em torno do sistema antigo e não mais

fazer o monólito ainda maior com código novo. Conforme as novas funcionalidades são incluídas, o sistema monolítico antigo fica obsoleto.

Além da *StranglerApplication*, pode-se analisar a aplicação e desmembrar suas funcionalidades com base nos serviços. Nesse contexto, para identificar as funcionalidades que podem ser refatoradas em microserviços, deve-se analisar os contextos delimitados (*bounded context)* e compreender as relações, mapeamentos e dependências desses contextos. Um *bounded context* busca delimitar o seu domínio com base em uma parte da lógica de negócio (Fowler, Martin. 2014).

3. TRABALHOS RELACIONADOS

Essa Seção traz um resumo do estado da arte do assunto abordado, com o objeto de enriquecimento do trabalho.

3.1. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud

No artigo de Mario Villamizar et al. (2015), os autores trabalharam com uma empresa de software para construir uma aplicação com abordagem monolítica e uma com o padrão de microserviços, suportando os serviços S1 e S2. A versão monolítica da aplicação foi desenvolvida com um framework MVC. Já em microserviços, os autores criaram um microserviço para cada serviço S da versão monolítica expondo-os através de REST em uma rede privada. Como o objetivo do artigo é comparar o desenvolvimento de uma aplicação monolítica com uma em microserviços, em ambas as abordagens se usou a mesma *stack* de tecnologias com o *deployment* em AWS (Amazon Web Service).

O estudo sobre monolítico vs. microserviços foi avaliado nas seguintes áreas de Performance, Metodologia de desenvolvimento e esforços, implementação, escalabilidade e *Continuous Delivery* e adoção de mudanças. Vale ressaltar que com relação a performance, observou-se que a abordagem em microserviços, nesse caso, não impactou de forma considerável a latência das respostas com o aumento de *hosts.* No entanto, o custo de infraestrutura reduziu em 17% devido granularidade.

Com isso, os autores identificaram os benefícios e desafios do padrão de microserviços. Um dos benefícios é a possibilidade de implementação de uma grande aplicação como pequenos serviços que podem ser mantidos de forma independente. Porém, essa arquitetura traz consigo alguns problemas dos sistemas distribuídos além

de exigir uma nova cultura de desenvolvimento e inovação complementadas por uma variedade de princípios e boas práticas a serem seguidas a nível de organização e aplicação.

3.2. Choosing a Microservices Deployment Strategy

No blog escrito por Richardson, Chris (2016), abre-se a discussão sobre as estratégias de deployment para microserviços. O principal tópico é a abordagem das diferentes formas de implementação que suprem as necessidades de uma arquitetura em microserviços, dentre elas *Multiple Services Instance per Host, Service Instance per Virtual Machine, Service Instance per Container* e *Serverless Deployment.*

Com o primeiro padrão, cria-se um ou mais hosts físicos ou virtuais para suportar múltiplos serviços em cada um. Uma grande vantagem dessa abordagem é o alto aproveitamento de recursos, pois diversos serviços dividem o mesmo servidor e sistema operacional. No entanto, os serviços não são devidamente isolados e pode acontecer de um serviço especifico consumir grande parte da memória afetando a performance dos demais.

Com o *Service Instance per Machine*, implementa-se cada serviço com uma imagem de uma máquina virtual Amazon EC2 Ami, por exemplo. Um dos grandes benefícios é que cada serviço trabalha de forma isolada com uma quantidade fixa de CPU e memoria, sem poder consumir recursos de outros serviços.

Uma forma mais leve de implementação do que máquinas virtuais, é com a utilização de containers Docker ou Solaris Zones para cada serviço. Os benefícios da utilização de containers são similares aos das máquinas virtuais em questões de isolamento e encapsulamento além de serem mais rápidos em termos de implementação. Contudo, por dividirem o mesmo kernel do sistema operacional no qual estão hospedados, tendem a ser menos seguros que as máquinas virtuais.

A Amazon dispõe de outra abordagem, sem servidores (*Serverless Deployment*), para a implementação de microserviços, chamanda de AWS Lambda. Nessa abordage, é necessário somente o upload de um arquivo zip com o código do serviço. O preço a ser cobrado é calculado com base em cada requisição e o quanto de tempo e memoria ela consumiu. Os desenvolvedores não necessitam se preocupar com aspectos de servidor, máquinas virtuais ou containers.

3.3. Architecting Microservices

O propósito de Di Francesco, Paolo. (2017), é descrever uma pesquisa de doutorado que investigou os aspectos chave da arquitetura de microserviços (MSA – *Microservice Architecture*). Através de três perguntas (RQ1, RQ2 e RQ3), os autores especificaram os desafios e as respectivas analises. As três perguntas chave são:

- RQ1 – Quais são as propriedades principais da arquitetura de microserviços?
- RQ2 – Como descrever MSA a fim de guiar uma análise arquitetural?
- RQ3 – Como estimar o impacto de migração para MSA?

Primeiramente, em RQ1, é necessário compreender quais as qualidades de software que MSA satisfaz de forma global e local, buscando entender o quão bem um software executa as suas tarefas em uma arquitetura de microserviços. Os autores concluem que a performance e a manutenibilidade estão entres os atributos de qualidade mais pesquisados.

Com o segundo ponto de pesquisa, analisou-se diversas linguagens de modelagem (AL – *Architectural language*) como uma abstração adequada para modelar MSA. A consistência dos diagramas pode indica uma grande necessidade de se propor uma linguagem de modelagem abrangente o suficiente para as peculiaridades dos microserviços (tamanho pequeno, comunicação somente via interfaces, times específicos). Nesse quesito, linguagens como UML são de grande ajuda para a modelagem de arquiteturas de microserviços, porém pesquisas nessas áreas ainda estão no começo e em andamento.

A terceira questão levanta um ponto importante, pois diversas organizações já possuem sistemas legados e tem como objetivos a migração para microserviços utilizando uma plataforma na nuvem. Ser capaz de identificar e estimar o impacto de cada fator em uma migração para MSA é uma vantagem significativa para as empresas, pois, com isso, é possível estimar tempo e recursos para um estudo de viabilidade, além de poder analisar e simular diferentes modelagens e níveis de granularidade para as aplicações. Os autores ainda irão investigar as ferramentas e práticas utilizadas nos processos de migração, baseados em estudos de caso.

De forma geral, a analise apresentada no artigo apresenta os passos inicias para um estudo aprofundado sobre a arquitetura de microserviços, a fim de contribuir com ferramentas para modelar uma arquitetura em microserviços além de prover fatores que possibilitem o estudo de viabilidade da migração para MSA.

3.4. Refactoring a Monolith into Microservices

Uma das questões em torno de microserviços, diz respeito ao processo de transformação de uma aplicação monolítica para essa nova arquitetura. De acordo com Richardson, Chris (2016), uma das estratégias que podem ser adotadas na modernização de uma aplicação é adotar a pratica de que qualquer funcionalidade nova que necessite ser desenvolvida, deve ser implementada separadamente como um microserviço. Neste caso, para suportar ambas as arquiteturas, as requisições de usuário passam por um roteador que as endereçam ou para um microserviço ou para os serviços monolíticos.

Outra maneira eficiente para transformação em microserviços é, inicialmente, separar a aplicação em três camadas (apresentação, lógica de negócios e camada de acesso a dados) e depois desmembrar os monolíticos restantes em microserviços de acordo com os módulos. O autor recomenda extrair primeiro os módulos mais simples, para se ganhar experiência no processo, e em seguida os módulos que requerem mais recursos do sistema. Para a extração de um módulo em microserviço, deve-se implementar uma API bidirecional com o monólito, pois há necessidade mútua de dados, e desenvolver uma interface para comunicação através de uma API utilizando IPC (*Inter-process communication*) para garantir a autonomia do serviço. Conforme os módulos tornam-se serviços independentes e isolados, a aplicação monolítica diminui e o número de microserviços cresce, aumentando a velocidade de desenvolvimento e a agilidade da equipe de desenvolvimento.

4. TRABALHO DESENVOLVIDO

Uma aplicação monolítica possui diversos módulos que são candidatos a se tornarem microserviços, no entanto a refatoração aleatória desses módulos não é sugerida. Isso posto, deve-se começar com os serviços que são mais fáceis de serem extraídos. Isso dará experiência inicial no processo de extração e criação de microserviços para a equipe, visando os casos mais complexos e de mais impacto arquitetural.

Nesse trabalho, desenvolveu-se uma metodologia com a utilização de métricas de qualidade de software como guia para priorização de serviços a serem extraídos, dado que os serviços já tenham sido identificados como potenciais candidatos a uma extração para a nova arquitetura em microserviço. Os dados resultantes são plotados

em um plano cartesiano como gráfico de dispersão para fins de visualização, permitindo uma análise comparativa.

Para identificar quais elementos são bons candidatos para uma extração facilitada, utiliza-se as métricas de profundidade da árvore de herança (DIT – *depth of inheritance tree*), acoplamento eferente (CE), acoplamento aferente (CA) e complexidade ciclomática (CC). Nas seções 4.1.1, 4.1.2 e 4.1.3 discorre-se sobre as características de cada métrica, o significado dos seus valores de referência dentro do projeto de software e a respectiva relevância de utilização para essa metodologia.

4.1. Métricas de Qualidade de Software

As métricas a seguir compõem a metodologia desenvolvida para a priorização de serviços a serem extraídos do monolítico e migrados para a arquitetura de microserviços. Para cada tipo de métrica de qualidade, considera-se o valor mais alto dentro do pacote de classes, por esse ser um valor de máxima representação, ou cenário de pior caso, para um dado serviço.

4.1.1. Métrica de Herança: Profundidade da Árvore de Herança

A profundidade da árvore de herança (DIT) é uma métrica de software que mede a profundidade de uma classe em uma árvore de herança, dando a distância máxima até o nó raiz da árvore (Del Esposte, A. D. M e Manzo, R. R). Se uma classe não herda nada, esta possui DIT igual a 0. Caso herde uma classe, tem profundidade igual a 1 e assim por diante. Isso significa que quanto mais profundo na hierarquia uma classe for declarada, provavelmente, maior será o número de métodos e atributos herdados por essa classe.

Altos valores de DIT indicam que uma classe herda de várias outras, tornando seu comportamento mais imprevisível, pois classes que herdam atributos e operações estão acoplados as suas superclasses (Júnior, M. R. P.). Quando da necessidade de uma mudança, quaisquer alterações realizadas nas superclasses são replicadas para as classes que herdam suas características, portanto toda a árvore de hierarquia deve ser migrada de forma conjunta para um microserviço. Além do mais, quanto maior o número de classes na árvore de hierarquia, maior a probabilidade de as classes herdadas serem dependentes ou dependerem de funcionalidades/atributos de outras classes não pertencentes a árvore de hierarquia, aumentando o acoplamento do

conjunto analisado. Logo, o valor de DIT nesse contexto representa um índice de dependência intra-serviço da funcionalidade a ser migrada.

De acordo com Sommerville (2010), quanto maior for o valor de DIT, mais complexo é o projeto. Os valores para DIT estabelecidos por Ferreira et.al (2012) são utilizados como referência e estabelecidos em Bom (valor menor que 2), Regular (maior que 2 e menor ou igual a 4) e Ruim (maior que 4).

Os valores de DIT dentro da faixa de referência Bom traduzem uma árvore de herança com no máximo dois níveis que mantem as classes filhas próximas da raiz, consequentemente com menor acoplamento dentro da árvore e com menor probabilidade de um alto acoplamento com classes externas a hierarquia.

Portanto, para a aplicação do método, considera-se o valor extremo de DIT dentro do conjunto de classes que representam o candidato a extração.

4.1.2. Métricas de Acoplamento

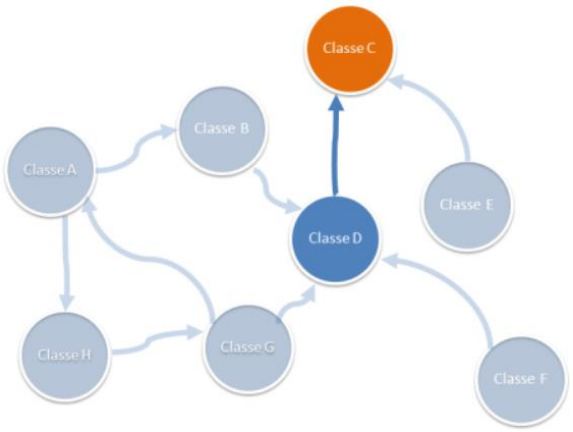
Utiliza-se duas métricas de acoplamento nesse método. As métricas medem o acoplamento do pacote de classes que representa o microserviço a ser extraído. Considera-se esses índices relevantes, pois, como os serviços não são extraídos do monolítico de uma única vez, durante a migração a organização terá duas arquiteturas funcionais, a monolítica e a de microserviços comunicando-se entre si devido as dependências existentes. Logo, a extração de um serviço que depende de muitas classes e/ou é dependente de muitos artefatos externos, aumenta o impacto e a complexidade do processo.

4.1.2.1. Acoplamento Eferente (CE)

De acordo com Martin, R (1994), a métrica de acoplamento eferente (CE) mede o número de classes internas a uma determinada categoria que dependem de classes externas. Segundo Martin, quanto maior o número CE, maior o número de funcionalidades providas por classes externas a essa categoria.

A Figura 6 ilustra o relacionamento entre diversas classes, destacando a métrica CE, tomando como base a classe D.

Figura 6: Exemplo de acoplamento eferente



Fonte: (Daniel, Leandro. 2012).

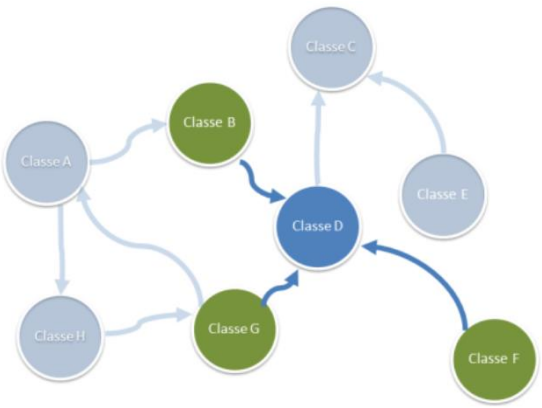
Uma classe ou pacote com muitas funcionalidades providas por classes externas, é um artefato mais instável. Ou seja, manter um valor de acoplamento eferente mínimo, significa uma classe com dependências externas limitadas, autossuficiente, que, portanto, prove um serviço mais estável com um proposito bem direcionado (Filó, T. G, 2014).

Segundo Filó, T.G. (2014), pode-se extrair valores de referência para CE, sendo Bom (menor ou igual a 6), Regular (maior que 6 e menor ou igual a 16) e Ruim (maior que 16).

4.1.2.2. Acoplamento Aferente (CA)

O acoplamento aferente refere-se a contagem de quantas classes utilizam funcionalidades da classe atual. A Figura 7 ilustra a perspectiva de acoplamento aferente, tomando como base a Classe D.

Figura 7: Exemplo de acoplamento aferente



Fonte: (Daniel, Leandro. 2012).

Nesse contexto, quanto maio o número CA, maior o número de classes externas dependentes das classes internas, ou seja, maior a quantidade de classes às quais o conjunto prove serviços, indicando uma maior responsabilidade e relevância dentro do projeto.

Segundo Martin, R. (1994), quanto maior o valor de CA, maior o acoplamento no projeto. Portanto, caso uma classe ou pacote seja altamente acoplado, esse pode se tornar um rico, tendo em vista que, no evento de alguma alteração, isso impacta diretamente em diversas classes e indiretamente afetando a estabilidade da aplicação, já que as classes afetadas também podem possuir seu próprio nível de acoplamento a outras classes. Assim como para o acoplamento eferente, Filó, T.G. (2014) extraiu valores de referência para CA sendo Bom (menor ou igual a 7), Regular (maior que 7 e menor ou igual a 39) e Ruim (maior que 39).

Considerando a migração para microserviços, serviços com alto acoplamento são mais difíceis de serem extraídos e dado o alto nível de CA, podem ter um grande impacto no comportamento da aplicação.

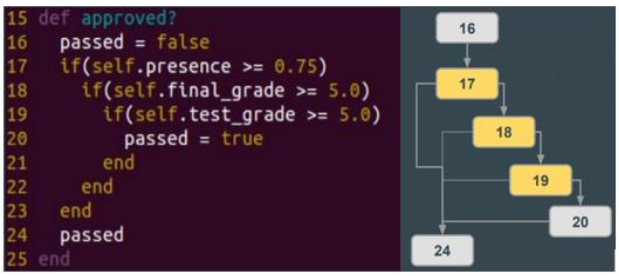
4.1.3. Métrica de Complexidade: Complexidade Ciclomática

Uma métrica de complexidade é utilizada nesse trabalho em comparação às métricas supracitadas. O valor mais expressivo calculado de complexidade de McCabe para o pacote é considerado. Essa métrica prove uma forma de mensurar a quantidade de caminhos de execução possíveis para uma classe. Um maior índice de complexidade por essa métrica não necessariamente está relacionado com a performance/recursos de CPU necessário, já que uma chamada para recuperar dados de um banco de dados pode exigir mais recursos de hardware que um agrupamento de condicionais e laços, por exemplo.

Complexidade ciclomática (CC) foi desenvolvida por McCabe, T. J. (1976) a fim de medir a quantidade de caminhos de execução independentes a partir do código-fonte. Um caminho independente é aquele que apresenta pelo menos uma nova condição, iteração representando um desvio de fluxo.

Também pode-se extrair valores de referência para a complexidade ciclomática. McConnell, Steve (2004) em seu livro *Code Complete* definiu os valores como Bom (menor que 5), Regular (maior que 5 e menor ou igual a 10) e Ruim (maior que 10). A Figura 8 exemplifica um código com clausulas IF.

Figura 8: Exemplo de código com grafo.



Fonte: Adaptado de (Del Esposte, A. D. M e Manzo, R. R.)

Nesse exemplo, o pedaço de código possui CC igual a 4. Esse resultado indica quantos testes precisam ser executados para se verificar todos os fluxos possíveis dentro da classe.

Dadas as métricas, suas relevâncias e aplicações nesse projeto, a Seção 4.2 ilustra os recursos utilizados e a aplicação pratica dos conceitos desenvolvidos.

4.2. Materiais e Métodos Para Aplicação da Metodologia

As seções 4.2.1 e 4.2.2 descrevem os materiais utilizados para desenvolvimento e testes de aplicação bem como os passos para a execução da metodologia.

4.2.1. Materiais

No desenvolvimento dos experimentos, são adotados projetos em linguagem de programação Java, utilizando-se da plataforma Eclipse Oxygen para análise dos mesmos.

As análises de código-fonte para o cálculo das métricas de qualidade de software se dá através do plugin Metrics, versão 1.3.8 (Sourceforge, Metrics). Esse plugin calcula diferentes métricas de qualidade para projetos Java. Os dados podem ser analisados a nível de classe, pacote ou projeto, contanto que não haja erros indicados pelo compilador do Eclipse.

Os projetos utilizados para a aplicação da metodologia são projetos monolíticos clonados de repositórios GitHub. Os possíveis serviços para cada projeto foram posteriormente elaborados pelo autor através do agrupamento de classes já existentes e funcionalidades semelhantes, seguindo a ideia de bounded context.

4.2.2. Método de Aplicação da Metodologia

O desenvolvimento dos experimentos consiste na aplicação das métricas de software em projetos com arquiteturas de abordagem monolítica. Para a aplicação da metodologia, os seguintes passos são aplicados. Como pré-requisito, as classes pertinentes a cada candidato a extração já estão delimitadas.

1. O plugin Metrics calcula os valores de cada métrica para os devidos pacotes, que representam os microserviços a serem extraídos.
2. Extrai-se os valores de DIT, CE, CA e CC e organiza-os em uma tabela com o respectivo serviço.
3. Soma-se o valor de DIT com o valor de CE ou CA, o qual tiver a maior representação, como descrito na Seção 4.1. A soma desses dois índices representa um grau de sensibilidade do pacote, dado que a soma de dois números é diretamente proporcional a cada uma das parcelas (Crespo, Antonio. 2002).
4. Normaliza-se o valor de sensibilidade bem como o valor de CC afim de dimensionalizar os valores. A equação: Z = (X – min) / (max – min) (Raschka, Sebastian, 2014) é utilizada para normalização dos dados entre 0 e 1. Considerando Z o valor normalizado, X a variável que se deseja normalizar e max e min são os valores máximos e mínimos dentro do conjunto de valores ao qual a variável X pertence.
5. Após a aplicação da equação no passo 4, tem-se os valores de sensibilidade e complexidade ciclomática normalizados para cada pacote. Portanto, pode-se plotar os pares ordenados em um plano cartesiano, representado por um gráfico de dispersão.

5. Avaliações e Resultados

O cenário abordado para implementar os demonstrativos práticos da aplicação da metodologia é um comum sistema de ponto de venda (PoS – Point of Sale)[1]. O software possui sistema de compra, venda, gerenciamento de produtos, usuários e emissão de relatórios. Para esse caso, foram identificados os seguintes candidatos a

[1] Disponível em https://github.com/sadatrafsanjani/JavaFX-Point-of-Sales. Acessado em 15 de junho de 2018

migração: *Admin*, *Category*, *Employee*, *Invoice*, *Login*, *Product*, *Purchase*, *Report*, *Sales* e *Supplier*.

Com a aplicação dos passos 1, 2 e 3 chega-se a uma tabela de valores, conforme ilustrado pela Tabela 2.

Tabela 2: Valores para PoS

Serviço	DIT	CE	CA	CC	Sensibilidade
Admin	2	4	8	2	10
Category	1	4	4	6	5
Employee	1	4	3	14	5
Invoice	1	5	6	5	7
Login	1	1	0	6	2
Product	1	5	6	11	7
Purchase	1	3	0	8	4
Report	1	3	0	2	4
Sales	1	2	2	2	3
Supplier	1	4	6	8	7

Fonte: Desenvolvido pelo autor.

Dada a tabela Tabela 2, executa-se o quarto passo, normalizando os valores de sensibilidade e complexidade, criando um par ordenado, como ilustra a Tabela 3.

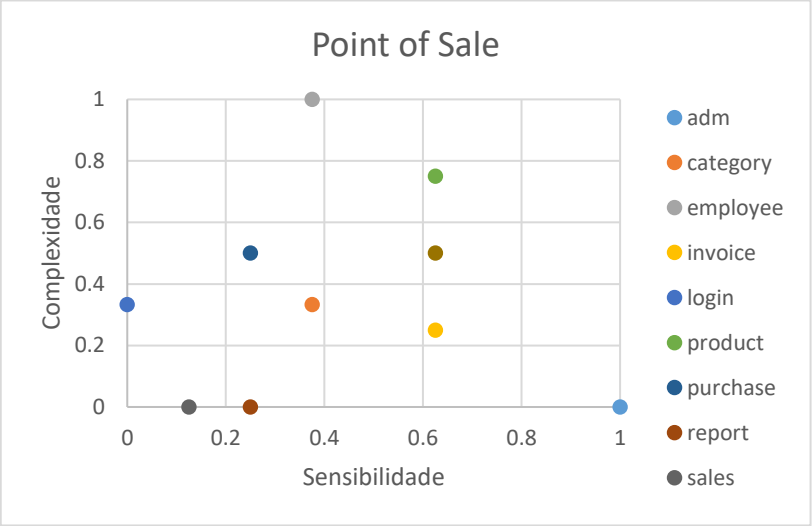
Tabela 3: Dados normalizados para PoS

Serviço	Sensibilidade Norm.	CC Norm.
Admin	1	0
Category	0,375	0,333333
Employee	0,375	1
Invoice	0,625	0,25
Login	0	0,333333
Product	0,625	0,75
Purchase	0,25	0,5
Report	0,25	0
Sales	0,125	0
Supplier	0,625	0,5

Fonte: Desenvolvido pelo autor.

Com base na Tabela 3, pode-se executar o quinto passo, criando um gráfico de dispersão ilustrado pela Figura 9.

Figura 9: Gráfico de Dispersão para PoS



Fonte: Desenvolvido pelo autor

Em virtude dos indicadores utilizados, quanto mais próximo do ponto (0,0) o serviço estiver menor o seu valor de sensibilidade, devido ao DIT e acoplamento, e menor o seu valor de complexidade. Logo, o serviço com maior proximidade ao ponto de origem é menos sensível e menos complexo, tornando a sua extração um processo de menor impacto para a arquitetura.

Com isso, o experimento apresenta uma estrutura base, possibilitando maior evidência nas características da metodologia aplicada. Nesse caso, portanto, o serviço de *Sales* teria a prioridade na migração, tendo em vista o aprimoramento da experiência da equipe para os casos mais complexos e de maior impacto arquitetural.

5.1.1. Outros Exemplos

Com base no método demostrado na Seção 4.2.2, aplicou-se a metodologia em demais projetos de software para comparação de resultados.

5.1.1.1. Sistema de Gestão Escolar

Esse sistema[2] foi desenvolvido para um projeto em uma universidade. Consiste de uma aplicação web escrita em Java e Javascript que possui interfaces para administradores do sistema responsáveis por gerenciar acesso de usuário, disciplinas e grupos. Os professores têm acesso a testes e gerenciamento de notas e os alunos

[2] Disponível em: https://github.com/LouisBarranqueiro/ZPareo. Acessado em 15 de junho de 2018

podem consultar seu livro de notas. Os seguintes candidatos a extração foram encontrados: *Adm, Gradebook, Group, Student, Subject, Teacher, Test* e *Util.*

Para os candidados citados, aplicou-se a metodologia como descrita na Seção 4.2.2 e obteve-se as duas Tabelas 4 e 5.

Tabela 4: Valores para o sistema de gestão escolar

Serviço	DIT	CE	CA	CC	Sensibilidade
Adm	3	4	16	2	19
Gradebook	3	2	2	1	5
Group	3	9	16	5	19
Student	3	9	7	8	12
Subject	3	9	11	5	14
Teacher	3	9	5	7	12
Test	3	9	3	9	12
Util	4	1	1	7	5

Fonte: Desenvolvido pelo autor.

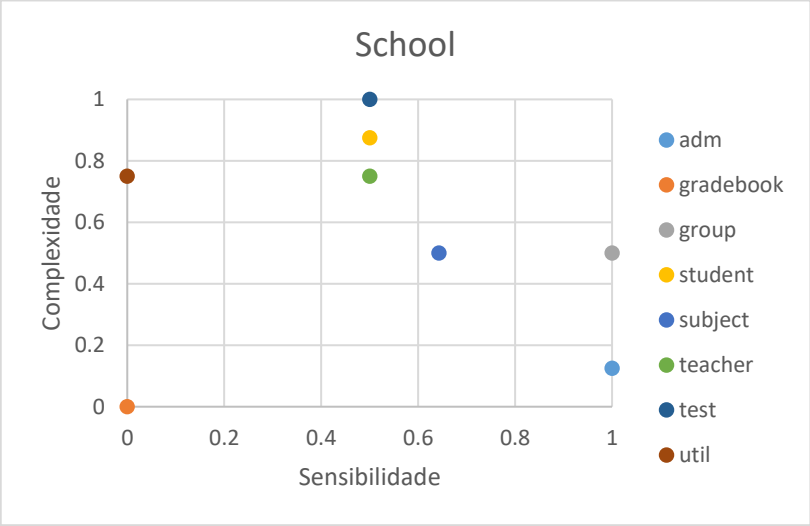
Tabela 5: Valores normalizados para o sistema de gestão escolar

Serviço	Sensibilidade Norm.	CC Norm.
Adm	1	0,125
Gradebook	0	0
Group	1	0,5
Student	0,5	0,875
Subject	0,642	0,5
Teacher	0,5	0,75
Test	0,5	1
Util	0	0,75

Fonte: Desenvolvido pelo autor.

A partir do par ordenado formado pelos valores normalizados, construiu-se o gráfico ilustrado na Figura 10. Como pode ser observado, o conjunto de funcionalidades e classes que representam o serviço de *Gradebook* é o que possui menor complexidade e sensibilidade, portanto o candidato com preferência para a extração inicial.

Figura 10: Gráfico de dispersão para o sistema de gestão escolar



Fonte: Desenvolvido pelo autor.

Como demostrado no gráfico da Figura 10, o serviço *Gradebook* encontra-se mais próximo ao ponto (0,0), portanto com preferência no processo de migração.

5.1.1.2. Sistema de Frente de Loja

O Plants by WebSphere[3] é um sistema de frente de loja especializado na venda de plantas e artigos de jardinagem. Com essa aplicação, os clientes podem abrir conta, pesquisar por items e emitir ordens. A aplicação foi desenvolvida em Java EE seguindo o padrão Model-View-Controller (MVC). Para esse sistema, os seguintes serviços foram identificados como possíveis candidatos: *Account, Adm, Cart, Image, Email, Product* e *Util.* Aplicando a metodologia proposta, tem-se as seguintes Tabelas 6 e 7.

Tabela 6: Valores para o sistema de frente de loja

Serviço	DIT	CE	CA	CC	Sensibilidade
Account	3	4	7	37	10
Adm	3	2	1	16	5
Cart	1	10	6	5	11
Image	3	1	0	6	4
Email	3	2	1	1	5
Product	1	2	14	4	15
Util	2	3	2	16	5

Fonte: Desenvolvido pelo autor.

[3] Disponível em: https://github.com/WASdev/sample.plantsbywebsphere. Acessado em 15 de junho de 2018.

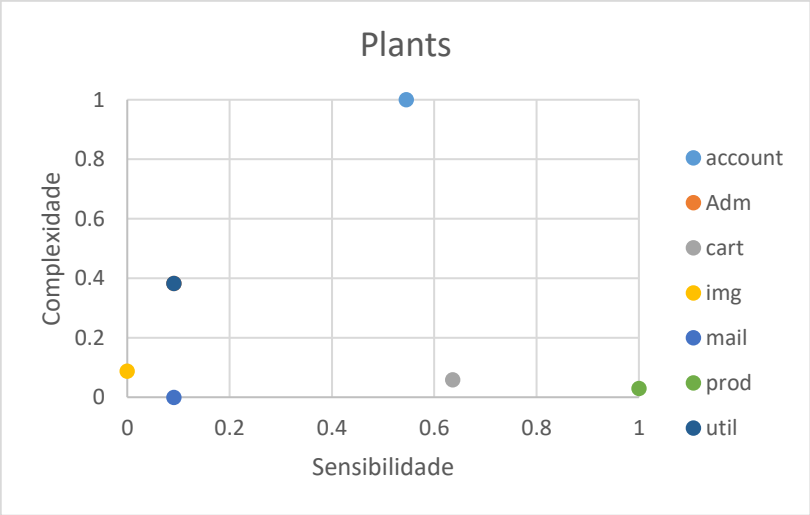
Tabela 7: Valores normalizados para o sistema de frente de loja

Serviço	Sensibilidade Norm.	CC Norm.
Account	0,545	1
Adm	0,090	0,382
Cart	0,636	0,058
Image	0	0,088
Email	0,090	0
Product	1	0,029
Util	0,090	0,382

Fonte: Desenvolvido pelo autor

Com base na tabela 7, pode-se plotar o gráfico ilustrado na Figura 11. Nota-se que devido aos mesmos resultados, *Adm* e *Util* ficam sobrepostos na Figura 11.

Figura 11: Gráfico de dispersão para o sistema de frente de loja



Fonte: Desenvolvido pelo autor.

Neste caso, os serviços *Image* e *Email* possuem distancia similar ao ponto (0,0), porém *Image* tem uma maior complexidade em relação ao serviço de *Email*, o que exigiria um maior número de testes. Por outro lado, *Email* possui uma maior sensibilidade, com possibilidade de impactar outras funcionalidades.

5.1.1.3. Aplicação E-commerce

Este sistema[4] refere-se a uma aplicação de vendas online, com controle de usuários, entregas, pagamentos e produtos. Essa aplicação é um exemplo didático

[4] Disponível em: https://github.com/xebia/microservices-breaking-up-a-monolith. Acessado em 15 de junho de 2018

utilizado em uma serie de blogs sobre refatoração. Os seguintes serviços foram identificados: *Account, LineItem, Order, Payment, Product, Ship, WebUser* e *Shop.*

Considerando os valores das métricas de qualidade de software para esse sistema, obtem-se as seguintes Tabelas 8 e 9.

Tabela 8: Valores para o sistema de E-commerce

Serviço	DIT	CE	CA	CC	Sensibilidade
Account	3	4	11	4	14
LineItem	2	3	7	1	9
Order	2	5	10	7	12
Payment	1	1	6	6	7
Product	3	4	7	4	10
Ship	2	5	3	5	7
WebUser	3	5	9	4	12
Shop	2	5	10	4	12

Fonte: Desenvolvido pelo autor.

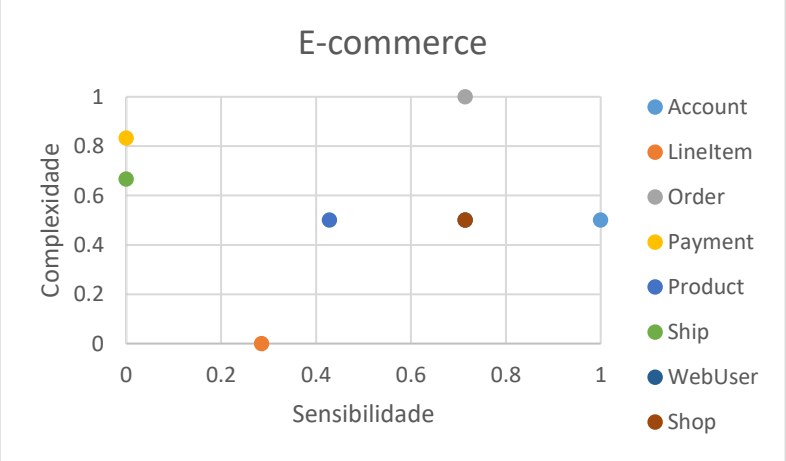
Tabela 9: Valores normalizados para o sistema de E-commerce

Serviço	Sensibilidade Norm.	CC Norm.
Account	1	0,5
LineItem	0,285	0
Order	0,714	1
Payment	0	0,833
Product	0,4285	0,5
Ship	0	0,666
WebUser	0,714	0,5
Shop	0,714	0,5

Fonte: Desenvolvido pelo autor.

Com base na Tabela 9, é possível construir o gráfico representado pela Figura 12.

Figura 12: Gráfico de dispersão para o sistema de E-commerce



Fonte: Desenvolvido pelo autor.

Nessa aplicação, o serviço representando as funcionalidades de *LineItem* tem a prioridade na migração.

6. Conclusão e Trabalhos Futuros

Com as pesquisas desse estudo, foi possível compreender a arquitetura de microserviços bem com grande parte dos conceitos envolvidos nesse tema, tais como diferenciação perante outras arquiteturas, impactos na estrutura de uma empresa, testes, processo de migração e construção de microserviços. Após a análise das pesquisas, identificou-se diversos pontos e considerações sobre o processo de migração de um sistema monolítico para a arquitetura de microserviços, cujas características deixam em aberto para diferentes interpretações relacionadas a diferentes aspectos de implementação, migração e deployment.

Para o desenvolvimento desse estudo, foram analisados quatro diferentes métricas de qualidade de software. As métricas de profundidade da árvore de herança, acoplamento aferente e eferente e complexidade ciclomática formam a base da metodologia desenvolvida, que leva em conta questões de hierarquia, acoplamento e complexidade de código para a priorização dos serviços candidatos a migração para a nova arquitetura.

Conclui-se, com isso, que a arquitetura em microserviços possibilita diversas interpretações devido a um grande leque de possiblidades e alternativas de escolha que suas características trazem, junto a isso, soma-se o acréscimo de complexidade que o processo de migração e a arquitetura distribuída trazem. Portanto, o método desenvolvido trabalha na seleção dos serviços com maior prioridade para migração, considerando aspectos de sensibilidade e complexidade, como forma de mitigar efeitos colaterais e falhas através do ganho de experiência da equipe, visando a extração de serviços de maior impacto arquitetural com menor propensão a inconsistências. Até então, não foram encontrados métodos similares, com o propósito e metodologia apresentados nesse trabalho.

Como trabalhos futuros, pode ser considerado melhorias no modelo através da análise de métricas de software e valores de referência que se adequem a linguagens de programação especificas. Como, por exemplo, a utilização de uma métrica que considere a quantidade de linhas de código, que tende a possuir diferentes valores dependendo da linguagem na qual o código foi escrito.

Outra possibilidade de trabalho futuro é a adequação do modelo para que, em um segundo momento, seja capaz de fazer a priorização com base na quantidade de recursos computacionais que um serviço requer, uma vez que transformado em microserviço, o processo de escalonamento é facilitado e mais proveitoso.

Com relação a simplificação no processo de obtenção de dados para os cálculos do modelo, sugere-se o desenvolvimento de uma interface ou plug-in para a leitura dos dados gerados pelo *Metrics* no Eclipse. No período desse trabalho, desenvolveu-se uma aplicação web que já automatiza os cálculos e a criação do gráfico, como pode ser visto no Apêndice A, porém a aplicação é alimentada com os dados iniciais ainda de forma manual.

MICROSERVICE MIGRATION:
A Service Prioritization Methodology Based on Software Quality Metrics

Abstract: Considering the rapid technological growth and the high demand for information and decision-making power in the current global marking, the transformation of companies and their business models is essential for the organization to adapt to market changes. The corporate applications are one of the bases that support business operations and allows companies' growth. A commonly used approach to the architecture of these applications is the monolithic pattern, for its simplicity, in general, when in small systems. However, this pattern is accompanied by major setbacks as the application becomes larger, increasing the need for scalability and frequent deliveries. In this context, the microservice architecture shows itself as a solution for several complications originating from a monolithic architecture in more complex applications. Based on this new concept of architecture, this work raises the main considerations involving microservices and proposes a methodology of prioritization of services that aims to assist the team in the process of architectural migration. The obtained results with the application of this methodology, shows the evidence of the characteristics of the developed method for the service prioritization.

Keywords: Monolithic. Microservices. Distributed systems. Migration.

REFERÊNCIAS

Borghezan, Brunna R. "A transformação digital não é mais uma questão de 'e se contecer', mas sim quando vai acontecer e a resposta é agora! Confira aqui o Road Map". Disponível em https://pt.linkedin.com/pulse/transforma%C3%A7%C3%A3o-digital-n%C3%A3o-%C3%A9-mais-uma-quest%C3%A3o-de-e-se-remor-borghezan , 2017 [Online. Acessado em 02-Outubro-2017]

Conway, M. E. (1968). How do committees invent. Datamation, 14(4), 28-31

Villamizar, Mario, et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud." *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 2015.

Escobar, Daniel, et al. "Towards the understanding and evolution of monolithic applications as microservices." *Computing Conference (CLEI), 2016 XLII Latin American*. IEEE, 2016.

Newman, S. (2015). Building microservices: *designing fine-grained systems*. " O'Reilly Media, Inc.".

Daya, S., Van Duy, N., Eati, K., Ferreira, C. M., Glozic, D., Gucer, V., ... & Narain, S. (2016). Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach. IBM Redbooks.

Fowler, Martin (2014, Março). MartinFowler | CircuitBreak [Online. Acessado em 08-Setembto-2017]. Disponível em: https://martinfowler.com/bliki/CircuitBreaker.html

Fowler, Martin (2014, Março). MartinFowler | Microservices, a definition of this new architectural term [Online. Acessado em 08-Agosto-2017]. Disponível em: https://martinfowler.com/articles/microservices.html

Heydari, Babak, and Kia Dalili. "Optimal System's Complexity, An Architecture Perspective." Procedia Computer Science 12 (2012): 63-68.

Cohn, Mike. "Succeeding with agile: software development using Scrum." (2010).

Richardson, Chris. “Refactoring a Monolith into Microservices”. Disponível em: https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/, 2016 [Online; acessado em 28-Agosto-2017]

Di Francesco, Paolo. "Architecting Microservices." Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on. IEEE, 2017.

Júnior, M. R. P. Estudo de métricas de código-fonte no sistema Android e seus aplicativos

Sommerville, I. (2010). Software Engineering. Addison-Wesley, Harlow, England, 9 edição. ISBN 978-0-13-703515-1.

Ferreira, K. A.; Bigonha, M. A.; Bigonha, R. S.; Mendes, L. F. & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. Journal of Systems and Software, 85(2):244--257. ISSN 01641212.

Martin, R. (1994). OO design quality metrics. *An analysis of dependencies*, 12, 151-170.

Daniel, Leandro. “Code Metrics (parte 3) – Medindo Acoplamento” Disponível em http://leandrodaniel.com/index.php/code-metrics-parte-3-medindo-acoplamento/ , 2012 [Online. Acessado em 14-Março-2018]

Filó, T. G., Bigonha, M. A., & Ferreira, K. A. M. (2014). Um Método de Extraç ao de Valores Referência para Métricas de Softwares Orientados por Objetos. WTDSoft 2014, 62.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.

McConnell, Steve. Code complete. Pearson Education, 2004

Del Esposte, A. D. M e Manzo, R. R. “Métricas de Software para POO”. Disponível em https://social.stoa.usp.br/articles/0046/2403/M_tricas_de_Software_para_POO.pdf [Online. Acessado em 05-Abril-2018]

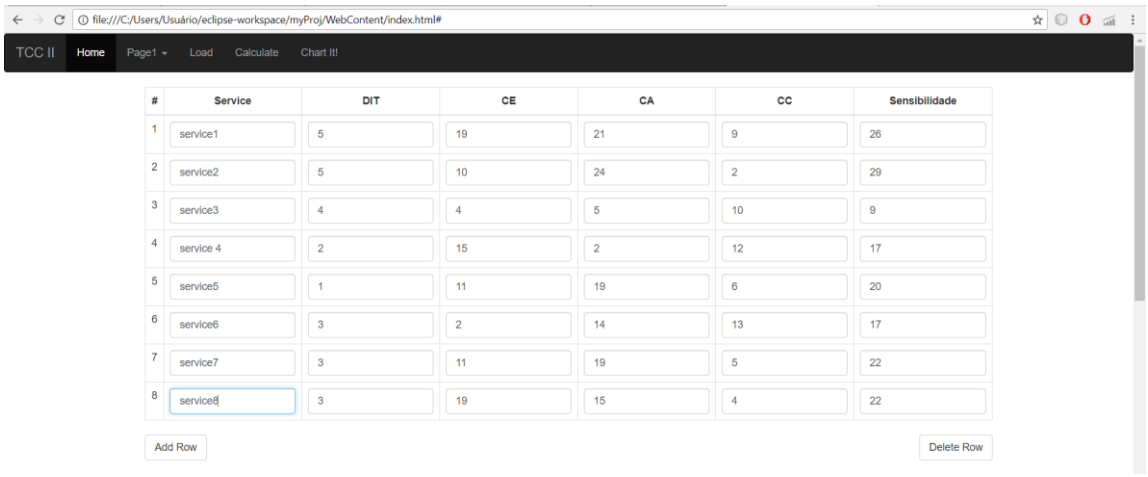
Raschka, Sebastian (2014). “About Feature Scaling and Normalization”. Disponível em http://sebastianraschka.com/Articles/2014_about_feature_scaling.html [Online. Acessado em 08-Junho-2018]

Crespo, Antônio Arnot. Matemática financeira fácil. Saraiva, 2002

APÊNCIDE A – Aplicação web para automatizar a metodologia desenvolvida

No curso desse trabalho, desenvolveu-se uma aplicação web baseada em JavaScript e HTML5 com a finalidade de agilizar os cálculos e a construção do gráfico. A aplicação possui as funções *Load*, que carrega valores pseudorrandômicos para DIT, CA, CE e CC para fins de teste. A função *Calculate* calcula a Sensibilidade bem como os valores normalizados de CC e Sensibilidade. Por fim, a função *Chart It!* constrói o gráfico de dispersão com base nos pares ordenados calculados. Figura 13 ilustra o menu de ações no topo e a tabela para o cálculo dos valores iniciais, como por exemplo na tabela 8.

Figura 13: Tabela de valores



#	Service	DIT	CE	CA	CC	Sensibilidade
1	service1	5	19	21	9	26
2	service2	5	10	24	2	29
3	service3	4	4	5	10	9
4	service 4	2	15	2	12	17
5	service5	1	11	19	6	20
6	service6	3	2	14	13	17
7	service7	3	11	19	5	22
8	service8	3	19	15	4	22

Fonte: Desenvolvido pelo autor.

A Figura 14 ilustra a tabela dos valores normalizados, criados a partir da função *Calculate* com base nos valores da Figura 13.

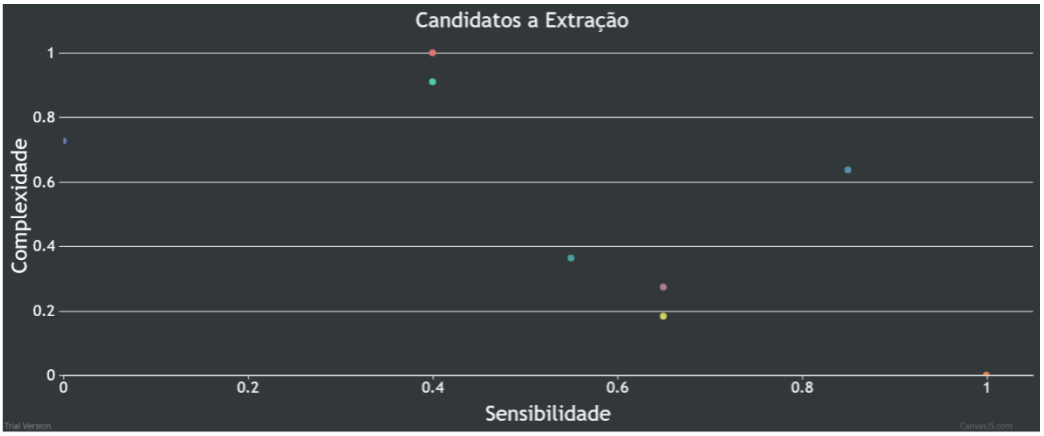
Figura 14: Valores normalizados

#	Sensibilidade Norm.	CC Norm.
1	0.85	0.6363636363636364
2	1	0
3	0	0.7272727272727273
4	0.4	0.9090909090909091
5	0.55	0.36363636363636365
6	0.4	1
7	0.65	0.2727272727272727
8	0.65	0.18181818181818182

Fonte: Desenvolvido pelo autor.

O gráfico representado pela Figura 15 foi criado com a função *Chart It!,* que utiliza os pares ordenados da Figura 14.

Figura 15: Gráfico de Distribuição



Fonte: Desenvolvido pelo autor.