Avaliação dos impactos da decomposição de uma aplicação monolítica para microsserviços: Um estudo de caso

Tulio Ricardo Hoppen Barzotto
Universidade do Vale do Rio do Sinos (Unisinos)
São Leopoldo, Rio Grande do Sul, Brasil
tulio.barzotto@gmail.com

Kleinner Farias
PPGCA, Universidade do Vale do Rio do Sinos (Unisinos)
São Leopoldo, Rio Grande do Sul, Brasil
kleinnerfarias@unisinos.br

arXiv:2203.13878v2 [cs.SE] 2 Aug 2022

ABSTRACT

Aplicações monolíticas estão sendo decompostas para uma arquitetura de microsserviços, visando melhorar a manutenabilidade, perfomance e modularização. Embora tais decomposições tenham sido amplamente realizadas atualmente na indústria, pouco é reportado na literatura sobre os impactos destas decomposições. Este trabalho, portanto, reporta um estudo de caso realizado para investigar os impactos da decomposição de uma aplicação real da indústria para a arquitetura de microsserviços. A aplicação alvo do estudo refere-se a uma operação de saque, realizada por uma instituição financeira, a qual foi extraída de uma aplicação monolítica para uma aplicação baseada em microsserviços. Em particular, métricas foram aplicadas nas aplicação monolítica e na baseada em microsserviços, visando quantificar o acoplamento, coesão, consumo de CPU e consumo de memória. Os resultados obtidos apontam que a arquitetura de microsserviços gerou melhores resultados para as métricas de modularidade de software, além de menor consumo de memória e CPU. Por fim, este trabalho traz reflexão e aponta para desafios e direções futuras de pesquisa que precisam ser exploradas pela academia e a indústria.

KEYWORDS

Arquitetura monolítica; Arquitetura de microsserviços; Modularização; Estudo de caso; Performance

1 INTRODUÇÃO

As aplicações monolíticas podem ser caracterizadas como um único artefato de software executável, tendo seus módulos altamente acoplados e os requisitos implementados de forma entrelaçada e espalhada entre os módulos da aplicação [46]. Atualmente, a arquitetura monolítica vem perdendo espaço para a arquitetura de microsserviços, evidenciando a popularidade, através de como as aplicações são entendidas, concebidas e desenhadas [14, 39]. Em particular, as constantes modificações dos requisitos e volatilidade dos ambientes de negócio provocam mudanças constantes das aplicações monolíticas [34, 40], dificultando as realizações das manutenções, aumentado as estimativas de esforço [11], elevando o esforço cognitivo de desenvolvedores na compreensão de código [25]. Além disso, a ausência de documentação das aplicações monolíticas nas empresas aumenta o desafio da realização das manutenções [20, 29].

Com o uso da arquitetura de microsserviços, que divide a aplicação em um conjunto de serviços, que gera bases de códigos menores, que podem ser compreendidos mais facilmente pelos desenvolvedores, permite o desenvolvimento contínuo, sem afetar a aplicação como um todo, assim como uma série de outros benefícios, não encontrados na arquitetura monolítica. Nesta linha, é salientada a importância dos temas “microsserviços” e “decomposição de aplicações monolíticas”, uma vez que possibilita ciclos de desenvolvimento de software menores, caracterizando aumento de performance e implantações mais ágeis, times menores e mais especializados. Destaca-se que a arquitetura monolítica possui uma forte desvantagem, já que, alterações na base de código ou implantações afetam a aplicação como um todo, seja por alterações de código ou disponibilidade durante novas implantações. Em contrapartida, a arquitetura de microsserviços possui uma estrutura modular que facilita as mudanças, tornando alterações de códigos e implantações pontuais, sem necessariamente afetar toda a aplicação ou necessidade de avisar todos os desenvolvedores sobre o novo código inserido.

Alguns trabalhos foram propostos, com finalidade de discussão do tema, por meio de análise comparativa. Autores como [14, 31] e outros, serviram como base para embasar o desenvolvimento do presente trabalho. Os trabalhos citados serviram especialmente para análise comparativa do tema, de acordo com critérios comparativos, como contexto de avaliação, domínio da aplicação, métodos de estudo empírico e tipos de métricas.

Ressalta-se que a problemática do tema em questão, é escassa na literatura, visto que a arquitetura de microsserviços é relativamente recente, especialmente carece de estudos que propunham a decomposição de aplicações monolíticas, como base para a arquitetura de microsserviços. Além disso, costuma-se dar ênfase na arquitetura monolítica na maioria dos artigos e trabalhos sobre temas similares, com exceção de trabalhos como [8, 26, 41, 43, 47], onde é realizada uma revisão sistemática, tanto da arquitetura monolítica, quanto na arquitetura de microsserviços e suas respectivas aplicações.

Neste sentido, essa pesquisa executa um estudo experimental, visando avaliar o impacto das métricas de consumo de CPU, consumo de memória e modularidade de software. Para isso foi executado um estudo empírico em que uma aplicação real da empresa hipotética *Cooperativa Utile*, onde foi analisada a versão da aplicação monolítica em comparação com a sua versão equivalente, seguindo a arquitetura baseada em microsserviços. A modularidade de software foi analisada sob os atributos de acoplamento e coesão, usando 5 métricas voltadas para aplicações que utilizam linguagem orientada a objetos. As métricas de consumo de CPU e consumo de memória foram analisadas através de testes de carga, simulando a execução do fluxo de negócio equivalente em ambas versões da aplicação.

O presente trabalho se encontra estruturado da seguinte forma. Seção 2 apresenta a fundamentação teórica do tema dividido em subtópicos; Seção 3 traz os trabalhos relacionados ao tema para análise comparativa, assim como a metodologia exposta de cada trabalho e subtópicos com a evidenciação das oportunidades do

presente trabalho; Seção 4 traz à luz, a metodologia utilizada no trabalho e subtópicos que salientam os objetivos e as questões de pesquisa explícitas, hipóteses, seleção da aplicação alvo, variáveis e métodos de quantificação, métricas selecionadas, procedimento de análise e experimental; Seção 5 traz os resultados obtidos; Por fim, a Seção 6 aborda as conclusões e os trabalhos futuros. Já na estruturação pós-textual, encontram-se as referências bibliográficas utilizadas.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção aborda os conceitos teóricos usados durante a construção e desenvolvimento do estudo. A Seção está dividida da seguinte forma: a Seção 2.1 descreve os conceitos sobre a arquitetura monolítica; a Seção 2.2 descreve os conceitos sobre a arquitetura de microsserviços; a Seção 2.3 descreve os conceitos sobre a performance de software; por fim a Seção 2.4 descreve os conceitos sobre a modularização de software.

2.1 Arquitetura monolítica

Para um melhor entendimento sobre a arquitetura de microsserviços e como a tecnologia evoluiu para isto, será preciso primeiramente entender a arquitetura monolítica tradicional.

Na arquitetura monolítica, todas as funcionalidades estão encapsuladas em uma única aplicação, fazendo com que os módulos não possam ser executados independentemente [14]. Este tipo de arquitetura torna a aplicação altamente acoplada e toda a lógica para processar a requisição é executada em um único processo, que usam os mesmos recursos de hardware, como memória *RAM*, *CPU* e armazenamento de dados. Devido ao fato de todo o desenvolvimento estar em um único executável, uma única alteração de código pode afetar todos os recursos que a aplicação provê, gerando necessariamente um novo *build* e um *redeploy* de toda a aplicação [38].

Enquanto a arquitetura monolítica é uma boa escolha para iniciar um projeto, já que isso permite que você explore a complexidade de um sistema e os limites de seus componentes [21]. Entretanto, os benefícios vão desaparecendo, conforme o código fonte da aplicação fica maior. Quanto maior o tamanho da aplicação, maior a complexidade, resultando em um grande número de dependências, o que causa alto acoplamento [14, 38]. Com o objetivo de modernização destas aplicações, o processo de decomposição surge como atividade central, visando utilizar tecnologias emergentes e de arquiteturas de software distribuídas e de alta disponibilidade [27].

2.2 Arquitetura de microsserviços

De acordo com [4], os microsserviços são resultados de uma abordagem arquitetônica, focada na decomposição de aplicações em serviços, com propósito único e com baixo acoplamento, sendo gerenciadas por equipes multifuncionais, para entrega e manutenção de sistemas de software complexos rapidamente. Para [23], a arquitetura de microsserviços consiste em uma abordagem para desenvolver um único aplicativo, como um conjunto de pequenos serviços, cada um sendo executado de forma isolada e se comunicando de forma leve, geralmente uma *API* de recursos *HTTP*.

Segundo [23], não existe em particular uma definição do que seja arquitetura de microsserviços, mas existem certas características que o tornam sujeitos a classificar como arquitetura de microsserviços, "Como acontece com qualquer definição que descreve características comuns, nem todas as arquiteturas de microsserviços têm todas as características, mas esperamos que a maioria das arquiteturas de microsserviços exibam a maioria das características" [23, Traduzido pelo autor]. Para [33], essas características são mais como princípios da arquitetura de microsserviços, os quais são definidos como:

(1) "Modelar em torno de conceitos de negócios", para serem representados como contextos limitados e modelos de domínio de acordo com padrões do *Domain-Driven Design* (DDD) [16].
(2) "Adotar uma cultura de automação" em testes e implantação; praticar entrega contínua.
(3) "Ocultar detalhes da implementação interna", como bancos de dados; definir Interfaces de programação de aplicativos (APIs) independentes de tecnologia.
(4) "Descentralizar todas as coisas": por exemplo, aplicar a governança compartilhada, prefira o serviço de coreografia ao invés de orquestração, use *middleware* burro, mas *endpoints* inteligentes.
(5) Tornar os serviços "implantáveis de forma independente", por exemplo, deixar versionado (serviço) *endpoints* coexistem; implantar apenas um serviço por *host* (virtual).
(6) "Isolar falha", por exemplo, introduz disjuntores para tornar os serviços robustos.
(7) Ser "altamente observável", por exemplo, por meio de monitoramento semântico com dados de agregação.

Para fins de conceituação dos diferentes autores, evidencia-se que ao contrário das nove características que [23] expõem, para [33], apenas sete princípios são válidos para definir a arquitetura de microsserviços. Porém, tanto os princípios de [33] quanto às características de [23], mesclam-se, seja através da modelagem da aplicação em torno do negócio, como a descentralização da governança.

2.3 Performance

Segundo [36], a performance é um fator crucial, pois ela impacta diretamente na experiência do usuário ao utilizar um determinado sistema. A performance está invariavelmente atrelada a capacidade da máquina e sua composição arquitetural, podendo ser medida em vários parâmetros, como taxa de transferência, latência e largura de banda do sistema [13]. Para fins de catalogação de medição da performance, é inerente que o desempenho da *CPU*, sendo a taxa de transferência utilizada como quesito para medir a saída de carga de trabalho, o desempenho da memória que tem como parâmetro a medição da largura da banda na velocidade de acesso à memória e operações, assim como o desempenho da rede, de disco e outros, os quais estão ligados à performance, tanto da arquitetura, quanto da máquina, sendo usados como atributos para conceituação [13].

De acordo com [39], são os recursos utilizados sob condições estabelecidas que representam o desempenho/performance da arquitetura. Esses mesmos recursos estão ligados à eficiência do desempenho em características, como comportamento do tempo (tempo de resposta), utilização de recursos (tipos de recursos utilizados por um produto), dentre outros. Também são atributos usados para medição

de performance, ou ao menos caracterização e conectividade deste desempenho, a compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade [39]. A arquitetura de microsserviços traz complexidade aos testes de desempenho, que são classificados como: tipo caixa preta, portanto, o tipo de teste mais compatível é o *end-to-end* [10]. Segundo os preceitos de [44], são os testes de desempenho que têm como finalidade a verificação do software e o cumprimento dos requisitos pré-estabelecidos, como o tempo de resposta, vazão e disponibilidade.

2.4 Modularização

A conceituação de modularização se dá através do compartilhamento de que é uma atividade na qual a estruturação em módulos é adotada, portanto, um sistema complexo é estruturado em vários subsistemas independentes (módulos) [37]. Ressalta-se a importância da diferenciação entre módulo (relacionado a uma unidade funcional independente em relação ao propósito do produto), modularização (estruturação em vários subsistemas) e modularidade (concepção de produtos complexos a partir da combinação de módulos relativamente simples) [37]. Para [45], a modularização se dá através de duas características inerentes ao conceito: 1) similaridade entre a arquitetura física e funcional do produto; 2) minimização do grau de interação entre os componentes físicos.

A partir do conceito de modularização, surge o termo *bad smells*, que segundo [22] *smells* são estruturas no código que sugerem a possibilidade de refatoração. Já [35], conceitua *smells* como *"symptoms of poor design and implementation choices"*, sendo descrito na literatura, uma catalogação de 104 *smells*, sendo os mais importantes *duplicate code* (DC), *large class* (LC), *feature envy* (FE) e outros. Alguns não apresentam técnicas, ou até mesmo ferramentas para identificação de suas instâncias, tornando assim o uso de estratégias, uma opção para identificação de alguns *bad smells*, portanto, basicamente os *bad smells*, descrevem possíveis problemas em determinado código, possibilitando oportunidades de refatoração [37]. A evidenciação de *bad smells*, reforça a necessidade de exposição da divida técnica.

A dívida técnica é nada mais do que reflexão dos compromissos técnicos, que podem resultar em benefícios a curto prazo, mas em contrapartida, podem causar danos e prejuízos à qualidade de um sistema de software a longo prazo [9]. Segundo ressalta [3], a dívida técnica é inevitável, sendo assim, o enfoque principal não é tentar eliminá-la, mas mantê-la sob controle, por meio de seu gerenciamento. É por conta disso que, a dívida técnica é causada a partir de uma tomada de decisão, ou um processo, uma ação, ou falta dela, resultando pela pressão de uma programação, ou de cronograma, pela indisponibilidade de uma pessoa chave, ou pela falta de informações sobre um recurso técnico [6].

3 TRABALHOS RELACIONADOS

Esta Seção realiza uma análise comparativa dos trabalhos relacionados. A análise tem por objetivo identificar, destacar as características comuns e as diferenças entre os estudos já realizados e o trabalho proposto. A seção está divido da seguinte forma: a Seção 3.1 descreve a metodologia utilizada para escolha dos trabalhos relacionados; a Seção 3.2 é realizada a análise de cinco artigos que satisfazem os critérios de seleção; a Seção 3.3 realiza a comparação dos trabalhos, mediante critérios definidos; por último as oportunidades de pesquisa identificadas.

3.1 Metodologia para escolha dos trabalhos

Este trabalho utilizou como base de dados o *Google Scholar*. Foi utilizada a *query* de pesquisa com as seguintes palavras-chave:

("*microservice**" **OR** "*micro-service*") **AND** ("*monolith**") **AND** ("*comparative study*" **OR** "*empirical study*" **OR** "*performance*")

Com base nos resultados obtidos na busca, foram selecionados cinco artigos pela similaridade com o tema em estudo.

3.2 Análise dos trabalhos relacionados

Nesta seção, será realizada uma análise comparativa de cinco trabalhos que abordam o tema de análise de performance entre aplicações com arquitetura monolítica e microsserviços.

Gos and Zabierowski (2020) [26]. Neste estudo, os autores propõem uma análise comparativa, entre uma aplicação desenvolvida em uma arquitetura monolítica e outra equivalente desenvolvida em uma arquitetura baseada em microsserviços. A aplicação alvo é um sistema de aluguel de carros, contendo as funcionalidades de consulta, cadastro, aluguel e atualização do status. A aplicação monolítica foi desenvolvida em *Java* utilizando *Spring* e um único banco de dados *PostgreSQL*. A aplicação baseada em arquitetura de microsserviços também foi desenvolvida em *Java* utilizando *Spring*, sendo composta por quatro serviços autônomos e independentes. Os testes de carga foram realizados utilizando a ferramenta *Gatling*, onde foram separados em dois cenários, o primeiro visa simular 1000 requisições feitas por 30 usuários simultaneamente, já no segundo cenário é um total de 10000 requisições feitas por 30 usuários simultaneamente. Ambos cenários foram aplicados em requisições *GET* e requisições *POST*. Os autores concluem que a arquitetura baseada em microsserviços é mais eficiente em lidar com um número grande de requisições simultâneas, além de permitir construir um software de alta qualidade, fácil de escalar, confiável e a longo prazo mais fácil de manter. Também consideram que a arquitetura monolítica é mais eficiente em cargas mais baixas e mais fácil de ser desenvolvida.

Villamizar *et al.* (2015) [47]. Neste estudo, os autores buscam avaliar em um cenário real as implicações de uso de uma arquitetura baseada em microsserviços. A aplicação alvo é um sistema de gestão de empréstimos realizados por uma instituição financeira aos seus clientes. O estudo selecionou apenas dois dos serviços mais utilizados, o primeiro serviço intitulado S^1 é responsável por gerar um plano de pagamento, contendo as parcelas do empréstimo contratado, o segundo serviço S^2 é responsável por retornar um plano de pagamento existente com o respectivo conjunto de parcelas. O S^1 utiliza algorítimos para gerar cada plano de pagamento, usando excessivamente recursos da *CPU*, o S^2 realiza a consulta do plano de pagamento através de um identificador único na base de dados. Considerando a aplicação desenvolvida na arquitetura monolítica onde ambos serviços estão implementados na mesma aplicação e expostos via *endpoints*, temos os seguintes tempos médios de respostas, S^1 com 3.000 milissegundos e S^2 em torno de 300 milissegundos. Na versão da aplicação desenvolvida utilizando arquitetura baseada em microsserviços, foram criados dois microsserviços, um

para implementar as regras do S^1 e outro as regras do S^2. Ambas abordagens da aplicação foram desenvolvidas com Java, utilizando o *framework Play* e hospedados em uma solução *cloud* da *Amazon Web Services (AWS)*. Utilizaram o *JMeter* para realizar os testes de carga e avaliar a performance, onde foram criados dois cenários de teste aplicados, nas duas aplicações, sendo que o primeiro cenário de teste é focado no S^1 executando 30 requisições por minuto, no segundo cenário de teste focado no S^2 foram executados 1.100 requisições por minuto. Os autores concluem que os microsserviços não impactam consideravelmente na latência de resposta usando mais instâncias, no entanto, a granularidade dos microsserviços permite escalonar pontos específicos da aplicação, reduzindo os custos de infraestrutura. Também consideram que há benefício no desenvolvimento de software utilizando arquitetura baseada em microsserviços, permitindo com que pequenas equipes trabalhem em pequenos microsserviços de forma independente. Os autores identificaram desafios de sistemas distribuídos ao introduzirem a arquitetura baseada em microsserviços, os quais são gerenciados de maneira mais simples, em uma arquitetura monolítica.

Rudrabhatla (2020) [41]. Este estudo aborda três técnicas de decomposição de aplicações monolíticas para microsserviços e realiza uma comparação de performance e latência entre elas. A aplicação alvo é um módulo de uma aplicação *e-commerce*, composta inicialmente por três entidades de negócios. Na primeira técnica de decomposição abordada, sugere o uso da técnica *Domain-Driven Design*, onde usando *Common Closure Principle* são identificadas as classes que são impactadas pela mesma regra de negócio, devem estar no mesmo pacote em um único microsserviço. No sentido de que cada microsserviço seja autônomo, cada um possui uma base dados própria e isolada dos demais. A segunda técnica aborda a decomposição baseada em *Business capability* onde as principais entidades são normalizadas e os microsserviços são construídos considerando a transação de negócio. A granularidade das entidades são definidas até o ponto em que possam ser consideradas autônomas e operações de *CRUD* são criadas para gerenciamento. Na terceira e última técnica abordada, a decomposição é baseada em uma abordagem híbrida, combinando as duas primeiras técnicas, onde são identificados subdomínios utilizando *Business capability* gerando microsserviços unificados. O autor realiza uma comparação de desempenho e latência entre as três abordagens, observando os tempos de respostas para persistência dos dados em banco de dados. Como resultado da análise comparativa, o autor pode demonstrar que a técnica *Domain-Driven Design* foi superior a *Business capability*, mas que em uma aplicação *real time* a abordagem híbrida produziu melhores resultados.

Tapia *et al.* (2020) [43]. Este estudo propõe uma análise comparativa de performance entre arquitetura monolítica e arquitetura baseada em microsserviços. A aplicação alvo de estudo é um fórum, contendo três entidades de negócio. Primeiramente são descritas as estruturas técnicas, onde aplicação monolítica é desenvolvida utilizando a linguagem *Node.js*, contendo as entidades de negócio implementadas na mesma aplicação, hospedada em uma máquina virtual. A aplicação equivalente em arquitetura baseada em microsserviços também é desenvolvida utilizando a linguagem *Node.js*. As entidades de negócios estão separadas em aplicações distintas, conteinerizadas, utilizando *Docker* e hospedadas na estrutura de *cloud* da *Amazon*. O processo de coleta de dados é feito através de testes de estresse, separados por dois cenários, onde o primeiro realiza requisições via *HTTP* para gerar dados no banco de dados, o segundo cenário realiza requisições via *HTTP* para selecionar dados persistidos no banco de dados. O estudo também aplica o modelo matemático *Non-Parametric Regression Model* para relacionar dados de recursos computacionais utilizados e assim poder avaliar os dados coletados. Os autores concluem que o uso da arquitetura baseada em microsserviços é mais vantajosa em relação aos recursos de hardware e redução de custos.

Bjørndal *et al.* (2020) [8]. Neste estudo os autores procuram verificar os benefícios de migração de uma arquitetura monolítica para uma arquitetura baseada em microsserviços, realizando experimentos de *benchmarking*. A aplicação alvo é um sistema fictício de biblioteca, contendo as funcionalidades necessárias para login e empréstimo de livros. A aplicação com arquitetura monolítica foi desenvolvida utilizando *ASP.NET Core* e base de dados centralizada utilizando *SQL Server*, já a aplicação com arquitetura baseada em microsserviços gerou quatro serviços, representando as entidades de domínio, cada um com seu contexto limitado e base de dados própria. Foram aplicados dois experimentos de *benchmarking*, onde a aplicação foi executada em uma estrutura local e outra hospedada no *Azure Cloud*. As métricas utilizadas em ambos experimentos foram *Throughput*, *Latency*, *Scalability* e recursos de hardware (*CPU*, *Memory*, *Network*). Os experimentos são compostos por uma execução simples e outra complexa, todos eles executados utilizando *JMeter*. Os autores concluem que a arquitetura monolítica foi melhor em todas métricas, exceto na escalabilidade, entretanto consideram pontos em que o ambiente e aplicação podem ter levado a este resultado, como o tamanho reduzido da aplicação, se comparado a uma aplicação real, assim como gargalos de uso entre *kubernetes*, *Docker* e os próprios bancos de dados. Concluem também que apesar das considerações anteriores, a arquitetura baseada em microsserviços se torna uma melhor alternativa, considerando uma grande quantidade de usuários simultâneos, devido a métrica de escalabilidade ter sido melhor.

3.3 Análise comparativa dos trabalhos relacionados

Nesta seção será realizada a comparação dos trabalhos selecionados no item 3.2 com base nos critérios comparativos definidos:

- **Contexto de Avaliação:** Este critério irá avaliar se a proposta do trabalho foi avaliada em um ambiente *acadêmico* ou na *indústria*.
- **Domínio da aplicação:** Este critério irá avaliar qual o domínio da aplicação alvo: *automotivo*, *comunicação*, *financeiro*, *literatura*, *varejo*.
- **Métodos de estudo empírico:** Este critério irá avaliar qual o método de avaliação foi utilizado para avaliar as arquiteturas propostas pelos trabalhos: *estudo de caso*, *experimento* ou *levantamento*.
- **Tipos de Métricas:** Este critério irá avaliar quais métricas de avaliação foram utilizadas pelos trabalhos: *custos*, *disponibilidade*, *escalabilidade*, *performance*, *qualidade de código*, *segurança*.

Sobre as principais contribuições, apenas o trabalho [47] foi aplicado na indústria. Os trabalhos [26], [41], [43] e [8] foram aplicados

Table 1: Comparação dos trabalhos relacionados

Critérios		Trabalhos				
		Gos and Zabierowski (2020) [26]	Villamizar *et al.* (2015) [47]	Rudrabhatla (2020) [41]	Tapia *et al.* (2020) [43]	Bjørndal *et al.* (2020) [8]
Contexto de avaliação	Acadêmico	+	-	+	+	+
	Indústria	-	+	-	-	-
Domínio da aplicação	Automotivo	+	-	-	-	-
	Comunicação	-	-	-	+	-
	Financeiro	-	+	-	-	-
	Literatura	-	-	-	-	+
	Varejo	-		+	-	-
Métodos de estudo empírico	Estudo de caso	+	+	+	+	+
	Experimento controlado	-	-	-	-	-
	Survey (Levantamento)	-	-	-	-	-
Tipo de Métricas	Custos	-	+	-	-	-
	Disponibilidade	-	-	-	-	-
	Escalabilidade	-	+	-	+	+
	Performance	+	+	+	+	+
	Qualidade de código	-	-	-	-	-
	Segurança	-	-	-	-	-

Legenda: (+) Aplicado (-) Não aplicado

na acadêmia. O domínio da aplicação alvo no estudo [26] foi automotivo, no estudo [47] foi financeiro, no estudo [41] foi varejo, no estudo [43] foi comunicação e no estudo [8] foi literatura. Como métodos de avaliação dos trabalhos selecionados, todos os trabalhos apresentaram um estudo de caso de suas proposta. Dos estudos selecionados apenas o [47] considerou os custos como métrica. Os estudos [43] e [8] avaliaram como métrica a escalabilidade. No entanto, todos os estudos consideraram a métrica de performance para avaliação.

Oportunidades de pesquisa. Após a análise dos artigos, as seguintes oportunidades de pesquisa foram identificadas: (1) avaliação entre ambas arquiteturas considerando a performance; e (2) avaliação da complexidade de código considerando métricas de modularidade de software;

Destas oportunidades, este trabalho procura gerar conhecimento empírico sobre os impactos da arquitetura monolítica e da arquitetura baseada em microsserviços na modularidade e na performance.

4 METODOLOGIA

Esta seção apresenta as principais decisões que fundamentam a análise comparativa do estudo de caso [7, 24]. Para começar, são apresentados os objetivos e as questões de pesquisa (Seção 4.1). Em seguida, são descritas as hipóteses (Seção 4.2). Na sequência a seleção da aplicação alvo (Seção 4.3). As variáveis e métodos de quantificação considerados, também são discutidos em detalhes (Seção 4.4). A seleção de métricas é abordada na sequência (Seção 4.5). A descrição do procedimento de análise (Seção 4.7). Finalmente, o procedimento experimental é apresentado (Seção 4.6). Todas essas etapas metodológicas foram baseadas em diretrizes práticas em estudos empíricos [19].

4.1 Objetivo e Questões de Pesquisa

Este estudo tenta essencialmente avaliar os efeitos do estilo de arquitetura de microsserviços na performance e modularidade de software. Busca-se investigar os efeitos da performance de hardware, através de duas métricas: consumo de memória e consumo de CPU [8]. No entanto, busca-se investigar os efeitos na modularidade de software através de duas variáveis [12]: acoplamento e coesão. Esses efeitos são investigados no contexto de uma aplicação real do sistema financeiro (Seção 4.3), a qual foi construída seguindo o estilo arquitetural de microsserviços, resultado de um processo de decomposição de uma aplicação monolítica (Seção 4.3.1). Com isso em mente, o objetivo deste estudo é estabelecido com base no modelo GQM [42].

analisar estilos arquiteturais
com o propósito de investigar seus efeitos
com relação à modularidade de software e performance
da perspectiva dos desenvolvedores
no contexto da evolução tecnológica

Este estudo tem como foco principalmente avaliar os efeitos da decomposição de uma aplicação monolítica em uma arquitetura de microsserviços. Desta forma, o foco será em duas questões de pesquisa:

- **QP1:** Qual seria o efeito na *modularidade de software* de aplicações monolíticas em comparação com aplicações baseadas em microsserviços?
- **QP2:** Qual seria o efeito na *performance* de aplicações monolíticas em comparação com aplicações baseadas em microsserviços?

4.2 Hipóteses

Hipótese 1. Conjectura-se que a arquitetura monolítica tende a gerar aplicações maiores, com um grande número de dependências, o que causa alto acoplamento do software [14, 38]. No entanto, o estilo arquitetural de microsserviços separa a aplicação em serviços independentes [16], promovendo o baixo acoplamento e a alta coesão [33]. Consequentemente, conjectura-se que os microsserviços resultantes da decomposição irão possuir maior modularização do software. Desta forma, declarando as hipóteses nula e alternativa da seguinte forma:

Hipótese Nula 1, H_{1-0}: Não há diferença entre as métricas de modularidade de software utilizando arquitetura de microsserviços em comparação com a arquitetura monolítica.

$$H_{1-0} : Modularidade(Microsservico) = Modularidade(Monolitica) \quad (1)$$

Hipótese alternativa 1, H_{1-1}: A arquitetura de microsserviços possui maior modularidade de software em comparação com a arquitetura monolítica.

$$H_{1-1} : Modularidade(Microsservico) > Modularidade(Monolitica) \quad (2)$$

Testando esta primeira hipótese, será confirmado (ou não) o aumento da modularidade de software, através de métricas avaliando cada aspecto do software gerado.

Hipótese 2. Conforme mencionado anteriormente, a arquitetura de microsserviços gera serviços independentes entre si, comunicando-se de maneira leve, com recursos de hardware próprios. Entretanto, a arquitetura monolítica segue uma abstração dos componentes que depende do compartilhamento de recursos do mesmo servidor onde os componente não são executáveis de forma independente [33]. No entanto, os ganhos de performance provenientes do uso de cada arquitetura não são evidentes [8]. Consequentemente, conjectura-se que a arquitetura baseada em microsserviços exigirá um menor consumo de memória e menor uso de CPU. No entanto, não é de forma alguma óbvio que esta hipótese seja válida. Talvez, a quantidade de requisições simultâneas gere um consumo menor de memória e CPU entre os microsserviços; ou ambas arquiteturas obtenham resultados semelhantes. Com base nesta declaração, declaro as hipóteses nula e alternativa da seguinte forma:

Hipótese Nula 2, H_{2-0}: Não há diferença no uso dos recursos de hardware utilizando arquitetura de microsserviços em comparação com a arquitetura monolítica.

$$H_{2-0} : ConsumoCPU(Microsservico) = ConsumoCPU(Monolitica) \quad (3)$$

$$H_{2-0} : ConsumoMemoria(Microsservico) = ConsumoMemoria(Monolitica) \quad (4)$$

Hipótese alternativa 2, H_{2-1}: Há uma redução do consumo de memória e CPU utilizando arquitetura de microsserviços em comparação com a arquitetura monolítica.

$$H_{2-1} : ConsumoCPU(Microsservico) < ConsumoCPU(Monolitica) \quad (5)$$

$$H_{2-1} : ConsumoMemoria(Microsservico) < ConsumoMemoria(Monolitica) \quad (6)$$

Testando esta segunda hipótese, será possível avaliar o uso dos recursos de hardware, comparando os recursos utilizados em ambas abordagens arquiteturais.

4.3 Seleção da Aplicação Alvo

Para a execução do estudo de caso, foram selecionados projetos no repositório de código fonte da *Cooperativa Utile*. A coleta dos projeto considerou algumas características:

(1) Aplicação ser caracterizada como monolítica;
(2) Aplicação escrita utilizando linguagem Java;
(3) Decomposição da aplicação estar finalizada ou em etapa de finalização;

Após a conclusão da busca dos projetos, obteve-se uma amostra inicial de 3 projetos. Os 3 projetos foram avaliados, tanto em qual etapa de entrega estava quanto sua complexidade. Os projetos em etapa de execução intermediária ou análise da decomposição foram descartados. A aplicação selecionada foi projetada para atender diversos recursos dos terminais de autoatendimento aos clientes da instituição financeira, o objeto de estudo será focado no módulo de saque em dinheiro.

Esta aplicação atualmente atende diversos tipos de terminais de autoatendimento, alguns dos recursos são: extrato de conta corrente, extrato e antecipação de cota capital, seguro veicular, transferência eletrônica, depósito de cheque, pagamento de DOC, entre outras funcionalidades.

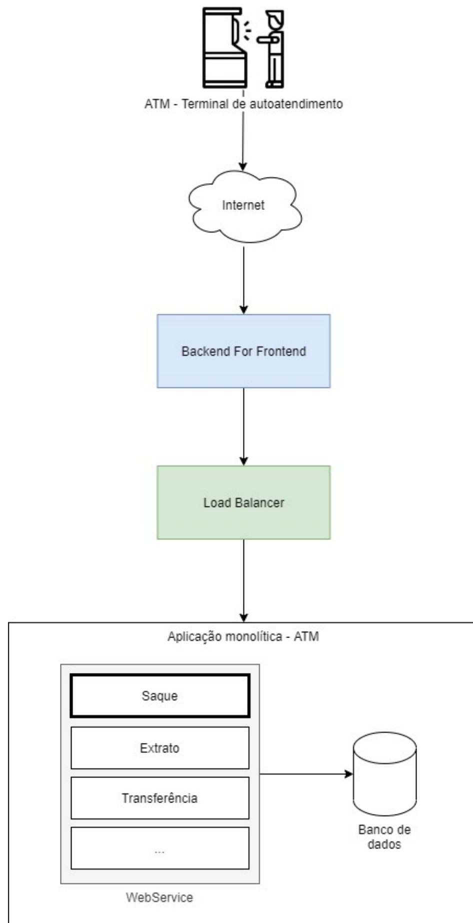

Figure 1: Diagrama geral da arquitetura monolítica

4.3.1 Decomposição. Para realizar o processo de decomposição da aplicação monolítica foram utilizados os princípios do *Domain-Driven Design* [16], desta forma serão definidos os domínios juntamente com os seus delimitadores. A aplicação alvo possui um alto nível de criticidade, desta forma será preciso realizar uma migração gradativa na utilização dos microsserviços decompostos. Haverá um chaveamento por tipo de terminal de atendimento e números de contas, desta forma será possível ter um controle maior sobre os possíveis clientes impactados. Os microsserviços resultantes da decomposição foram construídos utilizando a linguagem *Java* e o *framework Spring*. A escolha das tecnologias utilizadas se deu através do conhecimento e experiência comum entre os membros da equipe técnica. A comunicação entre os microsserviços é feita através do protocolo *HTTP*, seguindo o padrão *REST*.

Para a decomposição da base de dados, foi utilizado o *SQL Server*, onde a estrutura de dados foi implementada para ser migrada parcialmente, tendo em vista a grande dependência dos demais sistemas com as tabelas da aplicação alvo.

4.3.2 Testes de carga. Os testes de cargas serão aplicados utilizando a ferramenta *Gatling*, a qual é uma ferramenta de alta performance, uso simples e fácil manutenção. A ferramenta *Gatling* suporta protocolo HTTP, a qual será a forma de comunicação presente nas duas arquiteturas da aplicação. Os cenários serão escritos utilizando a linguagem *Scala*, o que torna necessário conhecer o básico da linguagem. Contudo, ao final dos testes é gerado um relatório contendo informações como: usuários ativos durante a simulação, requisições por segundo, respostas por segundo e uma variedade de percentuais sobre os tempos de resposta das requisições. A execução do teste de carga simulou 4 quantidades diferentes de requisições por segundo durante determinado período. A Table 2 lista as simulações executadas para o teste de carga.

Table 2: Tabela simulações do teste de carga

Quantidade requisições por segundo	Duração
10	10 minutos
20	10 minutos
30	10 minutos
40	10 minutos

4.4 Variáveis e Método de Quantificação

Variável dependente na primeira hipótese. A variável dependente na primeira hipótese será a de métricas de modularidade de software descritas na Table 3, onde o código será avaliado sob diversos aspectos da modularidade de software. O cálculo dessa variável permite estudar o impacto variável de cada métrica.

Variável dependente na segunda hipótese. A variável dependente na segunda hipótese será a de métricas de performance descritas na ??. O cálculo dessa variável permite estudar o impacto no uso de recursos de hardware em ambas abordagens arquiteturais, de acordo com uma carga de uso definida.

Variável independente. A variável independente dos hipóteses 1 e 2 será o estilo arquitetural monolítico e o baseado em microsserviços.

4.5 Métricas Selecionadas

Neste estudo serão utilizados os conjuntos de métricas *Acoplamento* e *Coesão* para avaliar a modularidade de software. Para avaliação de performance serão utilizadas métricas *Consumo de CPU* e *Consumo de memória.*

A Table 3 apresenta as métricas selecionadas para quantificar variáveis de modularidade, sendo acoplamento e coesão. A seleção das métricas foi feita com base em estudos empíricos anteriores [24, 28] que comprovam a validade destas métricas para análise de modularidade de software. As métricas utilizadas foram:

- **CBO:** Esta métrica mede quantas classes uma determinada classe depende. Em Orientação a Objetos, o baixo acoplamento entre objetos indica um bom grau de modularidade [12].
- **DIT:** Esta métrica é definida como o comprimento máximo do nó até a raiz da árvore. Quanto mais profunda uma classe está na hierarquia, maior o número de métodos que ela provavelmente herdará, tornando mais complexo prever seu comportamento [12].
- **WMC:** Métrica que mede a complexidade da classe, obtida em termos da complexidade de cada um de seus métodos. Valor maior indica maior tempo e esforço para desenvolver

e manter a classe. Quanto maior o número de métodos de uma classe, maior será o impacto potencial sobre os filhos, pois estes herdarão os métodos da classe pai [12].
- **NOC:** Métrica que mede a largura da hierarquia de uma classe. Quanto maior o número, maior o reaproveitamento, entretanto quanto maior o número de filhos, maior a probabilidade de abstração inadequada da classe pai, além da possibilidade de necessitar de mais testes dos métodos dessa classe [12].
- **LCOM:** Métrica que mede a falta de coesão de uma classe, selecionando todos os pares de métodos de uma classe e verificando se esses compartilham algum atributo. Quanto maior o número, maior é a falta de coesão, aumentando assim a complexidade, levando a um provável aumento do número de defeitos injetados no software [12].

Para comparação de performance entre a arquitetura monolítica e a arquitetura baseada em microsserviços, foram selecionadas as métricas de consumo de CPU e consumo de memória, visando determinar o desempenho de cada arquitetura [43]. O consumo de CPU se refere à porcentagem de unidade do recurso CPU virtual que está sendo usado pelo *Kubernetes* [1]. O consumo de memória se refere à porcentagem de memória que foi utilizada na execução do processo [2].

4.6 Processo Experimental

O processo experimental definido foi baseado em estudos empíricos publicados [15, 17–19], onde é identificado um conjunto de atividades, organizadas em três fases (Figura 2).

A Figura 2 mostra através de um processo experimental como as três fases foram organizadas. As atividades são descritas a seguir:

- **Buscar projetos:** foi realizada a busca de projetos no repositório de códigos da *Cooperativa Utile*.
- **Avaliar projetos:** foi realizada a avaliação dos projetos, para averiguar quais seriam os mais aptos a serem utilizados no estudo de caso. A seleção se baseou nas características definidas na Seção 4.3.
- **Mapear fluxo da aplicação:** identificado o fluxo da aplicação alvo e criação dos testes de carga.
- **Executar testes de carga:** testes de carga executados e os dados resultantes foram coletados.
- **Avaliar resultados:** avaliação dos resultados dos testes de carga usando o método de quantificação descrito na Seção 4.4.

4.7 Procedimento de Análise

A análise quantitativa dos dados de modularidade de software, será realizada através dos dados coletados pela ferramenta *Understand*[1]. Onde para a análise de distribuição dos dados de cada métrica, serão utilizados os seguintes métodos estatísticos: desvio padrão, valor máximo, mediana, média e percentual de variação entre as médias. Outros estudos usam desta abordagem para analisar métricas de software [28].

[1]Understand: https://www.scitools.com/

A análise quantitativa dos dados refentes à análise de performance, será realizada através da estatística descritiva para analisar sua distribuição normal [48] e inferência estatística para testar as hipóteses. O nível de significância dos testes de hipótese serão α = 0,05. Para testar as hipóteses, será aplicado o teste T das amostras [32].

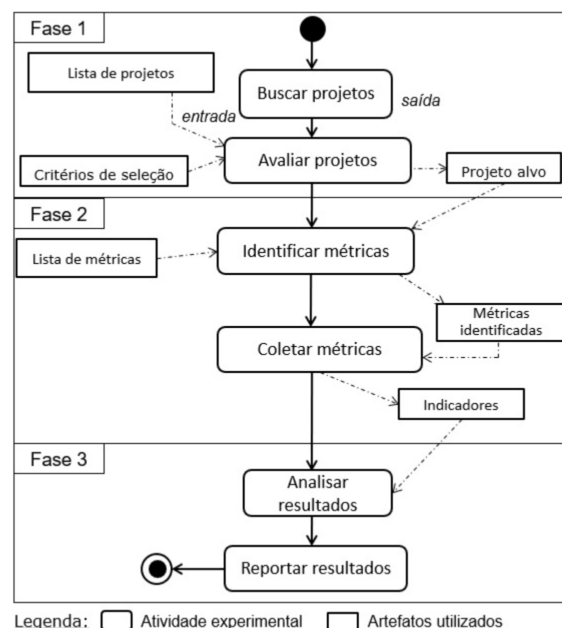

Figure 2: Processo experimental

5 RESULTADOS

Esta seção tem como objetivo apresentar os resultados referentes as questões de pesquisas formuladas na section 4. A Seção 5.1 discute os resultados obtidos referentes a performance de hardware. Na Seção 5.2 discute-se os resultados obtidos referentes a modularidade de software. Por fim a Seção 5.3 apresenta uma discussão adicional sobre os resultados obtidos.

5.1 Análise de performance

A aplicação alvo da empresa hipotética *Cooperativa Utile*, do ramo financeiro, foi projetada para atender operações realizadas no terminal de autoatendimento. Considerando as aplicações monolíticas já existentes na empresa, o valor médio para consumo de CPU é de 2,5 vCPU, já o valor médio de consumo de memória é de 1536 MiB.

5.1.1 Consumo de Memória. **Estatística descritiva.** A Table 5 apresenta os dados referentes a variável do consumo de memória. Embora esse valor baixo, é necessário verificar se esse valor obtido possui diferença estatisticamente relevante ao limite aceitável hipoteticamente definido pela *Cooperativa Utile*. Para isso será testada a hipótese no parágrafo à frente.

Table 3: Tabela métricas de modularidade de software [24]

Atributos	Métricas	Definições
Acoplamento	Coupling betweenobjects (CBO)	Esta métrica mede quantas classes uma determinada classe depende.
	Depth inheritancetree(DIT)	Esta métrica é definida como o comprimento máximo do nó até a raiz da árvore.
	Weighted Methods per Class (WMC)	Métrica que mede a complexidade da classe, obtida em termos da complexidade de cada um de seus métodos.
	Number of Children (NOC)	Métrica que mede a largura da hierarquia de uma classe.
Coesão	Lack of Cohesion in Methods (LCOM/LOCM)	Esta métrica mede a falta de coesão de uma classe, selecionando todos os pares de métodos de uma classe e verificando se esses compartilham algum atributo.

Table 5: Tabela resultados consumo de Memória (medidas em MebiByte - MiB)

Tamanho da amostra	Menor	Maior	Média	Desvio padrão
84	479,75	788,84	627,68	95,09

Teste de hipótese questão de pesquisa 1.

Resultados observados 1: *O consumo médio de memória (M = 627,68, SD = 95,09) foi menor do que o consumo médio padrão das aplicações monolíticas de 1536,00, uma diferença média estatisticamente significativa de 908,32, IC de 95% [607,05, 648,32], t (84) = -87,54, p-value = 0,001.*

5.1.2 Consumo de CPU. **Estatística descritiva.** A Table 6 apresenta os dados referentes à variável do consumo de CPU. Embora esse valor seja baixo, é necessário verificar se esse valor obtido possui diferença estatisticamente relevante ao limite aceitável hipoteticamente definido pela *Cooperativa Utile*. Para isso, será testada a hipótese no parágrafo à frente.

Table 6: Tabela resultados consumo de CPU (medidas em vCPU)

Tamanho da amostra	Menor	Maior	Média	Desvio padrão
84	0,14	0,92	0,45	0,16

A Tabela 5.1.2 apresenta os resultados do teste de hipótese para análise do consumo de *CPU*. Os valores apontam que o $p < 0{,}05$ indicando que há uma diferença estatisticamente relevante, onde o uso da arquitetura baseada em microsserviços apresenta uma diferença significativa, entre o valor de limite aceitável e o valor apresentado. Portanto, a arquitetura baseada em microsserviços se configura como sendo uma arquitetura promissora, corroborando com outros estudos [8, 43] que evidenciam menor consumo dos recursos de hardware.

Teste de hipótese questão de pesquisa 1.

Resultados observados 2: *O percentual de consumo médio de CPU (M = 0,45, SD = 0,16) foi menor do que o percentual de consumo médio padrão dos microsserviços de 2,50, uma diferença média estatisticamente significativa de 2,05, IC de 95% [0,42, 0,49], t (84) = -116,7, p-value = 0,001.*

5.2 Análise das métricas de modularidade de software

A Tabela 7 traz os indicadores estatísticos sobre o resultado das métricas coletadas, referentes à modularidade de software. Apresenta os resultados dos efeitos da arquitetura baseada em microsserviços em comparação à arquitetura monolítica, através dos atributos de Acoplamento e Coesão, incluindo o desvio padrão, mediana, média e percentual de variação entre as médias. Este conjunto de métricas visa identificar a modularidade do software, já a Table 8 apresenta valores de referência para as métricas [12], tendo como categorias: Baixo, Alto e Anomalia.

Table 8: Valores referência para as métricas CK em softwares desenvolvidos na linguagem Java [28]

	DIT	NOC	CBO	RFC	LCOM	WMC
Baixo	0	0	0	0	0	230
Alto	2,5	6,4	13,6	35,5	92,7	253,1
Anomalia	3	8	18	46	120	329

A categoria de Acoplamento traz as seguintes métricas para comparação: *CBO*, *DIT*, *WMC* e *NOC*. A arquitetura baseada em microsserviços apresentou resultados melhores em comparação à arquitetura monolítica. Isso pode ser identificado através da diferença entre as médias de cada arquitetura, representada pelo percentual de variação, sendo 30,76%, 76,67%, 209,85% e 121,35% respectivamente.

CBO. O percentual de variação de 30,76% para a arquitetura de microsserviços, indica um melhor grau de modularidade do código, onde os princípios da Orientação a Objetos foram melhores aplicados. Ambas arquiteturas obtiveram médias classificadas entre

Table 7: Tabela resultados das métricas de modularidade de software

Atributos	Métricas	Arquitetura	Desvio Padrão	Máximo	Mediana	Média	% de variação
Acoplamento	CBO	Monolítica	12,53	121	5	10,06	30,76%
		Microsserviços	7,22	38	6	7,70	
	DIT	Monolítica	0,92	4	1	1,62	76,67%
		Microsserviços	0,51	3	1	0,92	
	WMC	Monolítica	82,81	2327	7	19,34	209,85%
		Microsserviços	15,79	231	2,5	6,24	
	NOC	Monolítica	4,60	86	0	0,51	121,35%
		Microsserviços	0,62	4	0	0,23	
Coesão	LCOM	Monolítica	37,84	100	61	48,49	129,32%
		Microsserviços	30,02	100	0	21,15	

Baixo e Alto, segundo os valores de referência da Table 8. Entretanto, ambas arquiteturas tiverem classes com valores categorizados como Anomalia.

DIT. A arquitetura de microsserviços tendo um percentual de variação menor, indica menos complexidade e também a possibilidade de menos reutilização de código por meio de herança. Ambas arquiteturas obtiveram médias classificadas entre Baixo e Alto, segundo os valores de referência da Table 8. Contudo, os resultados obtidos indicam que ambas arquiteturas tiveram classes com valores acima do categorizado como Anomalia, indicando que ambos projetos possuem classes problemáticas.

WMC. A arquitetura monolítica teve percentual de variação superior em 209,85%, indicando classes com maior número de métodos e complexidades entre eles. Além do percentual de variação em comparação à arquitetura de microsserviços, esta métrica obteve classes com valores superiores ao categorizado como Anomalia.

NOC. A arquitetura monolítica apresentou um número maior desta métrica, a qual obteve classes com valor máximo de 86. Um alto valor desta métrica indica que há probabilidade da classe apresentar defeitos, devido à grande hierarquia das classes [28].

LCOM. Os resultados coletados desta métrica trazem uma redução de 129,32% para a arquitetura de microsserviços em comparação com a arquitetura monolítica. Entretanto, ambas arquiteturas obtiveram valores máximos de 100, podendo ser classificadas com categoria Alto, segundo a Table 8 com valores de referência.

Estes resultados eram esperados, pois a aplicação monolítica apresentou um número elevado de classes e linhas de códigos. Isso pode ser explicado pelo fato da aplicação monolítica possuir um conjunto de funcionalidades maiores, consequentemente necessitando de um número maior de classes e linhas de código, outros estudos indicam estes resultados [5]. Entretanto, a aplicação baseada em microsserviços foi resultado de um processo de decomposição da aplicação monolítica, onde foram extraídas as funcionalidades necessárias para o produto, isso facilitou um melhor entendimento do domínio deste produto [16], onde foram melhor aplicados os conceitos da linguagem orientada a objetos, diminuindo a complexidade do código. Esta última afirmação se fundamenta principalmente através dos melhores resultados para as métricas WMC e CBO, as quais foram utilizadas com maior chance de acerto em outros estudos [28] para prever classes com propensão a erros.

Resultados observados 3: *As métricas de acoplamento e coesão obtiveram valores médios considerados baixos para ambas arquiteturas [28]. No entanto, a aplicação com arquitetura baseada em microsserviços, obteve valores menores em todas as métricas, evidenciando um menor acoplamento e maior coesão.*

5.3 Discussão

Modularidade de software. A arquitetura monolítica obteve bons resultados considerando os valores de referência. Entretanto, ao analisar os resultados obtidos através dos conjuntos de métricas definidas, identificou-se que a arquitetura baseada em microsserviços obteve melhores resultados de forma geral. O baixo acoplamento indica um aumento na qualidade do *software* e possivelmente maior reuso dos componentes [30].

Análise de performance. A arquitetura monolítica obteve valores elevados em comparação com a arquitetura baseada em microsserviços. Os valores de consumo de memória para os microsserviços indicam uma redução considerável, tendo em vista o contexto de software reduzido e uma menor dependência de outros softwares. O consumo de CPU e memória, inferiores na aplicação com arquitetura baseada em microsserviços, indicam menor carga de processamento computacional.

5.4 Limitações do Estudo

O estudo de caso reportado se trata de um estudo inicial que explora um assunto pouco investigado na literatura. Desta forma, o estudo possui algumas limitações que devem ser consideradas. A aplicação monolítica considerada, possui diversas funcionalidades implementadas, além da funcionalidade alvo do estudo, podendo ter distorcido os resultados coletados. Tal consideração corrobora com o fato de ser uma aplicação antiga, onde não adotaram boas práticas e todos os conceitos da orientação a objetos. Sendo esse um dos motivos pelo qual houve um grande percentual de variação entre as métricas de modularidade de software. Outra dificuldade foi encontrada para coletar as métricas de consumo de memória e consumo de CPU, pela aplicação monolítica, onde não foi possível coletar devido a problemas técnicos no servidor onde os testes de carga foram executados. Neste caso, foram considerados valores baseados na experiência da equipe de desenvolvimento.

6 CONCLUSÃO E TRABALHOS FUTUROS

A arquitetura de microsserviços vem sendo adotada na indústria como forma de modernização de aplicações legadas. Surge como

alternativa para a arquitetura monolítica pois, traz maior escalabilidade e manutenibilidade da aplicação. No entanto, não há muitos estudos evidenciando os impactos da adoção deste estilo arquitetural em relação ao monolítico. Neste sentido, o estudo atual reportou um estudo de caso inicial com o propósito de comparar a arquitetura monolítica e a arquitetura de microsserviços. O estudo reportado procurou avaliar os impactos da utilização de ambas arquiteturas, através de métricas computacionais como, consumo de CPU, consumo de memória, além de métricas para medir a modularidade do software.

No estudo atual, as métricas selecionadas são avaliadas em uma aplicação financeira real, que atende as operações de saque, dos terminais de autoatendimento, da empresa fictícia *Cooperativa Utile*.

As descobertas indicam que o uso da arquitetura de microsserviços, apresentou bons resultados quanto as métricas de acoplamento e coesão. Contudo, ficou evidente a necessidade de um conjunto maior de métricas, para avaliar outros aspectos das aplicações, como por exemplo o acoplamento entre microsserviços.

As métricas computacionais para avaliar a performance tiveram valores hipotetizados para as métricas de consumo de CPU e consumo de memória. Novamente a arquitetura de microsserviços teve bons resultados em comparação a arquitetura monolítica. No entanto, mostrou-se necessário novas métricas para melhor avaliar a performance, como por exemplo a latência e a taxa de transferência [8].

A realização deste estudo, trouxe contribuições científicas referentes ao tema de decomposição de aplicações monolíticas em microsserviços. O uso de uma aplicação real como estudo de caso, aumentou o conhecimento empírico gerado sobre o tema.

Como trabalhos futuros, pretende-se: (1) definir um conjunto maior de métricas, visando aumentar a perspectiva de análise da modularidade do software e performance; (2) coletar mais dados em ambas arquiteturas, para realizar uma analise estatística sem a necessidade de hipotetizar os valores. Este trabalho pode ser tido como sendo um estudo inicial, de uma sequência de estudos de caso mais robustos, relacionados aos impactos da decomposição de aplicações monolíticas para microsserviços.

REFERENCES

[1] 2021. Managing Resources for Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu

[2] 2021. Managing Resources for Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-memory

[3] Eric Allman. 2012. Managing technical debt. *Commun. ACM* 55, 5 (2012), 50–55.

[4] Arun. 2015. A First Look at Microservices. *Java Magazine* sep/oct (2015).

[5] Tugrul Asik and Yunus Emre Selcuk. 2017. Policy enforcement upon software based on microservice architecture. In *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*. 283–287. https://doi.org/10.1109/SERA.2017.7965739

[6] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, Vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: a New OS Architecture for Scalable Multicore Systems. In *Proc. of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*. Big Sky, Montana, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[8] N Bjørndal, Antonio Bucchiarone, Manuel Mazzara, Nicola Dragoni, and Schahram Dustdar. 2020. Migration from Monolith to Microservices : Benchmarking a Case Study. (03 2020). https://doi.org/10.13140/RG.2.2.27715.14883

[9] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 47–52.

[10] André Stangarlin de Camargo et al. 2016. Uma abordagem para testes de desempenho de microservices. (2016).

[11] Carlos Eduardo Carbonera, Kleinner Farias, and Vinicius Bischoff. 2020. Software development effort estimation: a systematic mapping study. *IET Software* 14, 4 (2020), 328–344.

[12] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[13] Prashant Ramchandra Desai. 2016. A survey of performance comparison between virtual machines and containers. *Int. J. Comput. Sci. Eng* 4, 7 (2016), 55–59.

[14] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[15] Leandro Ferreira D'Avila, Kleinner Farias, and Jorge Luis Victória Barbosa. 2020. Effects of contextual information on maintenance effort: a controlled experiment. *Journal of Systems and Software* 159 (2020), 110443.

[16] Eric Evans. 2009. *Domain-driven design: atacando as complexidades no coração do software*. Alta Books.

[17] Kleinner Farias. 2016. Empirical evaluation of effort on composing design models. *arXiv preprint arXiv:1610.09012* (2016).

[18] Kleinner Farias, Alessandro Garcia, and Carlos Lucena. 2014. Effects of stability on model composition effort: an exploratory study. *Software & Systems Modeling* 13, 4 (2014), 1473–1494.

[19] Kleinner Farias, Alessandro Garcia, Jon Whittle, Christina von Flach Garcia Chavez, and Carlos Lucena. 2015. Evaluating the effort of composing design models: a controlled experiment. *Software & Systems Modeling* 14, 4 (2015), 1349–1365.

[20] Kleinner Farias, Lucian Gonçales, Vinicius Bischoff, Bruno Carreiro da Silva, Everton T Guimarães, and Jacob Nogle. 2018. On the UML use in the Brazilian industry: A state of the practice survey (S).. In *SEKE*. 372–371.

[21] Martin Fowler. 2015. Monolith First. https://martinfowler.com/bliki/MonolithFirst.html

[22] Martin Fowler, Kent Beck, and W Roberts Opdyke. 1997. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*.

[23] Martin Fowler and James Lewis. 2014. Microservices, a definition of this new architectural term. https://martinfowler.com/articles/microservices.html

[24] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. 2006. Modularizing design patterns with aspects: a quantitative study. In *Transactions on Aspect-Oriented Software Development I*. Springer, 36–74.

[25] Lucian Gonçales, Kleinner Farias, and Bruno C da Silva. 2021. Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and Software Technology* (2021), 106563.

[26] Konrad Gos and Wojciech Zabierowski. 2020. The Comparison of Microservice and Monolithic Architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE, 150–153.

[27] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 185–200.

[28] Renato Correa Juliano, Bruno AN Travençolo, and Michel S Soares. 2014. Detection of Software Anomalies Using Object-oriented Metrics. In *ICEIS (2)*. 241–248.

[29] Ed Júnior, Kleinner Farias, and Bruno Silva. 2021. A Survey on the Use of UML in the Brazilian Industry. In *Brazilian Symposium on Software Engineering*. 275–284.

[30] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. 1996. Scenario-based analysis of software architecture. *IEEE software* 13, 6 (1996), 47–55.

[31] Holger Knoche and Wilhelm Hasselbring. 2018. Using microservices for legacy software modernization. *IEEE Software* 35, 3 (2018), 44–49.

[32] David J Lilja. 2005. *Measuring computer performance: a practitioner's guide*. Cambridge university press.

[33] Sam Newman. 2015. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

[34] Anderson Oliveira, Vinicius Bischoff, Lucian José Gonçales, Kleinner Farias, and Matheus Segalotto. 2018. BRCode: An interpretive model-driven engineering approach for enterprise applications. *Computers in Industry* 96 (2018), 86–97.

[35] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 268–278.

[36] Masoud Rafighi, Yaghoub Farjami, and Nasser Modiri. 2015. Studying the deficiencies and problems of different architecture in developing distributed systems and analyze the existing solution. In *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. IEEE, 826–834.

[37] Cesar Coutinho Ramos. 2016. Análise e aplicação de métodos de modularização no desenvolvimento do produto. (2016).

[38] Chris Richardson. 2018. *Microservices patterns*. Manning Publications Company,.

[39] Diego Pereira da Rocha. 2018. Monólise: Uma técnica para decomposição de aplicações monolíticas em microsserviços.

[40] Maluane Rubert and Kleinner Farias. 2021. On the Effects of Continuous Delivery on Code Quality: A Case Study in Industry. *Computer Standards and Interfaces* (2021), 103588.

[41] Chaitanya Rudrabhatla. 2020. Impacts of Decomposition Techniques on Performance and Latency of Microservices. *International Journal of Advanced Computer Science and Applications* 11 (01 2020). https://doi.org/10.14569/IJACSA.2020.0110803

[42] Dag IK Sjoberg, Bente Anda, Erik Arisholm, Tore Dyba, Magne Jorgensen, Amela Karahasanovic, Espen Frimann Koren, and Marek Vokác. 2002. Conducting realistic experiments in software engineering. In *Proceedings international symposium on empirical software engineering*. IEEE, 17–26.

[43] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. 2020. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences* 10, 17 (2020), 5797.

[44] Sheetal Thakare, Savita Chavan, and PM Chawan. 2012. Software Testing Strategies and Techniques. *International Journal of Emerging Technology and Advanced Engineering* 2 (2012), 980–986.

[45] Karl Ulrich. 1994. Fundamentals of product modularity. In *Management of Design*. Springer, 219–231.

[46] Roger Gonçalves Urdangarin, Kleinner Farias, and Jorge Barbosa. 2021. Mon4Aware: A multi-objective and context-aware approach to decompose monolithic applications. In *XVII Brazilian Symposium on Information Systems*. 1–9.

[47] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 583–590.

[48] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.