UNIVERSIDADE FEEVALE

LEONARDO MOISÉS LEAL

IMPACTO DO NÍVEL DE GRANULARIDADE EM UM CÓDIGO COM ARQUITETURA DE MICRO SERVIÇOS

Novo Hamburgo

2022

LEONARDO MOISÉS LEAL

IMPACTO DO NÍVEL DE GRANULARIDADE EM UM CÓDIGO COM ARQUITETURA DE MICRO SERVIÇOS

Trabalho de Conclusão de Curso, apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Feevale.

Orientador: Profa. Dra. Adriana Neves dos Reis

Novo Hamburgo

2022

LEONARDO MOISÉS LEAL

Trabalho de conclusão do Curso Ciência da Computação, com título *Impacto do nível de granularidade em um código com arquitetura de micro serviços*, submetido ao corpo docente da Universidade Feevale, como requisito necessário para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovado por:

_____

Profa. Dra. Adriana Neves dos Reis

_____

Prof. Me. Edvar Bergmann Araujo

_____

Prof. Dr. Ricardo Ferreira de Oliveira

Novo Hamburgo, 2022

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

Primeiramente a Deus, pois sem ele nada disso seria possível.

A minha mãe Ursula e a minha avó Talya, que sempre me incentivaram a nunca desistir e acreditar, até quando eu mesmo não acreditava.

A minha irmã Bárbara, que me auxiliou na elaboração deste trabalho, com opiniões e críticas, além de ser meu maior exemplo.

Ao meu pai, Laerte, que mesmo não estando mais aqui, me proporcionou tudo que era necessário para chegar neste momento.

De modo muito especial, à minha orientadora Professora Doutora Adriana, pela sua dedicação, motivação e auxílio durante todo o desenvolvimento deste trabalho.

A todos que, de alguma forma, contribuíram para o desenvolvimento deste trabalho.

Muito obrigado!

RESUMO

Sendo *microservices* uma arquitetura de software e DDD uma proposta de modelagem de código, muitas vezes não se avaliam os impactos de suas utilizações dentro de uma aplicação, tornando a decisão de qual o nível ideal da granularidade dos serviços algo não definido. Com o auxílio da Engenharia de Software Baseada em Evidências (ESBE) como metodologia de estudo, são recolhidas evidências, para a elaboração de um modelo de recomendação que auxilie na definição granular de micro serviços. O modelo possui como passo inicial a identificação dos micro serviços em um formato que torne visível todo o contexto em que estão inseridos. O segundo passo visa limitar a existência de até três entidades de responsabilidades únicas dentro de um mesmo micro serviço. O objetivo do terceiro passo é a construção de micro serviços que se comunicam com no máximo três outros serviços. O quarto e último passo busca decompor serviços que constituem abstrações de termos utilizados pelos conhecedores do negócio. Para validação, o modelo é aplicado a partir de um projeto inicial. O modelo de recomendação contribui na construção de uma arquitetura com micro serviços que possuam contextos e reponsabilidades bem definidos, proporcionado qualidade e melhoria de desempenho. Tais benefícios são comprovados através de testes de carga dos micro serviços, que simulam usuários na utilização do sistema.

Palavras-chave: Granularidade de código. arquitetura de micro serviços. *Domain-Driven Design*.

ABSTRACT

Since microservices are a software architecture and DDD a code modeling proposal, the impacts of their uses within an application are often not evaluated, making the decision of what is the ideal level of granularity of services something undefined. With the help of Evidence-Based Software Engineering (ESBE) as a study methodology, evidence is collected for the elaboration of a recommendation model that assists in the granular definition of micro services. The model has as its initial step the identification of micro services in a format that makes visible the entire context in which they are inserted. The second step aims to limit the existence of up to three entities of single responsibilities within the same micro service. The objective of the third step is the construction of micro services that communicate with a maximum of three other services. The fourth and final step seeks to decompose services that constitute abstractions of terms used by business connoisseurs. For validation, the model is applied from an initial project. The recommendation model contributes to the construction of an architecture with micro services that have well-defined contexts and responsibilities, providing quality and performance improvement. Such benefits are proven through load tests of micro services, which simulate users in the use of the system.

Keywords: Code granularity. microservice architecture. Domain-Driven Design.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE SIGLAS

AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
BC	*Bounded Context*
BDD	*Behavior Driven Development*
BFF	*Backend-For-Frontend*
CRUD	*Create, Read, Update e Delete*
DDD	*Domain-Driven Design*
DevOps	*Development Operations*
DM	*Domain Model*
DTO	*Data Transfer Object*
ESBE	Engenharia de Software Baseada em Evidências
HTTP	*Hypertext Transfer Protocol*
IaaS	Infraestrutura como Serviço
ICSAE	*International Conference on Sustainable Agriculture and Environment*
IDE	*Integrated Development Environment*
MSA	*Microservice Architecture*
OWL	*Web Ontology Language*
PaaS	Plataforma como Serviço
SaaS	Software como Serviço
SOA	*Service-Oriented Architecture*
UML	*Unified Modeling Language*
URI	*Uniform Resource Identifier*
W3C	*World Wide Web Consortium*

SUMÁRIO

1 INTRODUÇÃO

Profundas mudanças ocorrem constantemente, tornando crescentes e cada vez mais aceleradas as inovações tecnológicas, colocando à disposição dos profissionais e usuários os mais diversos tipos de tecnologia (BARRA *et al.*, 2006). Com o avanço no desenvolvimento de software, mudanças significativas ocorrem e com elas novas tecnologias são construídas. Com isso, as empresas de desenvolvimento buscam migrar os seus produtos e serviços utilizando novas tecnologias como, por exemplo, novas linguagens de programação, novos paradigmas, modelos de processos com novas perspectivas, *designs* de código e conceitos de arquitetura de software.

Arquitetura de software trata-se de uma representação que auxilia na compreensão de como um sistema irá se comportar (CMU, 2015), servindo como modelo para o sistema e projeto que está sendo construído. Grande parte das tomadas de decisões visam uma melhor qualidade do software, em que se espera que requisitos como extensibilidade, capacidade de manutenção, reutilização de código, desempenho, escalabilidade, usabilidade e confiabilidade dos dados sejam atendidos (ENGHOLM JR., 2010).

Buscando esses requisitos para a melhoria no desenvolvimento de software, muito se discute qual o tamanho ideal para a granularidade dos códigos, ou seja, o quão detalhado é seu código. Um grande desafio na hora de projetar um aplicativo é determinar a granularidade apropriada, o que geralmente é feito por arquitetos de software utilizando seu julgamento (VERA-RIVERA *et al.*, 2020). Cada arquitetura possui uma diretriz que deve ser observada pelo arquiteto e seguida pela equipe de desenvolvimento. Neste sentido, novas propostas surgem e nem sempre é fácil tomar a decisão de qual estratégia seguir, pois existem vários padrões que podem ser adotados, tornando difícil a decisão de qual se adequa melhor ao propósito da aplicação.

Uma arquitetura chamada de *Microservices*, ou micro serviços, surgiu durante a *International Conference on Sustainable Agriculture and Environment* (ICSAE) em maio de 2011 e vem sendo bastante difundida desde então na comunidade de arquitetos, engenheiros de softwares e desenvolvedores. Micro serviços são partes, específicas e independentes, de uma aplicação maior e possuem responsabilidade única (AMARAL; CARVALHO, 2017).

O uso desta abordagem precisa ser bem avaliado pelo arquiteto no início do projeto, pois não é a solução para todos os cenários devido a sua complexidade de desenvolvimento. Mesmo possuindo uma boa aceitação dentro da comunidade, ainda é um desafio chegar a um consenso de como definir o tamanho ideal de cada micro serviço. Alcançar um nível ideal de granularidade é, portanto, de grande interesse para encontrar alternativas que o tornem menos complexos, que tenham baixa latência e menor número de transações, ou seja, que não tenha atrasos e com o menor número possível de comunicação entre serviços.

É comum que os desenvolvedores usem a própria funcionalidade para definir o escopo que determina o tamanho de um micro serviço. Porém, uma alternativa conhecida é na utilização dos conceitos empregados ao adotar o *Domain-Driven Design* (DDD). DDD é uma abordagem que agrupa técnicas e conceitos onde o foco está no domínio e na lógica para criar um *Domain Model* (modelo de domínio). Dentro dos conceitos que compõem o DDD estão os *Bounded Context* (contexto delimitado ou BC) que visam facilitar e dar coerência no desenvolvimento (EVANS, 2019). Com isso surgiram algumas propostas de criar micro serviços com a modelagem DDD e seus BCs.

Porém, a utilização deste método não consegue definir o melhor nível de granularidade, e o impacto desta decisão ainda não está bem claro, deixando essa lacuna aberta dentro da comunidade de desenvolvedores. Assim, assume-se como questão de pesquisa: “Como auxiliar na tomada de decisão de todos os envolvidos em um projeto de software, quanto à granularidade ideal de um micro serviço, quando aplicados conceitos do DDD, buscando melhorar a qualidade do software?”.

Para atender a questão de pesquisa, neste trabalho são apresentados resultados quanto a *performance* de um projeto de software com diferentes níveis de granularidade quando implementados em um arquitetura de micro serviços e aplicados conceitos do DDD. A análise dos resultados não visa determinar com exatidão qual nível é o melhor para o desenvolvimento de ponta-a-ponta. Visto que dentro de um único sistema com micro serviços, podem ser empregados múltiplos níveis para que cada funcionalidade tenha seu melhor desempenho possível. Vale ressaltar que o foco é a utilização da arquitetura de micro serviços e DDD, não sendo consideradas outras arquiteturas ou abordagens de software.

1.1 OBJETIVOS

1.1.1 Objetivo geral

O objetivo deste trabalho é elaborar, utilizando uma abordagem de arquitetura de micro serviços e *Domain-Driven Design*, um modelo de definição de granularidade de código, com o propósito de auxiliar na tomada de decisão com as práticas que podem ser adotadas. A definição do modelo baseia-se nas evidências encontradas dentro da comunidade de software através da bibliografia e os resultados obtidos em experimentos realizados com codificação.

1.1.2 Objetivos específicos

- Pesquisar trabalhos correlacionados entre micro serviços e DDD;
- Mapear os princípios de arquitetura de software relacionados à granularidade de código;
- Descrever o que foi encontrado de relevante na literatura;
- Realizar experimentos em código;
- Descrever os resultados encontrados através do código implementado;
- Propor um modelo consolidando recomendações para definição de granularidade em código.

1.2 ESTRUTURA DO TRABALHO

Inicialmente, no capítulo 2, é apresentada a metodologia de pesquisa que foi optada. No capítulo 3 é realizada uma breve explicação dos assuntos bases. Em seguida, no capítulo 4, é exposto o emprego de micro serviços com a abordagem DDD. No capítulo 5, é detalhado o processo de desenvolvimento de um caso de uso. Na sequência, no capítulo 6, é apresentado o modelo inicial e são descritos os passos para a alteração granular. No capítulo 7, o modelo é aplicado a partir de um projeto inicial. No capítulo 8, é detalhado o processo de experimentação, com testes de carga. Os resultados são discutidos e o modelo de recomendação é atualizado no capítulo 9. Por fim, as limitações e conclusões do trabalho são apresentadas.

2 METODOLOGIA

A questão de pesquisa apontada na introdução deste trabalho deixa explícita a necessidade de buscas por evidências e experimentos que proporcionem uma resposta. Profissionais de software e gerentes que procuram meios para melhorar a qualidade de seus processos de desenvolvimento de software, muitas vezes adotam novas tecnologias sem evidências o suficientes de que irão ser eficazes, enquanto outras tecnologias são ignoradas, apesar de evidências de que provavelmente serão mais úteis (DYBÅ; KITCHENHAM; JØRGENSEN, 2005). Neste contexto, se faz necessária uma análise da literatura para que as informações possam ser realmente avaliadas e que gerem resultados válidos.

Evidenciar algo é quando são reunidas provas suficientes que o torne incontestável, que não deixa dúvidas e que qualquer pessoa também pode verificar o mesmo. Exemplos disso são quando um detetive reúne pistas que viram evidências a partir de seu conhecimento ou quando um juiz criminal leva em consideração os depoimentos de um número de vítimas para obter o que realmente ocorreu no crime (KITCHENHAM; BUDGEN; BRERETON, 2016). As abordagens metodológicas baseadas em evidências já foram utilizadas em diferentes contextos, tais como a medicina, psiquiatria, enfermagem, política social e educação (MAFRA; TRAVASSOS, 2006). A confiança sobre um determinado conhecimento aumenta com a existência de diferentes fontes de evidências (KITCHENHAM; BUDGEN; BRERETON, 2016).

Por volta do fim dos anos 80 e início dos anos 90 foram realizadas pesquisas no meio da medicina que comprovaram que análises sistemáticas, como metodologia de pesquisa, estavam custando vidas. Como forma de melhorar isso começaram a surgir trabalhos adotando o paradigma baseado em evidência que mesmo com seus pontos críticos é considerado um sucesso para a instrução e treinamentos na área (DYBÅ; KITCHENHAM; JØRGENSEN, 2005). É com esse intuito, unindo as melhores evidências atuais de pesquisa com experiência prática, que surgiu a Engenharia de Software Baseada em Evidências (ESBE). Essa abordagem visa responder a uma pergunta principal: O que funciona, para quem, onde, quando e por que (LAUX, 2019). Semelhante a Medicina Baseada em Evidências, a ESBE deve aplicar na prática o que for encontrado na literatura atual para fechar as lacunas ou encontrar outras linhas de pesquisa (MISIRLI; BENER,

2014). A ESBE também pode ser considerada como um mecanismo para suportar e melhorar as decisões relacionadas à adoção de tecnologias (LAUX, 2019).

Dybå, Kitchenham e Jørgensen (2005) propõem que na aplicação da ESBE deve se seguir cinco passos estruturados. O primeiro passo se faz necessário propor uma questão que precisa ser respondida com as evidências que serão buscadas na literatura. Essa questão deve seguir um formato que proporcione detalhes suficientes sobre o contexto em que a pesquisa está inserida, deixando claro o que será agregado de valor, a quem é de interesse e o impacto que pode ser gerado. Um exemplo de pergunta de pesquisa seria: "*Pair programming* leva a uma melhor qualidade do código-fonte quando praticado por desenvolvedores de software profissionais?", sendo "*pair programming*" o contexto de pesquisa, "desenvolvedores de software profissionais" é a possível parte interessada e "qualidade de código-fonte" é o que será agregado de valor e o possível impacto (DYBÅ; KITCHENHAM; JØRGENSEN, 2005).

O segundo passo tem como foco a busca por evidências por meio de revisão da literatura em artigos científicos, livros e revistas que possam chegar a uma resposta da pergunta formulada no passo anterior (DYBÅ; KITCHENHAM; JØRGENSEN, 2005). Para contemplar esse passo, diferentes fontes de dados podem ser utilizadas (artigos científicos, entrevistas com usuários ou entrevistas com especialistas), bem como estratégias distintas podem ser aplicadas para conquistar esses dados, como revisão da literatura, revisão sistemática e entrevistas (LAUX, 2019).

No terceiro passo todo o material reunido no segundo passo deve ser colocado a prova. As evidências precisam ser criticadas através de experimentos e testes para que possam ser validadas ou não. Dybå, Bergensen e Sjøberg (2016) apontam que as evidências na engenharia de software costumam ser sensíveis ao contexto da organização, por isso é recomendável realizar experimentos para validar as evidências encontradas. Embora existam centenas de publicações sobre vários tópicos de engenharia de software, os profissionais têm poucas evidências sobre usabilidade, limitações, riscos e benefícios das técnicas propostas em diferentes configurações nessas publicações (MISIRLI; BENER, 2014).

O quarto passo é o momento em que conexões entre as evidências, coletadas no passo dois e validadas no passo três, são feitas com a pergunta de pesquisa elaborada no passo um. Essas conexões têm como objetivo iniciar uma

resposta válida para o contexto de estudo. O uso do conhecimento adquirido exige a aplicação ou a adaptação das evidências a situações específicas na prática (DYBÅ; KITCHENHAM; JØRGENSEN, 2005). Nesse passo é muito importante que se tenha entendimento e convicção das informações, pois a partir daqui poderão ser utilizadas pela indústria para a tomada de decisão.

No quinto e último passo é construída uma avaliação sobre tudo o que foi feito até então, dando relevância ao que responde à pergunta de pesquisa e realizando melhorias em pontos que são passíveis de melhorias. Precisa considerar o quão bem foi executada cada etapa do ESBE e como pode melhorar o uso, ou seja, o quanto as evidências coletadas estão integradas com a experiência prática, os resultados do cliente e o conhecimento das circunstâncias específicas (DYBÅ; KITCHENHAM; JØRGENSEN, 2005). Abaixo segue o Quadro 1, resumindo os cinco passos.

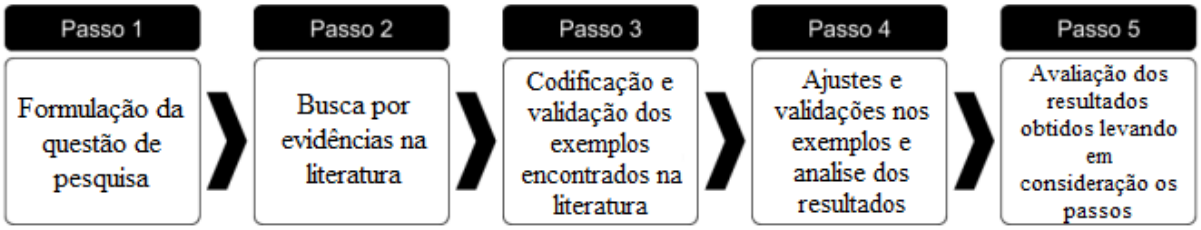
Quadro 1 – Passos da ESBE

Passo	Objetivo
1	Formular uma questão a partir da necessidade de informação sobre um determinado assunto.
2	Buscar as melhores evidências com as quais seja possível responder à questão de pesquisa.
3	Criticar as evidências quanto a validade, impacto e aplicabilidade.
4	Aplicar as evidências no contexto do estudo para geração de resultados.
5	Avaliar a efetividade e a eficiência dos passos 1 a 4 e buscar formas de melhorar na próxima execução do método.

Fonte: LAUX (2019)

De conhecimento dos passos necessários para a elaboração de uma pesquisa utilizando como metodologia a ESBE, e a necessidade de busca por evidências para responder à questão de pesquisa, considera-se esta metodologia como adequada para o presente estudo. É considerado todo referencial teórico obtido através de livros e trabalhos científicos junto com validações através da codificação de pequenos projetos de exemplo contendo as tecnologias pesquisadas. Esses projetos são simulados através de testes, aplicando possíveis melhorias, para que os resultados sejam apresentados. Na Figura 1 é apresentada a estrutura da metodologia aplicada ao contexto do trabalho.

Figura 1 – Execução da ESBE no contexto do trabalho



Fonte: Elaborado pelo autor

A questão de pesquisa, apontada como primeiro passo da ESBE, já foi descrita na introdução do trabalho. O passo seguinte da metodologia proposta está contido nos capítulos 3 e 4, onde todo o referencial teórico é apresentado. Os passos três e quatro são realizados na sequência do trabalho, onde o referencial teórico é codificado, testado e os resultados são apresentados.

A codificação é implementada utilizando a linguagem de programação C Sharp, a IDE Visual Studio da Microsoft e demais *frameworks* que visam auxiliar no desenvolvimento. Os testes são automatizados buscando o estresse máximo do sistema. Considera-se a apresentação desses resultados adequada ao problema, pois tem como objetivo expor exemplos de níveis de granularidade, gerando um modelo de definição, para auxiliar na decisão para o desenvolvimento de software. Os resultados, pertencentes ao passo 5, estão contidos no capítulo 9.

Com isso todos os passos necessários para a elaboração da metodologia escolhida são realizados. Os procedimentos da ESBE foram apresentados antes do referencial teórico para que fiquem claros todos os seus conceitos. A partir do próximo capítulo serão apresentados os assuntos base deste trabalho, extraídos da literatura para auxiliar na melhor definição de granularidade para um código com arquitetura de micro serviços e DDD.

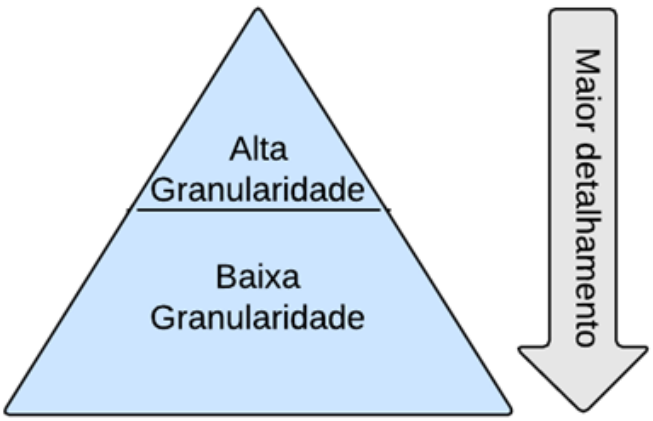
3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais pontos que envolvem este estudo: 1) Granularidade e; 2) Arquitetura de micro serviços, seus conceitos e comparações e; 3) *Domain-Driven Design*, seu objetivo e suas características. Todas as informações visam um embasamento sobre cada um dos assuntos para auxiliar no entendimento da problemática principal deste trabalho.

3.1 GRANULARIDADE

Granularidade deve ser vista como o nível de detalhamento que algo deve ter, conforme ilustrado na Figura 2. Granularidade baixa ocorre quando seu código é bem dividido e granularidade alta quando se tem o nível mais grosseiro, quando seu código realiza diversas funcionalidades sem divisões e atribuições. A definição desse nível ganha força quando discutida em conjunto a reutilização de código. Objetos pertinentes devem ser identificados para que sejam transformados em classes no nível correto de granularidade, para assim definir as interfaces das classes, as hierarquias de herança e estabelecer as relações-chaves entre eles (GAMMA *et al.*, 2000).

Figura 2 – Nível de granularidade



Fonte: Elias (2014)

Granularidade fina define que são necessários muitos “grãos”, enquanto na granularidade grossa poucos “grãos”, mas maiores, são criados. Ou seja, na granularidade fina, são criados serviços ou objetos com poucas operações, mas

essas operações estarão contidas em vários serviços ou objetos. Já na granularidade grossa ocorre o oposto, poucos serviços e objetos são construídos, mas cada um conterá uma gama muito maior de operações (ELIAS, 2014). Quando é construída uma granularidade fina, existem pequenos “blocos” de funcionalidades bem específicas e independentes. Isso torna o código mais facilmente atualizável, distribuído e gerenciável, mas deixa o serviço mais complexo. Por outro lado, a granularidade grossa torna o código mais enxuto, podendo deixar o serviço muitas vezes com uma responsabilidade exagerada, tornando o serviço pesado e causando um grande acoplamento com outras funcionalidades (AÉCE, 2009).

3.2 MICRO SERVIÇOS

Segundo Fowler e Lewis (2014), uma arquitetura de micro serviços (MSA) trata-se de uma abordagem de desenvolvimento de software que busca a construção de uma única aplicação composta por pequenos serviços independentes. Por serem independentes cada serviço possui seu processo bem definido ou um servidor dedicado, deve ser escalável e independente, projetado tendo como foco as regras de negócio, e os serviços devem se comunicar por meio de mecânicas leves, como por exemplo através de *Hypertext Transfer Protocol* (HTTP) ou *Application Programming Interface* (API). Um micro serviço em si não é um aplicativo, mas sim um bloco de construção de software.

Tanto no meio acadêmico como na comunidade de software não se tem definido o tamanho que um micro serviço deve possuir. Grande parte das equipes de desenvolvimento utilizam da experiência para definir a melhor granularidade para cada serviço. Porém, Newman (2021), Fowler e Lewis (2014) definiram alguns princípios que orientam engenheiros de software a projetarem seus micro serviços, são eles:

- **Modelo em torno de conceitos de negócio:** Os micro serviços precisam ser planejados tendo como parâmetros seus contextos bem definidos, visando sempre as regras de negócio.
- **Ocultar detalhes internos de implementação:** Para que a independência dos micro serviços seja respeitada, detalhes de implementação e banco de dados precisam ser omitidos ou ocultados, para isso a comunicação entre os

serviços deve ser feita por meio de alguma API. Esse princípio é de suma importância para mudanças futuras na arquitetura.

- **Descentralização:** Os micro serviços devem possuir o mais alto nível de coesão e com o mínimo de acoplamento, com isso qualquer alteração necessária não afetará outros serviços.
- **Implementação independente:** Esse princípio exige que qualquer micro serviço precisa ter a garantia que, quando colocado em produção, não afetará ou exigirá algo de qualquer outro micro serviço. Com isso, a velocidade de publicação de novos recursos será mais rápida e proporcionará mais autonomia para as equipes de desenvolvimento.
- **Cultura de automação:** A implantação de micro serviços acrescenta uma maior complexidade ao desenvolvimento devido ao maior número de serviços independentes dentro do sistema. Visto isso, é recomendável que as empresas adotem a cultura de automação, construindo processos e utilizando ferramentas que auxiliem na entrega contínua.
- **Isolar falhas:** Os micro serviços precisam ser projetados buscando a tolerância a falhas. Com isso, caso ocorra qualquer falha, toda a aplicação precisa responder a isto de uma maneira que o usuário final sinta o menor impacto possível.
- **Altamente observável:** A quebra de um sistema em serviços proporciona uma granularidade fina e gera uma maior complexidade para monitorar o comportamento de múltiplos serviços. Contudo, pode-se adotar técnicas que possibilitam monitorar automaticamente os micro serviços, como por exemplo o monitoramento semântico.

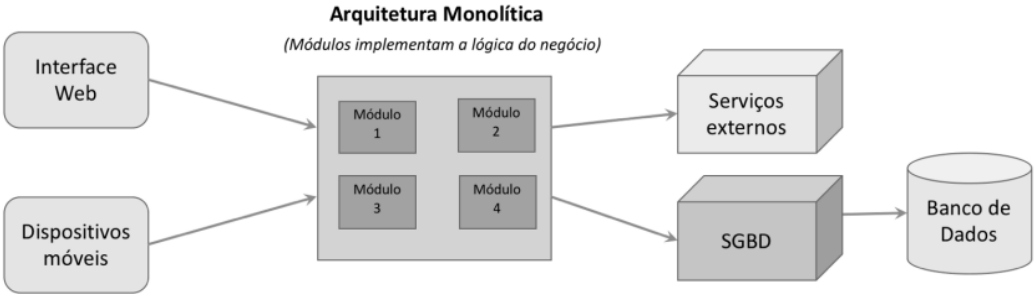
3.2.1 Arquitetura monolítica x Micro serviços

Para tornar claro o que é uma arquitetura de micro serviços, pode-se diferenciar ela em comparação com a já tradicional e difundida arquitetura monolítica. As características principais da arquitetura monolítica é possuir uma única aplicação responsável em realizar todas as tarefas do sistema e ser independente de outras aplicações.

Nesse modelo de arquitetura o seu desenvolvimento é projetado sem a necessidade de se preocupar com modularidade externa pois a aplicação não será

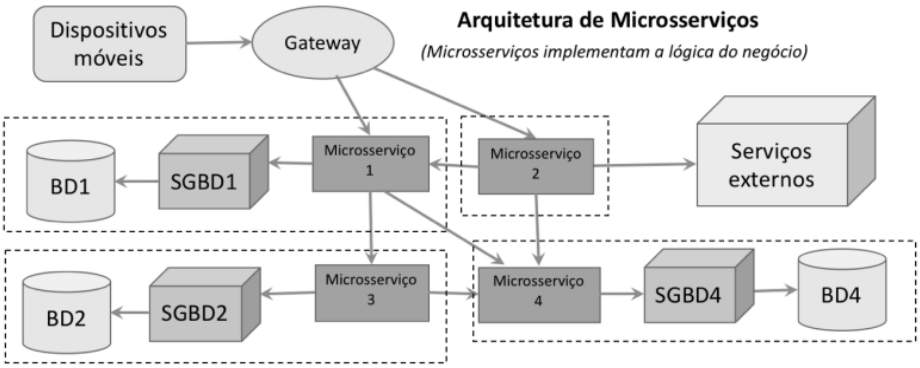
base para outras. Já sua modularidade interna é realizada através de ligações entre os módulos que formam a aplicação final. Quando é realizada a compilação do projeto todos os módulos são necessários para que não ocorram erros. Nas Figuras 3 e 4 são apresentados exemplos de arquitetura monolítica e de micro serviços respectivamente.

Figura 3 – Exemplo de arquitetura monolítica



Fonte: Adaptado de Richardson (2015)

Figura 4 – Exemplo de arquitetura de micro serviços



Fonte: Adaptado de Richardson (2015)

Em ambas as arquiteturas existem vantagens e desvantagens e a decisão em qual aplicar no projeto de uma aplicação deve ser muito bem avaliada para que a melhor opção seja adotada dentro do cenário ao qual se está incluído. As principais vantagens do uso de micro serviços estão relacionadas com a heterogeneidade tecnológica, resiliência, escalabilidade e facilidade de implantação (MOREIRA; BEDER, 2016).

A heterogeneidade tecnológica é considerada uma vantagem principalmente quando os micro serviços são desenvolvidos em linguagens diferentes, pois garante que se comuniquem com naturalidade, trabalhando como se utilizasse somente uma tecnologia. A experimentação de novas tecnologias também se torna mais favorável

e menos custosa, já que caso não funcione conforme o esperado, basta refazer um pequeno serviço específico de forma rápida e com bem menos custo do que seria em uma arquitetura monolítica.

A resiliência é a capacidade de uma aplicação identificar uma falha e responder a ela da melhor maneira possível. Visto isso, na arquitetura de micro serviços esta característica implica em um grande benefício, pois caso ocorra algum problema em algum serviço, apenas ele ficará indisponível, sem afetar outros serviços que continuarão funcionando normalmente. Já em uma arquitetura monolítica a aplicação inteira enfrentará o problema, ficando totalmente indisponível.

Na questão da escalabilidade, em micro serviços é aplicado apenas nos serviços específicos conforme a necessidade dos clientes. Já em aplicações monolíticas é necessário escalar todo o sistema, tornando menos flexível. A última característica listada é a facilidade de implantação. Isso é, a possibilidade de realizar o *deploy* somente das *features* necessárias, garantindo um maior controle do processo e o tornando mais fácil de ser realizado.

As principais desvantagens do uso de micro serviços são a complexidade de desenvolvimento, chamadas remotas e gerenciamento de múltiplos bancos de dados (MOREIRA; BEDER, 2016). Dentre as características apontadas, a que gera um maior receio quanto a utilização de micro serviços por uma equipe de desenvolvimento está no aumento da complexidade. O gerenciamento do processo de desenvolvimento é mais fácil em aplicações monolíticas pois tudo está integrado em um único projeto, se faz necessário apenas a importação de pacotes e classes. Já em projetos com micro serviços precisam ser adotadas medidas para que não ocorram problemas com versões desatualizadas ou ruído nas trocas de informações. Para isso uma boa gerência de dependências entre os módulos dos micro serviços é necessária, criando comunicação limpa e transparente entre a equipe envolvida no projeto.

Outra característica desfavorável na utilização de micro serviços está nas chamadas remotas, pois a comunicação externa entre os serviços exige um maior custo do que uma comunicação entre classes internas de uma aplicação monolítica. O gerenciamento de múltiplos bancos de dados também precisam ser analisados com atenção, pois exige procedimentos complexos e de alto custo para que uma série de tratamentos contra falhas dos serviços sejam implementados, possibilitando

um controle real nas transações e nos dados. O Quadro 2 apresenta vantagens e desvantagens da arquitetura de micro serviços.

Quadro 2 – Vantagens e desvantagens da arquitetura de micro serviços

Vantagens	Desvantagens
Heterogeneidade tecnológica	Complexidade de desenvolvimento
Resiliência	Chamadas remotas
Escalabilidade	Gerenciamento de múltiplos bancos de dados
Facilidade de implantação	

Fonte: Elaborado pelo autor

3.2.2 Escalabilidade

Um dos motivos para que uma equipe de desenvolvimento adote a arquitetura de micro serviços é para tornar a aplicação escalável. A escalabilidade, além dos benefícios que fornece em relação a desempenho, auxilia para garantir disponibilidade e tolerância a falhas (DRAGONI *et al.*, 2017). Para isso, alguns pontos devem ser observados na hora de desenvolver uma aplicação com micro serviços altamente escalável, são eles:

- **Distribuição:** Parece natural pelo fato de uma arquitetura de micro serviços já ser distribuída. O fato de que cada serviço seja projetado em um tamanho relativamente pequeno e independente, leva a distribuição ao mais alto degrau possível. Isso proporciona para a aplicação uma escalabilidade maior diante da distribuição, tornando-a mais eficaz do que em aplicações monolíticas.
- **Portabilidade:** A implementação de *containers* contendo todas as bibliotecas, banco de dados e demais ferramentas necessárias para funcionar independente de plataforma, possibilita a fácil replicação de serviços individuais em plataformas heterogêneas.
- **Elasticidade:** Arquitetura de micro serviços são elásticas devido a facilidade em replicar serviços individuais, permitindo assim escalar dinamicamente e de acordo com a necessidade, levando em consideração o tamanho da carga e o número de uso, em um período pré-determinado e conforme cada cliente.
- **Disponibilidade:** Tendo em vista o que foi dito sobre elasticidade, a arquitetura de micro serviços proporciona, com todos os seus serviços, uma

grande disponibilidade. Isso é possível pois cada serviço trabalha de forma autônoma, conforme a carga e uso.

- **Robustez:** Com a tolerância a falhas proporcionada pelo uso de *containers* e processos independentes, a arquitetura de micro serviços agrega vantagens para a robustez pois cada serviço é isolado de outros e, caso ocorra uma falha, os demais serviços não serão necessariamente afetados.

3.2.3 SOA x Micro serviços

Deve-se tomar cuidado para não confundir *Service-Oriented Architecture* (SOA) e micro serviços. Alguns autores entendem a arquitetura de micro serviços como uma abordagem mais específica de SOA ou como um padrão de SOA. Onde SOA emprega o uso de serviços com granularidade grossa e micro serviços indica o desenvolvimento de serviços com granularidade fina (RICHARDSON, 2016). Di Francesco, Malavolta e Lago (2017) sugerem que micro serviços surge de SOA, mas apresenta diferenças importantes. Micro serviços foca em características específicas de SOA, tais como, governança descentralizada entre serviços, definição de responsabilidades de pequenos serviços leves, gestão independente dos dados de cada serviço, utilização de automação de infraestrutura com entrega contínua e uso de práticas ágeis e DevOps para desenvolvimento. Outros apontam a arquitetura de micro serviços como sendo algo completamente novo.

As principais diferenças são quanto aos seus tamanhos e as características que cada uma delas tem como padrões definidos. Micro serviços são pequenos, independentes e descentralizados (NEWMAN, 2021). A forma de utilizar um micro serviço, como por exemplo o meio de comunicação entre os serviços, é algo mais importante do que simplesmente projetar o seu tamanho (CRAMON, 2017). Micro serviços foca em coreografia enquanto SOA foca em orquestração e coreografia (DI FRANCESCO; MALAVOLTA; LAGO, 2017).

3.3 DDD – *DOMAIN-DRIVEN DESIGN*

Desenvolver um software não é algo simples, dentre diversas decisões a serem tomadas no começo do projeto, a definição de um bom modelo de domínio para monitorar a complexidade é uma delas. Esse modelo deve abranger elementos

que são pertinentes e significativos para a criação do domínio, de forma que os desenvolvedores consigam aproveitar ao máximo (SANTOS, 2015). Buscando assegurar a agilidade nas técnicas para o desenvolvimento de um software surgiram vários métodos e técnicas criadas para auxiliar em diversos pontos dentro de um projeto de desenvolvimento de software. Dentre elas está o *Domain-Driven Design* (DDD), voltado para o domínio do problema (MATTOS; DOLL; ALMEIDA, 2010).

O termo DDD surgiu pela primeira vez em 2003 no livro "*Domain-Driven Design*: Atacando as Complexidades no Coração do Software" por Eric Evans (MATTOS; DOLL; ALMEIDA, 2010). Evans (2003) descreve que o DDD é uma maneira de pensar e um conjunto de princípios que ajudam os desenvolvedores a criar sistemas poderosos, robustos e sustentáveis. Portanto, não se trata de uma tecnologia ou metodologia, mas sim de uma abordagem de projeto de software, focado no domínio e na lógica do sistema. Agregado na engenharia de software, essa abordagem de projeto de software engloba técnicas e métodos que enfatizam a linguagem do cliente (DA SILVA; FILHO; DA SILVA, 2018). O DDD é executado como uma maneira de guiar métodos de desenvolvimento de modo ágil (EVANS, 2003).

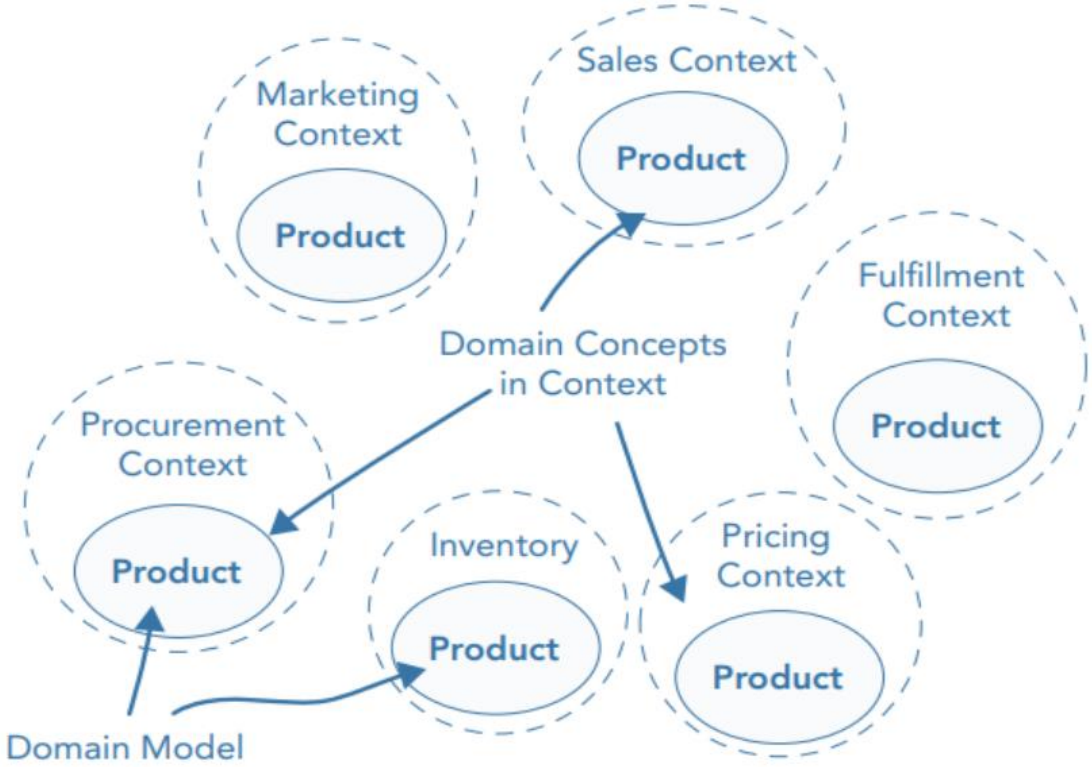
3.3.1 Modelo de domínio (*Domain Model*)

O DDD é focado em auxiliar na elaboração do domínio, ou seja, ele não leva em consideração as demais camadas necessárias para a construção de uma aplicação. Para isso, o modelo de domínio é construído mantendo uma relação próxima, de interação e de constante aprendizado entre os entendedores do domínio e todos os envolvidos no desenvolvimento da aplicação. Na engenharia de software, um modelo de domínio é conhecido por uma representação conceitual contendo o conhecimento e comportamentos que são característicos da aplicação.

Em *Domain-Driven Design*, o modelo de domínio é construído utilizando uma linguagem ubíqua, servindo para a disseminação do conhecimento do domínio e na elaboração do código, deixando bem claro e conciso com relação ao domínio. A elaboração da modelagem estratégica do negócio, a identificação do domínio principal, os subdomínios e possíveis domínios genéricos, facilitam na definição das entidades, objetos de valor e agregados.

Quando uma aplicação começa a ganhar maiores proporções surgem subdomínios integrados a um domínio principal. Os subdomínios, além de aumentar a complexidade, podem gerar termos iguais, mas que possuem um sentido diferente, pois cada modelo é constituído para representar um ponto distinto do problema (MILLETT; TUNE, 2015). Um termo, dependendo do seu contexto, pode ser empregado de formas diferentes, causando impactos na lógica empregada no desenvolvimento do projeto, conforme é mostrado na Figura 5.

Figura 5 – Modelos em diversos contextos



Fonte: Millet e Tune (2015, p. 81)

3.3.2 Linguagem Ubíqua

Uma das principais diferenças do DDD para outras filosofias de *design* de projeto é a ênfase dada à modelagem, onde o entendimento dos requisitos e a implementação da aplicação não são consideradas como atividades diferentes, mas sim que trabalham juntas constantemente, tornando possível a construção de um modelo refletido no código (MACEDO, 2009). Os domínios geralmente são compreendidos como substantivos apontados pelas partes interessadas. A carga de

informações pode ser grande, porém, o modelo serve justamente como dispositivo redutor dessa sobrecarga de informação (SANTOS, 2015).

Para a construção do modelo e para que o software siga otimizado dentro dos padrões do DDD é necessário que uma Linguagem Ubíqua seja previamente estipulada. Essa linguagem deve ser definida de acordo com as conversas entre o cliente e a equipe de desenvolvimento, sendo compreendida por todos, sem ambiguidades (CUKIER, 2010). O cliente possui especialistas portadores do entendimento do negócio, porém com um vocabulário técnico limitado para o desenvolvimento de soluções tecnológicas, sendo esse vocabulário compreendido pela equipe de desenvolvimento. Com isso, é necessária a utilização de uma linguagem comum entre todos os envolvidos com o projeto (SANTOS, 2015). Evans (2003) define a Linguagem Ubíqua sendo uma linguagem comum, com termos claramente definidos, que integram o domínio do negócio e que são aplicados por todos os envolvidos no processo de desenvolvimento de software.

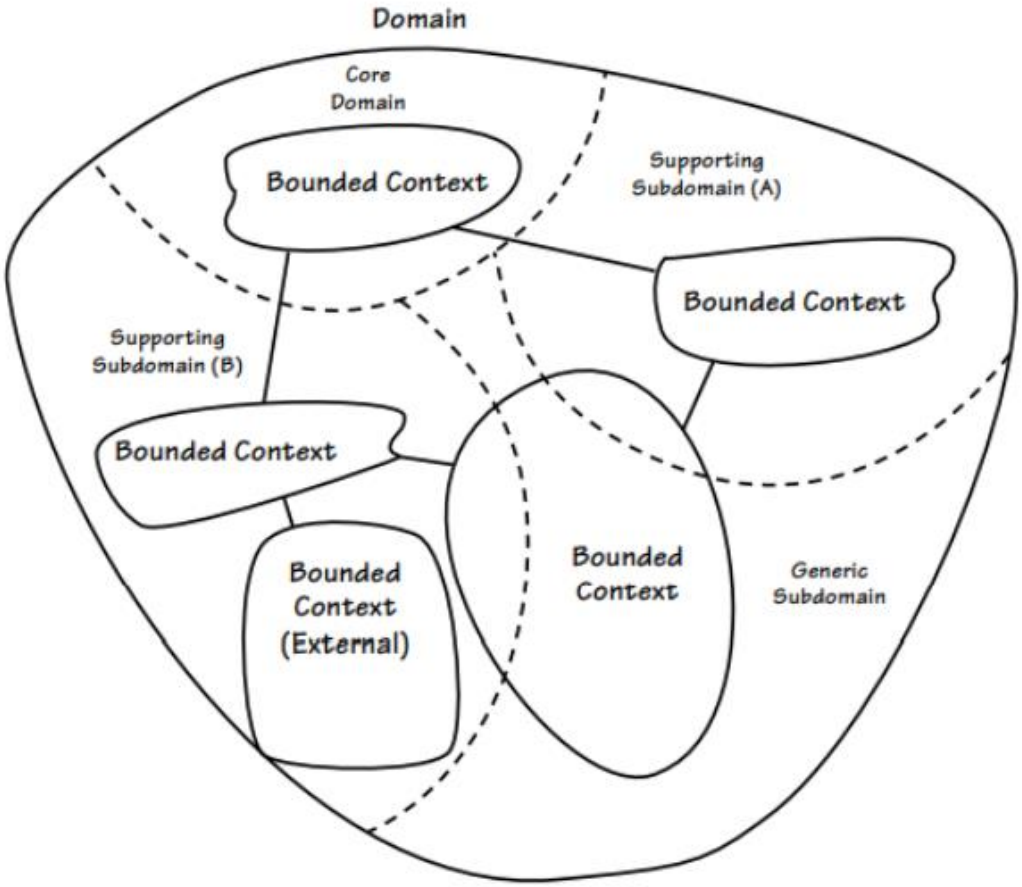
3.3.3 Contextos delimitados (*Bounded Context*)

Um dos conceitos mais importantes do DDD são os *bounded context*, ou contextos delimitados. Possuem a função de delimitar ou agrupar domínios de um determinado negócio. As entidades de um contexto devem ser criadas respeitando as características do contexto ao qual elas serão integradas. Esse contexto deve agrupar um conjunto de comportamentos expostos pelos conhecedores da regra de negócio. Evans (2020) afirma que, ao delimitar um contexto, todos os integrantes da equipe de desenvolvimento do projeto passam a possuir um entendimento claro e compartilhado do que deve compor um determinado contexto, se tornando consistente para que possa se comunicar com outros contextos.

Cada contexto específico deve possuir o seu modelo de domínio consistente e significativo para o entendimento da complexidade, definido pela Linguagem Ubíqua aderida pelos especialistas do domínio e a equipe de desenvolvimento (MILLET; TUNE, 2015). Assim, os contextos contidos no domínio principal e seus subdomínios podem ser delimitados, conforme mostrado na Figura 6. O desenvolvimento deve sempre manter o modelo logicamente unificado, não se preocupando com contextos fora do limite do escopo identificado. Assim, para

entender onde o modelo se aplica é necessário examinar o projeto como ele é, e não como deveria ser no mundo ideal (EVANS, 2020).

Figura 6 – Conceito abstrato de um modelo de domínio contendo subdomínios e contextos delimitados



Fonte: Vernon (2016, p. 50)

3.3.4 Mapa de contexto (*Context Map*)

O mapa de contexto é a ferramenta usada para tornar explícitos os limites entre os contextos delimitados. Pode ser construído através de um documento, uma imagem, um rascunho anexado na parede da equipe de desenvolvimento ou qualquer outro método que torne mais fácil o entendimento dos contextos da aplicação. A integridade do modelo conforme o mapa de contexto é de extrema importância para garantir que o domínio seja bem compreendido e modelado (EVANS, 2003). É necessário possuir muito bem fundamentado, em união aos especialistas de domínio, toda a Linguagem Ubíqua, para que o mapa de contexto sirva como viga de sustentação de todo o projeto de desenvolvimento.

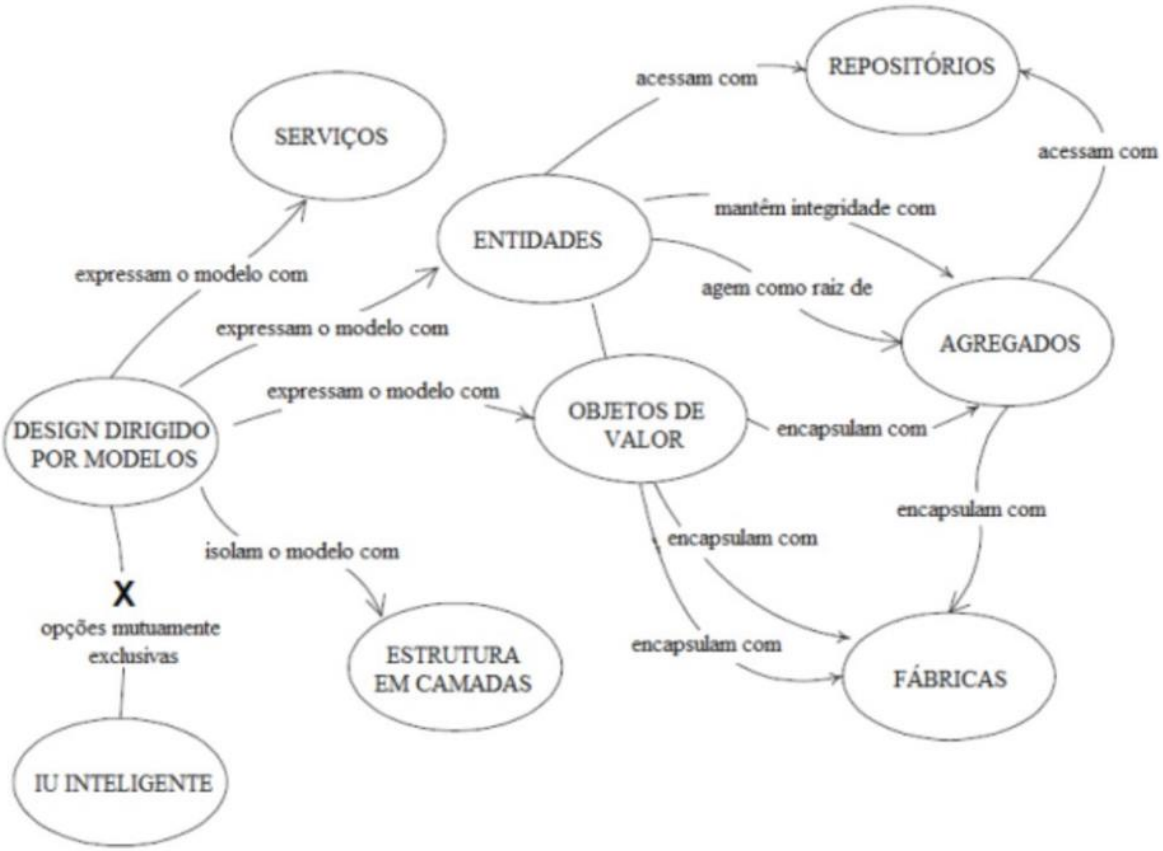
3.3.5 Padrões de *Design*

Na Figura 7 são demonstrados os padrões que o DDD possui para guiar o desenvolvimento na construção do *design* baseado em modelos. Os padrões são:

- **Entidades:** Classes de objetos que carecem de uma identidade, a definição desse objeto deve ser simples, estabelecida pela sua identidade e não seus atributos (SANTOS, 2015). A identidade não é estabelecida pela união dos atributos ou pelo estado do objeto, mas sim por uma identidade, sendo que identidades mal acordadas podem gerar uma incongruência de dados (MACEDO, 2009).
- **Objetos de valor:** Ao contrário das entidades, os objetos de valor não possuem uma identidade ou descreve alguma característica, sendo que na maioria das vezes são imutáveis e possuem um curto ciclo de vida (MATTOS; DOLL; ALMEIDA, 2010).
- **Serviços:** São as classes que contém lógica de negócio, porém não pertencem a uma entidade ou objeto de valor, mas devem possuir algum significado para o domínio (MACEDO, 2009).
- **Módulos:** No decorrer da evolução do modelo, cresce o número de conceitos com o qual o desenvolvedor irá lidar, assim como o número de classes e suas associações, gerando assim a inevitável necessidade de agrupar as classes em módulos, que nada mais são do que blocos lógicos onde o conjunto de classes da aplicação são particionados (MACEDO, 2009).
- **Agregações:** Este bloco exerce a manutenção da integridade do modelo. Podem ser definidos como conjuntos de entidades ou objetos de valor que são agrupados em uma única classe (MATTOS; DOLL; ALMEIDA, 2010).
- **Fábricas:** São empregados para agrupar a informações necessárias para a elaboração de objetos complexos ou agregados, gerando uma interface que reproduz os desejos do cliente e uma visão subjetiva do objeto a ser construído (MATTOS; DOLL; ALMEIDA, 2010).
- **Repositórios:** Têm como objetivo agrupar toda a lógica necessária para atingir a referência de um objeto, onde estes não necessitam lidar com a infraestrutura para apanhar informações essenciais de outros objetos de domínio (SANTOS, 2015). São classes responsáveis por controlar o ciclo de

vida de outros objetos como entidades, objetos de valor ou agregados. Concentram operações de criação, alteração e remoção de objetos (MACEDO, 2009).

Figura 7 – Mapa de navegação da linguagem do DDD



Fonte: Santos (2015)

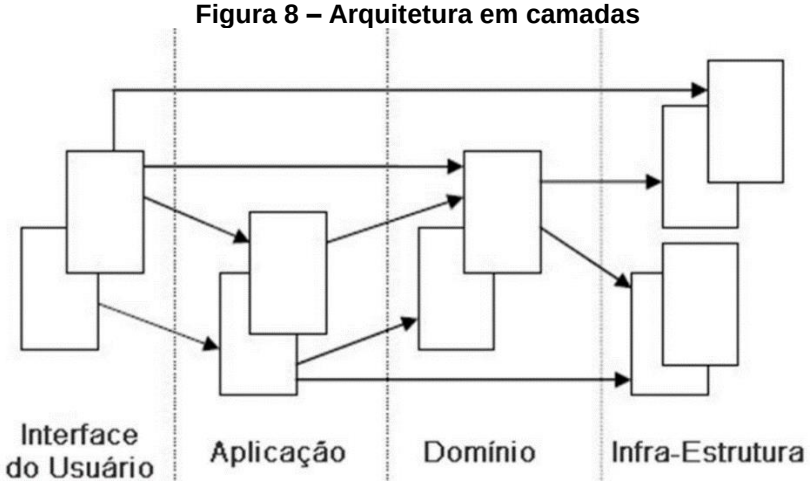
3.3.6 Camadas do DDD

Evans (2003) afirma que o DDD é focado na criação do domínio do software, porém é recomendada, conforme Figura 8, a utilização de uma arquitetura contendo quatro camadas essenciais, são elas:

- **Camada de interface:** Responsável por apresentar informações e interpretar comandos do usuário (MATTOS; DOLL; ALMEIDA, 2010).
- **Camada de aplicação:** Onde as atividades do aplicativo são coordenadas, essa camada não deve conter a lógica do negócio, deve se restringir apenas ao estado do progresso de uma tarefa de aplicação (MATTOS; DOLL; ALMEIDA, 2010).

- **Camada de domínio:** Onde estão localizadas todas as regras de negócio. O centro do DDD se encontra nela, deve se manter mais isolada possível de todas as outras camadas (SANTOS, 2015).
- **Camada de infraestrutura:** Age como uma biblioteca para as outras camadas, fornecendo a comunicação entre elas e realizando a persistência de dados (MATTOS; DOLL; ALMEIDA, 2010).

Figura 8 – Arquitetura em camadas



Fonte: Avram e Marinescu (2007, p. 29)

O foco do DDD está na camada de domínio da arquitetura, que é responsável pelos conceitos do domínio, dos casos de uso e das regras de negócio. O estado dos objetos é armazenado nesta camada, porém a persistência é realizada pela camada de infraestrutura (CUKIER, 2010). A utilização das práticas sugeridas pelo DDD afeta diretamente a extensibilidade, usabilidade, testes e manutenção de um sistema. Cada uma das camadas deve se limitar a exercer apenas as funções as quais são de sua responsabilidade (MATTOS; DOLL; ALMEIDA, 2010).

3.4 CONSIDERAÇÕES DO CAPÍTULO

Tendo visto um apanhado básico dos principais conceitos e tecnologias que serão utilizados neste trabalho, são perceptíveis características que apontam uma boa integração dentro de um projeto de desenvolvimento de software. Arquitetura de micro serviços tem em sua essência dispor de uma granularidade fina e o conceito de contextos delimitados do DDD pode auxiliar na decisão do quão refinado pode ser um serviço.

No capítulo 4 são apresentados os conceitos encontrados na literatura quanto a aplicabilidade dessas tecnologias em conjunto, buscando indícios de como estimar a granularidade que melhor se adeque dentro do contexto.

4 GRANULARIDADE EM MICRO SERVIÇOS

Definir tamanhos adequados para micro serviços é importante porque sua granularidade influencia na qualidade do serviço. A questão a ser respondida é estabelecer onde os limites dos componentes devem estar (FOWLER; LEWIS, 2014). Wolff (2016) afirma que o tamanho de um micro serviço deve ser pequeno e focado em alcançar uma funcionalidade. Embora na arquitetura de micro serviços o *design* de serviços pequenos seja incentivado, serviços muito refinados geram uma quantidade ineficientemente alta de interações necessárias para atender a um único propósito (NEWMAN, 2021).

A escolha da granularidade deve ser orientada pelo equilíbrio entre os custos de garantia de qualidade e o custo de implementação (GOUIGOUX; TAMZALIT, 2017). A principal dificuldade na busca da granularidade correta é que não há um estado da arte sobre esse assunto (NEWMAN, 2021). Não existe uma definição comumente aceita do tamanho de um micro serviço (FOWLER; LEWIS, 2014). Josélyne *et al.* (2018) apontam fatores como rota de mensagens, mecanismos síncronos ou assíncronos, o método de publicação e conexão a eventos, escolha de protocolo de *gateway* e API, e a maneira de integrar com o banco de dados, capazes de influenciar no tamanho de um micro serviço. Algumas estratégias para definir o tamanho apropriado de um micro serviço tem sido abordadas por profissionais, tais como:

- **Linha de código:** Alguns membros da comunidade de desenvolvimento de software apontam que o tamanho de um micro serviço é dado pelo número de linhas de código implementadas. Para isso é recomendado que um serviço ideal deva possuir entre 10 e 100 linhas de código (SCHERMANN; CITO; LEITNER, 2015). O objetivo desta abordagem é monitorar constantemente o limite de linhas de código dos serviços implementados, para que o torne mais flexível, de mais fácil dimensionamento e que gere menores preocupações quando forem necessárias alterações ou até mesmo remoções de micro serviços. No entanto, essa estratégia se torna inviável para micro serviços, pois os serviços podem ser construídos com diferentes tecnologias, que se diferenciam quanto a linha de código.
- **Unidade de implantação:** Nessa estratégia, cada micro serviço é visto como uma unidade de desenvolvimento de implementação. Os serviços são

exibidos na nuvem através de infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e software como serviço (SaaS), para implantar pequenos serviços que em conjunto formam uma grande aplicação. Com isso, os pequenos serviços podem ser testados, dimensionados, operados e atualizados de forma independente uns dos outros. Os serviços são particionados conforme são registrados e geralmente são organizados no processo de implementação e atualização (VILLAMIZAR *et al.*, 2016).

- **Capacidade de negócios:** Essa estratégia constrói métricas para definir como uma organização realiza com sucesso as atividades necessárias para o desenvolvimento do sistema. Sua intenção é combinar dados que possuem relações, como funcionalidade, ao invés de utilizar coleções de entidades de dados que expõem métodos no estilo CRUD (*Create, Read, Update and Delete*), utilizando uma metodologia de desenvolvimento ágil. Assim, a capacidade de negócios busca garantir o entendimento sobre o impacto da definição de um serviço na arquitetura da aplicação antes do desenvolvimento. Porém, o entendimento de qual nível granular um recurso de negócio deve possuir ainda é um desafio deste método para os desenvolvedores de software (RICHARDS, 2016). Neste sentido, caso um micro serviço seja demasiadamente refinado, aparece a necessidade de orquestração de dentro da camada de interface do usuário ou na camada de API. Outra desvantagem é provocar o aumento da complexidade do projeto para garantir a comunicação entre serviços diferentes, quando for necessário processar um pedido simples do cliente.

Outra estratégia adotada para definir o tamanho de um micro serviço, similar a capacidade de negócios, é o *design* orientado a domínio. Arquitetura de micro serviços se concentra no projeto e implementação de sistemas de software distribuídos altamente escaláveis, onde o tamanho dos serviços impacta diretamente no desempenho, flexibilidade e usabilidade. Para analisar o domínio do negócio e sua decomposição em serviços, o *Domain-Driven Design* é comumente aplicado.

4.1 DDD PARA DEFINIR UM MICRO SERVIÇO

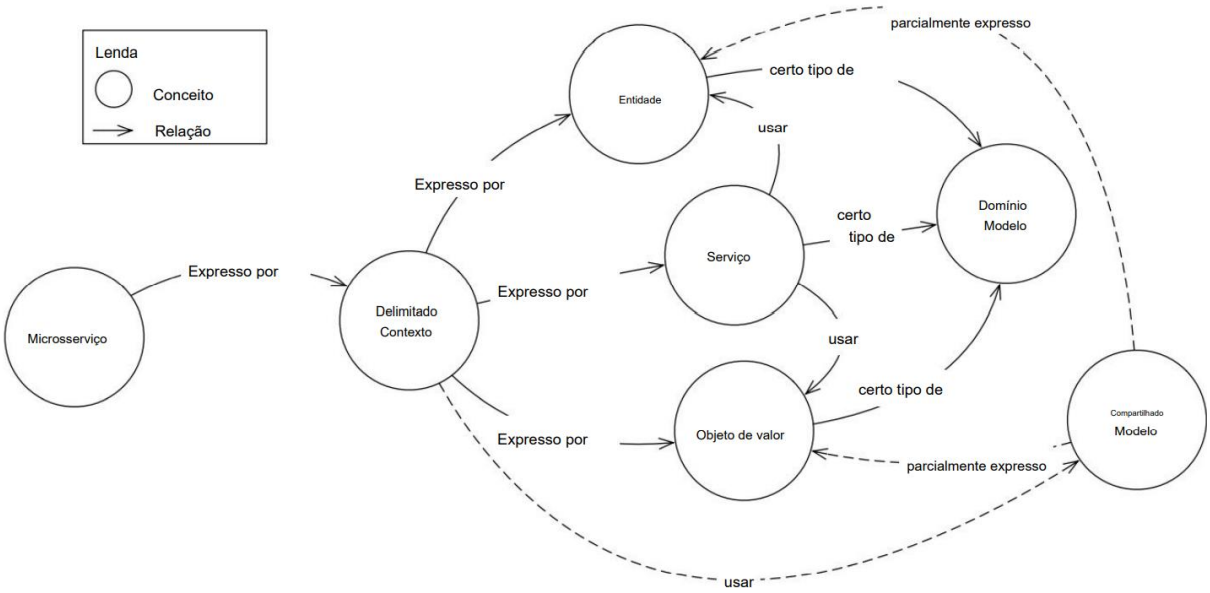
O *design* orientado a domínio oferece conceitos e etapas importantes para a criação de aplicativos baseados em uma arquitetura de micro serviços (HIPPCHEN

et al., 2017). O uso de DDD é um fator crítico de sucesso na construção de aplicativos baseados em micro serviços (NEWMAN, 2021). De acordo com Evans (2003), o DDD fornece os meios para decompor domínios em contextos, cada um agrupando conceitos de domínio coerentes. Esses contextos correspondem a micro serviços funcionais que fornecem recursos de negócio distintos (RADEMACHER; SORGALLA; SACHWEH, 2018).

A abordagem DDD fornece um meio de representar o mundo real na arquitetura, por exemplo, usando contextos delimitados (BC) para expressar unidades organizacionais e ajudar a identificar o foco de domínio central. Ambas as características levam a uma melhor qualidade da arquitetura de software (LANDRE; WESENBERG; RØNNEBERG, 2006). O BC é usado para modelar os conceitos de domínio encapsulados por um micro serviço relacionado ao negócio (NEWMAN, 2021).

Uma abordagem bem estabelecida para construir um software com micro serviços é o DDD (NADAREISHVILI *et al.*, 2016), pois utiliza o modelo de domínio (DM) para expressar conceitos do negócio. A aplicação do padrão de DM para agrupar contextos coerentes, ajuda a expressar os limites de um micro serviço (WOLFF, 2016). A Figura 9 auxilia no entendimento dos conceitos da modelagem DDD que são relevantes para um projeto de arquitetura de micro serviços orientado por domínio.

Figura 9 – Mapa de navegação para *design* dirigido por domínio em MSA



Fonte: Diepenbrock, Rademacher e Sachweh (2017, p. 3)

No entanto, os contextos delimitados e o modelo de domínio carecem de suporte para a expressão da semântica. Quando aplicado o DDD para o *design* de micro serviços isso se torna ainda mais problemático. Devido ao fato de a MSA propor uma independência de equipe (YU; SILVEIRA; SUNDARAM, 2016), os contextos diluídos pela arquitetura podem evoluir independentes, podendo afetar contextos que compartilham a mesma semântica ou são semanticamente equivalentes. Além disso, a independência entre as equipes pode levar a compreensões distintas dos conceitos de domínio, gerando ainda mais complexidade a todo o projeto (EVANS, 2003). Outros obstáculos como latência da rede, disponibilidade limitada para comunicação síncrona, manutenção da consistência dos dados entre outros serviços, obtenção de uma visão consistente dos dados, e classes que promovem dependências, devem ser considerados na hora de projetar micro serviços utilizando o DDD (NEWMAN, 2021).

4.2 PROTÓTIPOS RELACIONADOS

Na busca por evidências, foram encontrados trabalhos relacionados contendo protótipos que exemplificam a utilização de DDD em uma arquitetura MSA. Levando em consideração os benefícios e as dificuldades impostas pela abordagem, Diepenbrock, Rademacher e Sachweh (2017) constroem um protótipo para gerenciar teleconferências e trabalhos como forma de exemplificar uma abordagem de representação semântica. Utilizam uma linguagem para comunicação em equipes de produção de software, conhecida como UML (*Unified Modeling Language*), para a elaboração de um mapa de contextos. Mesmo não possuindo uma comunicação com os especialistas de domínio, os autores empregam uma Linguagem Ubíqua para evitar diferentes compreensões dos conceitos do domínio. Posteriormente, utilizam os diagramas UML já amadurecidos para criar representações por meio de OWL (*Web Ontology Language*).

OWL é um conjunto de linguagens para criar uma representação do conhecimento aplicadas na construção de ontologias, tendo como base o padrão W3C (*World Wide Web Consortium*). Essas representações expressam o conhecimento semântico como forma de documentar os contextos. De forma mais objetiva, Petrasch (2017) cria diagramas UML para a construção de um mapa de contextos bem definido, aplicando os conceitos de DDD, tendo como foco o

desenvolvimento de uma aplicação para a reserva de táxis. Esses dois trabalhos, apesar de identificarem o foco na elaboração de uma MSA com a modelagem do DDD, não avançaram para uma etapa de desenvolvimento, focando apenas em esclarecer a importância em formar uma Linguagem Ubíqua com o cliente e de realizar quantas iterações forem necessárias, para modelar os contextos do projeto.

Já Zaschke (2019) implementa um projeto chamado CSD Server, realizando reuniões com especialistas de domínio para obter conhecimento sobre o domínio e formar uma Linguagem Ubíqua com o cliente. Após isso, constrói um domínio inicial, onde novas interações ocorreram com os especialistas e alguns termos são corrigidos ou adicionados, refinando e tornando a Linguagem Ubíqua mais consistente. A partir disso, um mapa de contextos é definido, tornando visível todos os contextos pertencentes ao projeto. O que torna o protótipo de Zaschke (2019) diferente dos anteriores é a utilização do conceito de BDD (*Behavior Driven Development*). BDD é uma técnica de desenvolvimento ágil e derivada do desenvolvimento orientado a testes, que encoraja a colaboração entre desenvolvedores, setor de qualidade e pessoas não-técnicas, ou de negócios, em um projeto de software (OKOLNYCHYI; FÖGEN, 2016). Semelhante ao princípio de Linguagem Ubíqua do DDD, os padrões do BDD podem ser uma adição que agregue ao processo de construção de um mapa de contextos.

Indo um pouco mais além, Hippchen *et al.* (2017) realizam a migração de um sistema que administra teses do departamento de informática do Instituto de Tecnologia de Karlsruhe, para uma arquitetura de micro serviços. Detalhando ao máximo todo o processo, elaboram um levantamento de requisitos com os especialistas de domínio utilizando BDD, possibilitando a construção de testes para validação antes do início do desenvolvimento. Após realizarem diversas reuniões e iterações, estabelecem um modelo de domínio amplamente conhecido entre todos os envolvidos. Com o modelo em mãos, os desenvolvedores elaboram um protótipo como forma de validar o conhecimento do domínio adquirido. O protótipo é apresentado aos especialistas, que validam e, junto com a equipe de desenvolvimento, criam o mapa de contextos. Com isso, é iniciada a codificação do projeto, tendo como base o desenvolvimento em camadas "*onion architecture".* O Scrum é adotado como método de desenvolvimento ágil do projeto.

Os protótipos analisados possuem foco na adoção de uma Linguagem Ubíqua, para que o entendimento do domínio seja refinado, possibilitando assim a

construção do mapa de contextos. A busca por definir micro serviços de acordo com os contextos delimitados dentro da modelagem DDD é realizada. Porém, nenhum dos estudos encontrados apresenta mais de uma granularidade para analisar seus desempenhos. Os estudos definem uma única granularidade para cada contexto delimitado e até mesmo os que efetivamente codificam seus projetos, não apresentam dados sobre seu desempenho.

4.3 CONSIDERAÇÕES DO CAPÍTULO

Aplicar DDD em um projeto de MSA é recomendável e já amplamente difundido na comunidade. Definir micro serviços através dos contextos delimitados, demonstrados no mapa de contextos, é uma forma prática de modelar seu tamanho. Entretanto, apontar a granularidade ideal para um micro serviço ainda não é bem clara, diversos fatores podem interferir nessa decisão. Analisando os protótipos descritos neste capítulo é perceptível que uma boa modelagem do domínio e a construção do mapa do contextos são de extrema importância como ponto de partida do projeto. Porém, os estudos pecam em demonstrar os resultados quanto ao desempenho e uma análise com diferentes níveis de granularidade dentro do domínio requer mais atenção. No capítulo 5, o protótipo de Hippchen *et al.* (2017) é detalhado e demonstrado como base para a elaboração do projeto modelo.

5 CODIFICAÇÃO E ANÁLISE DE PROTÓTIPO

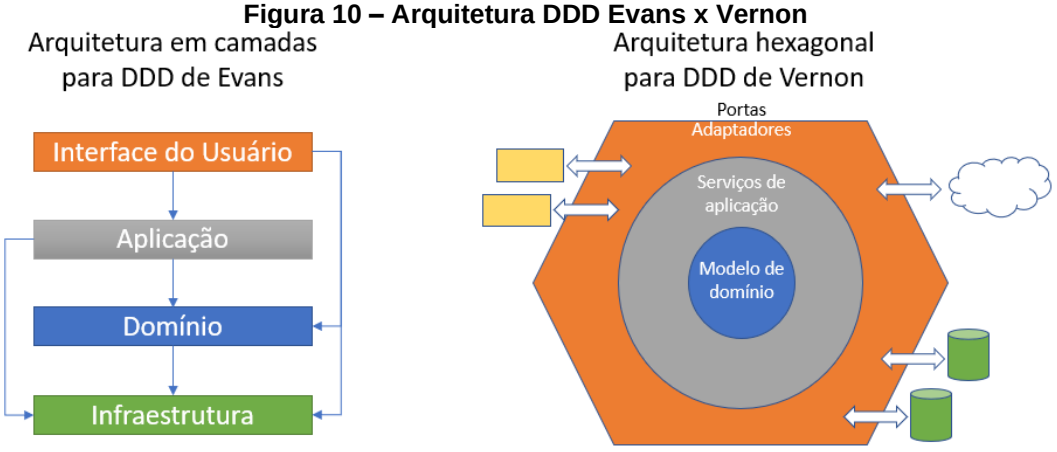
Como visto na seção 4.2, todos os trabalhos analisados possuem ênfase na exemplificação da utilização de *design* orientado a domínio, respeitando seus princípios e padrões de como projetar e desenvolver o domínio, na construção de um projeto utilizando arquitetura de micro serviços. Porém, não são demonstrados resultados referentes a desempenho e não levam em consideração uma maior ou menor granularidade dos micro serviços. O trabalho de Hippchen *et al.* (2017) é notoriamente o mais detalhado, aplicam DDD de ponta-a-ponta, visando a construção de um processo de desenvolvimento sistemático para a melhor aplicabilidade das especificações. Além disso, buscam abranger as demais camadas fora do domínio, para uma cobertura total no desenvolvimento de micro serviços.

5.1 MODELO ARQUITETÔNICO PARA MICRO SERVIÇOS

Inicialmente, Hippchen *et al.* (2017) apontam a necessidade de se adotar uma arquitetura diferente da proposta por Evans (2003). O *design* orientado a domínio requer uma estruturação em camadas para separar o domínio de outras preocupações. Na disposição em quatro camadas, vista no final do capítulo 3, a camada de interface de usuário tem acesso direto às demais camadas, não sendo ideal para uma arquitetura de micro serviços. Como alternativa, Vernon (2016) introduz a arquitetura hexagonal no contexto DDD, sendo amplamente aceito para a implementação de uma arquitetura de micro serviços.

No modelo hexagonal, cada lado representa uma porta específica de comunicação do micro serviço aos clientes. Apesar da representação da arquitetura vista na Figura 10 possuir seis lados, um micro serviços pode conter diversas portas para expor e receber informações. Baseado no princípio de inversão de dependência, cada porta possui um adaptador responsável por representar uma camada de anticorrupção. A camada de domínio, com sua lógica de negócio, se torna independente das demais camadas, assim as dependências são constituídas das camadas mais externas para as internas.

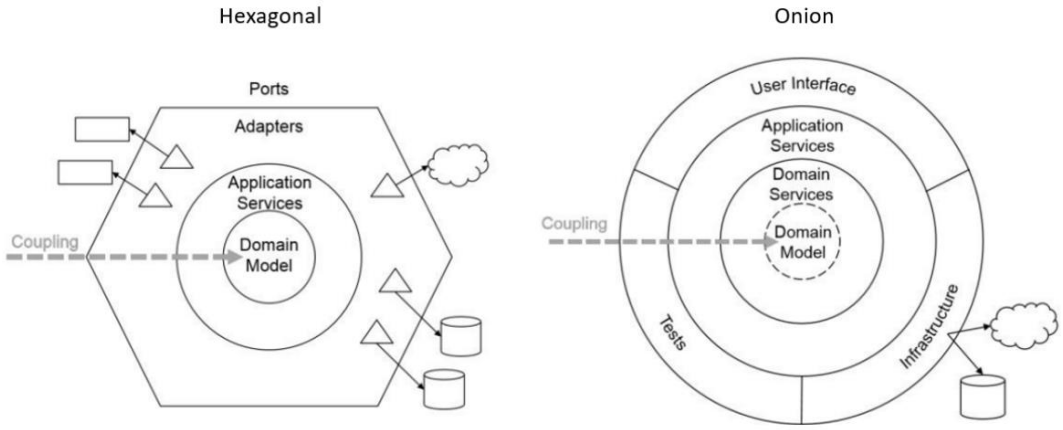
Figura 10 – Arquitetura DDD Evans x Vernon



Fonte: Elaborado pelo autor

Semelhante a arquitetura hexagonal, Palermo (2008) propôs a denominada "*onion architecture*". A diferença é a inclusão da camada de serviço de domínio, que representa o comportamento que não pode ser mapeado para objetos de domínio (EVANS, 2003). Outra diferença está na camada de infraestrutura, na *onion architecture* encontra-se na parte externa, junto com a interface. Com essas adições feitas por Palermo (2008), que beneficiam o uso de micro serviços e DDD, a arquitetura *onion* é adotada. Na Figura 11 é demostrada a representação das duas arquiteturas.

Figura 11 – Arquitetura Hexagonal x Onion



Fonte: Hippchen *et al.* (2017, p. 435)

5.2 METODOLOGIA DE DESENVOLVIMENTO ÁGIL

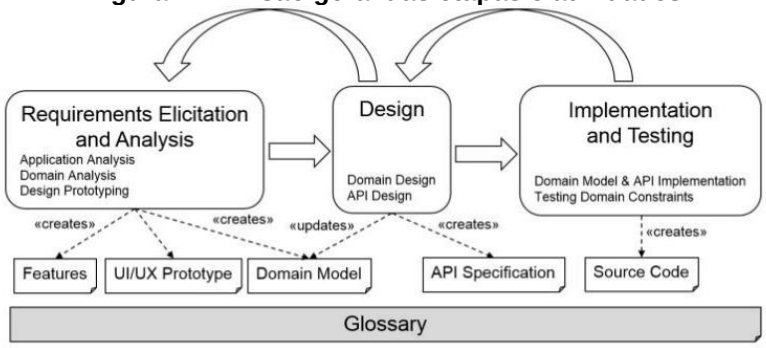
Durante o processo de análise de requisitos Hippchen *et al*. (2017) percebem a necessidade de adotar uma metodologia de desenvolvimento ágil para auxiliar na

aplicação de DDD e MSA. As atividades podem ser executadas utilizando outros modelos de processo de software, mas a necessidade específica do DDD em ajustar constantemente a compreensão do domínio, torna os processos de desenvolvimentos ágeis mais adequados. Levando em consideração a designação de uma pequena equipe para cada micro serviços e a necessidade de interação contínua entre todas as etapas, optou-se pelo Scrum. Popularmente conhecido dentro da comunidade de software, esse *framework* para gerenciamento de projetos possui diversos padrões a serem seguidos. Por exemplo, o *product backlog*, que ajuda no entendimento do que deve ser realizado, pode ser elaborado em conjunto com os recursos do BDD e histórias de usuários.

5.3 ETAPAS DO PROJETO

Após escolher um modelo de arquitetura em camadas e decidir a metodologia de desenvolvimento ágil para a elaboração do projeto, Hippchen *et al*. (2017) optam por dividir o projeto em etapas. A divisão é baseada na abordagem de engenharia de software de Bruegge e Dutoit (2009), que elencam as etapas de levantamento de requisitos, análise, projeto de sistemas, projeto de objetos, implementação e testes. Levantamento de requisitos e análise são unidas na etapa de elicitação de requisitos, projeto de sistemas e projeto de objetos constituem a etapa de *design*, por fim implementação e testes formam uma última etapa. Bruegge e Dutoit (2009) descrevem a etapa de implementação e manutenção, porém não é discutida. Visando a interação contínua entre as etapas, Hippchen *et al*. (2017) criam o fluxograma mostrado na Figura 12, para elucidar o processo.

Figura 12 – Visão geral das etapas e atividades



Fonte: Hippchen *et al.* (2017, p. 436)

5.3.1 Elicitação de requisitos

Com o DDD, os requisitos referentes ao domínio são reunidos e modelados, isso resulta no entendimento do domínio entre todos os envolvidos no projeto. Porém, as demais camadas também precisam ser atendidas conforme a necessidade do cliente. Para abranger todas as camadas, a elicitação é dividida nas atividades de análise da aplicação, exploração do domínio e desenho de um protótipo, que devem ser realizadas simultaneamente e com trocas de informações constantes para garantir a consistência dos artefatos gerados. Ao discutir o protótipo, o conhecimento do domínio é aprofundado, com o entendimento do domínio, termos e fluxos de trabalho podem mudar, aguçando ainda mais os objetivos da aplicação. Essa divisão ajuda na cobertura total da análise de requisitos.

A análise da aplicação busca adquirir o entendimento da camada de aplicação do micro serviço, tanto para consumos externos via API como da própria interface de usuário. Para isso, a utilização do BDD se torna altamente recomendável, pois requer o envolvimento total dos conhecedores do domínio. As ações esperadas pelo cliente são mapeadas através de recursos e cenários, onde são criados e validados pelos desenvolvedores e clientes. Nessa etapa, tanto desenvolvedores como clientes devem ter a capacidade de elaborar recursos e possuir o entendimento, gerando um caráter informal e uma fácil compreensão à especificação.

Durante essa etapa de análise, estudantes e membros da banca examinadora são procurados, como portadores das regras de negócio, para uma série de discussões referentes aos comportamentos básicos necessários para a gestão das teses. Cada comportamento é descrito em recursos na linguagem Gherkin, conforme Figura 13, utilizada para escrever formalmente os artefatos. A ligação entre o BDD e DDD é notória, onde a aplicabilidade da Linguagem Ubíqua é de suma importância para a manutenção e coerência de todos os artefatos.

Figura 13 – Exemplo de recurso com linguagem Gherkin

```
Feature: Teses
    Passos para um aluno se vincular a uma tese.

Scenario: Atribuir um aluno a uma proposta de tese.
    Given uma proposta de tese é aprovada.
    When um aluno é recomendado para a proposta de tese.
    And o aluno aceita participar da proposta de tese.
    Then o aluno é vinculado para a tese.
    And a oferta de tese não está mais disponível para outros estudantes.
```

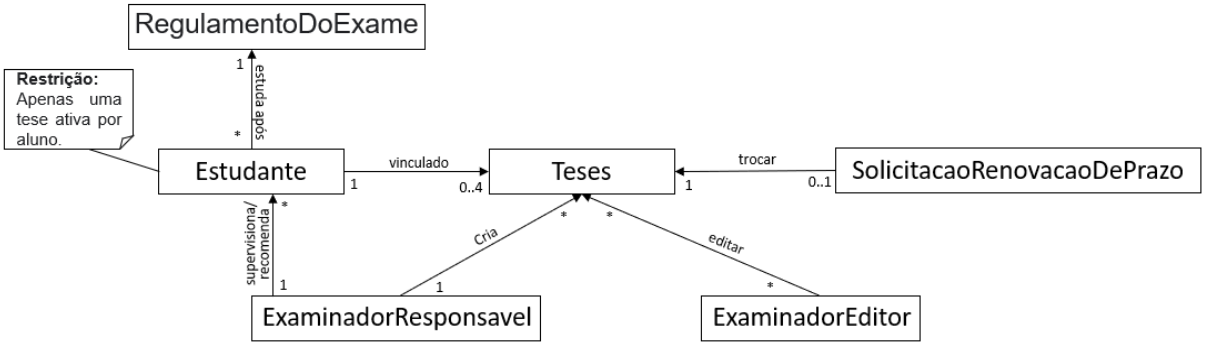
Fonte: Adaptado de Hippchen *et al.* (2017, p. 441)

Essa atividade resulta em uma especificação de requisitos abrangente, possibilitando testes automatizados na camada de aplicação. O BDD permite a criação de uma “documentação viva”, através da forte ligação entre os recursos, implementação e testes. Portanto, o BDD específica o acesso ao núcleo do aplicativo, possibilitando assim realizar testes de aplicação e de domínio. Porém, requer esforço para que atualizações sejam realizadas em todas as fases do projeto.

Na atividade de exploração do domínio são realizadas três tarefas para o início de uma modelagem consistente. A ideia central é entender a necessidade do cliente, abordando todos os conceitos que são de conhecimento dos especialistas de domínio. Nessa etapa, membros do comitê que administra as teses do departamento de informática do Instituto de Tecnologia de Karlsruhe, são chamados para discussões.

A primeira tarefa consiste em realizar a “trituração” do domínio, ou seja, o processamento do conhecimento adquirido durante as discussões. A equipe de desenvolvimento realiza simultaneamente conversas com os clientes e a modelagem do domínio. Esse processo é iterativo até ambas as partes concordarem com o modelo gerado. Primeiro são explorados conceitos e relações referentes à tese, posteriormente são constatadas restrições. Ao final do processo de “trituração”, o objetivo é possuir representações que esbocem todas as informações obtidas, cada representação constitui uma seção do modelo de informação, conforme é ilustrado na Figura 14.

Figura 14 – Seção do modelo de informação mostrando conceitos do domínio tese



Fonte: Adaptado de Hippchen *et al.* (2017, p. 441)

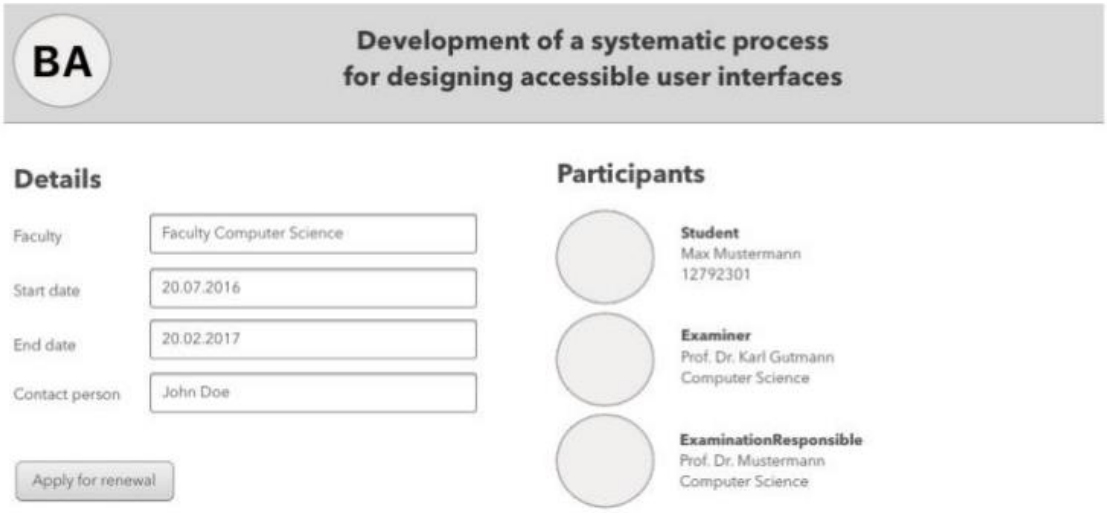
Em domínios complexos, podem ocorrer dificuldades para o cliente compreender o modelo de domínio. Comportamentos dinâmicos, como fluxos de trabalho, são conceitos relevantes para o domínio. Para tornar esse entendimento mais facilitado, Hippchen *et al*. (2017) propõem a segunda tarefa de criação de visões do domínio, para modelar diferentes aspectos. Dois *stakeholders* podem referenciar um mesmo domínio de formas diferentes, gerando duas visões distintas sobre o mesmo processo. Neste cenário, as partes interessadas tendem a possuir interesses conflitantes que afetam a estrutura do modelo. As representações dessas visões podem ser realizadas através de fluxogramas de processos. A relação entre o interesse de um *stakeholder* e o tipo de visão de domínio torna possível determinar as pessoas certas para a discussão do conteúdo de cada visão.

A Linguagem Ubíqua é um fator central quando se utiliza DDD. Vernon (2016) aponta ser de grande valia a elaboração de um glossário para capturar os termos utilizados para a elaboração do domínio. Cada termo da Linguagem Ubíqua é listado e descrito por algumas frases. A elaboração do glossário é a terceira tarefa na exploração do domínio. O glossário é um artefato transversal, deve ser criado e atualizado durante todas as etapas do projeto, conforme ilustrado na Figura 12. Manter o glossário atualizado exige esforço, mas os benefícios superam os esforços. O processo pelo qual o modelo de domínio é desenvolvido tem influência, na maior parte, pela exploração e experimentação (EVANS, 2003). Implementar um modelo de domínio que não seja completamente satisfatório é mais importante que refinar repetidamente o modelo de domínio sem realmente realizar a implementação (EVANS, 2003).

Com base nos cenários elaborados na análise da aplicação e nas discussões com os *stakeholders*, cada recurso criado deve ser representado no chamado

protótipo de projeto. A atividade de desenho do protótipo, concentra-se em validar a camada de interface e experiência do usuário. Como o cliente geralmente interage pela interface, essa atividade gera um artefato para discussões com o cliente sobre o modelo de domínio. No protótipo, as interações são necessárias para que atendam todas as necessidades do cliente. O feedback do cliente é coletado e analisado para identificar as mudanças necessárias antes da próxima iteração. O processo de prototipagem só é concluído quando todas as necessidades do cliente são atendidas. A Figura 15 apresenta uma página contendo as informações de uma tese específica, sendo utilizada, juntamente com todo o protótipo, para validar o conhecimento do domínio.

Figura 15 – Página de detalhes de tese pertencentes ao protótipo



Fonte: Hippchen *et al.* (2017, p. 441)

5.3.2 *Design* do projeto

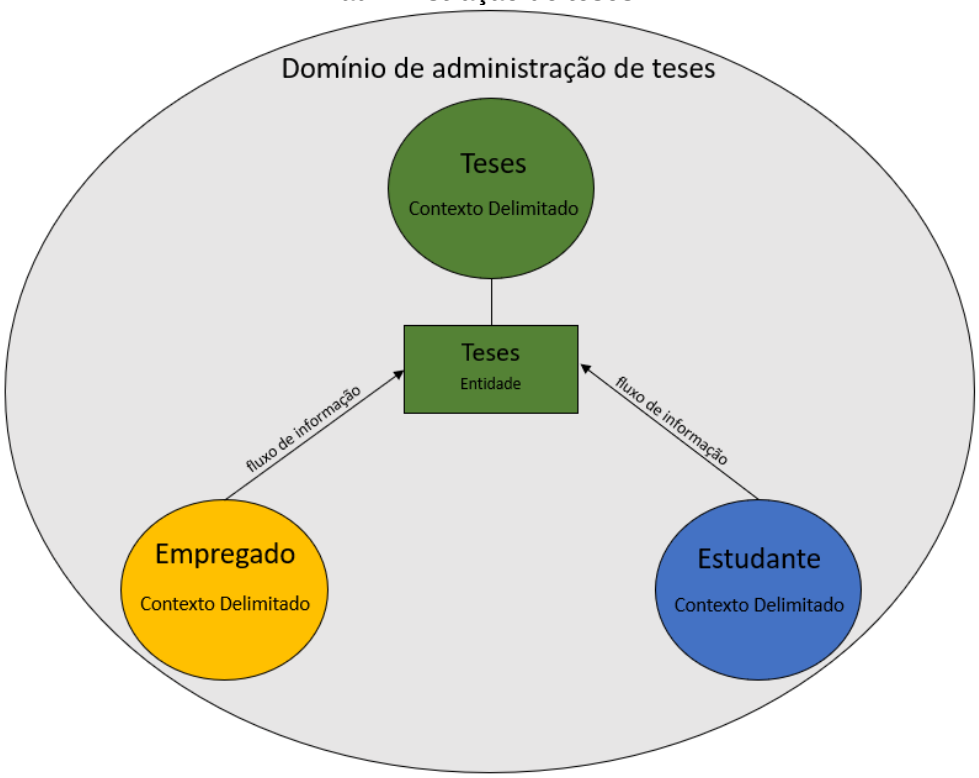
A fase de *design* do projeto é composta por duas tarefas, o *design* do domínio e o *design* da API. Essas tarefas requerem o modelo de informação e o protótipo criados na fase de elicitação de requisitos. Para que os padrões de *design* sejam bem aplicados e um modelo de informação seja bem definido, discussões adicionais com os especialistas de domínio e um protótipo avaliado e aceito pelo cliente são de extrema importância.

O *design* do domínio é focado na ideia central do DDD, a camada de domínio. O modelo de informação é dividido em contextos delimitados para que seja

entendido e refinado, aplicando padrões de projeto que satisfaçam os requisitos do aplicativo. Essa tarefa requer responsáveis experientes e diversas iterações. Os objetos contidos em cada contexto devem possuir indicações a qual tipo eles pertencem, ou seja, raiz agregada, entidade, objeto de valor ou evento de domínio, para que a equipe de desenvolvimento responsável pelo contexto possa identificar os requisitos necessários em conjunto com os recursos do BDD e o protótipo. Em seguida, repositórios, fábricas e serviços de domínio são adicionados de acordo com as necessidades da aplicação. Os especialistas de domínio e outras fontes de informação devem estar mutuamente envolvidos para uma análise contínua do conhecimento.

Com isso, Hippchen *et al*. (2017) elaboram, a partir do seu modelo de informação e das visões do domínio, um mapa de contextos que descreve as relações entre os contextos delimitados elegidos para se tornarem micro serviços. Teses, empregado e estudante são definidos como contextos delimitados dentro da administração de teses conforme é mostrado na Figura 16.

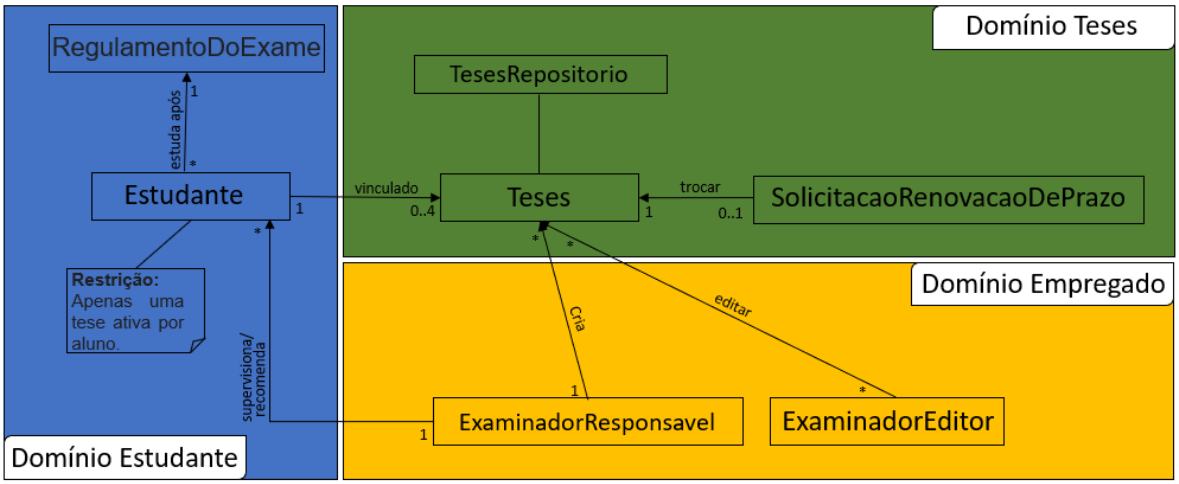
Figura 16 – Mapa de contextos representando os contextos delimitados do domínio de administração de teses



Fonte: Adaptado de Hippchen *et al.* (2017, p. 442)

Considerando uma arquitetura de micro serviços, o propósito de um mapa de contextos não é apenas obter o entendimento do domínio, mas também proporcionar uma fonte de conhecimento dos micro serviços existentes e suas responsabilidades, para que toda a organização possa gerenciar seus cenários. Além disso, cada contexto delimitado possui sua própria Linguagem Ubíqua, sendo baseada no conhecimento do domínio, formando um contrato de comunicação entre os membros do projeto e as partes interessadas. Cada micro serviço tem seu modelo de domínio construído com a identificação de suas entidades, objetos de valor e demais padrões do DDD. O *design* de domínio resulta em um modelo de domínio consolidado que deve ser vinculado aos artefatos de implementação. Na Figura 17 consta o modelo de domínio para o micro serviço de administração de teses.

Figura 17 – Modelo de domínio para micro serviço de administração de teses

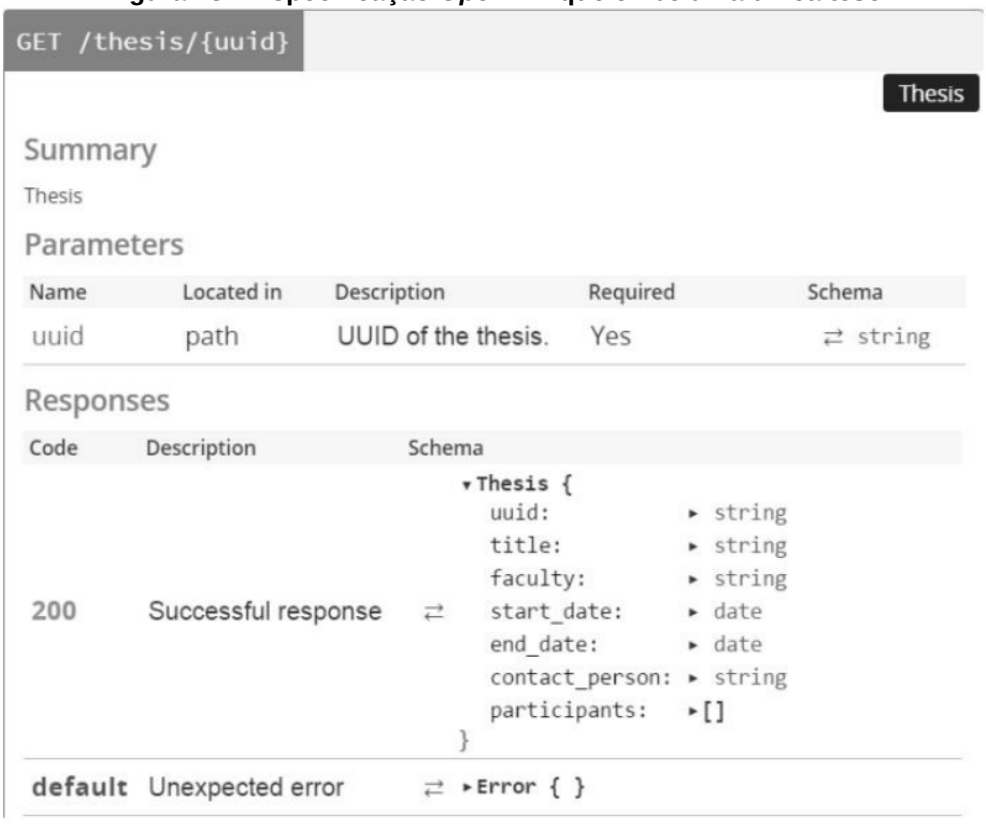


Fonte: Adaptado de Hippchen *et al.* (2017, p. 442)

A tarefa de *design* da API é utilizada para coordenar os micro serviços no mapeamento de fluxos de trabalho ou oferecer funcionalidades do negócio a terceiros. Os artefatos obtidos na análise da aplicação e na elaboração do protótipo tem como função auxiliar no *design* de API. Os recursos elaborados com o BDD formam casos de uso, que representam os requisitos que devem ser atendidos pela API. Para que os casos de uso tenham relações quanto ao ciclo de vida dos recursos, devem ser realizadas investigações de possíveis interações, levando em consideração os comportamentos encontrados no modelo de domínio.

Hippchen *et al*. (2017) projetam suas APIs utilizando REST, um padrão já comumente conhecido e aceito. De posse dos casos de uso devidamente separados em seus contextos delimitados, as URIs são pensadas para que os recursos possam ser corretamente endereçados sem ambiguidade, garantindo a mais alta reutilização. As APIs são estruturadas de acordo com as especificações da *OpenAPI*, que tem como objetivo definir uma interface padrão, agnóstica de linguagem e que permita que humanos e computadores descubram e compreendam as responsabilidades de cada serviço, sem a necessidade de possuir acesso ao código-fonte ou realizar uma verificação por meio de inspeção de tráfego de rede. Na Figura 18 uma chamada da API para recuperar uma tese pelo seu código único é exemplificada.

Figura 18 – Especificação *OpenAPI* que exibe uma única tese



Fonte: Hippchen *et al.* (2017, p. 442)

Apesar de não demonstrarem o avanço no que se diz respeito à construção de uma interface de usuário, Hippchen *et al*. (2017) apontam que nessa fase também é construído o *design* da camada de *Backend-For-Frontend* (BFF). A interface do usuário é responsável por servir de comunicação entre os usuários e os diversos micro serviços que constituem todos os requisitos de negócio do sistema. A

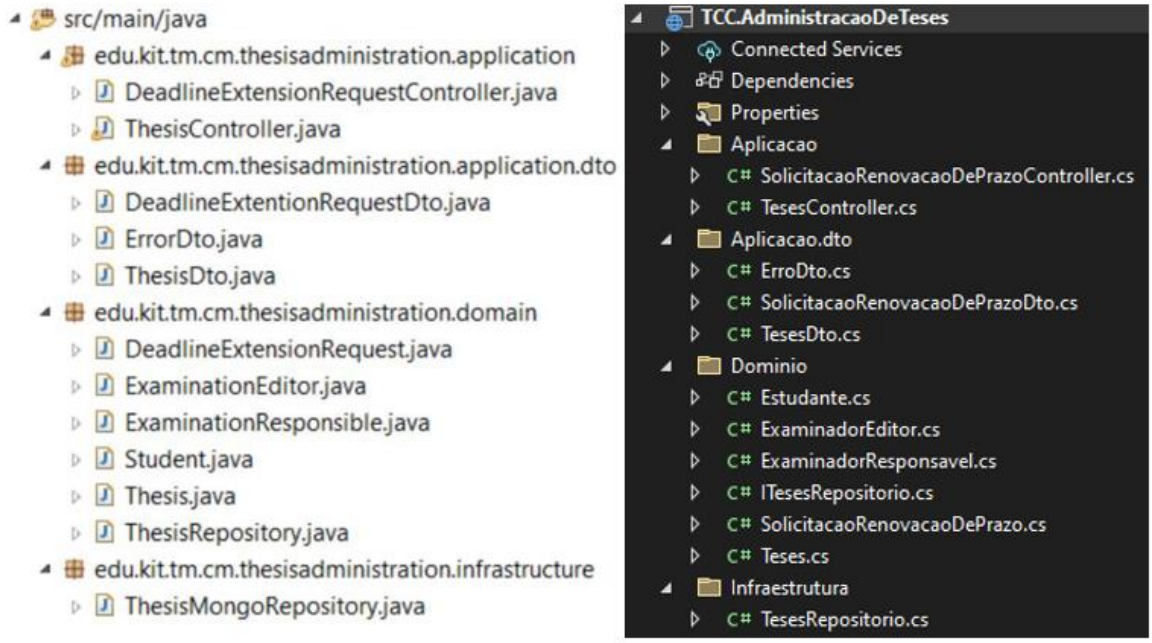
camada de BFF deve ser pensada para evitar que muitas solicitações sejam realizadas para obter as informações necessárias, pois cada micro serviço é projetado considerando seu contexto específico e não os casos de uso. O BFF deve ser vista como parte da interface do usuário, pois possibilita que o time responsável se concentre nos seus requisitos específicos e não nas APIs expostas por cada micro serviço.

5.3.3 Implementação e testes

Com os artefatos criados na elicitação de requisitos e amadurecidos no *design* do projeto, é iniciada a implementação e testes dos micro serviços. Utilizando o mapa de contextos construído no *design* do domínio, Hippchen *et al*. (2017) codificam os micro serviços para cada contexto delimitado, levando à risca todo o modelo de domínio elaborado nas etapas anteriores. Após isso, as camadas de aplicação, infraestrutura e BFF são desenvolvidas conforme o *design* de API e os comportamentos esperados pelos recursos. Por fim, uma cobertura de testes automatizados utilizando BDD é recomendada para a validação da conformidade do desenvolvimento com a elicitação de requisitos.

Pela complexidade e importância do domínio em um projeto com DDD, a implementação do domínio, necessita dos desenvolvedores mais experientes e que compreendam todos os contextos existentes. O *design* orientado a domínio recomenda o uso de integração contínua, portanto o *pipeline* é configurado e um controle de versão é adotado para auxiliar na implementação. Seguindo os padrões da *onion architecture*, cada micro serviço é implementado respeitando as camadas, sendo separadas em pastas ou pacotes e seguindo um fluxo de referências de fora para dentro, auxiliando os desenvolvedores a focar em sua camada de trabalho e diminuindo o acoplamento interno. Para exemplificação da implementação proposta por Hippchen *et al*. (2017), é elaborada a mesma solução proposta na linguagem C#, levando em consideração os *frameworks* necessários. Na Figura 19 é possível visualizar a estrutura construída no trabalho referencial e na codificada pelo autor.

Figura 19 – Estrutura na linguagem Java x C#

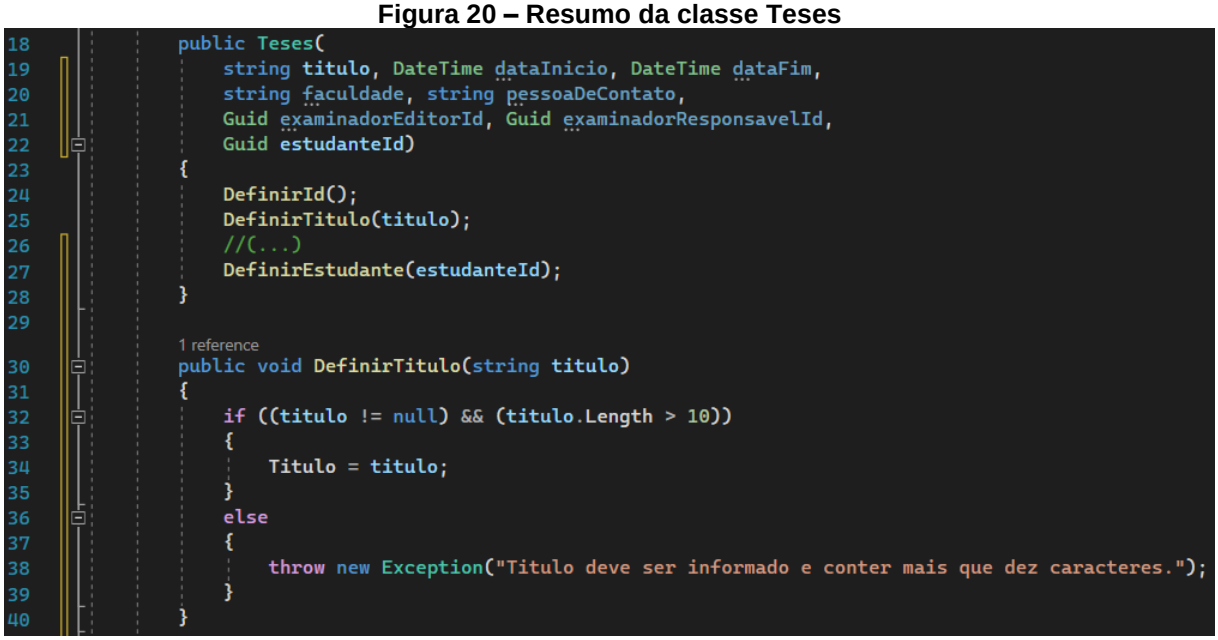


Fonte: Elaborado pelo autor

Indo de encontro ao conceito de Linguagem Ubíqua, os objetos e suas nomenclaturas definidas nas etapas anteriores são mantidas na construção das classes. Os objetos de domínio devem conter suas restrições, como multiplicidade e lógica de domínio, para que se tornem objetos inteligentes. A relação entre dois objetos deve ser exercida utilizando seu identificador. A criação e manipulação de objetos de domínio não pode conter informações que não são consistentes com o modelo de domínio.

Todos os projetos elaborados por este trabalho estão em https://drive.google.com/file/d/1Z1No1fptVZjTo33ruKoppxEWUCABVV8P e podem ser consultados no Apêndice A. A partir daqui os códigos de Hippchen *et al*. (2017) para exemplificação serão demonstrados através do código contido no projeto “TCC.AdministracaoDeTeses”. Na Figura 20 o objeto de domínio da tese é codificado e demonstrado resumidamente.

Figura 20 – Resumo da classe Teses



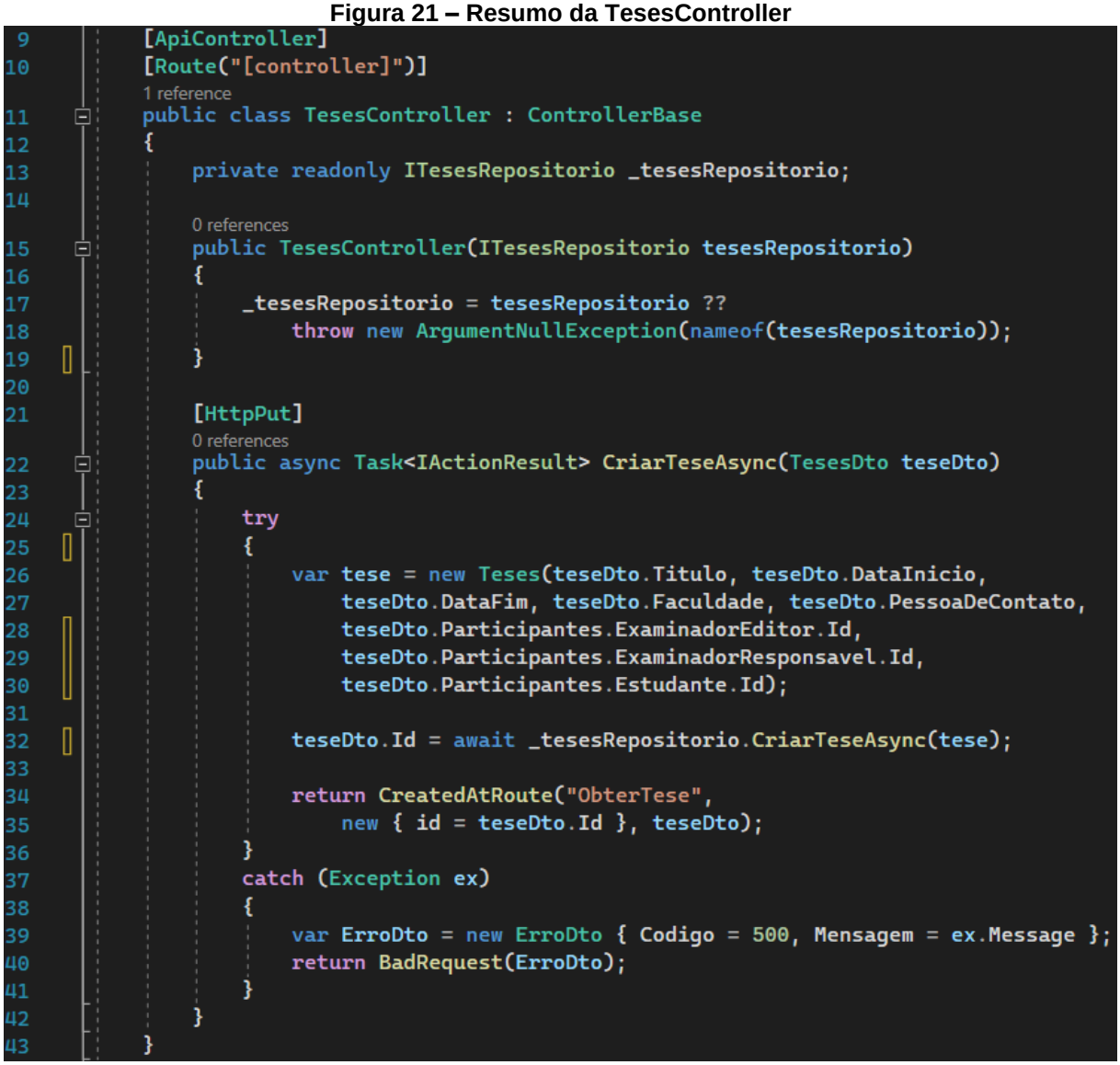
```
18      public Teses(
19          string titulo, DateTime dataInicio, DateTime dataFim,
20          string faculdade, string pessoaDeContato,
21          Guid examinadorEditorId, Guid examinadorResponsavelId,
22          Guid estudanteId)
23      {
24          DefinirId();
25          DefinirTitulo(titulo);
26          //(...)
27          DefinirEstudante(estudanteId);
28      }
29
        1 reference
30      public void DefinirTitulo(string titulo)
31      {
32          if ((titulo != null) && (titulo.Length > 10))
33          {
34              Titulo = titulo;
35          }
36          else
37          {
38              throw new Exception("Titulo deve ser informado e conter mais que dez caracteres.");
39          }
40      }
```

Fonte: Elaborado pelo autor

O construtor espera todos os atributos necessários para uma tese válida. A partir da linha 24 os dados são vinculados às suas respectivas propriedades através de métodos que representam o comportamento do objeto. Dentro do método *DefinirTitulo*, na linha 32, são realizadas duas validações para permitir a criação do objeto *Tese*. Caso essas restrições não sejam atendidas o construtor retorna uma exceção, não permitindo a criação do objeto.

O *design* da API define os pontos de entrada para cada micro serviço, assim a implementação da aplicação é realizada de maneira específica utilizando REST. As partes de responsabilidade da aplicação são separadas do domínio, gerando uma camada de anticorrupção, utilizada para obter uma separação limpa entre termos de aplicação e domínio. Cada micro serviço deve possuir uma camada de aplicação no topo, com abstrações dos objetos do domínio, para que pequenas alterações não influenciem na implementação da interface, reduzindo assim o acoplamento entre o domínio e a API. Para isso, são criados objetos de transferência de dados (DTO), conforme as especificações elaboradas no *design* de API. Na Figura 21 é demonstrado, em um formato resumido, a classe *TesesController*, responsável por fazer o mapeamento de entrada de requisições para o micro serviço de teses, através da camada de aplicação e da serialização dos dados para *teseDto*.

Figura 21 – Resumo da TesesController



```
 9       [ApiController]
10       [Route("[controller]")]
         1 reference
11       public class TesesController : ControllerBase
12       {
13           private readonly ITesesRepositorio _tesesRepositorio;
14
             0 references
15           public TesesController(ITesesRepositorio tesesRepositorio)
16           {
17               _tesesRepositorio = tesesRepositorio ??
18                   throw new ArgumentNullException(nameof(tesesRepositorio));
19           }
20
21           [HttpPut]
             0 references
22           public async Task<IActionResult> CriarTeseAsync(TesesDto teseDto)
23           {
24               try
25               {
26                   var tese = new Teses(teseDto.Titulo, teseDto.DataInicio,
27                       teseDto.DataFim, teseDto.Faculdade, teseDto.PessoaDeContato,
28                       teseDto.Participantes.ExaminadorEditor.Id,
29                       teseDto.Participantes.ExaminadorResponsavel.Id,
30                       teseDto.Participantes.Estudante.Id);
31
32                   teseDto.Id = await _tesesRepositorio.CriarTeseAsync(tese);
33
34                   return CreatedAtRoute("ObterTese",
35                       new { id = teseDto.Id }, teseDto);
36               }
37               catch (Exception ex)
38               {
39                   var ErroDto = new ErroDto { Codigo = 500, Mensagem = ex.Message };
40                   return BadRequest(ErroDto);
41               }
42           }
43       }
```

Fonte: Elaborado pelo autor

Na linha 10, a *tag Route(“[controller]”)* identifica o nome da classe como o caminho de recebimento das requisições, no caso demonstrado é Teses. Ao utilizar injeção de dependência no construtor da classe, a partir da linha 15, as interfaces e suas implementações são separadas, pois o repositório de persistência de dados, que nesse projeto faz comunicação com um banco de dados relacional *Sql Server*, é de responsabilidade da camada de infraestrutura, não fazendo parte da camada de aplicação. A partir da linha 21, a *tag [HttpPut]* identifica que o método *CriarTeseAsync* é o responsável por receber e processar requisições de criação de teses. Através do parâmetro *teseDto*, os dados enviados para a criação das teses são serializados e, a partir da linha 26, transformados no objeto *Teses* do domínio. Por fim é persistido através da *TesesRepositorio*. Caso a requisição termine sem

erros, *teseDto* é retornado ao requerente, caso contrário é retornada uma exceção contendo *ErroDto*.

A camada de infraestrutura é responsável pelas funcionalidades necessárias para acessar bancos de dados, registrar eventos, realizar autorizações e manipulação de *cache.* A infraestrutura serve como suporte de demandas necessárias pelo domínio e aplicação, que ficam fora das suas respectivas responsabilidades. Se um micro serviço possuir repositório, deve estar contido nessa camada. Cada micro serviço pode possuir um ou mais banco de dados, de diferentes estruturas e tecnologias, responsáveis por realizar a persistência de dados. Para isso, a implementação da infraestrutura deve viabilizar a capacidade de atender as necessidades de cada uma delas. Na Figura 22 é demonstrado o repositório utilizando o *framework* Dapper para a persistência em bancos de dados.

O repositório para as teses recebe, em seu construtor, a *string* de conexão com o banco de dados através de injeção de dependência. Para persistir uma tese criada, na linha 18 o método *CriarTeseAsync* recebe como parâmetro um objeto do domínio. Seguindo o padrão do Dapper, uma instrução de *insert* para um banco de dados *Sql Server* é elaborada e a conexão com o banco de dados é aberta. Após isso, a partir da linha 29, a instrução recebe os dados contidos no objeto tese e é enviada ao banco de dados através do método *QuerySingleAsync*. Caso não ocorram erros, o método retorna o identificador do objeto persistido, caso contrário retornará uma exceção. Por fim, a conexão com o banco de dados é fechada.

Figura 22 – Repositório de Teses

```
 9   public sealed class TesesRepositorio : ITesesRepositorio
10   {
11       private readonly string _stringDeConexao;
12
             1 reference
13       public TesesRepositorio(string stringDeConexao)
14       {
15           _stringDeConexao = stringDeConexao;
16       }
17
             2 references
18       public async Task<Guid> CriarTeseAsync(Teses tese)
19       {
20           const string insercao = @"INSERT INTO Teses (Id, Titulo, (...), EstudanteId)
21                                     OUTPUT INSERTED.Id
22                                     VALUES (@Id, @Titulo, (...), @EstudanteId);";
23
24           using var conexao = new SqlConnection(_stringDeConexao);
25           try
26           {
27               await conexao.OpenAsync();
28
29               var id = await conexao.QuerySingleAsync<Guid>(insercao,
30               new
31               {
32                   tese.Id,
33                   tese.Titulo,
34                   //(...)
35                   tese.EstudanteId
36               });
37
38               return id;
39           }
40           catch (Exception ex)
41           {
42               throw new Exception($"Erro ao tentar inserir uma tese. ({ex.Message})", ex);
43           }
44           finally
45           {
46               await conexao.CloseAsync();
47           }
48       }
```

Fonte: Elaborado pelo autor

Hippchen *et al*. (2017) discutem a implementação do BFF como uma aplicação que oculte a arquitetura de micro serviços. Sua maior atribuição é servir como intermediador entre a interface de usuário e a aplicação dos micro serviços, realizando a orquestração das requisições e respostas. Sua implementação deve ser mantida simples e possuir coerência com as especificações criadas no *design* de API para BFF. Atividades como autenticação e controle de acesso também podem ser implementadas, porém o recomendado é que seja criado um micro serviço específico para essas tarefas.

Para validar e garantir que o modelo de domínio foi implementado corretamente, uma abordagem de teste automatizado é recomendada. Hippchen *et al*. (2017) utilizam BDD para reunir recursos baseados em comportamentos na

análise da aplicação. Esses recursos são as principais fontes de conhecimento para todo o processo iterativo de desenvolvimento e testes. A aplicação dessa abordagem, em conjunto com testes unitários, facilita os testes das funcionalidades do projeto. Caso um recurso não passe nos testes, o desenvolvimento é retomado até que o teste seja aceito. Em cada iteração com o desenvolvimento, a equipe explora ainda mais os requisitos da aplicação e pode adicionar mais recursos para novas implementações. Assim, o processo é conduzido e repetido várias vezes, para a construção da solução pretendida.

5.4 CONSIDERAÇÕES DO CAPÍTULO

Hippchen *et al*. (2017) realizam a construção de um projeto de ponta-a-ponta utilizando DDD e micro serviços. A adoção da arquitetura *onion* e do Scrum mostrou-se correta em todas as etapas do projeto. A elicitação de requisitos constrói um alicerce para a projeção de um *design*. A partir do *design* o projeto é implementado e testado. Durante a fase de implementação e testes, são necessárias novas revisões nos requisitos e no *design* para adequação do entendimento entre desenvolvedores e os clientes. O processo descrito neste capítulo serve como base para a construção do projeto modelo utilizado para testar métodos de alterações granulares em micro serviços. Tendo em vista que o modelo de recomendação é independente de regra de negócio, no capítulo 6 é apresentado um modelo inicial e as abordagens de alterações granulares aplicadas.

6 MODELO DE RECOMENDAÇÃO PARA DEFINIÇÃO GRANULAR

A ideia de construção do modelo de recomendação se deu através da busca na literatura por um modelo que fosse capaz de indicar os passos que pudessem ser utilizados para a redução da granularidade em projetos com arquitetura de micro serviços e *Domain*-Driven *Design*. Como tal modelo não foi encontrado, observou-se que sua elaboração pode trazer benefícios na construção de projetos que necessitam de contextos e responsabilidades bem definidos. Visando auxiliar na tomada de decisão das técnicas mais apropriadas a serem aplicadas, os passos levam em consideração os benefícios de qualidade e desempenho da aplicação. O modelo inicialmente foi estruturado em um quadro contendo quatro passos que serão discutidos ao longo deste capítulo.

6.1 ELABORAÇÃO DE UM MODELO VISUAL

Primeiramente, é necessária a elaboração de um modelo visual, como por exemplo uma lista ou mapa mental, contendo todos os micro serviços que constituem o projeto e suas respectivas entidades. Esse modelo possui a função de tornar visível os micro serviços e suas conexões. A origem e objetivos do modelo visual são herdadas do mapa de contextos vindo do DDD. Portanto, a ideia é que esse passo seja executado entre equipe de desenvolvimento e cliente, para que no final, o modelo visual possibilite um entendimento por todos os envolvidos no projeto e tornem explícitos todos os contextos delimitados. O modelo visual é necessário para a execução dos demais passos do modelo de recomendação, tornando clara a localização de pontos de melhoria dentro da arquitetura de micro serviços.

6.2 QUANTIDADE DE ENTIDADES COM RESPONSABILIDADES ÚNICAS

O segundo passo é identificar, a partir do modelo visual, o número de entidades que possuem responsabilidades próprias dentro de cada micro serviço. Para os micro serviços que contém mais do que três entidades com preocupações distintas, o recomendado são suas decomposições em novos micro serviços, garantindo assim a responsabilidade única e diminuindo o nível granular. Responsabilidade única é um padrão utilizado pelo DDD para garantir que cada

entidade realize apenas o que é de sua característica. Para micro serviços, esse padrão é ainda mais necessário, tendo em vista que cada serviço precisa se limitar em realizar suas tarefas, tornando-o independente.

6.3 QUANTIDADE DE RAÍZES AGREGADAS

O terceiro passo consiste em localizar e decompor os micro serviços que possuam mais do que três raízes agregadas, ou seja, realizam comunicações com mais do que três outros serviços. Esses micro serviços, além de compor uma granularidade alta, tendem a possuir muita responsabilidade. Reduzir o número de mensagens enviadas por um micro serviço auxilia na performance da arquitetura e na velocidade de propagação das informações entre os serviços. O que difere esse passo do anterior é que, o anterior tem relação direta com o domínio, enquanto o passo atual realiza alterações significativas em todo o contexto arquitetônico dos micro serviços.

6.4 DECOMPOSIÇÃO DE ABSTRAÇÕES

O quarto passo visa remover abstrações que causam ruído, principalmente na Linguagem Ubíqua. Micro serviços que representam um conjunto de termos diferentes, mas que possuem características idênticas, sendo diferenciadas geralmente por uma única propriedade. Para decompor a abstração e diminuir a granularidade é necessário identificar os novos termos candidatos a se tornarem micro serviços.

Esse passo tem relação direta com a fase de elicitação de requisitos, onde um entendimento equivocado na elaboração das visões do domínio, tendem a criar micro serviços abstratos no modelo de domínio. Este passo, apesar de demonstrar uma ligação forte com o domínio, também causa impacto em toda a arquitetura, pois os serviços que se comunicavam antes com um serviço abstrato, agora precisam possuir inteligência para remanejar as informações corretamente entre as decomposições.

6.5 CONSIDERAÇÕES DO CAPÍTULO

Para realizar uma análise que visa evidenciar um modelo de definição granular em micro serviços, quatro passos são elaborados. Esses passos possuem focos distintos e realizam diferentes mudanças dentro do mapa de contextos, demandando esforço em todas as etapas do projeto. No Quadro 3 é demonstrado cada um dos passos do modelo de recomendação. Este quadro tem como objetivo servir como um modelo de recomendação para equipes que pretendem reduzir o nível granular de seus micro serviços, visando uma melhor qualidade e desempenho. No capítulo 7, cada passo será aplicado em um projeto modelo baseado no portal do TC-Online da Universidade Feevale.

Quadro 3 – Modelo de recomendação inicial

Passo	Regra	Impacto
Criar um modelo visual contendo cada micro serviço com suas entidades.	Todos os micro serviços com suas entidades devem ser visíveis.	Tornar o modelo de domínio mais visível para a aplicação dos próximos passos.
Contabilizar o número de entidades que possuem responsabilidades próprias em cada micro serviço.	Caso o micro serviço possua mais do que três responsabilidades, sua granularidade é alta, devendo ser considerada a decomposição.	Dispor de uma separação de responsabilidades entre os micro serviços, diminuindo a granularidade.
Contabilizar o número de comunicações que cada micro serviço realiza com outros serviços.	Caso o micro serviço realize comunicações com mais do que três outros micro serviços, sua granularidade é alta, devendo ser considerada a decomposição.	Reduzir o número de comunicações que um micro serviço precisa realizar.
Analisar se o micro serviço representa uma abstração.	Caso o micro serviço realiza a abstração de termos parecidos do negócio, deve ser considerado decompor, para que os termos sejam claros e tenham suas responsabilidades especificas.	Remover abstração torna o contexto mais visível e diminui a granularidade para cada termo.

Fonte: Elaborado pelo autor

7 APLICAÇÃO DO MODELO EM UM PROJETO

Seguindo os passos contidos no capítulo 5 para a construção de um projeto com DDD e micro serviços, os contextos do Portal TC-Online da Universidade Feevale são utilizados para a validação dos passos que compõem o modelo de recomendação visto no capítulo 6. Esse portal tem como função administrar todas as atividades necessárias durante o desenvolvimento do trabalho de conclusão para cursos que possuem esse componente obrigatório para a colação de grau.

7.1 CONSTRUÇÃO DO PROJETO MODELO

Para o primeiro passo do modelo de recomendação, um projeto modelo inicial é construído e um mapa de contextos é elaborado como um modelo de visão. Conforme visto durante todo o capítulo 5, as interações com pessoas que detém conhecimento da regra de negócio são necessárias. Porém, como o modelo de recomendação é independente de regra de negócio, essas interações não são realizadas e o mapa de contexto, junto com o modelo de domínio, são projetados de acordo com a análise do portal, levanto em consideração as nomenclaturas utilizadas e funcionalidades realizadas. Portanto, o portal funciona como um protótipo já consolidado, servindo como base para a análise da aplicação e a exploração do domínio.

7.1.1 Elicitação de requisitos

Inicialmente são detectados todos os termos elegíveis para se tornarem entidades ou objetos de valor. Ao acessar o portal, o primeiro termo encontrado são os cursos. Logo após, termos como trabalhos, compromissos e orientadores são visíveis na barra de menu do portal. Trabalho de conclusão I e trabalho de conclusão II são apresentados como estados do trabalho, podendo ser abstraídos como situações do trabalho. Funcionalidades como alteração dos dados do trabalho e registro de ata também são visíveis para os alunos dentro do portal. Os termos encontrados são listados e classificados, tais como:

- **Aluno:** São os responsáveis pelos trabalhos, possuem um portal do aluno com uma série de informações e dados básicos. Para o contexto do portal do TC-online é importante seu nome e matrícula.
- **Áreas de interesse:** Termo utilizado para identificar um objeto de valor atrelado tanto para a entidade trabalho como para professores.
- **Ata:** Termo encontrado vinculado ao aluno, são informações das atividades realizadas e serve como confirmação da frequência durante a realização do trabalho.
- **Avaliação:** Cada situação de trabalho possui sua avaliação inserida por um orientador ou avaliador. Dependendo da situação, os parâmetros de avaliação mudam, criando uma forte ligação com a situação do trabalho.
- **Avaliador:** Cada trabalho possui dois avaliadores responsáveis por realizar avaliações durante todo o ciclo de trabalho.
- **Currículo:** Um curso possui um ou mais currículos, onde são reunidas todas as disciplinas necessárias para a conclusão do aluno.
- **Curso:** Sendo o primeiro termo encontrado ao acessar o portal, todos os alunos, trabalhos e professores possuem um vínculo com algum curso.
- **Compromisso:** Pode ser compreendido como uma agenda, cada situação de trabalho possui uma série de compromissos que devem ser atendidas pelo aluno durante o processo de desenvolvimento do trabalho de conclusão.
- **Disciplina:** Compõem o currículo do curso. Cada disciplina possui um ou mais professores atribuídos.
- **Documento:** Cada situação de trabalho possui um documento atrelado, que serve como base avaliativa.
- **Instituto:** Os cursos da universidade são agrupados em institutos de acordo com sua área.
- **Modelo:** Termo utilizado para identificar um objeto de valor atrelado a entidade curso. Podendo ser atribuído com valores específicos, por exemplo bacharelado ou licenciatura.
- **Orientador:** Professor responsável por orientar o aluno durante toda a elaboração do trabalho de conclusão. Orientadores possuem mais propriedades atreladas ao contexto do trabalho que os avaliadores.

- **Professor:** Todo orientador e avaliador é um professor. Possui informações pertinentes para identificação e análise para escolha de orientação.
- **Situação do trabalho:** Serve para identificar qual estágio dentro do trabalho de conclusão de curso o aluno está inserido.
- **Trabalho:** Entidade central de todo o contexto, possui informações e referências importantes para os contextos dos demais serviços.

Após reunir os termos para a construção do modelo de informação e exploração do domínio, recursos são criados para a análise da aplicação. Assim como Hippchen *et al*. (2017), o BDD foi utilizado nessa fase, aderindo à linguagem Gherkin. Uma série de funcionalidades são descritas para auxiliar na implementação e testes do projeto modelo. Na Figura 23 é disponibilizado um exemplo de recurso implementado. No Quadro 4 são descritos os recursos e cenários criados.

Figura 23 – Cenários do recurso Situação

```
Feature: Situacao
    Funcionalidades para o contexto de situação de trabalho.

Scenario: Alterar um compromisso.
    Given o administrador precisa alterar um compromisso.
    And insere todos os dados com as alterações.
    When envia todas as informações.
    Then o compromisso é alterado com sucesso.

Scenario: Alterar uma situação.
    Given o administrador precisa alterar uma situação.
    And insere todos os dados com as alterações.
    When envia todas as informações.
    Then a situação é alterada com sucesso.

Scenario: Incluir um novo compromisso.
    Given o administrador precisa incluir um novo compromisso.
    And insere todos os dados necessários.
    When envia todas as informações.
    Then o compromisso é incluído com sucesso.

Scenario: Incluir uma nova situação.
    Given o administrador precisa incluir uma nova situação.
    And insere todos os dados necessários.
    When envia todas as informações.
    Then a situação é incluída com sucesso.
```

Fonte: Elaborado pelo autor

Quadro 4 – Recursos e cenários

Recursos	Cenários
Aluno	Alterar um aluno.
	Alterar uma ata.
	Autorizar ata pelo orientador.
	Incluir um novo aluno.
	Incluir uma nova ata.
	Obter todos os alunos.
Curso	Alterar um instituto.
	Alterar um currículo.
	Alterar um curso.
	Alterar uma disciplina.
	Incluir um novo instituto.
	Incluir um novo currículo.
	Incluir um novo curso.
	Incluir uma nova disciplina.
Professor	Adicionar um curso ao professor.
	Adicionar uma disciplina ao professor.
	Alterar um professor.
	Incluir um novo professor.
	Remover um curso ao professor.
	Remover uma disciplina ao professor.
Situação	Alterar um compromisso.
	Alterar uma situação.
	Incluir um novo compromisso.
	Incluir uma nova situação.
Trabalho	Adicionar um avaliador ao trabalho.
	Alterar dados básicos de um trabalho.
	Alterar um trabalho completo
	Alterar uma situação do trabalho.
	Avaliar um trabalho.
	Incluir um documento para situação.
	Incluir um novo trabalho.
	Incluir uma nova situação para o trabalho.
	Obter todos os trabalhos.
	Obter um trabalho.
	Remover um avaliador ao trabalho.

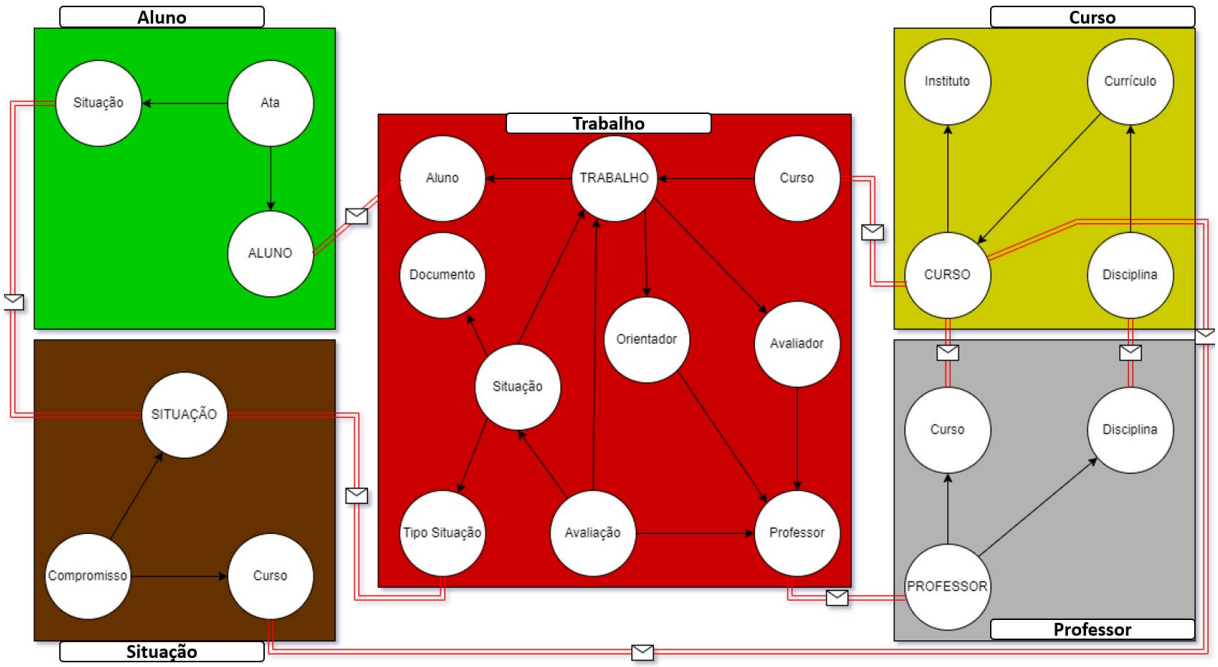
Fonte: Elaborado pelo autor

7.1.2 *Design* do projeto

Após a fase de elicitação de requisitos, o *design* do projeto é iniciado. O *design* do domínio é realizado, agrupando os termos de acordo com o domínio central, formando assim contextos delimitados. Cada contexto delimitado tem suas entidades conectadas através de referências, que permitem visualizar quais entidades são classificadas como raízes agregadas. Os agregados realizam a comunicação com outro micro serviço. O projeto modelo inicial é constituído por cinco micro serviços que possuem responsabilidades distintas e se comunicam

através de mensagens. O mapa de contextos elaborado é apresentado na Figura 24. O mapa de contextos do projeto modelo é a visão elaborada como o primeiro passo do modelo de recomendação.

Figura 24 – Mapa de contextos do projeto modelo

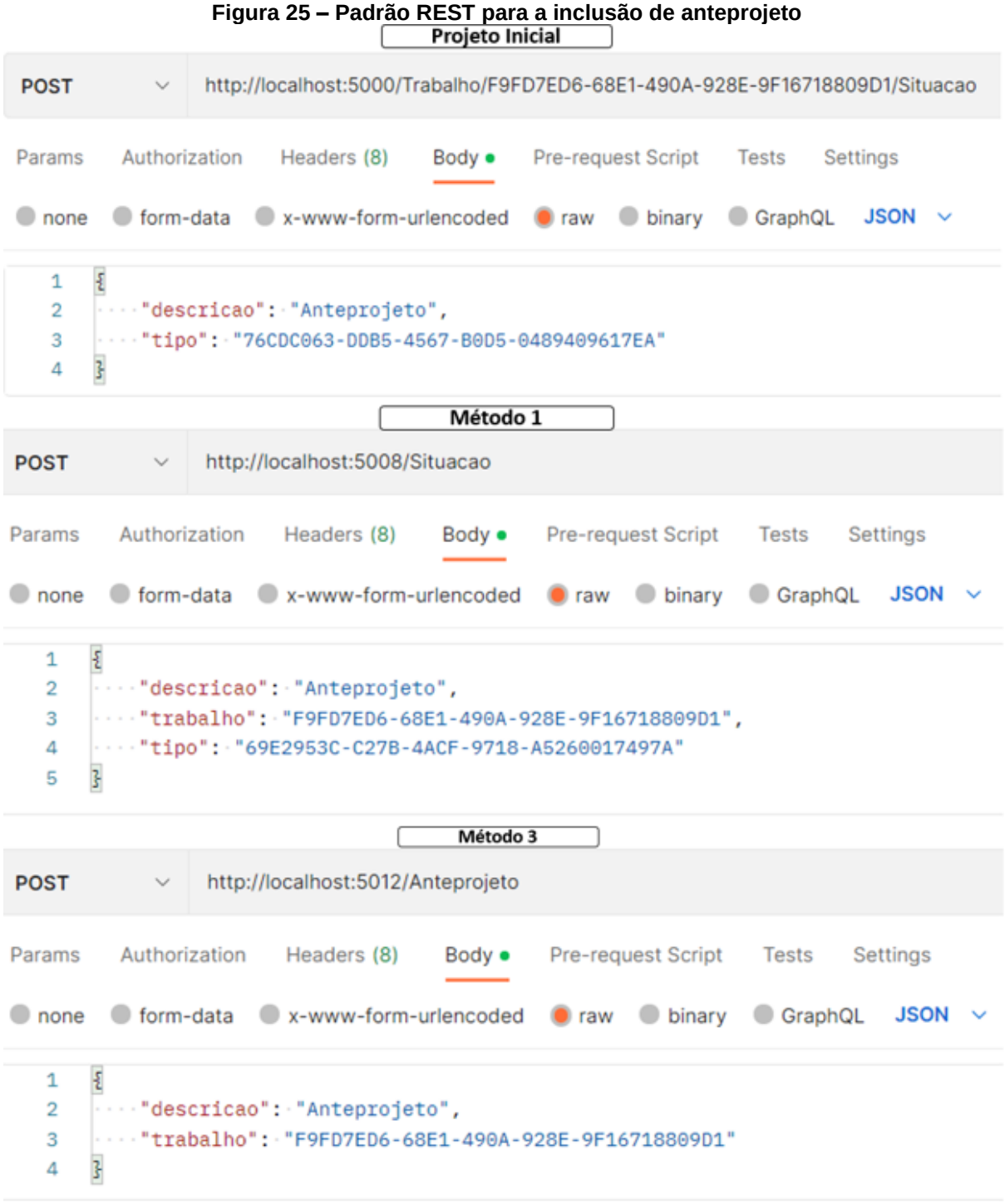


Fonte: Elaborado pelo autor

7.1.3 Implementação e testes

Assim como no trabalho de Hippchen *et al*. (2017), o projeto é implementado utilizando a arquitetura *onion*, possuindo a aplicação como a camada mais exposta do projeto. Sendo assim, a camada de interface de usuário e anticorrupção foram desconsideradas. A vantagem aqui é limitar os testes somente ao desempenho dos micro serviços, não tendo impacto de processamento de camadas de interação com o usuário. Antes da implementação, é realizado o *design* da API REST, responsável por receber uma requisição para os serviços e retornar o que foi solicitado. Para isso, o padrão *OpenAPI* é respeitado no *design* de API. Na Figura 25, o modelo de chamada para a inclusão da situação de anteprojeto de um trabalho é apresentado.

Figura 25 – Padrão REST para a inclusão de anteprojeto



Fonte: Elaborado pelo autor

A chamada HTTP muda de acordo com a decomposição realizada no micro serviço. No projeto inicial é necessário informar na URI o trabalho ao qual a situação anteprojeto será inserida. Para a decomposição do serviço com um número elevado de entidades de responsabilidades únicas, a requisição é enviada diretamente para o novo micro serviço, passando o identificar do trabalho no corpo da requisição. Por fim, para o quarto passo, a abstração da situação é removida, sendo a requisição

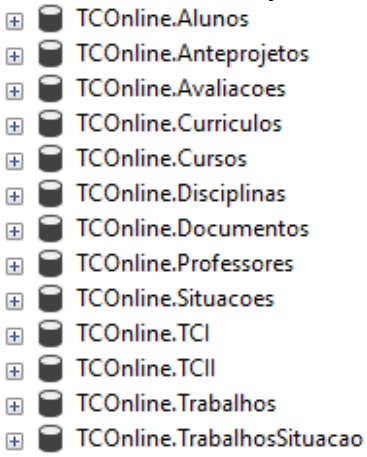
enviada diretamente para o micro serviço anteprojeto, não sendo necessário informar um identificador de tipo no corpo da requisição.

Após realizar o *design* da API, são construídos quatro projetos que podem ser visualizados através do link disponibilizado no Apêndice A. As construções utilizam os mapas de contextos oriundos de cada um dos passos do modelo de recomendação, são eles:

- **TCC.ProjetoInicial:** Micro serviços obtidos através da execução do primeiro passo.
- **TCC.Passo2:** Projeto com os micro serviços contidos no modelo visual após a realização do segundo passo.
- **TCC.Passo3:** Projeto com os micro serviços contidos no modelo visual após a realização do terceiro passo.
- **TCC.Passo4:** Projeto com os micro serviços contidos no modelo visual após a realização do quarto passo.

Cada micro serviço possui apenas um controlador na camada de aplicação, responsável por receber as requisições dos clientes, serializados em DTOs, e realizar os processos e validações de acordo com as regras contidas na camada de domínio. Caso seja necessária a persistência de dados, a camada de infraestrutura efetua o procedimento com o banco de dados. Este processo é idêntico ao descrito na seção 5.3.3. Vale ressaltar que cada micro serviço possui sua base de dados separadamente, conforme mostrado na Figura 26. Após a implementação, um processo de interação com os testes foi iniciado tendo como base os recursos gerados com o BDD na fase de análise da aplicação.

Figura 26 – Lista de bases de dados para cada micro serviço



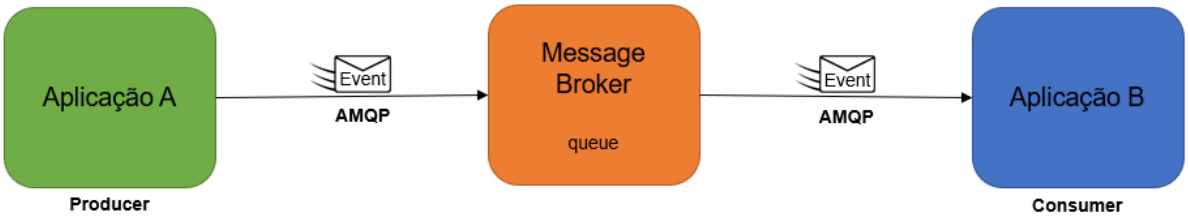
Fonte: Elaborado pelo autor

Durante o processo de desenvolvimento, Hippchen *et al.* (2017) não apontam como os micro serviços realizam a comunicação entre eles. A comunicação entre as raízes agregadas dos micro serviços é um fator crítico quando discutido o desempenho da arquitetura. Pois cada micro serviço necessita de uma série de dados que são administrados por outros serviços. Portanto, a troca de informações é necessária para o bom funcionamento, podendo gerar gargalos na comunicação e inconsistência nos dados.

O método de comunicação entre os serviços mais utilizado dentro da comunidade é através de um mecanismo de mensageria. Em sistemas distribuídos, mensageria é a comunicação realizada por troca de mensagens, geralmente por meio de registros de eventos. As mensagens são gerenciadas por um *Message Broker*, responsável por intermediar a comunicação entre os serviços, enfileirando as mensagens e garantindo que os clientes interajam com a fila. É como uma caixa de correio, onde as mensagens são cartas depositadas e retiradas por alguém que tenha interesse em ler essas cartas.

Conforme mostrado na Figura 27, o *Message Broker* possui uma *queue* responsável por armazenar mensagens geradas por um *producer*. O *producer* nada mais é do que a aplicação que envia mensagens através de *events*. Todo *event* é enviado através de um protocolo de comunicação, sendo geralmente utilizado o *Advanced Message Queuing Protocol* (AMQP). O mesmo padrão de *event* é utilizado para o envio da mensagem para os *consumers*, que são as aplicações que possuem interesse na mensagem.

Figura 27 – Fluxo de mensageria



Fonte: Elaborado pelo autor

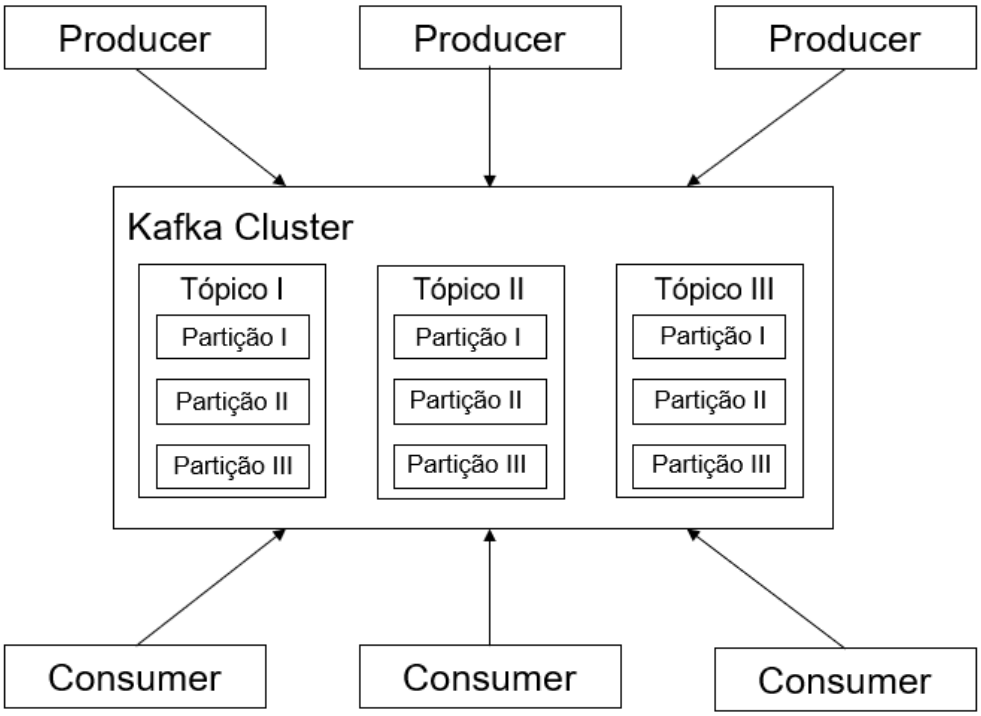
Buscando minimizar ao máximo problemas decorrentes da troca de informações entre os micro serviços, a ferramenta Apache Kafka[1] foi adotada. Trata-se de uma plataforma de *streaming* de eventos distribuídos, sendo comumente utilizado para *pipelines* de dados de alto desempenho, análise de *streaming*, integração de dados e aplicativos de missão crítica. Tornando capaz publicar, subscrever, armazenar e processar fluxos de registros em tempo real.

Kafka realiza a transmissão das mensagens de forma assíncrona. Em sua estrutura, possui *consumers* e *producers* orquestrados por um *cluster*, que agrupa um conjunto de *brokers*, contendo tópicos e partições, conforme mostrado na Figura 28. Kafka é projetado para suportar quedas e falhas, quando um determinado *broker* fica fora do ar, um *broker* reserva assume as atividades para manter a entrega das mensagens e a alta disponibilidade, sem perder nenhuma informação. O mesmo comportamento é aplicado nos tópicos, que possuem múltiplas partições, replicadas a partir de uma partição principal.

Cada *consumer* é responsável por buscar os eventos enviados para seus tópicos de interesse, e realizar seus processos de acordo com a informação. Caso alguma erro ocorra durante o processo, o *consumer* deve informar ao tópico, para que os demais interessados possam utilizar a informação conforme o esperado. Aplicado no contexto do trabalho, na Figura 29 é demonstrado o fluxo a partir do *producer* curso, enviado *events* para os seus *consumers*.

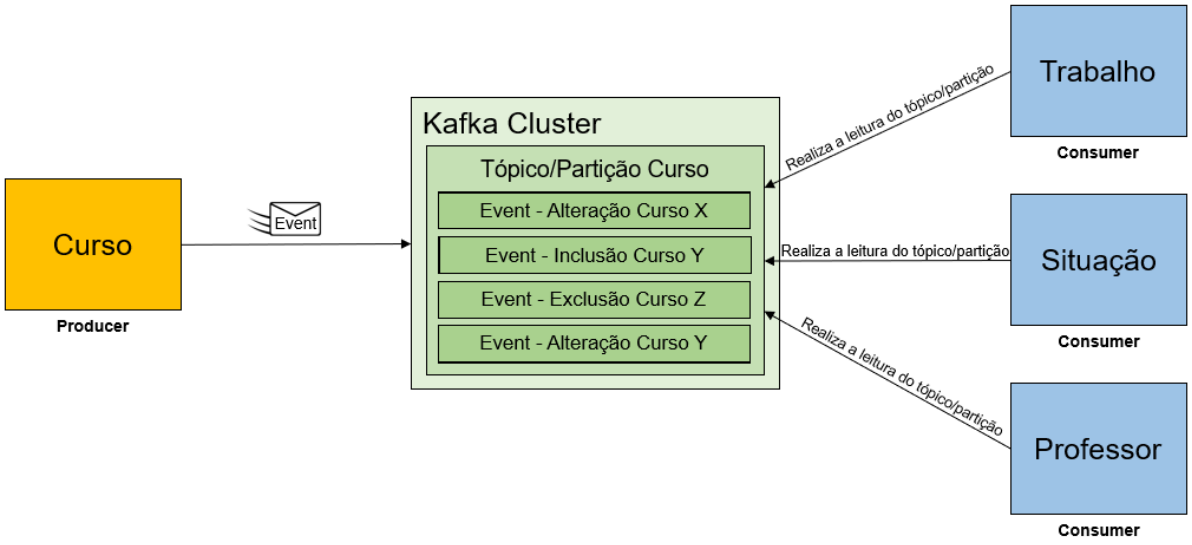
[1] A ferramenta Apache Kafka pode ser obtida através do link https://kafka.apache.org/.

Figura 28 – Fluxo do Apache Kafka



Fonte: Elaborado pelo autor

Figura 29 – Kafka aplicado no projeto modelo inicial



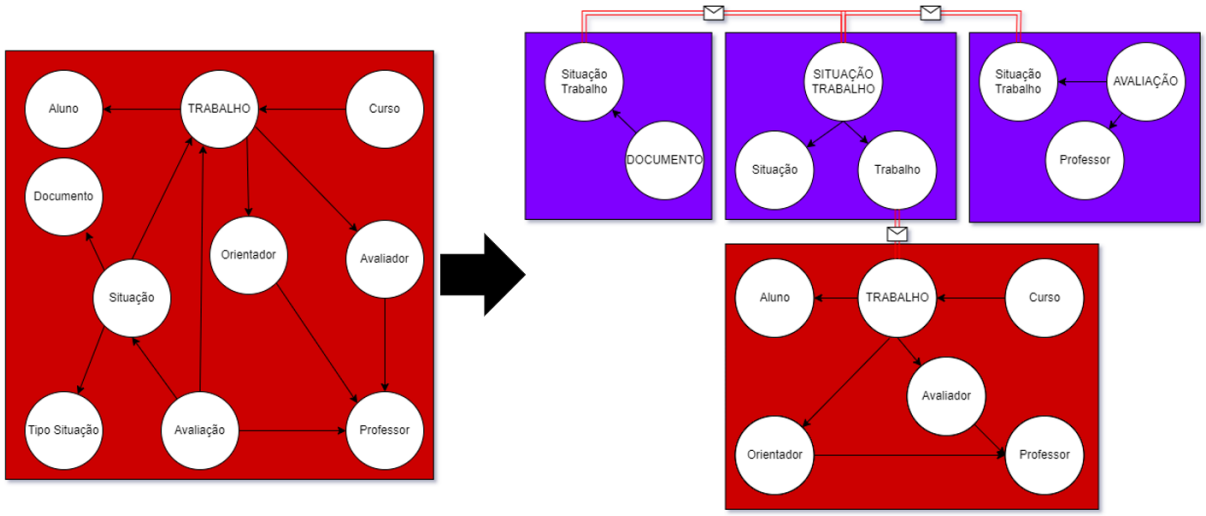
Fonte: Elaborado pelo autor

7.2 QUANTIDADE DE ENTIDADES COM RESPONSABILIDADES ÚNICAS

A partir do modelo de visão gerado na seção 7.1.2, é contabilizado o número de entidades com responsabilidades únicas para cada micro serviço. Micro serviços que possuem mais de três entidades identificadas são isolados para que o segundo passo do modelo de recomendação seja realizado. As entidades aptas a possuir

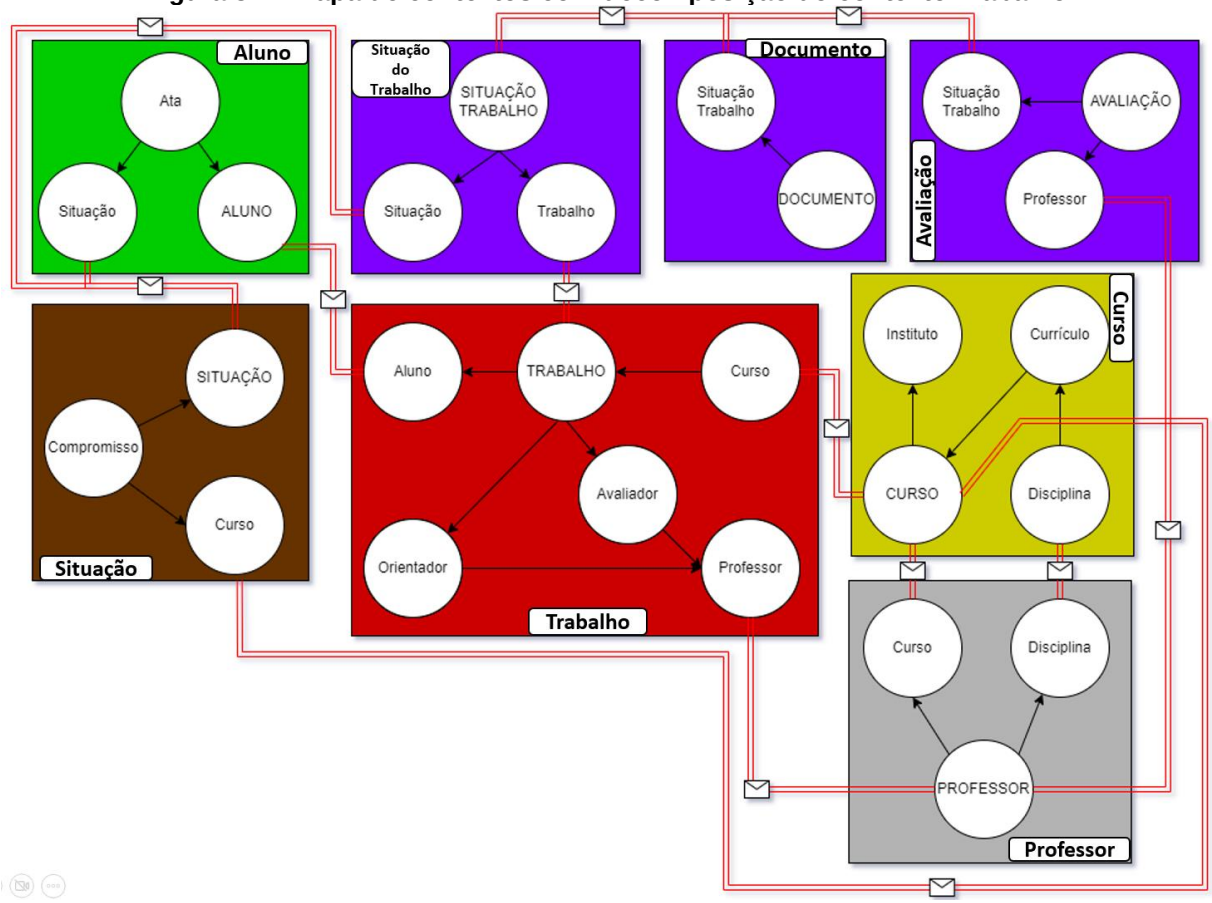
autonomia com relação ao contexto atual, são transformadas em micro serviços, auxiliando na escalabilidade da aplicação. No projeto modelo, o contexto de trabalho é o que possui mais responsabilidades entre suas entidades. Para reduzir a granularidade nesse contexto, as entidades de avaliação, situação do trabalho e documento são separadas do contexto trabalho e projetadas em novos micro serviços. Na Figura 30 é demonstrado a modificação e na Figura 31 o mapa de contextos é atualizado.

Figura 30 – Decomposição do contexto Trabalho



Fonte: Elaborado pelo autor

Figura 31 – Mapa de contextos com decomposição do contexto Trabalho



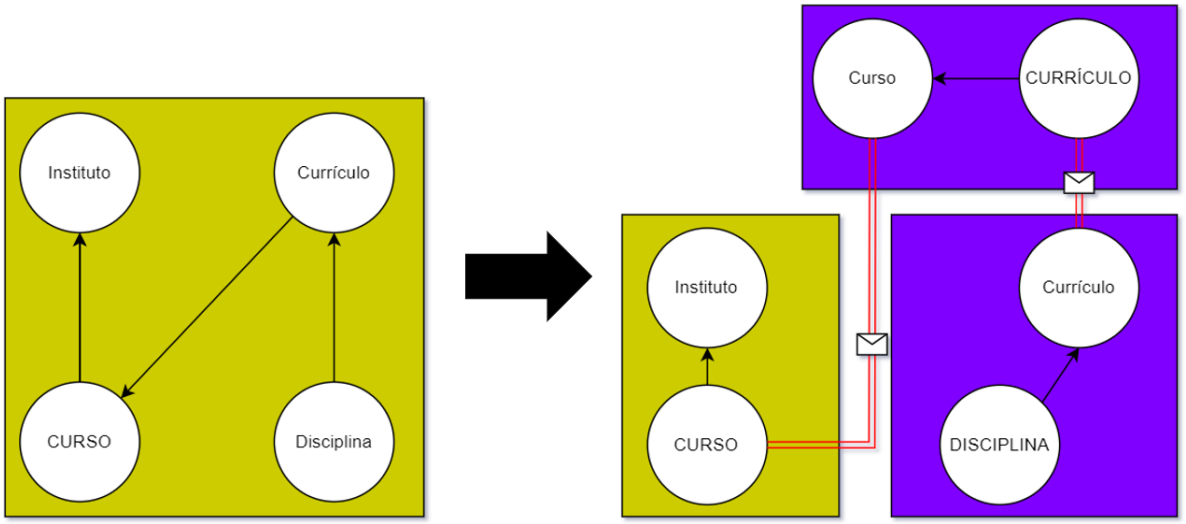
Fonte: Elaborado pelo autor

7.3 QUANTIDADE DE RAÍZES AGREGADAS

O terceiro passo do modelo de recomendação é localizar o micro serviço que realiza o maior número de comunicações com outros serviços, ou seja, aquele com diversas raízes agregadas dentro do mapa de contextos. Aquele micro serviço que necessita realizar interações com vários outros serviços existentes no contexto geral. No projeto modelo, é possível perceber que, quando ocorre uma alteração no contexto de curso, é necessário comunicar essa alteração para os contextos de trabalho, professor e situação.

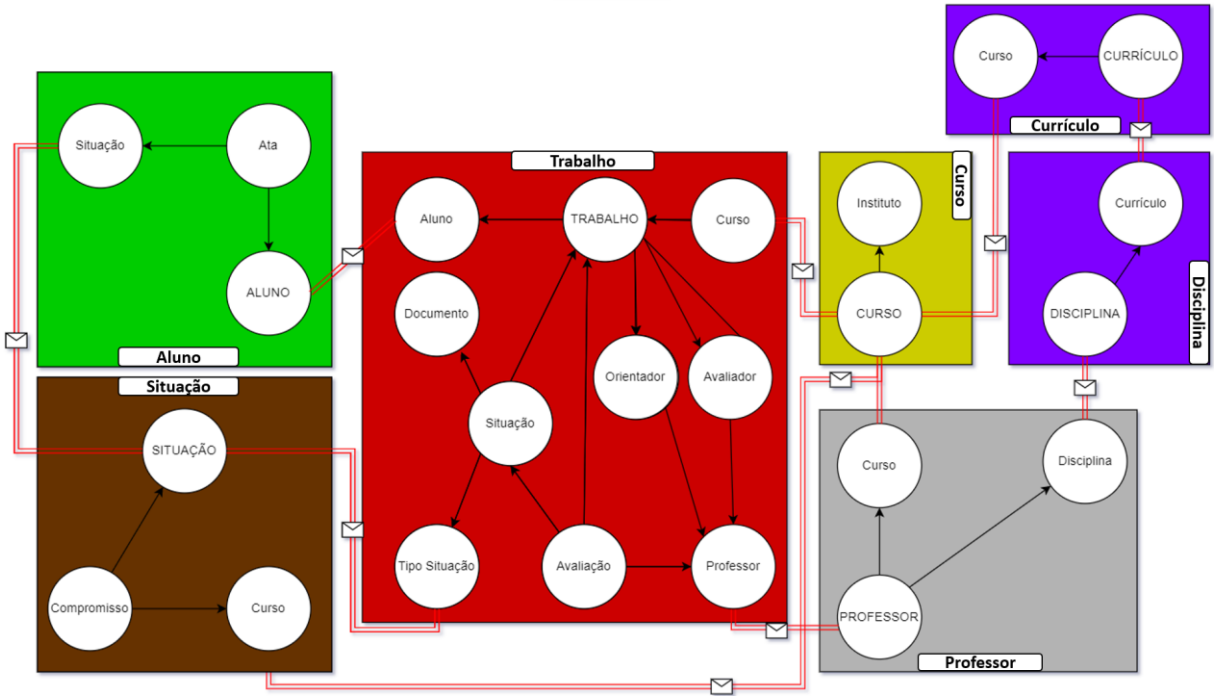
Essa tarefa requer uma certa complexidade técnica, pois exige maior esforço da comunicação deste micro serviço com os demais. Para atender o terceiro passo, as entidades currículo e disciplina são transformadas em serviços específicos dentro do contexto geral, reduzindo as responsabilidades do micro serviço de curso. Na Figura 32 é demonstrada a decomposição do micro serviço de curso e na Figura 33 o mapa de contextos é atualizado.

Figura 32 – Decomposição do contexto Curso



Fonte: Elaborado pelo autor

Figura 33 – Mapa de contextos com decomposição do contexto Curso



Fonte: Elaborado pelo autor

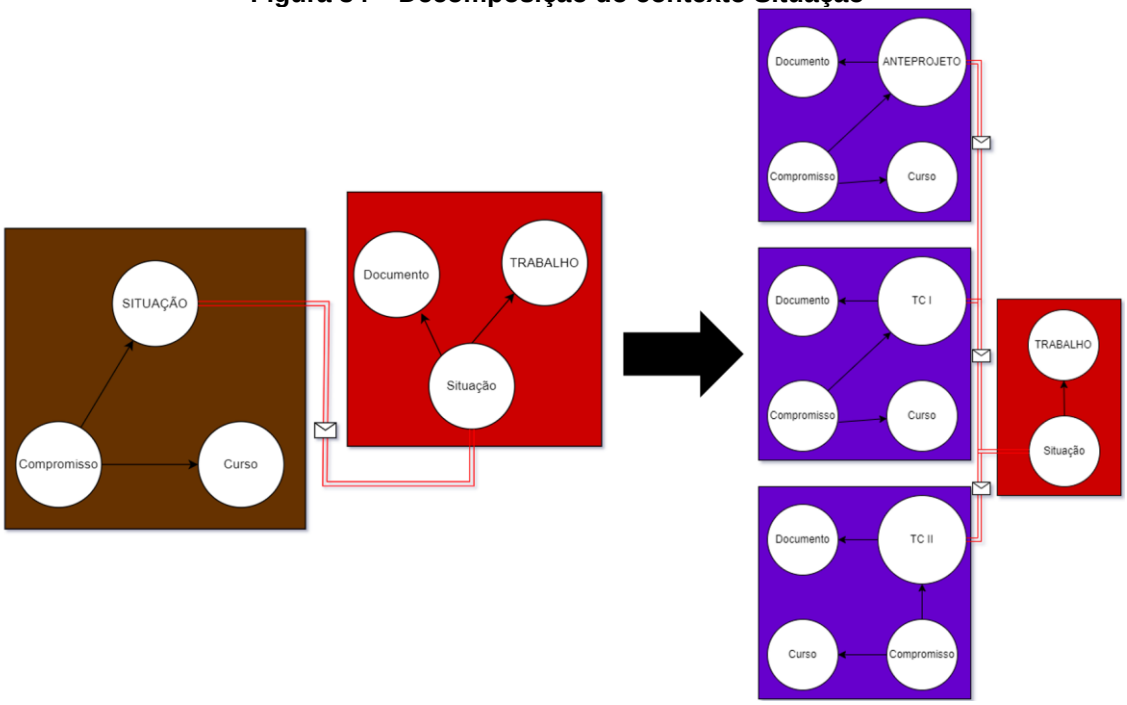
7.4 DECOMPOSIÇÃO DE ABSTRAÇÕES

A intenção do quarto passo é especificar micro serviços que têm em sua essência uma abstração, ou seja, que possuam um significado genérico dentro do mapa de contextos. No projeto modelo inicial, as etapas de anteprojeto, trabalho de conclusão I e trabalho de conclusão II, são classificadas como situações do trabalho e são tratadas como um único micro serviço. Para remover essa abstração o mapa

de contextos é repensado para que o contexto de situação seja substituído por três novos contextos, um para cada situação de trabalho.

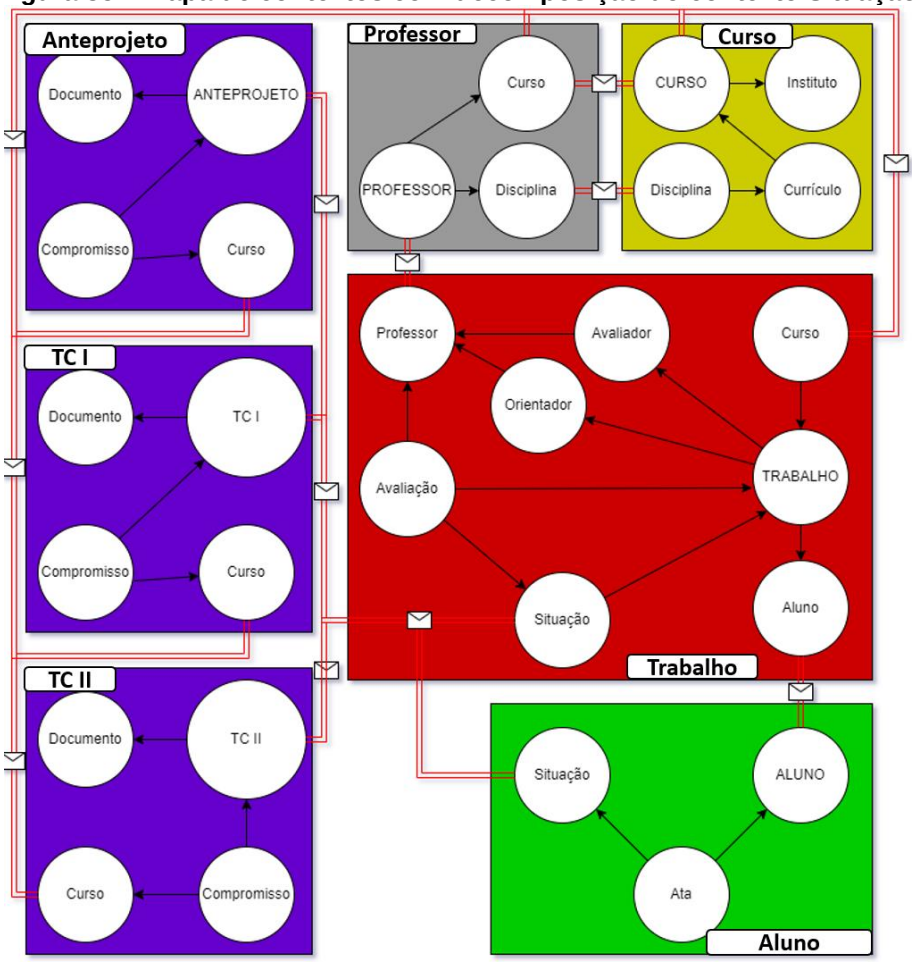
Note que esse passo é o único que, além de incluir novos micro serviços, aplica a remoção de um micro serviço já existente. Isso implica na complexidade de implementação em projetos consolidados, pois a remoção de um micro serviço impacta diretamente no formato de comunicação entre os serviços. A comunicação que antes era abstrata, agora necessita ser especificada e orquestrada. Na Figura 34 é demonstrado a remoção da abstração da situação do trabalho e na Figura 35 o mapa de contextos é atualizado.

Figura 34 – Decomposição do contexto Situação



Fonte: Elaborado pelo autor

Figura 35 – Mapa de contextos com decomposição do contexto Situação



Fonte: Elaborado pelo autor

7.5 CONSIDERAÇÕES DO CAPÍTULO

Até aqui foram elaborados quatro mapas de contextos com diferentes níveis granulares entre eles. Cada mapa foi devidamente projetado, implementado e testado seguindo os passos descritos por Hippchen *et al*. (2017) no capítulo 5. No capítulo 8 os modelos são submetidos a testes de carga para análise dos resultados.

8 TESTES DE CARGA

Uma maneira de realizar testes com relação a performance de um micro serviço é submetê-lo a um número de requisições por segundo para os *endpoints* fornecidos pela camada de aplicação. Esse tipo de teste é conhecido como carga e é qualificado como teste estrutural. Seu objetivo é determinar quais transações poderão impactar na execução da aplicação, definindo a configuração de arquitetura mínima que permitirá ao software atender ao que foi especificado, minimizando os riscos relacionados aos requisitos de desempenho.

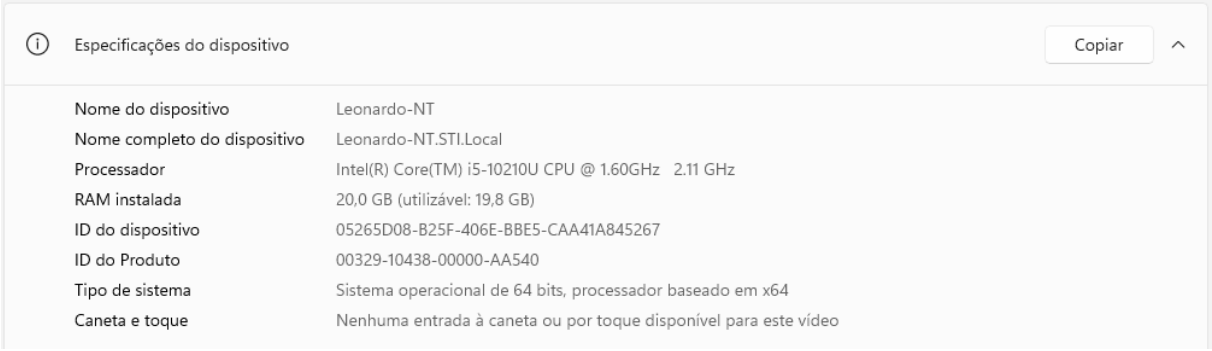
Realizando simulações com diferentes números de usuários e transações simultâneas, é possível fazer análises, monitorar os resultados e fornecer indicadores importantes sobre a capacidade de carga que a aplicação pode suportar. Adequando ao escopo deste trabalho, o projeto modelo inicial e os passos de alteração no nível granular dos micro serviços, apresentados no capítulo 6, são submetidos a uma série de requisições com diferentes quantidades de usuários por segundo, executadas em baterias de dez execuções para cada quantidade de usuários.

8.1 CONDIÇÕES E CONFIGURAÇÕES DE TESTES

Os testes são executados localmente em uma máquina com um processador Intel I5 de 10ª geração e 20Gb de memória RAM. Os bancos de dados encontram-se na mesma máquina e todos os testes são executados sem nenhum processamento paralelo. Na Figura 36 estão as especificações da máquina. Para a realização dos testes é utilizada a ferramenta *Open Source* da Apache, conhecida como JMeter. Essa ferramenta possui uma série de funcionalidades, mas para o escopo deste trabalho serão aplicados testes estáticos de carga, consumindo as APIs REST de cada um dos micro serviços desenvolvidos.

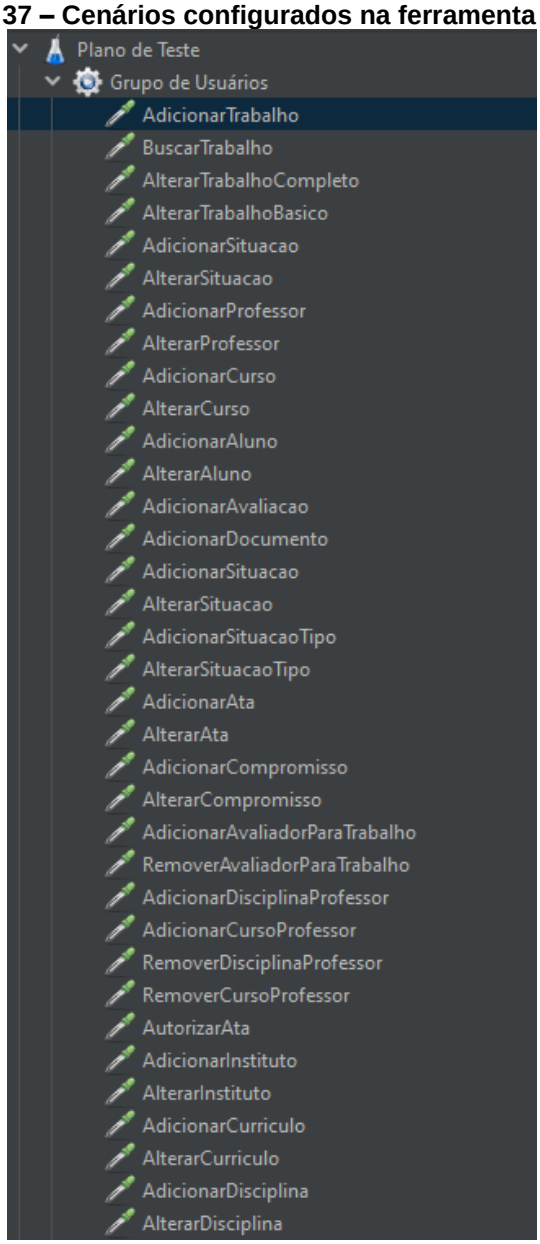
Os cenários criados na análise da aplicação, através do BDD, e listados no Quadro 4, são configurados e especificados como requisições de teste. Cada um dos recursos consome seu respectivo micro serviço e possui seu próprio comportamento, ou seja, segue o seu fluxo de requisições e não depende de outro cenário de teste para ter sucesso. Na Figura 37 é demostrada a lista de cenários configuradas dentro da ferramenta JMeter.

Figura 36 – Especificações da máquina de teste



Fonte: Elaborado pelo autor

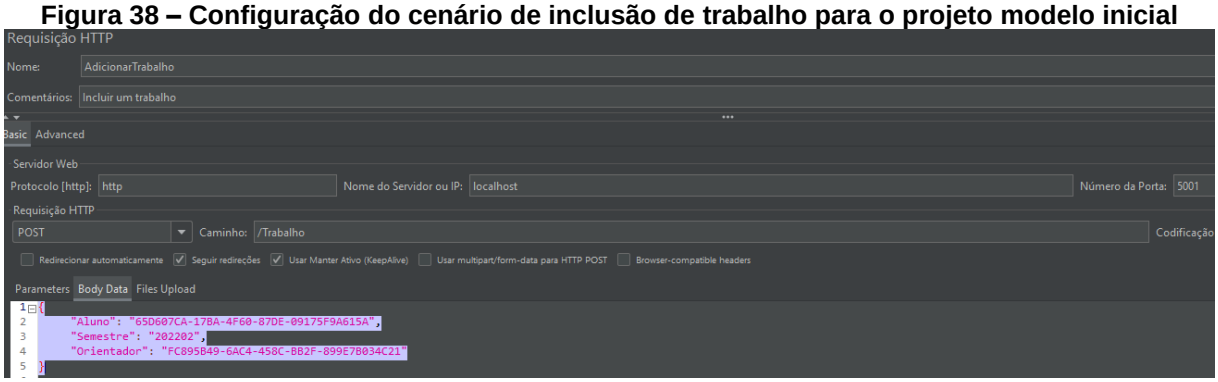
Figura 37 – Cenários configurados na ferramenta JMeter



Fonte: Elaborado pelo autor

Para cada um dos cenários é configurado o URI do micro serviço, através do protocolo HTTP e sua respectiva porta. Seguindo os padrões de APIs REST o verbo é definido e, caso seja necessário, o *body* é preenchido com os DTOs esperados através de JSON. Todos os 35 cenários são testados nos quatro projetos desenvolvidos, as alterações são apenas de rotas, conforme Figura 25, para que cada projeto tenha seus micro serviços devidamente utilizados de acordo com o mapa de contextos. A configuração da inclusão de um trabalho no projeto modelo inicial é mostrada na Figura 38. A ordem de configuração das chamadas para cada cenário é planejada para que nenhuma falha decorrente de possíveis gargalos do mecanismo de mensageria ocorra. As trocas de mensagens, dado o número baixo de usuários simultâneos, não impactam em nenhuma requisição dentro dos micro serviços.

Figura 38 – Configuração do cenário de inclusão de trabalho para o projeto modelo inicial



Fonte: Elaborado pelo autor

8.2 EXECUÇÃO DOS TESTES

Para cada projeto desenvolvido de acordo com os passos do modelo de recomendação, são aplicadas três cargas de usuários. Devido às limitações da máquina, as cargas foram de 10, 25 e 50 usuários simultâneos. Os usuários iniciam as requisições no mesmo segundo e realizam uma interação com cada cenário de teste. Essa configuração é executada 10 vezes para cada quantia de usuários simultâneos. Informações como média, mediana e desvio padrão são recolhidas em milissegundos para a análise. A Tabela 1 apresenta todos os resultados obtidos.

Tabela 1 – Resultados dos testes de carga

Proj.	Usu.	Vlr.	1º	2º	3º	4º	5º	6º	7º	8º	9º	10º
Ini	10	Média	180	183	175	171	202	169	209	174	204	176
		Desvio	150	125	142	142	153	137	114	131	121	124
		Mediana	132	147	119	120	145	125	197	136	193	144
	25	Média	401	368	396	374	392	445	411	415	412	414
		Desvio	349	330	337	326	371	356	354	402	403	343
		Mediana	278	235	282	252	241	371	326	246	285	301
	50	Média	763	853	845	775	809	852	755	861	812	890
		Desvio	1002	845	787	933	918	866	1265	852	933	849
		Mediana	331	480	532	485	417	405	392	522	425	565
1º Mtd	10	Média	195	193	196	179	166	162	169	170	189	197
		Desvio	138	148	143	158	134	137	146	151	140	144
		Mediana	158	137	145	119	114	112	113	110	138	151
	25	Média	362	353	358	361	370	377	387	361	383	359
		Desvio	400	375	412	438	420	415	441	381	415	440
		Mediana	245	227	211	240	233	230	215	235	231	225
	50	Média	590	683	651	817	730	719	654	677	721	673
		Desvio	1050	963	942	982	1010	1074	1313	1053	1030	988
		Mediana	287	260	280	471	275	265	283	431	425	370
2º Mtd	10	Média	186	185	183	215	187	176	211	204	191	173
		Desvio	171	161	138	175	152	161	148	142	144	150
		Mediana	50	52	56	159	130	114	163	175	150	112
	25	Média	425	417	415	421	397	407	396	390	397	396
		Desvio	331	311	302	298	353	392	364	388	373	322
		Mediana	238	259	235	265	246	263	279	171	229	241
	50	Média	702	812	739	749	777	813	738	825	809	789
		Desvio	881	822	699	874	666	835	904	889	797	783
		Mediana	259	309	277	299	303	280	299	320	287	359
3º Mtd	10	Média	262	221	206	193	208	202	199	211	197	203
		Desvio	242	150	189	144	157	172	179	157	156	168
		Mediana	140	171	129	138	167	142	127	159	140	143
	25	Média	508	454	465	458	471	468	393	416	416	432
		Desvio	465	416	525	481	443	448	453	457	535	517
		Mediana	351	314	307	291	308	325	276	299	241	254
	50	Média	875	718	945	813	955	1088	849	794	809	868
		Desvio	1271	1317	1046	1238	1419	1314	1015	1123	995	936
		Mediana	511	357	597	531	466	577	500	358	402	483

Fonte: Elaborado pelo autor

8.3 CONSIDERAÇÕES DO CAPÍTULO

A amostragem para cada número de usuários simultâneos são de 350, 875 e 1750 respectivamente. Esses números se dão pela soma de requisições realizadas por cada usuário. Com a configuração e execução dos testes, os resultados são coletados e são analisados no capítulo 9.

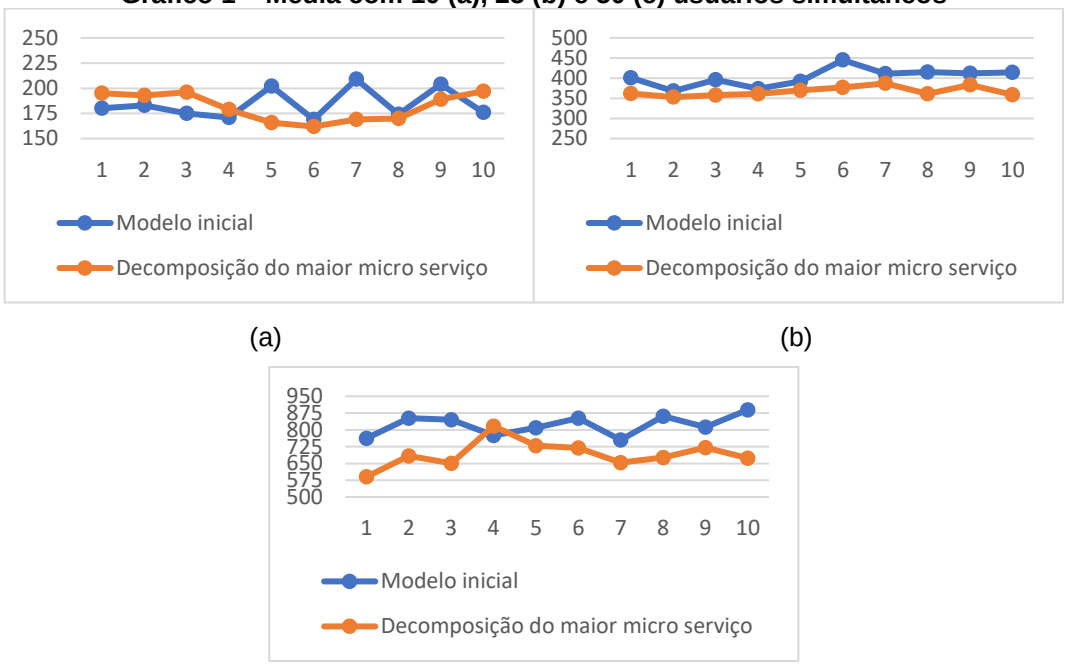
9 ANÁLISE DOS RESULTADOS

Para construir um modelo de recomendação para definição de nível granular em micro serviços utilizando DDD, os resultados gerados e apresentados na Tabela 1 são analisados comparando cada um dos passos com relação ao modelo inicial apresentado no capítulo 7. O primeiro passo é responsável por tornar visível todo o mapa de contextos do projeto inicial. Portanto, a análise é realizada a partir do segundo passo.

9.1 QUANTIDADE DE ENTIDADES COM RESPONSABILIDADES ÚNICAS

Ao comparar os resultados obtidos entre o projeto inicial e o projeto referente a aplicação do segundo passo, é perceptível uma melhora na média entre os testes de carga. Porém, o desvio padrão aponta um aumento da variação dos resultados em torno da média, isso impacta diretamente na performance das requisições, pois elas tendem a possuir tempos de resposta discrepantes. Para visualização dos resultados são gerados gráficos para cada uma das três quantidades de usuários simultâneos, utilizados nos testes de carga. No Gráfico 1 são apresentadas as comparações das médias entre os projetos. As comparações do desvio padrão estão contidas no Gráfico 2.

Gráfico 1 – Média com 10 (a), 25 (b) e 50 (c) usuários simultâneos

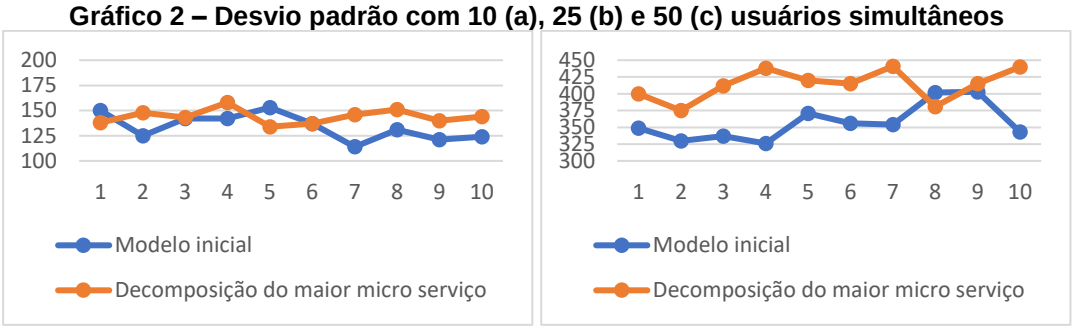


(a) (b)

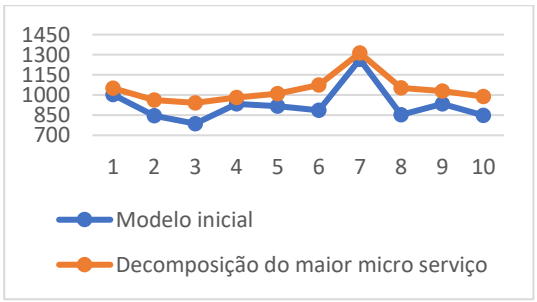
(c)

Fonte: Elaborado pelo autor

Gráfico 2 – Desvio padrão com 10 (a), 25 (b) e 50 (c) usuários simultâneos



(a) (b)

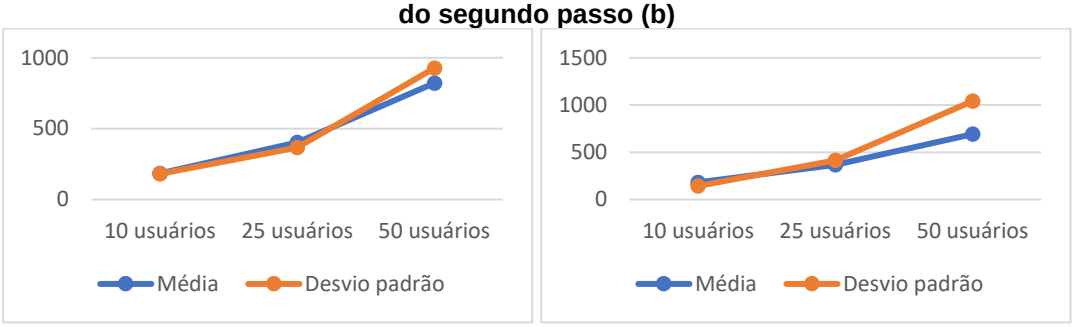


(c)

Fonte: Elaborado pelo autor

No Gráfico 3, são apresentados os crescimentos da média e do desvio padrão, para o projeto inicial e o projeto do segundo passo, com relação ao número de amostras. Tornado mais visível o aumento exponencial do desvio padrão com relação à média.

Gráfico 3 – Crescimento a partir do aumento da amostragem para projeto inicial (a) e projeto do segundo passo (b)



(a) (b)

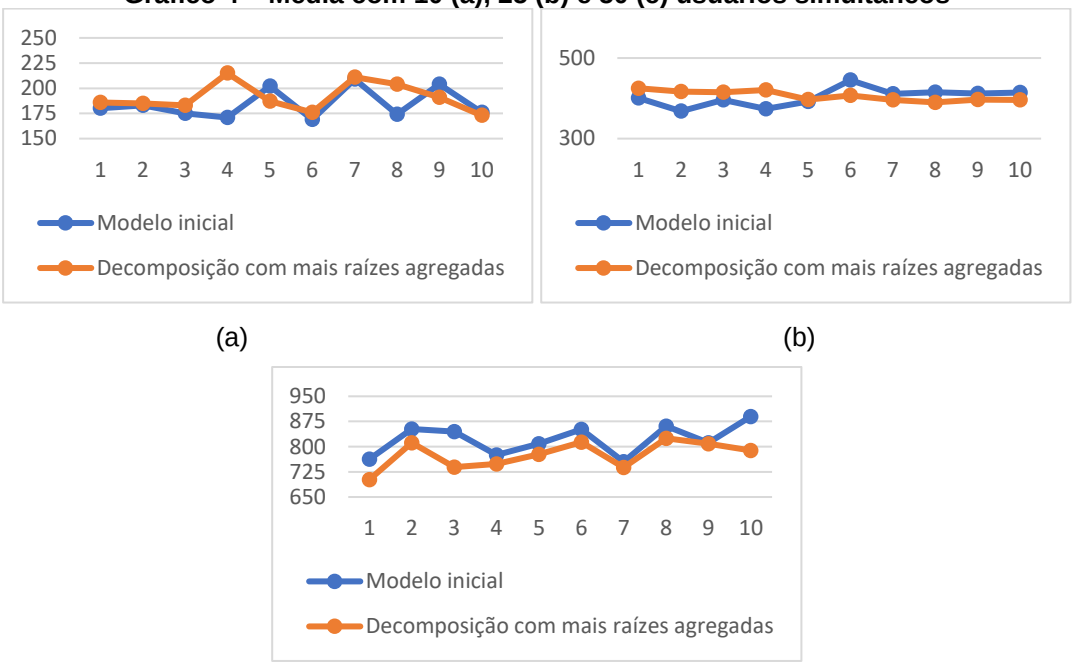
Fonte: Elaborado pelo autor

Os testes de carga para o projeto que alterou o nível granular a partir da decomposição do micro serviço com mais de três entidades com responsabilidades únicas se mostrou pouco eficaz. O aumento gradativo do desvio padrão afeta a integridade do sistema, principalmente na troca de mensagens entre os micro serviços, causando o aumento no número de erros por inconsistência de dados entre entidades agregadas.

9.2 QUANTIDADE DE RAÍZES AGREGADAS

Ao decompor o projeto modelo inicial a partir do micro serviço que possui a maior quantidade de comunicação com outros serviços, os resultados apresentados pelos testes de carga com relação à média permanecem semelhantes, não demonstrando uma melhora significativa como visto no passo anterior. Porém, com o aumento das amostragens o desvio padrão diminui, garantindo assim uma maior confiabilidade da média. Como na seção anterior, as comparações entre o projeto inicial e a decomposição pelo micro serviço com mais raízes agregadas são apresentadas. No Gráfico 4 estão as médias e no Gráfico 5 estão os desvios padrão.

Gráfico 4 – Média com 10 (a), 25 (b) e 50 (c) usuários simultâneos

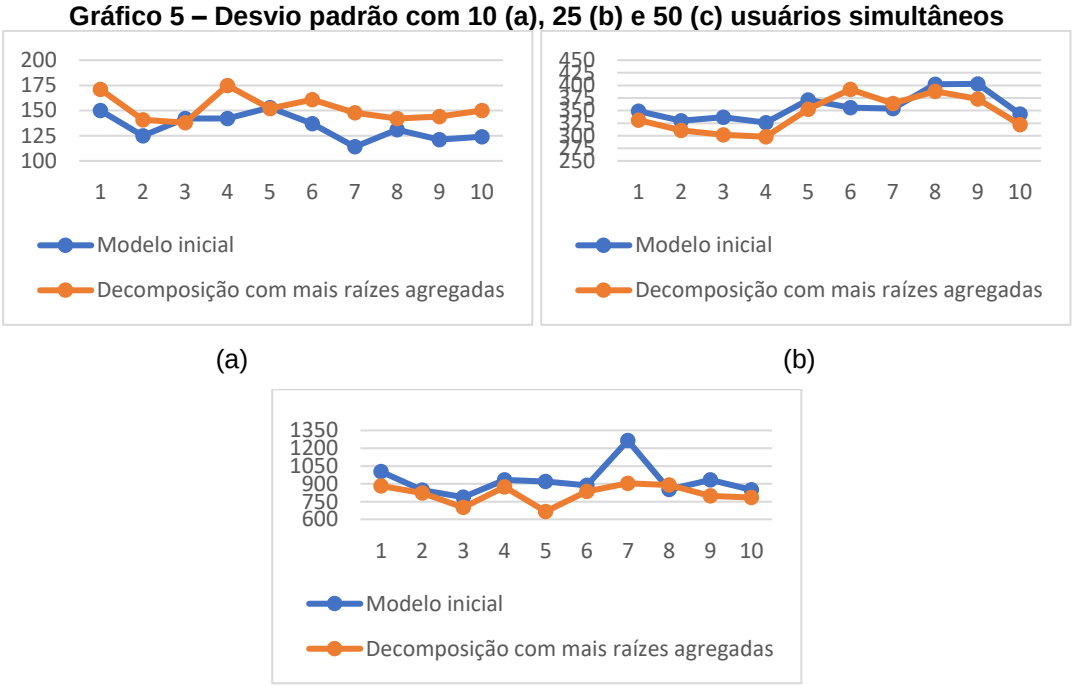


(a) (b)

(c)

Fonte: Elaborado pelo autor

Gráfico 5 – Desvio padrão com 10 (a), 25 (b) e 50 (c) usuários simultâneos



(a) (b)

(c)

Fonte: Elaborado pelo autor

Ajustar o nível granular de uma arquitetura de micro serviços a partir dos serviços que possuem mais raízes agregadas mostrou-se promissor, visto que tanto a média como o desvio padrão melhoram com o aumento de amostras,

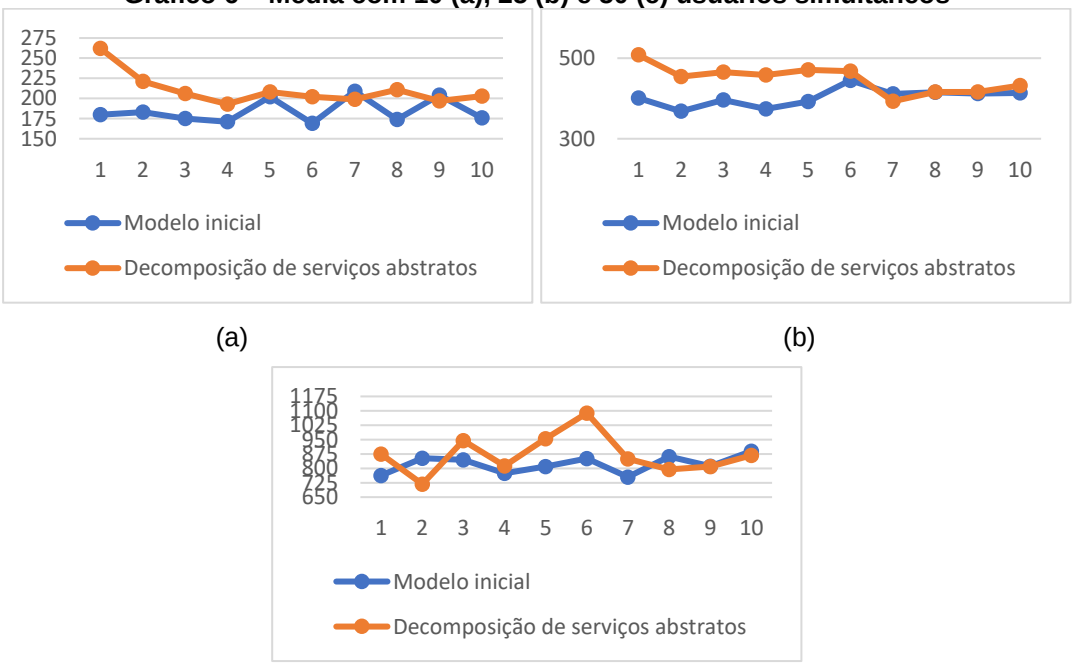
proporcionando assim serviços com tempo de resposta mais baixos e de menor latência.

9.3 DECOMPOSIÇÃO DE ABSTRAÇÕES

O quarto passo aplicado no projeto inicial foi reduzir a abstração de um micro serviço, transformando-o em contextos mais específicos dentro do modelo. Os resultados apresentados nos testes de carga, com relação ao projeto inicial, mostraram-se o mais ineficiente entre todos os passos abordados por esse trabalho. Pois tanto a média, como o desvio padrão, aumentam seu valor, independentemente da quantidade de amostras.

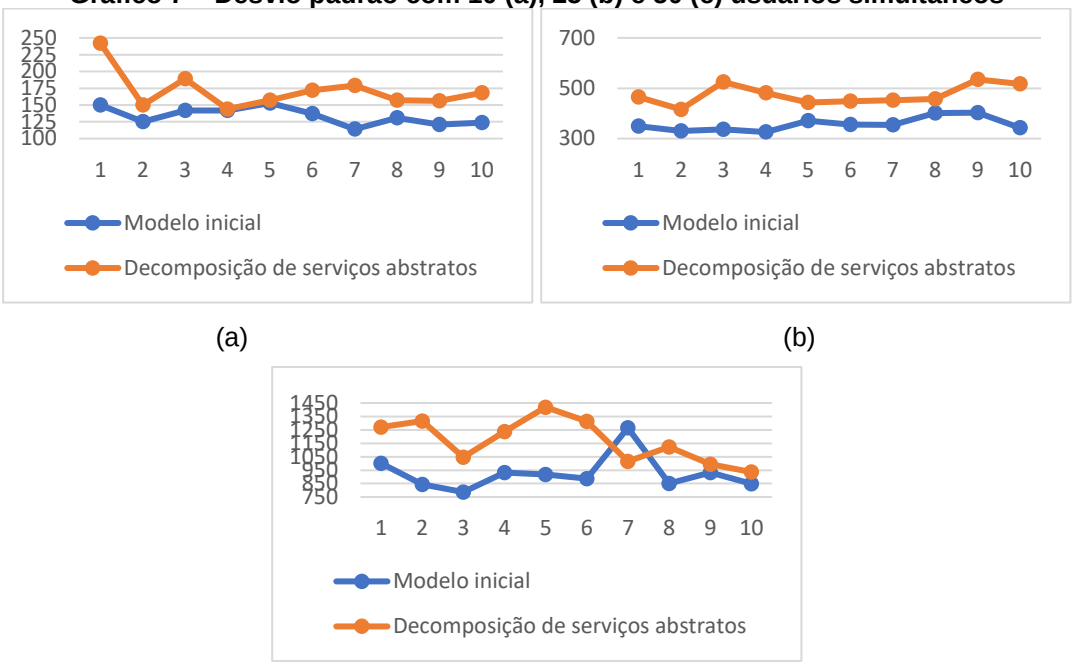
Apesar de ser o método que apresentou uma linearidade de valores, não pode ser considerado pois houve um aumento no tempo de execução comparado ao projeto inicial. Esse comportamento deve-se principalmente ao fato de que, quando removida a abstração, uma quantia maior de micro serviços são criados para atender ao modelo, gerando um acréscimo considerável no número de mensagens para a comunicação entre serviços. No Gráfico 6 são apresentados os valores comparativos para a média. Os desvios padrão são demonstrados no Gráfico 7.

Gráfico 6 – Média com 10 (a), 25 (b) e 50 (c) usuários simultâneos



Fonte: Elaborado pelo autor

Gráfico 7 – Desvio padrão com 10 (a), 25 (b) e 50 (c) usuários simultâneos



(a) (b)

(c)

Fonte: Elaborado pelo autor

9.4 MODELO DE RECOMENDAÇÃO ATUALIZADO

Após a análise nos resultados dos testes de carga, nos diferentes passos de decomposição do modelo inicial, o passo que decompõe um micro serviço com a maior quantidade de raízes agregadas se mostrou a melhor alternativa. A diminuição do nível granular a partir da decomposição do serviço com o maior número de entidade com responsabilidades únicas, apesar de demostrar melhoria na média, tende a aumentar o desvio padrão, gerando inconsistências no tempo de resposta das requisições. Esse passo pode ser válido, se aplicado em conjunto com o segundo passo, mas esse escopo requer novos testes e deve ser tratado em trabalhos futuros. A decomposição de abstrações não é recomenda, pois, além de possuir uma implementação de alto custo, do ponto de vista de adequações dentro do desenvolvimento do projeto, também gerou resultados insatisfatórios e que pioram o desempenho dos serviços.

O modelo de recomendação para a definição granular é atualizado de acordo com os resultados. A decomposição por abstração não se mostrou válida, portando, é retirada do modelo. A decomposição a partir dos micro serviços que possuem mais de três raízes agregadas é o primeiro método a ser aplicado, já que garantiu os melhores resultados. A decomposição de micro serviços que possuem mais de três

entidades com responsabilidades únicas, deve ser aplicada como passo após a decomposição de raízes agregadas, para que melhore a média e amenize o aumento do desvio padrão. No Quadro 5 o modelo de recomendação final é proposto.

Quadro 5 – Modelo de recomendação final

Passo	Regra	Impacto
Criar um modelo visual contendo cada micro serviço com suas entidades.	Todos os micro serviços com suas entidades devem ser visíveis.	Tornar o modelo de domínio mais visível para a aplicação dos próximos passos.
Contabilizar o número de comunicações que cada micro serviço realiza com outros serviços.	Caso o micro serviço realize comunicações com mais do que três outros micro serviços, sua granularidade é alta, devendo ser considerada a decomposição.	Reduzir o número de comunicações que um micro serviço precisa realizar.
Contabilizar o número de entidades que possuem responsabilidades próprias em cada micro serviço.	Caso o micro serviço possua mais do que três responsabilidades, sua granularidade é alta, devendo ser considerada a decomposição.	Dispor de uma separação de responsabilidades entre os micro serviços, diminuindo a granularidade.

Fonte: Elaborado pelo autor

10 CONCLUSÃO

O estudo exposto objetivou desenvolver um modelo de recomendação para o nível granular em projetos que utilizam arquitetura de micro serviços e DDD, possibilitando assim melhoria de desempenho e na qualidade do produto. Através do referencial teórico, fundamentou-se os conhecimentos com relação a utilização de micro serviços e DDD, suas utilizações em conjunto e na preocupação em manter uma granularidade baixa para os contextos. Todavia, existe a necessidade de um modelo que seja evidenciado, contendo passos de elaboração de ajustes granulares em micro serviços.

Neste contexto, foi proposto um modelo de recomendação, apresentado no capítulo 6, para a definição de granularidade em projetos que utilizam arquitetura de micro serviços e DDD, desenvolvido inicialmente em quatro passos. Como primeiro passo, um modelo de visão é formalizado através do mapa de contextos. Após isso, são reconhecidos todos os micro serviços que contenham mais de três entidades com responsabilidades únicas, o que torna o serviço com uma granularidade alta. Como terceiro passo, micro serviços que realizam comunicação com mais de três outros serviços são divididos. O último passo é a decomposição de serviços que realizam abstrações de termos utilizados pelos conhecedores do negócio.

O capítulo de construção do projeto modelo relatou os requisitos apresentados como essenciais. Princípio de reponsabilidade única e mínima de MSA e a recomendação do DDD de modelar contextos delimitados com o menor número possível de raízes agregadas, para que não ocorra aumento desnecessário na complexidade do domínio, visando assim maior qualidade e melhora no desempenho das aplicações.

Conforme visto no capítulo 9, os resultados obtidos através da análise dos resultados, mostraram que o passo de decomposição de micro serviços que realizam mais do que três comunicações com outros serviços, obteve resultados satisfatórios e deve ser aderido por equipes que pretendam tornar seus projetos mais rápidos e confiáveis. O passo de decompor serviços que contenham mais do que três entidades com responsabilidades únicas, apesar de demonstrar oscilações no tempo de requisição através do desvio padrão, é aplicável em conjunto com o passo anterior. Já o passo de decompor abstrações não deve ser aplicado, pois os resultados foram negativos.

Conforme consta na seção 9.4, o modelo de recomendação gerado por este trabalho possui três passos para a obtenção da redução do nível granular em projetos que utilizam MSA e DDD. Após a análise dos resultados, o primeiro passo é mantido, pois é essencial para que os demais passos sejam realizados. O segundo passo é a verificação e decomposição dos micro serviços que possuem mais que três raízes agregadas. O último passo é a identificação e separação dos micro serviços que contenham mais do que três entidades que possuam responsabilidades únicas. O modelo contribui para que equipes de desenvolvimento de software realizem melhorias em seus projetos para uma boa definição da granularidade de seus serviços.

Este estudo se mostrou muito importante no âmbito profissional do autor, pois proporcionou um maior conhecimento de micro serviços e DDD, assim como o desenvolvimento de um projeto através dos modelos implementados no capítulo 7. Micro serviços podem ser desenvolvidos com várias tecnologias, diversos banco de dados e são publicados em diferentes servidores com uma infraestrutura escalável. Portanto, em trabalhos futuros é recomendável a aplicação e avaliação dos métodos em estruturas mais próximas da realidade. Testes considerando falhas decorrentes de possíveis gargalos provocados pelo mecanismo de mensageria devem ser aplicados. Outros métodos que visam diminuir o nível granular de micro serviços devem ser identificados e avaliados como passos, assim como possíveis combinações, para que vantagens e avanços sejam dados ao modelo de recomendação do nível granular em arquiteturas de micro serviços e DDD.

REFERÊNCIAS

AÉCE, Israel. **Granularidade de Serviços.** 2009. Disponível em: <http://www.linhadecodigo.com.br/artigo/2599/granularidade-de-servicos.aspx>. Acesso em: 13 jun. 2022.

AMARAL, Odravison. CARVALHO, Marcus. **Arquitetura de Micro Serviços: uma Comparação com Sistemas Monolíticos**. 2017. Universidade Federal da Paraíba (UFPB), Rio Tinto, PB, 2017.

BARRA, Daniela Couto C. DO NASCIMENTO, Eliane Regina P. MARTINS, Josiane de Jesus. ALBUQUERQUE, Luiz. ERDMANN, Alacoque L. **Evolução Histórica e Impacto da Tecnologia na Área da Saúde e da Enfermagem.** 2006. Revista Eletrônica de Enfermagem, v. 08, n. 03, p. 422-430, 2006.

BRUEGGE, Bernd. DUTOIT, Allen H. **Object-Oriented Software Engineering: Using UML, Patterns and Java.** 2009. Editora Pearson, 3rd edition. 2009.

CMU - SOFTWARE ENGINEERING INSTITUTE. **Software Architecture**. 2015. Disponível em: <https://www.sei.cmu.edu/our-work/software-architecture/>. Acesso em: 31 mar. 2022.

CRAMON, Jeppe. **Microservices: Usage Is More Important than Size.** 2017. Disponível em: <https://www.infoq.com/news/2014/05/microservices-usage-size/>. Acesso em: 13 jun. 2022.

CUKIER, Daniel. **DDD - Introdução a Domain Driven Design.** 2010. Disponível em: <http://www.agileandart.com/2010/07/16/ddd-introducao-a-domain-driven-design/>. Acesso em: 13 jun. 2022.

DA SILVA, Teófilo A. FILHO, Cláudio de Castro C. DA SILVA, Carlos Tiago M. **Investigação Ontológica da Obra da Arte Digital: Linguagem Ubíqua, Modelo de Domínio e Programação Voltada para as Artes Visuais.** 2018. V Simpósio Internacional de Inovação em Mídias Interativas. p. 388-416. UFG. Goiânia.

DI FRANCESCO, P. MALAVOLTA, I. LAGO, P. **Research on Architecting Microservices: Trends, Focus and Potential for Industrial Adoption.** 2017. In: Software Architecture (ICSA). IEEE International Conference, p. 21-30, 2017.

DIEPENBROCK, Andreas. RADEMACHER, Florian. SACHWEH, Sabine. **An Ontology-Based Approach for Domain-Driven Design of Microservice Architectures.** 2017. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, p. 1777-1791, 2017.

DRAGONI, Nicola. GIALLORENZO, Saverio. LAFUENTE, Alberto L. MAZZARA, Manuel. MONTESI, Fabrizio. MUSTAFIN, Ruslan. SAFINA, Larisa. **Microservices: Yesterday, Today and Tomorrow.** 2017. Ulterior Software Engineering, p. 195-216, 2017.

DYBÅ, Tore. BERGENSEN, Gunnar R. SJØBERG, Dag IK. **Evidence-based Software Engineering.** 2016. Perspectives on Data Science for Software Engineering. Cambridge: Elsevier. p. 149-153. 2016.

DYBÅ, Tore. KITCHENHAM, Barbara A. JØRGENSEN, Magne. **Evidence-based software engineering for practitioners**. 2005. IEEE Software, v. 22, n. 1, p. 58-65, 2005.

ELIAS, Diego. **A Granularidade de Dados no Data Warehouse.** 2014. Disponível em: <https://canaltech.com.br/business-intelligence/a-granularidade-de-dados-no-data-warehouse-26310/>. Acesso em: 13 jun. 2022.

ENGHOLM JR, Hélio. **Engenharia de Software na Prática.** 2010. Editoria Novatec, 1nd ed.

EVANS, Eric. **Defining Bounded Contexts – Eric Evans at DDD Europa**. 2019. Disponível em: <https://www.infoq.com/news/2019/06/bounded-context-eric-evans/>. Acesso em 31 mar. 2022.

EVANS, Eric. **Domain-Driven Design Europe 2020 - Bounded Contexts**. 2020. Disponível em: <https://www.youtube.com/watch?v=am-HXycfalo>. Acesso em 10 jun. 2022.

EVANS, Eric. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. 2003. Addison-Wesley Professional, 1 ed. 2003.

FOWLER, Martin. LEWIS, James. **Microservices a definition of this new architectural term**. 2014. Disponível em <https://martinfowler.com/articles/microservices.html>. Acesso em: 31 mar. 2022.

GAMMA, Erich. HELM, Richard. VLISSIDES, John. JOHNSON, Ralph. **Padrões de Projetos: Soluções Reutilizáveis de Software.** 2000. Editora Bookman, 1nd ed.

GOUIGOUX, Jean-Philippe. TAMZALIT, Dalila. **From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture.** 2017. IEEE International Conference on Software Architecture Workshops, 2017.

HIPPCHEN, Benjamin. GIESSLER, Pascal. STEINEGGER, Roland H. SCHNEIDER, Michael. ABECK, Sebastian. **Designing Microservice-Based Applications by Using a Domain-Driven Design Approach.** 2017. International Journal on Advances in Software, vol. 10, no. 3 & 4, p. 432-445, 2017.

JOSÉLYNE, Munezero I. TUHEIRWE-MUKASA, Doreen. KANAGWA, Benjamin. BALIKUDDEMBE, Joseph. **Partitioning Microservices: A Domain Engineering Approach.** 2018. ACM/IEEE Symposium on Software Engineering in Africa, p. 43-49, 2018.

KITCHENHAM, Barbara A. BUDGEN, David. BRERETON, Pearl. **Evidence-Based Software Engineering and Systemic Reviews.** 2016. Boca Raton, FL: CRC Press, 2016.

LANDRE, Einar. WESENBERG, Harald. RØNNEBERG, Harald. **Architectural Improvement by Use of Strategic Level Domain-Driven Design.** 2006. OOPSLA: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages and applications, p. 809-814, 2006.

LAUX, Marina Letícia. **Técnica de Apoio à Construção do Modelo de Maturidade do Time Scrum**. 2019. Universidade Feevale, Novo Hamburgo, RS, 2019.

MACEDO, Otávio Augusto Cardoso. **Diretrizes para Desenvolvimento de Linhas de Produtos de Software com Base em Domain-Driven Design e Métodos Ágeis.** 2009. Tese de Doutorado. Universidade de São Paulo, SP, 2009.

MAFRA, Sômulo Nogueira. TRAVASSOS, Guilherme Horta. **Estudos Primários e Secundários Apoiando a Busca por Evidências em Engenharia de Software.** 2006. 33 f. Relatório Técnico (Programa de Engenharia de Sistemas e Computação) - Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, 2006.

MARINESCU, Floyd. AVRAM, Abel. **Domain-Driven Design Quickly.** 2007. Lulu Press, 1nd ed.

MATTOS, Karla Marturelli. DOLL, Luciano Mathias. ALMEIDA, João. **Domain-Driven Design e Test-Driven Development.** 2010. 5º Encontro de Engenharia e Tecnologia dos Campos Gerais, 2010.

MILLETT, Scott. TUNE, Nick. **Patterns, Principles and Practices of Domain-Driven Design.** 2015. Editora Wrox, 1nd ed.

MISIRLI, Ayse. BENER, Ayse Basar. **Bayesian Networks for Evidence-Based Decision-Making in Software Engineering.** 2014. IEEE Transactions on Software Engineering, vol. 40, no. 6, 2014.

MOREIRA, Pedro Felipe M. BEDER, Delano M. **Desenvolvimento de Aplicações e Micro Serviços: Um Estudo de Caso.** 2016. Revista TIS, vol. 4, no. 3, 2016.

NADAREISHVILI, Irakli. MITRA, Ronnie. MCLARTY, Matt. MIKE, Amundsen. **Microservice Architecture: Aligning Principles, Practices and Culture.** 2016. Editora O'Reilly Media, 1nd ed.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems**. 2021. O'Reilly Media; 2nd ed. 2021.71

OKOLNYCHYI, Anton. FÖGEN, Konrad. **A Study of Tools for Behavior-Driven Development.** 2016. Faculty of Mathematics, Computer Science and Natural Sciences. Seminar Winter Term, p. 7-12, 2016.

PALERMO, Jeffrey. **The Onin Architecture.** 2008. Disponível em: <https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/>. Acesso em: 13 jun. 2022.

PETRASCH, Roland. **Message-Oriented Middleware for System Communication: A Model-based Approach.** 2017. International Conference on Computing and Information Technology. IC2IT: Recent Advances in Information and Communication Technology, p. 253-263, 2017.

RADEMACHER, Florian. SORGALLA, Jonas. SACHWEH, Sabine. **Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective.** 2018. IEEE Computer Society Magazines, may-june, 2018.

RICHARDSON, Chris. **Introduction to Microservices.** 2015. Disponível em: <https://www.nginx.com/blog/introduction-to-microservices/>. Acesso em: 14 jun. 2022.

RICHARDSON, Chris. **Pattern: Microservice Architecture.** 2016. Disponível em: <https://microservices.io/patterns/microservices.html>. Acesso em: 14 jun. 2022.

RICHARDS, Mark. **Microservices vs. Service-Oriented Architecture.** 2016. Disponível em: <https://www.oreilly.com/radar/microservices-vs-service-oriented-architecture/>. Acesso em: 14 jun. 2022.

SANTOS, Eloisa Cristina S. **Integração da Abordagem Domain-Driven Design e da técnica Behaviour-Driven Development no desenvolvimento de aplicações web.** 2015. Tese de Pós-Graduação em Ciência da Computação pela Universidade Federal de São Carlos (UFSCar), São Carlos, SP, 2015.

SCHERMANN, Gerald. CITO, Jürgen. LEITNER, Philipp. **All the Services Large and Micro: Revisiting Industrial Practice in Services Computing.** 2015. In International Conference on Service-Oriented Computing, p. 36-47, 2015.

VERA-RIVERA, Fredy H. PUERTO-CUADROS, Eduard G. ASTUDILLO, Hermán. GOANA-CUEVAS, Carlos M. **Microservices Backlog - A Model of Granularity Specification and Microservice Identification.** 2020. Services Computing. 17th Internation Conference Held as Part of the Services Conference Federation. p. 85-102, september, 2020.

VERNON, Vaughn. **Implementando Domain-Driven Design.** 2016. Editora Alta Books, 1nd ed.

VILLAMIZAR, Mario. GARCES, Oscar. OCHOA, Lina. CASTRO, Harold. SALAMANCA, Lorena. VERANO, Mauricio. CASALLAS, Rubby. GIL, Santiago. VALENCIA, Carlos. ZAMBRANO, Angee. LANG, Mery. **Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures.** 2016. 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, p.179-182, 2016.

WOLFF, Eberhard. **Microservices: Flexible Software Architecture.** 2016. Editora Addison-Wesley Professional, 1nd ed.

YU, Yale. SILVEIRA, Haydn. SUNDARAM, Max. **A Microservice Based Reference Architecture Model in the Context of Enterprise Architecture.** 2016. In IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference, p.1856-1860, 2016.

ZASCHKE, Christian. **Elaboration of a Domain Model for Migrating the Monolithic Software Architecture of a Data Management Server into a Microservice Architecture.** 2019. In 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, p. 212-218, 2019.

APÊNDICE A – LINK COM CÓDIGOS DE PROJETOS IMPLEMENTADOS

Para evidenciar o estudo realizado por este trabalho, foram codificados projetos para que o modelo de recomendação fosse testado. Os projetos podem ser obtidos em https://drive.google.com/file/d/1Z1No1fptVZjTo33ruKoppxEWUCABVV8P e estão nomeados como:

- **TCC.AdministracaoDeTeses:** Projeto referente ao protótipo encontrado na literatura e descrito no capítulo 5.
- **TCC.ProjetoInicial:** Projeto construído através do mapa de contextos elaborado como primeiro passo do modelo de recomendação.
- **TCC.Passo2:** Projeto construído após a aplicação da decomposição do micro serviço que possui três ou mais entidades com responsabilidades únicas.
- **TCC.Passo3:** Projeto construído após a aplicação da decomposição do micro serviço que realiza três ou mais comunicações com outros serviços.
- **TCC.Passo4:** Projeto construído após a aplicação da decomposição de micro serviços que realizam abstrações de termos utilizados pelos conhecedores do negócio.