



INSTITUTO FEDERAL  
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
Fluminense

MINISTÉRIO DA  
EDUCAÇÃO



PÁTRIA AMADA  
**BRASIL**  
GOVERNO FEDERAL

Raphael Collin Teles Manhães

# **Desenvolvimento de Sistemas com Microserviços e *Domain-Driven Design*: um estudo de caso realista**

Campos dos Goytacazes-RJ

Outubro de 2024

Raphael Collin Teles Manhães

## **Desenvolvimento de Sistemas com Microserviços e *Domain-Driven Design*: um estudo de caso realista**

Trabalho de Conclusão apresentado ao curso Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia Fluminense, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Instituto Federal de Educação, Ciência e Tecnologia Fluminense

Orientador: Prof. D.Sc. Mark Douglas de Azevedo Jacyntho

Campos dos Goytacazes-RJ


Outubro de 2024

Raphael Collin Teles Manhães

## **Desenvolvimento de Sistemas com Microsserviços e *Domain-Driven Design*: um estudo de caso realista**


Trabalho de Conclusão apresentado ao curso Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia Fluminense, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Campos dos Goytacazes-RJ, 17 de outubro de 2024.

Documento assinado digitalmente  
 **MARK DOUGLAS DE AZEVEDO JACYNTHO**  
Data: 12/11/2024 19:59:37-0300  
Verifique em <https://validar.iti.gov.br>


---

**Prof. D.Sc. Mark Douglas de Azevedo Jacyntho (orientador)**  
Instituto Federal Fluminense (IFF)

Documento assinado digitalmente  
 **FERNANDO LUIZ DE CARVALHO E SILVA**  
Data: 12/11/2024 21:37:30-0300  
Verifique em <https://validar.iti.gov.br>

---

**Prof. D.Sc. Fernando Luiz de Carvalho e Silva**  
Instituto Federal Fluminense (IFF)

Documento assinado digitalmente  
 **PHILIPPE LEAL FREIRE DOS SANTOS**  
Data: 13/11/2024 15:36:27-0300  
Verifique em <https://validar.iti.gov.br>

---

**Prof. D.Sc. Philippe Leal Freire dos Santos**  
Instituto Federal Fluminense (IFF)

Campos dos Goytacazes-RJ  
Outubro de 2024

# Agradecimentos

Em primeiro lugar, agradeço a Deus por ter me dado força e sabedoria para superar os desafios e dificuldades ao longo da minha jornada acadêmica.

Agradeço também a minha mãe que sempre me apoiou e incentivou a seguir em frente na vida acadêmica e profissional desde os primeiros momentos. Sou muito grato por todo o amor, carinho e dedicação que sempre me proporcionou. Se hoje estou aqui, é graças a você.

Agradeço ao meu orientador, Prof. Dr. Mark Douglas, por todo o apoio, orientação e ensinamentos durante suas aulas, nossas conversas e ao longo deste trabalho. Sua paciência, dedicação e conhecimento foram fundamentais para o desenvolvimento deste estudo.

Por fim, sou extramente grato a todos professores, autores, colegas e amigos que em algum momento me ensinaram algo e contribuíram para a minha formação acadêmica e profissional. Acredito firmemente que a educação é a chave para não apenas para atingir nossos objetivos pessoais, mas também para transformar nosso país e o mundo em um lugar mais próspero, solidário e justo.

# Resumo

Com a popularização da computação em nuvem, a crescente demanda de mudanças cada vez mais frequentes e o crescimento das equipes de tecnologia da informação em grandes organizações, a arquitetura tradicional para o desenvolvimento de aplicações corporativas, baseada em monólitos, é confrontada com desafios significativos. Nesse cenário, surge como alternativa a Arquitetura de Microserviços. Paralelamente a isso, a abordagem *Domain-Driven Design* (DDD) tem sido cada vez mais utilizada para modelagem de domínios de negócio complexos. Essas duas estratégias podem ser combinadas no *design* e desenvolvimento de sistemas. De um lado, microserviços possibilitam escalabilidade independente, implantação desacoplada e utilização de múltiplas tecnologias para determinados casos de uso. Por outro lado, DDD fornece uma abordagem para modelagem do domínio de negócio, diversos padrões para resolver problemas de modelagem recorrentes e facilidade de entendimento e manutenção de código. Além disso, DDD é uma excelente estratégia para definição de limites entre microserviços. No entanto, a utilização combinada desses dois conceitos apresenta nuances e diferentes possibilidades. Assim, esse trabalho apresenta um estudo de caso realista de uma locadora de veículos com o emprego dessas estratégias visando oferecer uma contribuição significativa para a compreensão da aplicação integrada dessas abordagens.

**Palavras-chaves:** Microserviços, *Domain-Driven Design*, Arquitetura de Microserviços.

# Abstract

Considering the popularization of cloud computing, the increasing demand for more frequent changes and the growth of information technology teams in large organizations, the traditional architecture for developing enterprise applications, based on monoliths, is confronted with significant challenges. In this scenario, the Microservices Architecture emerges as an alternative. At the same time, the Domain-Driven Design (DDD) methodology has been increasingly used to model complex business domains. These two strategies can be combined in the design and development of systems. On one side, microservices enable independent scalability, decoupled deployment, and the use of multiple technologies for certain use cases. On the other hand, DDD provides an approach to modeling the business domain, several patterns to solve recurring modeling problems, and ease of understanding and maintaining code. In addition, DDD is an excellent strategy for defining boundaries between microservices. However, the combined use of these two concepts faces nuances and different possibilities. Therefore, this work presents a realistic case study of a vehicle retailer with the use of these strategies aiming to offer a significant contribution to the understanding of the integrated application of these approaches.

**Keywords:** Microservices, Domain-Driven Design, Microservices Architecture.

# Lista de Figuras

Figura 1 – Etapas de desenvolvimento da pesquisa . . . . .	16
Figura 2 – Arquitetura Hexagonal . . . . .	28
Figura 3 – Número de publicações ao longo dos anos . . . . .	32
Figura 4 – Publicações por localização . . . . .	32
Figura 5 – Diagrama de caso de uso . . . . .	40
Figura 6 – Exemplo de quadro <i>Kanban</i> . . . . .	41
Figura 7 – Ciclo de Vida do Desenvolvimento de Software . . . . .	50
Figura 8 – Modelo conceitual do sistema . . . . .	51
Figura 9 – Bounded Contexts do sistema . . . . .	52
Figura 10 – Microsserviços do sistema . . . . .	53
Figura 11 – Ciclo de vida de uma reserva . . . . .	55
Figura 12 – Criar reserva . . . . .	56
Figura 13 – Realizar check-in . . . . .	57
Figura 14 – Realizar check-out . . . . .	58
Figura 15 – User Service . . . . .	59
Figura 16 – Customer Service . . . . .	60
Figura 17 – Employee Service . . . . .	61
Figura 18 – Inventory Service . . . . .	62
Figura 19 – Booking Service . . . . .	63
Figura 20 – Utilização da CPU . . . . .	74
Figura 21 – Utilização da Memória RAM . . . . .	75
Figura 22 – Tempo de Resposta . . . . .	75
Figura 23 – Porcentagem de Erros . . . . .	76

# Lista de quadros

Quadro 1 – Estilos de comunicação entre microserviços . . . . .	22
Quadro 2 – Padrões de comunicação do tipo <b>"um para um"</b> . . . . .	23
Quadro 3 – Padrões de comunicação do tipo <b>"um para muitos"</b> . . . . .	23
Quadro 4 – Tipos de acoplamento . . . . .	25
Quadro 5 – Expressão de busca utilizada . . . . .	30
Quadro 6 – Publicações selecionadas . . . . .	34
Quadro 7 – Anti-padrões . . . . .	38
Quadro 8 – Registrar cliente . . . . .	41
Quadro 9 – Listar veículos . . . . .	42
Quadro 10 – Reservar um veículo . . . . .	42
Quadro 11 – Realizar <i>check-in</i> . . . . .	43
Quadro 12 – Realizar <i>check-out</i> . . . . .	44
Quadro 13 – Listar reservas . . . . .	44
Quadro 14 – Registrar veículo . . . . .	45
Quadro 15 – Métricas de comparação . . . . .	48



# Lista de codigos

5.1	Método para realizar login . . . . .	64
5.2	Método para realizar cadastro de cliente . . . . .	64
5.3	Método para criar reserva . . . . .	65
5.4	Métodos para realizar check-in . . . . .	66
5.5	Métodos para realizar check-out . . . . .	67
5.6	Método para obter um veículo do <i>Inventory Service</i> . . . . .	68
5.7	Código para realizar comunicação assíncrona entre microserviços . . . . .	69
5.8	Teste unitário simples . . . . .	70
5.9	Teste de integração . . . . .	71
5.10	Teste de sistema . . . . .	72
5.11	Teste de carga . . . . .	73

# Siglas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ACL	Anti-Corruption Layer
AMS	Arquitetura de Microserviços
API	Application Programming Interface
AWS	Amazon Web Services
BC	Bounded Context
BDD	Behavior Driven Development
DDD	Domain-Driven Design
IDE	Integrated Development Environment
IPC	Inter-Process Communication
OHS	Open Host Service
POO	Programação Orientada a Objetos
SOA	Service-Oriented Architecture
TDD	Test-driven development
VO	Value Object

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
<b>1.1</b>	<b>Problema e contexto</b>	<b>13</b>
<b>1.2</b>	<b>Justificativa</b>	<b>14</b>
<b>1.3</b>	<b>Objetivos</b>	<b>15</b>
1.3.1	Objetivo Geral	15
1.3.2	Objetivos Específicos	15
<b>1.4</b>	<b>Metodologia</b>	<b>15</b>
<b>1.5</b>	<b>Estrutura do Trabalho</b>	<b>15</b>
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>17</b>
<b>2.1</b>	<b><i>Domain-Driven Design (DDD)</i></b>	<b>17</b>
2.1.1	Modelo e Domínio	17
2.1.2	Entidade	18
2.1.3	<i>Value Object (VO)</i>	18
2.1.4	Serviço	18
2.1.5	<i>Aggregate</i>	19
2.1.6	Bounded Context (BC)	19
<b>2.2</b>	<b>Sistema Distribuído</b>	<b>20</b>
2.2.1	Service-Oriented Architecture (SOA)	20
<b>2.3</b>	<b>Microserviços</b>	<b>21</b>
2.3.1	Comunicação entre microserviços	22
2.3.2	Estratégias para delimitação de microserviços	23
2.3.3	Desafios na implementação de microserviços	26
<b>2.4</b>	<b>Arquitetura Hexagonal</b>	<b>27</b>
<b>3</b>	<b>TRABALHOS RELACIONADOS</b>	<b>29</b>
<b>3.1</b>	<b>Protocolo</b>	<b>29</b>
3.1.1	Questões de pesquisa	29
3.1.2	Estratégia de busca	29
3.1.3	Expressão de busca	30
3.1.4	Estratégia de seleção	30
3.1.5	Estratégia para extração de dados e análise	31

3.2	Caracterização dos estudos . . . . .	31
3.3	Estratégias para elaboração de sistemas com microserviços e DDD	33
3.4	Desafios na utilização de DDD como estratégia de delimitação de microserviços . . . . .	35
3.5	Anti-padrões a serem evitados . . . . .	37
3.6	Discussões . . . . .	37
4	<b>ESTUDO DE CASO: REQUISITOS, ORGANIZAÇÃO E MÉTODOS</b>	39
4.1	Contexto . . . . .	39
4.2	Processo de Desenvolvimento . . . . .	39
4.3	Requisitos . . . . .	39
4.3.1	Requisitos Funcionais . . . . .	41
4.3.2	Requisitos não Funcionais . . . . .	45
4.4	Design . . . . .	45
4.5	Implementação . . . . .	46
4.6	Validação . . . . .	47
4.7	Implantação . . . . .	48
5	<b>ESTUDO DE CASO: DESIGN, IMPLEMENTAÇÃO E TESTES</b> . .	50
5.1	Ciclo de Vida do Desenvolvimento de Software . . . . .	50
5.2	Design . . . . .	51
5.2.1	Bounded Contexts . . . . .	52
5.2.2	Microserviços . . . . .	53
5.2.3	Interações entre Microserviços . . . . .	54
5.2.4	Principais casos de uso . . . . .	54
5.2.4.1	Criar reserva . . . . .	55
5.2.4.2	Realizar check-in . . . . .	56
5.2.4.3	Realizar check-out . . . . .	57
5.2.5	User Service . . . . .	58
5.2.6	Customer Service . . . . .	59
5.2.7	Employee Service . . . . .	60
5.2.8	Inventory Service . . . . .	61
5.2.9	Booking Service . . . . .	62
5.3	<b>Implementação</b> . . . . .	63
5.3.1	Login de usuário . . . . .	63
5.3.2	Cadastro de Cliente . . . . .	64
5.3.3	Realizar Reserva . . . . .	65
5.3.4	Realizar Check-in . . . . .	65
5.3.5	Realizar Check-out . . . . .	66
5.3.6	Comunicação síncrona entre Microserviços . . . . .	67

5.3.7	Comunicação assíncrona entre Microserviços . . . . .	68
<b>5.4</b>	<b>Testes . . . . .</b>	<b>69</b>
5.4.1	Testes Unitários . . . . .	69
5.4.2	Testes de Integração . . . . .	70
5.4.3	Testes de Sistema . . . . .	71
5.4.4	Testes de Carga . . . . .	72
<b>6</b>	<b>RESULTADOS E DISCUSSÕES . . . . .</b>	<b>74</b>
<b>6.1</b>	<b>Resultados . . . . .</b>	<b>74</b>
6.1.1	Utilização da CPU . . . . .	74
6.1.2	Utilização da Memória RAM . . . . .	75
6.1.3	Tempo de Resposta . . . . .	75
6.1.4	Porcentagem de erros . . . . .	76
<b>6.2</b>	<b>Trabalhos Futuros . . . . .</b>	<b>76</b>
<b>6.3</b>	<b>Discussões . . . . .</b>	<b>76</b>
<b>7</b>	<b>CONCLUSÃO . . . . .</b>	<b>78</b>
	<b>REFERÊNCIAS . . . . .</b>	<b>79</b>

# 1 Introdução

## 1.1 Problema e contexto

A abordagem convencional no desenvolvimento de sistemas corporativos, na qual toda a lógica de negócio, persistência e apresentação são encapsuladas em uma única aplicação (e um único executável), é a mais natural para construção desse tipo de sistema. Essa arquitetura, conhecida comumente como monólito, tem alcançado sucesso devido à sua simplicidade no desenvolvimento, testes, implantação e monitoramento, entre outros fatores. No entanto, com a popularização da computação em nuvem, a crescente demanda de mudanças cada vez mais frequentes e o crescimento das equipes de tecnologia da informação em grandes organizações, essa arquitetura é confrontada com desafios significativos. Inicialmente, destaca-se o acoplamento nos ciclos de mudanças, onde qualquer alteração em uma pequena parte requer a compilação e implantação de todo sistema. Além disso, há dificuldade em escalar horizontalmente apenas funcionalidades específicas que estão sendo mais demandas e existe complexidade na utilização de diferentes linguagens de programação em diversas áreas de um sistema (FOWLER; LEWIS, 2014). Por esses motivos, a adoção da [Arquitetura de Microserviços \(AMS\)](#) cresceu de forma exponencial nos últimos anos, com 77% dos profissionais afirmando que suas organizações utilizam esse estilo arquitetural (LOUKIDES; SWOYER, 2020).

Por outro lado, [Domain-Driven Design \(DDD\)](#) é uma abordagem para o desenvolvimento de software, na qual os elementos de software como pacotes, classes, interfaces, métodos e nomes de variáveis devem corresponder aos conceitos do domínio de negócio (FOWLER, 2020). Adicionalmente, o DDD se configura como uma estratégia para harmonizar especialistas de negócios, desenvolvedores e projetistas, promovendo uma otimização nas interações ao reduzir o mapeamento entre termos de negócio e termos técnicos.

Em agosto de 2008, um grande incidente interrompeu as operações da *Netflix*, empresa que nesse período alugava DVDs através de seu site na *web*. Uma corrupção no banco de dados impediu que a companhia pudesse realizar envios por três dias. A partir desse momento, os engenheiros perceberam que necessitavam reduzir pontos únicos de falha, tais como grandes bases de dados relacionais armazenadas em somente um *data center*. Em vez disso, optaram por adotar sistemas distribuídos na nuvem, escalonados horizontalmente, para garantir maior confiabilidade e minimizar o impacto das falhas (IZRAILEVSKY; MESHENBERG, 2014).

Essa transição resultou em vários benefícios para a *Netflix*, destacando-se a conquista

de uma disponibilidade de 'quatro noves' (99,99%) e significativa redução de custos. Além disso, a ams permitiu que diferentes aplicações utilizassem diferentes tecnologias, facilitando contratações e extraindo o melhor de cada opção. Assim, em 2016 (8 anos após a migração) a empresa tinha 8 vezes mais assinantes, muito mais ativos, com o número de visualizações crescendo em três ordens de grandeza nesse período (IZRAILEVSKY; MESHENBERG, 2014).

A AMS e DDD podem ser combinados no *design* e desenvolvimento de sistemas. De um lado, microsserviços possibilitam escalabilidade independente, implantação desacoplada e utilização de múltiplas tecnologias para determinados casos de uso. Por outro lado, DDD fornece uma abordagem para modelagem do domínio de negócio, diversos padrões para resolver problemas de modelagem recorrentes e facilidade de entendimento e manutenção de código. Além disso, DDD é uma excelente estratégia para definição de limites entre microsserviços. Ao modelar os serviços com base em *Bounded Context (BC)*s coesos, é possível o desenvolvimento de novas funcionalidades mais rapidamente e também se torna mais simples a recombinação de microsserviços para fornecer novas funcionalidades aos usuários (NEWMAN, 2021).

## 1.2 Justificativa

Recentemente, observou-se um notável crescimento em popularidade e adoção da AMS e da abordagem DDD no desenvolvimento de sistemas corporativos. Porém, essas tecnologias são utilizadas de maneira equivocada em muitas implementações devido à falta de compreensão das suas vantagens e desvantagens, entre outros fatores. Por exemplo, a equipe de desenvolvimento do *Amazon Prime Video* recentemente anunciou uma migração de um serviço de monitoramento de vídeo, que foi inicialmente desenvolvido com microsserviços, para uma arquitetura monolítica com objetivo de aumentar a escalabilidade e reduzir custos (KOLNY, 2023).

Paralelamente a isso, é possível identificar uma carência de estudos de caso realistas envolvendo a utilização simultânea da AMS e DDD, especialmente no cenário nacional. Enquanto algumas publicações concentram-se exclusivamente em um desses conceitos e outras apresentam exemplos simplificados, proporcionando uma compreensão limitada de como essas duas técnicas podem ser efetivamente empregadas em conjunto.

Nesse contexto, este trabalho visa preencher essa lacuna ao oferecer uma contribuição significativa para a compreensão da aplicação integrada dessas estratégias no desenvolvimento de sistemas. Por meio de um estudo de caso realista, busca-se não apenas enriquecer o conhecimento acadêmico, mas também fornecer uma base sólida para a implementação prática dessas tecnologias em diversos setores da indústria.

## 1.3 Objetivos

### 1.3.1 Objetivo Geral

O objetivo geral deste trabalho de conclusão de curso é apresentar um estudo de caso realista integrando a [AMS](#) e [DDD](#), visando demonstrar como essas tecnologias podem ser utilizadas para aumentar a escalabilidade, desempenho e resiliência de sistemas complexos.

### 1.3.2 Objetivos Específicos

- Apresentar uma estratégia para transformar requisitos funcionais em um *design* com [AMS](#) e [DDD](#).
- Prover informações relevantes para definição dos estilo de comunicação adequado entre microsserviços.
- Demonstrar desempenho e escalabilidade do sistema desenvolvido através de testes de carga.

## 1.4 Metodologia

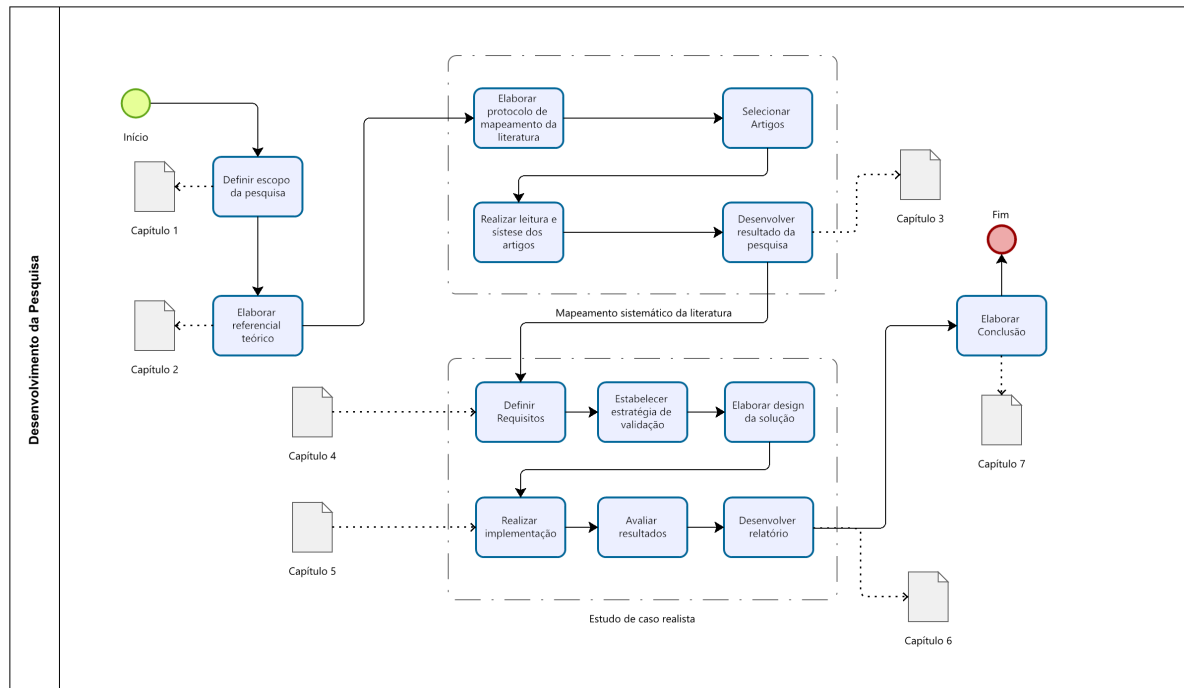
Pode-se observar na [Figura 1](#) as etapas de execução dessa pesquisa. Inicialmente, o escopo é definido e o primeiro capítulo é elaborado. Nesse momento, busca-se apresentar o contexto do estudo, as justificativas e os objetivos a serem atingidos. Em seguida, a fundamentação teórica com os conceitos-chave é construída, visando fornecer uma base sólida para o desenvolvimento do estudo de caso.

Posteriormente, é realizado um mapeamento da literatura buscando trabalhos similares. Esse mapeamento é essencial para identificar lacunas no conhecimento e definir o escopo do estudo de caso. Além disso, ele permite a identificação de desafios, práticas e padrões comuns na utilização de [DDD](#) e [AMS](#) em conjunto.

Por fim, um estudo de caso realista com a utilização da [AMS](#) e diversos conceitos do [DDD](#) é desenvolvido. Nesse estudo, são apresentados os requisitos, métodos e organização do sistema. Além disso, são apresentados trechos de código chave da implementação e os resultados obtidos com a execução dos testes de carga. O objetivo é demonstrar de maneira prática como essas tecnologias podem ser utilizadas para aumentar a escalabilidade, desempenho e resiliência de sistemas complexos. Finalmente, um relatório com análises quantitativa e qualitativa é produzido apresentando os resultados obtidos com a execução dos testes de carga e uma discussão sobre trabalhos futuros, desafios e benefícios da utilização de [DDD](#) e [AMS](#).



Figura 1 – Etapas de desenvolvimento da pesquisa



Fonte: o autor

## 1.5 Estrutura do Trabalho

Este trabalho está dividido em sete capítulos. O **Capítulo 1** expõe o contexto do estudo, as justificativas desta pesquisa e os objetivos a serem atingidos. O **Capítulo 2** apresenta conceitos de **DDD**, **AMS** e afins. O **Capítulo 3** expõe o protocolo e o resultado do mapeamento da literatura sobre a utilização combinada das estratégias mencionadas. Da mesma forma, o **Capítulo 4** descreve os requisitos, métodos e organização do estudo de caso. Em seguida, o **Capítulo 5** apresenta o estudo de caso desenvolvido, incluindo o *design* do sistema, trechos de código chave da implementação. Posteriormente, o **Capítulo 6** apresenta os resultados obtidos com a execução dos testes de carga e uma discussão sobre os desafios e benefícios da utilização de **DDD** e **AMS**. Por fim, o **Capítulo 7** apresenta as conclusões obtidas com o desenvolvimento deste trabalho.

## 2 Fundamentação Teórica

Este capítulo apresenta os conceitos de [DDD](#), Sistemas Distribuídos, [Service-Oriented Architecture \(SOA\)](#), Microserviços e Arquitetura Hexagonal.

### 2.1 *Domain-Driven Design (DDD)*

*Domain-Driven Design (DDD)* é uma abordagem para o desenvolvimento de software focada na construção de um modelo de domínio que representa de maneira sofisticada os processos e regras de negócio do sistema em questão ([FOWLER, 2020](#)). [DDD](#) é muito efetivo para desenvolver sistemas com regras de negócio complexas, que possuem, por exemplo, diversos atores, entidades e validações. A [Programação Orientada a Objetos \(POO\)](#) é geralmente empregada para a implementação do modelo de domínio, visto que possui uma série de recursos que facilitam a modelagem de objetos do mundo real ([EVANS, 2004](#)).

Um ponto central da metodologia [DDD](#), no desenvolvimento de softwares complexos, é a utilização de uma linguagem ubíqua que insere terminologia e jargões do domínio em componentes de software como atributos, métodos, classes, interfaces e pacotes ([FOWLER, 2020](#)). Assim, tanto especialistas de negócio quanto desenvolvedores e projetistas compartilham um vocabulário comum que facilita na comunicação e reduz a necessidade de realizar o mapeamento entre termos técnicos e termos do domínio. Além disso, a estrutura de sistema passa a representar a estrutura do negócio, tornando-se, assim, mais fácil de ser compreendida e modificada.

O nome dessa estratégia vem de um livro publicado em [2004](#) por Eric Evans que descreve a abordagem, uma série de conceitos e um catálogo de padrões comumente utilizados para resolução de problemas de modelagem de domínios. Os mais relevantes dentre estes são descritos a seguir.

#### 2.1.1 Modelo e Domínio

Os conceitos de modelo e domínio, no contexto da engenharia de software, são fundamentais para o entendimento do [DDD](#). Modelo é uma simplificação, uma interpretação da realidade que abstrai os aspectos relevantes para resolver o problema em questão e ignora detalhes superficiais ([EVANS, 2004](#)). Da mesma forma, todo sistema computacional está relacionado a alguma atividade ou interesse do usuário final. Assim, a área para qual o usuário utiliza o sistema é domínio do software ([EVANS, 2004](#)). Alguns exemplos de

domínios comuns são: financeiro, médico, logística e *marketplace*.

### 2.1.2 Entidade

No contexto da POO, muitos objetos não são fundamentalmente definidos por seus atributos, mas por um fio de continuidade e identidade (EVANS, 2004). Eles podem ter distintas representações como, por exemplo, em memória ou em um banco de dados. No entanto, eles necessitam ser distinguíveis mesmo se todas suas propriedades forem iguais. Em um domínio financeiro, por exemplo, uma transação composta por valor, data e usuário necessita ser diferenciável de outra transação com as mesmas características. Por outro lado, para uma classe fictícia chamada *Money* composta por valor e moeda, dois objetos com os mesmos atributos podem ser geralmente considerados iguais, sem gerar corrupções de dados ou impactos para os usuários finais do sistema. Em resumo, quando um objeto definido principalmente por sua identidade, ele é chamado de Entidade (EVANS, 2004).

### 2.1.3 Value Object (VO)

Muitos objetos não possuem identidade conceitual, sendo somente utilizados para descrever características de outro objeto. São objetos que descrevem coisas, classificados como VO (EVANS, 2004). Para um sistema de uma loja de roupas, por exemplo, um objeto que representa cor de cada peça formado pelas propriedades *red*, *green* e *blue*, não possui identidade. Não necessita ser diferenciável de outro objeto com as mesmas propriedades. Este objeto somente descreve uma característica de uma peça de roupa. É importante ressaltar, porém, que a diferenciação entre Entidade e *Value Object* depende do contexto do software. Para um sistema de software de computação gráfica, a cor é um conceito chave no sistema e pode possuir identidade. Pode-se pensar que é mais vantajoso sempre projetar os objetos do sistema como Entidade, pois adicionaria identidade, facilitaria o rastreamento de mudanças e agregaria mais recursos ao objeto. No entanto, a adição de identidade para objetos que não necessitam desta classificação pode prejudicar o desempenho do sistema, adicionar mais esforço de análise e aumentar a complexidade do modelo de domínio (EVANS, 2004). Por esse motivo, recomenda-se a utilização de VO sempre que for possível, pois se tratam de objetos mais fáceis de gerenciar.

### 2.1.4 Serviço

Em muitos casos, há operações importantes no modelo que não pertencem conceitualmente a nenhuma Entidade ou VO. Dessa forma, surge o conceito de Serviço: uma operação fornecida como uma interface autônoma no modelo, sem encapsular estado como outros tipos de objeto (EVANS, 2004).

Um erro comum é abusar da utilização de Serviços, desistindo de encontrar o objeto apropriado para uma operação de domínio. Da mesma forma, a inserção de um método que não está alinhado com a definição do objeto resulta na perda de coesão desse objeto, tornando-o mais difícil de compreender e refatorar (EVANS, 2004). Considerando esses pontos, um serviço deve ser utilizado quando um processo ou transformação relevante no modelo de domínio não faz parte da responsabilidade natural de uma Entidade ou VO.

Além disso, Evans (2004) destaca três características de um bom Serviço:

- A operação não possui estado (*stateless*).
- A interface é definida em termos de outros elementos do modelo de domínio.
- A operação se relaciona a um conceito do modelo que não é parte natural de uma Entidade ou VO.

### 2.1.5 *Aggregate*

Um *Aggregate* é um conjunto de objetos associados, tratados como uma única unidade para o propósito de alterações de dados (EVANS, 2004). Em um sistema de grande porte, é difícil garantir a consistência de alterações em objetos com relacionamentos complexos. As invariantes que precisam ser mantidas geralmente envolvem um grupo de objetos intimamente associados, ao invés de objetos isolados. Assim, o padrão *Aggregate* propõe delimitações definidas e restrições de acesso visando o melhor tratamento das associações entre objetos.

Cada *Aggregate* possui uma raiz e uma delimitação, que define o que está dentro do *Aggregate* (EVANS, 2004). A raiz é a única entidade global - que possui identidade em todo o sistema - contida dentro do *Aggregate*, também conhecida como *Aggregate root* (EVANS, 2004). Objetos fora da delimitação do *Aggregate* só podem possuir referências a raiz e não a outros objetos dentro do *Aggregate*. Excluindo a raiz, outras entidades somente possuem identidade local, necessitando somente ser distinguíveis dentro dos limites do *Aggregate*, visto que nenhum objeto de fora consegue obtê-las desassociadas da entidade raiz (EVANS, 2004).

### 2.1.6 *Bounded Context (BC)*

Em grandes projetos, muitos modelos coexistem em diferentes contextos. Quando a base de código, no entanto, combina diferentes modelos sem que haja uma separação clara, o *software* se torna defeituoso, frágil e difícil de ser compreendido (EVANS, 2004). Assim, é necessário especificar em qual o contexto um determinado modelo está sendo aplicado. Assim, um BC delimita a aplicabilidade de um modelo, de tal forma que os membros da

equipe tenham um entendimento claro do que precisa ser consistente e a relação do modelo com outros contextos. Um [BC](#) engloba tanto componentes de modelo como *Aggregates*, Entidades, *Value Objects* e Serviços, quanto detalhes técnicos como persistência. É uma parte completa e funcional de software sob responsabilidade de uma única equipe.

Por exemplo, um *software* para um *e-commerce* possui um módulo que lida com pedidos, em que cada pedido contém um ou mais produtos. Da mesma forma, outra equipe é responsável por um módulo que realiza o gerenciamento de produtos em estoque. O conceito de um produto para um sistema de pedidos é distinto de uma aplicação de estoque. Enquanto este pode ter um enfoque no armazém, na categoria e no estoque do produto, aquele está mais preocupado com o preço, nome e disponibilidade. Assim, ao tratar esses dois conceitos da maneira igual, confusões e defeitos são gerados.

## 2.2 Sistema Distribuído

Um sistema distribuído é um conjunto de componentes independentes entre si que se apresenta aos seus usuários de maneira transparente como um sistema único ([TANENBAUM, 2010](#)). São sistemas compostos por diversas partes cooperantes, que são executadas geralmente em processos diferentes interconectados por rede a fim de realizar alguma tarefa.

Diferentemente de sistemas monolíticos, nos quais todos os módulos estão interligados em uma única unidade, os sistemas distribuídos dividem o processamento entre as partes. Essa abordagem traz vantagens significativas, como a possibilidade de escalabilidade horizontal e maior flexibilidade para atualizar e substituir componentes individuais sem afetar o sistema como um todo e o compartilhamento de recursos.

Um dos principais desafios desse tipo de sistema é a concorrência e coordenação entre os componentes. Nessa abordagem, vários componentes podem estar operando simultaneamente e em diferentes locais físicos, o que gera problemas de concorrência, onde múltiplos processos tentam acessar os mesmos recursos compartilhados ou dados simultaneamente. Por esse motivo, torna-se essencial a coordenação entre esses processos para evitar conflitos e garantir a consistência dos dados.

### 2.2.1 [Service-Oriented Architecture \(SOA\)](#)

A arquitetura orientada a serviços ou [Service-Oriented Architecture \(SOA\)](#) é estilo arquitetural no qual um sistema é dividido em componentes de software chamados de serviços, responsáveis por satisfazer uma necessidade de negócio específica.

As políticas, práticas e *frameworks* permitem que a funcionalidade da aplicação

seja fornecida e consumida como conjuntos de serviços publicados em uma granularidade relevante para o consumidor do serviço. Os serviços podem ser invocados, publicados e descobertos, e são abstraídos da implementação usando uma única forma de interface baseada em padrões (PROTT; WIKES, 2009).

As principais vantagens da SOA são: reusabilidade dos serviços, simplicidade de manutenção, facilidade de escalar os serviços, maior disponibilidade do sistema e possibilidade de utilização de diferentes tecnologias nos serviços.

Porém, essa arquitetura possui uma maior complexidade para *design*, desenvolvimento, teste e implantação. Além desses fatores, é importante ressaltar um nível maior de dificuldade para depurar erros e encontrar profissionais qualificados para atuar nessa área.

## 2.3 Microserviços

A *Arquitetura de Microserviços (AMS)* é um subtipo da SOA. No entanto, trata-se de uma subcategoria que orienta como as fronteiras entre os serviços devem ser traçadas e que possui como ponto central a independência de implantação entre os serviços (NEWMAN, 2021).

Esse estilo arquitetural foca na criação de um sistema robusto, escalável e de fácil manutenção, desenvolvido através da composição de serviços pequenos, flexíveis e que resolvam uma necessidade negócio específica (NEWMAN, 2021). Além disso, cada microserviço deve possuir sua própria base de dados. A troca de informações entre os componentes deve ser realizada através das interfaces fornecidas.

A implantação de um sistema com a utilização de microserviços aumenta a resiliência geral do sistema, visto que possui melhor isolamento de falhas. Caso ocorra um mal-funcionamento em algum componente, somente uma parte das funcionalidades ficará indisponível. Tratando-se de um monólito, por outro lado, grande parte ou a totalidade dos recursos são afetados, pois todo o sistema é acoplado em uma única unidade.

Cada serviço na AMS pode ser escalado de maneira independente de outros serviços utilizando, por exemplo, escalonamento horizontal ou vertical (RICHARDSON, 2018). Ademais, para cada serviço pode ser selecionado o *hardware* mais adequado para seu tipo de processamento. Dessa forma, uma parte do sistema que requer maior utilização da CPU pode ser implantada em uma infraestrutura diferente de outra seção que necessita realizar mais operações de entrada e saída.

Outro benefício dessa arquitetura é o fato de cada serviço ser relativamente pequeno, sendo, assim, de fácil compreensão por todos que trabalham na base de código. Adicionalmente, um serviço pequeno possibilita um bom desempenho da *Integrated Development*

[Environment \(IDE\)](#) e também pode ser executado localmente de maneira rápida. Esses fatores contribuem para um aumento de produtividade da equipe ([RICHARDSON, 2018](#)).

Por outro lado, com um sistema composto de múltiplos componentes, torna possível a utilização de diferentes tecnologias dentro de cada serviço. Dessa forma, a ferramenta mais apropriada para resolver a necessidade específica do serviço pode ser selecionada, eliminando a obrigatoriedade de escolher uma única tecnologia para todo o sistema ([NEWMAN, 2021](#)).

### 2.3.1 Comunicação entre microsserviços

Quando um sistema é composto por um único monólito, a comunicação entre os diferentes módulos é simplificada, pois se trata de um único processo em nível do sistema operacional e, portanto, compartilha a área da memória principal. No entanto, microsserviços são implantados em diferentes processos (usualmente em diferentes computadores), assim se faz necessária a utilização de algum mecanismo de comunicação entre processos, também conhecido como [Inter-Process Communication \(IPC\)](#).

Existe no mercado uma série de tecnologias, protocolos e ferramentas que viabilizam a comunicação entre microsserviços. Porém, antes de selecionar o mecanismo de [IPC](#), é importante compreender os estilos de interações entre sistemas ([RICHARDSON, 2018](#)). Essa abordagem coloca foco nos requisitos da aplicação, no lugar de detalhes de um mecanismo específico.

Os modos de interação entre um cliente e um serviço podem ser classificados em duas dimensões: a quantidade de serviços que processam a requisição e a sincronicidade. No que diz respeito à primeira dimensão, existem duas opções: "**um para um**", em que cada requisição é processada por um único serviço; e "**um para muitos**", em que cada requisição pode ser processada por vários serviços. Por outro lado, a comunicação pode ser **síncrona**, onde o cliente espera uma resposta imediata e geralmente aguarda essa resposta para continuar seu processamento; ou **assíncrona**, onde o cliente envia a requisição e continua seu processamento sem esperar por uma resposta ([RICHARDSON, 2018](#)).

Quadro 1 – Estilos de comunicação entre microsserviços

	Um para um	Um para muitos
<b>Síncrona</b>	<i>Request/Response</i>	-
<b>Assíncrona</b>	<i>Asynchronous request/response</i> e <i>One-way notifications</i>	<i>Publish/subscribe</i> e <i>Publish/async responses</i>

Fonte: [Richardson \(2018\)](#).

No [Quadro 1](#), observam-se alguns estilos de comunicação categorizados pelas dimensões anteriormente mencionadas. No [Quadro 2](#) são apresentados os diferentes tipos de comunicação **"um para um"**:

Quadro 2 – Padrões de comunicação do tipo **"um para um"**

Nome	Descrição
<i>Request/Response</i>	O serviço cliente realiza uma requisição a outro serviço e espera(geralmente bloqueando sua execução) por uma resposta.
<i>Asynchronous request/response</i>	O serviço cliente envia uma solicitação a um serviço, que responde de forma assíncrona. O cliente não fica bloqueado enquanto espera, pois o serviço pode não enviar a resposta por um longo período.
<i>One-way notifications</i>	O serviço cliente envia uma requisição, mas não espera nenhum tipo de resposta do serviço que processará a requisição.

Fonte: [Richardson \(2018\)](#).

Frequentemente, em ambientes de sistemas distribuídos, há a necessidade de vários serviços consumirem uma mensagem produzida por um determinado serviço. Nesse sentido, no [Quadro 3](#) são listadas algumas opções de padrões para realizar comunicações do tipo **"um para muitos"**:

Quadro 3 – Padrões de comunicação do tipo **"um para muitos"**

Nome	Descrição
<i>Publish/Subscribe</i>	Um cliente publica uma mensagem, que é consumida por zero ou mais serviços..
<i>Publish/async responses</i>	Um cliente publica uma mensagem contendo uma requisição e espera por um determinado período por respostas.

Fonte: [Richardson \(2018\)](#).

### 2.3.2 Estratégias para delimitação de microsserviços

Uma decisão de grande impacto no projeto de sistema que utilize a [AMS](#) é o escopo de cada serviço. Em outras palavras, que funcionalidades serão parte de um determinado serviço e quais serão delegadas a outro componente.

[Newman \(2021\)](#) argumenta que a possibilidade de alterar um microsserviço de maneira isolada é fundamental para essa arquitetura. Além disso, [AMS](#) é apenas outra



forma de decomposição modular, afirma o autor. Assim, os conceitos de *information hiding*, coesão e acoplamento podem ser aplicados nesse contexto para criar as delimitações eficientes entre os serviços.

*Information hiding* é uma abordagem para desenvolvimentos de módulos, na qual se busca esconder o máximo de detalhes possíveis detrás da interface de um módulo (NEWMAN, 2021). Dessa forma, os benefícios esperados com a construção de módulos de alta qualidade são:

- **Tempo de desenvolvimento aprimorado:** com a possibilidade de módulos serem construídos de forma independente, o desenvolvimento pode ser realizado de maneira paralela.
- **Compreensibilidade:** cada módulo pode ser entendido de maneira isolado. Assim, se torna mais fácil compreender o que o sistema como um todo faz.
- **Flexibilidade:** módulos podem ser modificados de maneira independente, possibilitando, dessa forma, alterações em uma parte do sistema sem a necessidade de alterações em outros módulos.

(PARNAS, 2012).

Coesão diz respeito à medida que os elementos dentro de um módulo estão organicamente relacionados entre si (YOURDON; CONSTANTINE, 1979). Outra definição possível é código que, alterado conjuntamente, deve estar localizado no mesmo lugar (NEWMAN, 2021). No contexto da AMS, a adição ou atualização de funcionalidades deve, idealmente, envolver um único serviço. Dessa forma, é possível disponibilizar novos recursos aos usuários de maneira mais ágil (NEWMAN, 2021). Por esse motivo, buscar alta coesão é importante no momento da definição do escopo de cada serviço. O objetivo principal é evitar que sejam necessárias alterações em múltiplos serviços no desenvolvimento de funcionalidades para o sistema.

Quando serviços são fracamente acoplados, uma alteração em um serviço não requer alteração no outro (NEWMAN, 2021). A grande vantagem da AMS é a possibilidade de realizar mudanças em partes do sistema de maneira isolada. Nesse sentido, a projeção de serviços com alta coesão e fraco acoplamento é essencial para que os objetivos da utilização dessa tecnologia sejam atingidos. Nem todo acoplamento é necessariamente maléfico. De fato, algum nível de acoplamento entre os serviços é inevitável. Além disso, há diferentes tipos de acoplamento listados no Quadro 4.

Quadro 4 – Tipos de acoplamento

Tipo	Descrição
<i>Domain Coupling</i>	Descreve uma situação em que um microserviço necessita interagir com outro serviço para fazer uso de sua(s) funcionalidade(s). Na <a href="#">AMS</a> , esse tipo de interação é quase inevitável. No entanto, é necessário que cada serviço exponha o mínimo de detalhes possível ( <i>information hiding</i> ) para não gerar um nível de acoplamento muito alto.
<i>Pass-Through Coupling</i>	Acontece quando um microserviço envia dados a outro serviço somente porque essas informações são necessárias por um terceiro serviço. Trata-se de um dos tipos de acoplamento mais problemáticos, pois implica que o cliente da primeira requisição conheça tanto o serviço que inicialmente recebe a requisição, quanto o que irá, de fato, utilizar os dados.
<i>Common Coupling</i>	Ocorre quando dois ou mais microserviços compartilham o mesmo conjunto de dados. O exemplo mais comum desse tipo de acoplamento é a utilização de uma base de dados compartilhada. O principal problema dessa abordagem é o fato de modificações ou falhas na base de dados compartilhadas afetar mais de um serviço.
<i>Content Coupling</i>	Descreve uma situação em que o serviço acessa e modifica estados internos de outro serviço. É semelhante ao <i>Common coupling</i> , porém nesse caso as linhas de responsabilidades de cada serviço se tornam confusas. Assim, desenvolvedores encontram grandes dificuldades para realizar alterações.

Fonte: [Newman \(2021\)](#).

Visando a construção de serviços com alta coesão e baixo acoplamento, portanto estáveis, os conceitos de [DDD](#) podem ser utilizados. Uma abordagem interessante e comumente aplicada é mapear um [Bounded Context \(BC\)](#) para cada serviço ([NEWMAN, 2021](#)). Como os [Bounded Contexts](#) representam uma parte do domínio onde um modelo se aplica, eles permitem que microserviços sejam modelados ao redor de uma secção coesa e completamente funcional que ofereça funcionalidades de negócio. Outra opção para modelagem de microserviços é mapear cada [Aggregate](#) para um serviço. Essa estratégia oferece um nível de granularidade menor que um [Bounded Context \(BC\)](#), visto que este geralmente engloba múltiplos [Aggregates](#). No entanto, [Newman \(2021\)](#) sugere recorrer primeiro a delimitações maiores na construção de microserviços, englobando um (ou até

mais) [Bounded Context \(BC\)](#). Dessa forma, caso a equipe sinta necessidade de subdividir o serviço, poderá fazê-lo sem complicações.

Apesar de [DDD](#) ser uma excelente maneira de delimitar microsserviços, não é a única técnica disponível ([NEWMAN, 2021](#)). São listadas a seguir algumas alternativas frequentemente utilizadas:

- **Volatilidade:** decomposição baseada em volatilidade refere-se a uma abordagem, na qual partes do sistema que são frequentemente alteradas são extraídas em seus próprios serviços, de forma que possam ser trabalhadas de maneira mais efetiva. Essa estratégia por ser útil em organizações com ritmo de mudanças acelerado. No entanto, se o objetivo da utilização da [AMS](#) é a possibilidade de escalar partes do sistema de maneira independente, decomposição baseada em volatilidade não atingirá a meta.
- **Dados:** a natureza das informações que necessitam ser armazenadas pode direcionar a estratégia de decomposição. Por exemplo, é muito usual restringir quais microsserviços podem acessar informações de identificação pessoal para cumprir com legislações como a Lei Geral de Proteção de Dados Pessoais.
- **Tecnologia:** A necessidade de se utilizar diferentes tecnologias também pode ser um fator a se considerar na definição do escopo de um microsserviço. Se há a necessidade, por exemplo, de alto desempenho em uma parte do sistema que demande a utilização de uma linguagem de programação diferente, este pode ser um ponto importante para delimitação do serviço.

([NEWMAN, 2021](#)).

É importante mencionar que essas opções não são mutualmente exclusivas. Em um sistema real, provavelmente uma mistura de estratégias necessitará ser aplicada para satisfazer os requisitos funcionais e não-funcionais.

### 2.3.3 Desafios na implementação de microsserviços

Certamente, não há nenhuma tecnologia que não possua limitações. Isso não é diferente com a [AMS](#) ([RICHARDSON, 2018](#)).

Primeiramente, como microsserviços baseiam-se no conceito de [Sistema Distribuído](#), uma série de novos problemas estão presentes nessa arquitetura. Como os serviços necessitam utilizar algum mecanismo de [IPC](#), há um grande aumento na complexidade do ciclo de desenvolvimento como um todo. [IDEs](#) e outras ferramentas de desenvolvimento possuem maior foco no desenvolvimento de aplicações monolíticas. Adicionalmente, escrever

testes automatizados envolvendo múltiplos serviços é uma tarefa complexa. Além disso, a AMS introduz dificuldades significantes de operação, como necessidade de alto nível de automação e ferramentas de monitoramento avançadas. Assim, uma equipe técnica altamente qualificada se torna necessária, podendo inclusive ser útil a contratação de consultorias especializadas nessa tecnologia (RICHARDSON, 2018).

Por outro lado, a AMS engloba desafios para manter a consistência dos dados. Enquanto em um sistema monolítico, dados são usualmente armazenados em uma única base de dados relacional, que oferece suporte a transações ACID, em um sistema distribuído, no qual múltiplos banco de dados - possivelmente de tipos e fornecedores diferentes - são empregados, esse tipo de segurança não pode ser atingido facilmente (NEWMAN, 2021). Transações distribuídas podem ser aplicadas para reduzir problemas de consistência, no entanto, essas técnicas adicionam grande complexidade ao sistema (NEWMAN, 2021).

## 2.4 Arquitetura Hexagonal

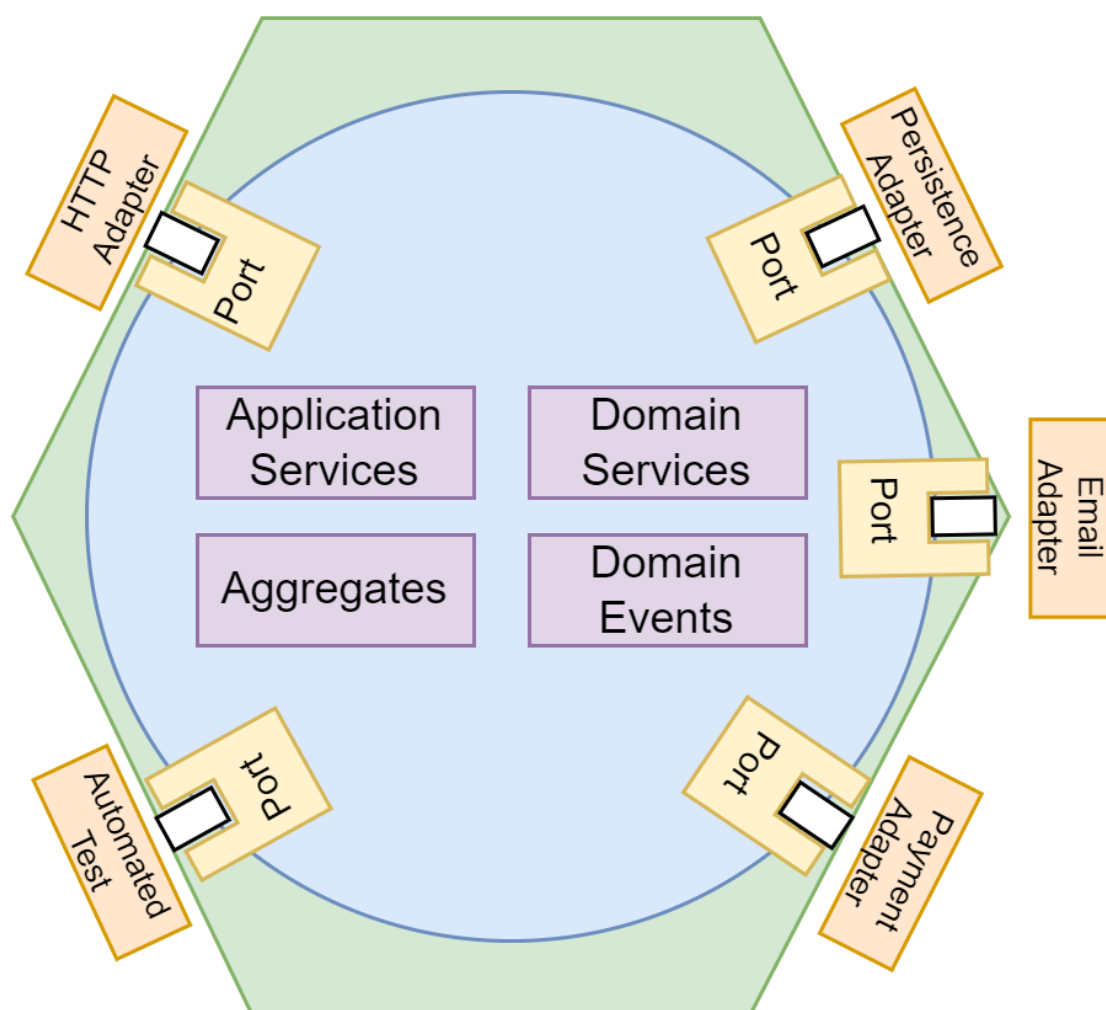
A arquitetura hexagonal, também conhecida como *Ports and Adapters*, possui como objetivo isolar regras de negócio de agentes externos (COCKBURN, 2005). Nesse sentido, o que está dentro do hexágono representa o modelo de domínio. Do mesmo modo, componentes de fora representam detalhes de implementação, como persistência, interface de usuário, envio de *e-mails*, entre outros. Um ponto crucial nessa arquitetura é a direção das dependências: o que está dentro do hexágono não pode depender do que está fora. Assim, é garantida a independência e o isolamento de regras de negócio das dependências técnicas.

Na Figura 2, pode-se observar que a injeção de agentes externos, como persistência em um banco de dados, é realizada por meio de portas definidas dentro do núcleo da aplicação. As portas, nessa arquitetura, são geralmente representadas por interfaces em linguagens orientadas a objetos. Então, um adaptador fornece uma implementação de uma tecnologia específica para a porta. Assim, implementações para tecnologias específicas são definidas em função do modelo de domínio.

Outros benefícios importantes, além do isolamento das regras de negócio, são: facilidade de testar o núcleo da aplicação com a utilização de *mocks* e *stubs*; possibilidade de utilização da mesma aplicação via interfaces de usuário, testes automatizados e *batches*.

Enquanto a AMS visa organizar um sistema como um conjunto de serviços independentes, cada um executando uma função específica, a arquitetura hexagonal concentra-se na organização interna de um serviço, propondo uma separação clara entre a lógica de negócios e as implementações técnicas, usando portas e adaptadores. Por outro lado, DDD apresenta-se como uma excelente abordagem para desenvolvimento do núcleo da aplicação

Figura 2 – Arquitetura Hexagonal



Fonte: o autor

na arquitetura hexagonal. Assim, esses três conceitos: microserviços, arquitetura hexagonal e [DDD](#) podem ser combinados para o desenvolvimento de sistemas altamente coesos, desacoplados, escaláveis, inteligíveis e sustentáveis.

## 3 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados ao objeto de pesquisa obtidos utilizando o protocolo descrito na seção a seguir.

### 3.1 Protocolo

Esta seção apresenta o protocolo de mapeamento sistemático da literatura usado para atingir os objetivos da pesquisa.

#### 3.1.1 Questões de pesquisa

- Q1: Quais são as principais estratégias para elaboração de sistemas com microserviços e [DDD](#)?
- Q2: Quais são os principais desafios na utilização de [DDD](#) como estratégia de delimitação de microserviços?
- Q3: Quais são os anti-padrões a serem evitados na utilização de [DDD](#) com microserviços?

#### 3.1.2 Estratégia de busca

Esta seção apresenta a estratégia de buscas de artigos científicos e livros relacionados à pesquisa. As ferramentas utilizadas para realizar as buscas são:

- **Periódicos Capes:** É uma ferramenta disponibilizada pelo governo federal para uso de estudantes e pesquisadores. Acessando através da instituição de ensino ou pesquisa, é possível ter acesso completo a uma grande quantidade de artigos científicos publicados em variadas revistas, conferências e universidades. A principal vantagem dessa ferramenta é a possibilidade de ler o conteúdo integral de grande parte das publicações disponíveis. Por outro lado, as expressões de busca atualmente suportadas são bem limitadas.
- **Scopus:** Trata-se de uma ferramenta similar ao Periódicos Capes. No entanto, o *Scopus* permite a elaboração de expressões de buscas mais complexas e sofisticadas, servindo para descobrir publicações não detectadas pelas outras plataformas. Além disso, possui um acervo bem mais amplo que o Periódicos Capes. Entretanto, algumas publicações não podem ser vistas na íntegra de forma gratuita.

- **Google Docs:** Ferramenta desenvolvida pela *Google LLC* que permite a criação e edição de documentos de texto. Suas grandes vantagens em relação a ferramentas de outros fornecedores são as avançadas ferramentas de colaboração e a possibilidade de acesso por meio de navegadores *web*, sem necessidade de instalação de *software* específico.
- **Google Sheets:** Com as mesmas características e vantagens do *Google Docs*, essa ferramenta fornece recursos para elaboração de planilhas de cálculo. É muito útil para realizar análise de dados simples e também visualizar e apresentar dados tabulares.

Como grande parte das publicações na área de computação são em inglês, esta pesquisa utiliza esse idioma para fazer buscas nas ferramentas indicadas. Além disso, [Arquitetura de Microsserviços \(AMS\)](#) e [Domain-Driven Design \(DDD\)](#) são relativamente recentes, as buscas se limitaram a publicações feitas nos últimos 20 anos.

Os termos-chave para realização das buscas são: Microsserviço, [DDD](#) e [Domain-Driven Design](#). Como a busca é feita em inglês, se usará *microservice* nas buscas.

### 3.1.3 Expressão de busca

Quadro 5 – Expressão de busca utilizada

Expressão de Busca
$( ( TITLE-ABS-KEY ( microservice ) AND TITLE-ABS-KEY ( domain-driven AND design ) ) OR ( TITLE-ABS-KEY ( microservice ) AND TITLE-ABS-KEY ( ddd ) ) )$

Fonte: o autor

No [Quadro 5](#), percebe-se que a expressão de busca pretende retornar todas as publicações que contenham as palavras chaves no título, resumo ou na seção de *keywords*.

### 3.1.4 Estratégia de seleção

A seguir são apresentados critérios para inclusão de publicações na pesquisa.

- Texto completo disponível de forma gratuita pelo portal Periódicos Capes.
- Materiais relacionados ao tópico de interesse, ou seja, título ou resumo.
- Publicações com ao menos 5 citações.

Por outro lado, estes são os critérios para exclusão de publicações.

- Publicações duplicadas.
- Materiais que não dispõem de informação relevante para responder às questões de pesquisa.

### 3.1.5 Estratégia para extração de dados e análise

Para atingir o objetivo do mapeamento da literatura, são filtrados manualmente nos artigos selecionados segundo os critérios de inclusão e exclusão. A partir da listagem reduzida, todas as publicações são lidas de forma integral. Adicionalmente, todos os gráficos e tabelas nos artigos selecionados são avaliados visando extrair algum dado que permita realizar a comparação entre aspectos quantitativos das estratégias como, tempo de resposta, latência e taxa de transferência.

Informações qualitativas como recomendações, destaques, conceitos e estudos de casos são registrados em um documento no *Google Docs*. Dados quantitativos como taxa de transferência, latência, tempo de processamento são armazenados em uma planilha no *Google Sheets*.

A pesquisa é executada seguindo os passos a seguir:

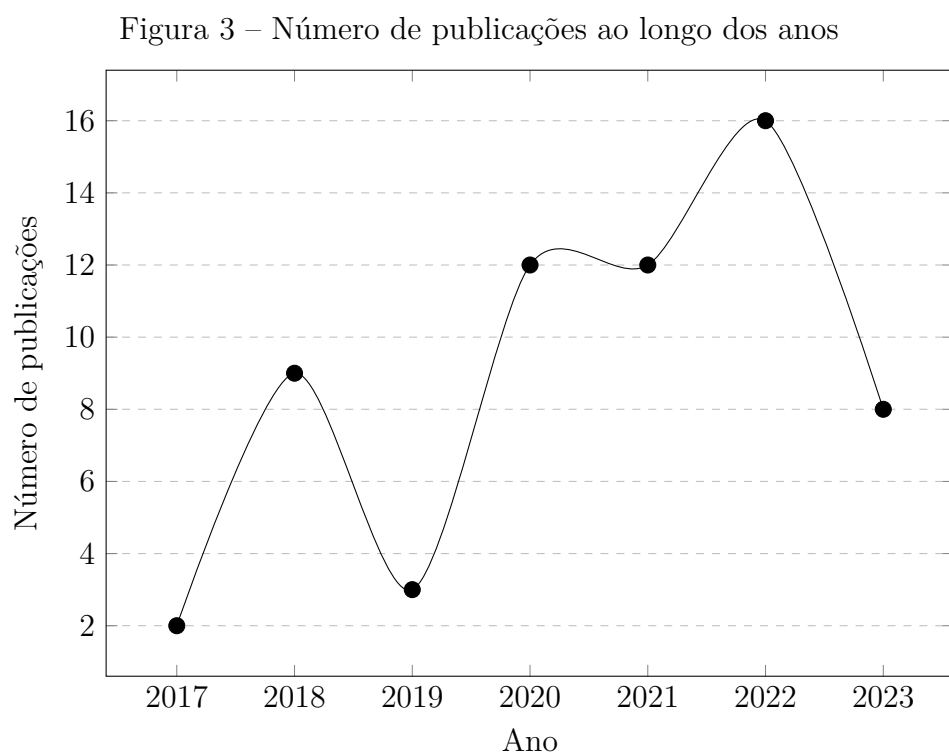
1. As expressões de busca citadas são inseridas nas ferramentas mencionadas.
2. É feito o armazenamento das publicações retornadas em uma *planilha de cálculo*.
3. As publicações retornadas são filtradas conforme os critérios de inclusão e exclusão.
4. Em cada artigo selecionado, é realizada a extração dos dados relevantes para responder às questões de pesquisa.
5. Finalmente, são produzidas respostas para questões de pesquisa com as informações extraídas das publicações.

## 3.2 Caracterização dos estudos

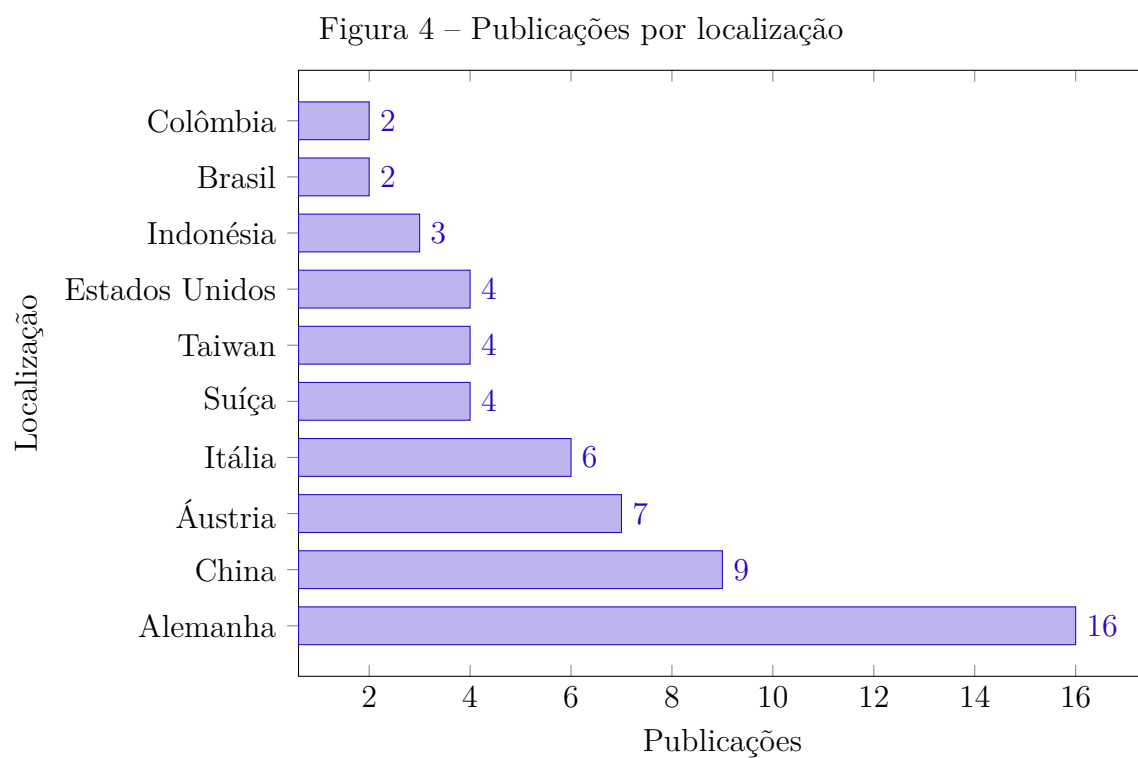
Na *Figura 3*, pode-se observar a quantidade de publicações por ano, encontradas a partir da expressão de busca descrita na seção 3.1.3. No total, foram obtidos 65 resultados. Percebe-se, claramente, um aumento exponencial no número de publicações sobre o tema, principalmente a partir de 2019, demonstrando crescimento no interesse de se realizar pesquisas sobre o assunto.

A Alemanha lidera em número de publicações, como pode ser visualizado na *Figura 4*. Por outro lado, o Brasil só possui duas entre as sessenta e cinco publicações retornadas.





Fonte: *Scopus*



Fonte: *Scopus*

Utilizando-se os critérios de inclusão e exclusão descritos nas seções 3.1.4 e 3.1.4, respectivamente, dos 65 artigos retornados, 27 foram selecionados inicialmente. Após realizada as leituras, apenas 16 publicações mostraram-se relevantes para responder às questões de pesquisa e/ou apoiar na elaboração do estudo de caso. Esta listagem está disponível no [Quadro 6](#).

### 3.3 Estratégias para elaboração de sistemas com microserviços e DDD

Todas as publicações revisadas sugerem o mapeamento de cada [Bounded Context \(BC\)](#) para um microserviço. [Vural e Koyuncu \(2021\)](#) afirmam que o objetivo principal da delimitação é alcançar serviços com baixo acoplamento e alta coesão. Porém, o grande desafio está na definição de maneira apropriada do escopo de cada [BC](#).

[Singjai et al. \(2021a\)](#) e [Ma et al. \(2022\)](#) mencionam os padrões *Open Host Service (OHS)* e *Anti-Corruption Layer (ACL)* como estratégias para comunicação, conceitualmente, entre [Bounded Contexts](#). No [OHS](#), o serviço que envia as mensagens implementa uma camada extra com objetivo de realizar a tradução para um formato que possa ser processado pelo serviço receptor. Dessa forma, os detalhes de implementação do serviço cliente não são expostos, diminuindo o acoplamento entre as partes. Semelhantemente, o [ACL](#) é uma camada extra inserida no serviço que recebe as mensagens. Trata-se de uma abordagem útil quando o serviço receptor não deseja aderir ao contrato do componente que produz as mensagens ([EVANS, 2004](#)).

Outro ponto importante levantando no material revisado é o mapeamento do modelo de domínio para uma [Application Programming Interface \(API\)](#) para possibilitar o consumo das funcionalidades tanto por outros serviços quanto de aplicações cliente. Algumas opções citadas por [Singjai et al. \(2021a\)](#) são:

- Expor todo o modelo de domínio em uma relação de 1 para 1 com a [API](#).
- Expor uma parte do modelo como uma [API](#).
- Expor cada [BC](#) como uma [API](#).
- Expor parte dos [BCs](#) como [APIs](#).

As alternativas mais indicadas são a exposição total ou de grupo de [BCs](#) como [APIs](#). Além disso, [Singjai et al. \(2021a\)](#) e [Singjai et al. \(2021b\)](#) levantam abordagens para definição do contrato de [API](#) de acordo com modelo de domínio. As duas principais opções apresentadas pelos autores são: explicitamente definir o contrato de [API](#) e extrair o

<b>Título</b>	<b>Ano</b>
<i>Microservice Migration Using Strangler Fig Pattern and Domain-Driven Design</i>	2022
<i>Microservice architecture and model-driven development: Yet singles, Soon Married ?</i>	2018b
<i>Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?</i>	2021
<i>Modeling Microservices with DDD</i>	2020
<i>Following Domain Driven Design principles for Microservices decomposition: Is it enough?</i>	2021
<i>An Ontology-based Approach for Domain-driven Design of Microservice Architectures</i>	2017
<i>Challenges of domain-driven microservice design: A model-driven perspective</i>	2018a
<i>Model-based engineering for microservice architectures using Enterprise Integration</i>	2017
<i>Patterns on Deriving APIs and their Endpoints from Domain Models</i>	2021b
<i>Refactoring with domain-driven design in an industrial context: An action research report</i>	2023
<i>A microservice based reference architecture model in the context of enterprise architecture</i>	2017
<i>Design of Domain-driven Microservices-based Software Talent Evaluation and Recommendation System</i>	2022
<i>Partitioning microservices: A domain engineering approach</i>	2018
<i>Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory</i>	2021a
<i>A Systematic Framework of Application Modernization to Microservice based Architecture</i>	2021
<i>Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling</i>	2020

Quadro 6 – Publicações selecionadas

contrato de [API](#) a partir do modelo. [Singjai et al. \(2021a\)](#) argumenta que ambas opções auxiliam na obtenção da separação do contrato de [API](#) e das responsabilidades do modelo de domínio.

Para este trabalho de conclusão de curso, a estratégia de mapeamento de um [BC](#) para um microserviço é utilizada. Com objetivo de habilitar a comunicação entre subdomínios, o padrão [Open Host Service \(OHS\)](#) é empregado. Adicionalmente, o contrato de [API](#) será definido explicitamente a partir do modelo de domínio. Assim, pretende-se utilizar as estratégias mais indicadas pelos autores revisados na elaboração do estudo de caso.

### 3.4 Desafios na utilização de DDD como estratégia de delimitação de microserviços

[Rademacher et al. \(2018a\)](#) menciona três principais desafios da utilização de [DDD](#) no contexto da [AMS](#). Adicionalmente, [Diepenbrock et al. \(2017\)](#) cita desafios semânticos juntamente com sua proposta de metamodelo.

Para fomentar o foco em conceitos relevantes e *design* efetivo, modelos de domínio tipicamente omitem informações obrigatórias para extração de microserviços, como:

- Interfaces e operações;
- Parâmetros e tipos de retorno das operações
- *Endpoints*, protocolos e formatos de mensagens.

([RADEMACHER et al., 2018a](#)).

Informações específicas sobre esses pontos são cruciais para implementação dos serviços. Considerando diferentes [BCs](#) conectados entre si no modelo, e que cada [BC](#) é mapeado para um microserviço, um componente vai necessitar acessar instâncias de um outro serviço. Assim, uma operação desse tipo deve ser fornecida e usualmente não é especificada no modelo, deixando espaço para ambiguidade ([RADEMACHER et al., 2018a](#)).

Modelos de domínio intencionalmente não compreendem componentes de infraestrutura da [AMS](#) ([RADEMACHER et al., 2018a](#)). Esses componentes incluem *API Gateways*, *containers*, banco de dados, entre outros. Por outro lado, requisitos do modelo podem afetar questões técnicas e esses pontos devem ser documentados separadamente do modelo de domínio ([RADEMACHER et al., 2018a](#)). Por exemplo, caso um microserviço que

realize o gerenciamento de usuários necessite ser acessível externamente para faturamento, configurações em diferentes componentes técnicos como *API Gateway* necessitarão ser aplicadas.

Responsabilidade sobre um microserviço é geralmente atribuída a um único time, devido à alta coesão e baixo acoplamento (RADEMACHER et al., 2018a). Essa equipe é responsável pela implementação do serviço, operação, *design* e manutenção desse *Bounded Context* (BC). Dessa forma, surgem desafios relacionados a visibilidade dos modelos e gerenciamento de alterações.

Primeiramente, é crucial definir a visibilidade de cada BC, ou seja, especificar quais equipes terão acesso a quais modelos de domínios (RADEMACHER et al., 2018a). Além disso, a permissão para realizar alterações é uma consideração essencial. Embora seja possível conceder a outras equipes privilégios para modificar outros BC, essa abordagem apresenta a desvantagem de possibilitar alterações, por vezes críticas, efetuadas por profissionais que não estão familiarizados com o contexto específico. No entanto, restringir exclusivamente à equipe responsável a capacidade de realizar mudanças pode resultar em gargalos significativos, especialmente em projetos envolvendo diversos contextos.

Diepenbrock et al. (2017) apresenta uma série de desafios semânticos na elaboração de microserviços com DDD. Inicialmente, um problema de semântico típico ocorre quando um atributo de um conceito de domínio é derivado de outro atributo. Por exemplo, quando um atributo de uma entidade é criado a partir da concatenação de dois outros atributos de outra entidade, ocorre um problema de semântica porque os dados ficam fragmentados.

Simultaneamente, é importante reconhecer que diferentes BCs podem interpretar os conceitos de domínio compartilhados de maneiras distintas (DIEPENBROCK et al., 2017). Nesse contexto, surgem desafios como atributos com nomes diversos, mas significados idênticos, bem como propriedades com o mesmo nome, porém, com significados diferentes. Esse risco é amplificado no contexto da AMS, onde é comum que diferentes BCs sejam desenvolvidos por equipes distintas. Esse ambiente descentralizado pode potencializar a disparidade de interpretações e a falta de consistência nos conceitos de modelos compartilhados.

Além disso, um mecanismo de definição de identificadores únicos deve ser definido entre as equipes para permitir a identificação semântica de diferentes conceitos do modelo com objetivo de tornar os serviços capazes de distinguir diferentes elementos em um contexto distribuído (DIEPENBROCK et al., 2017).

Por essas motivações, Diepenbrock et al. (2017) apresenta uma abordagem para DDD no contexto da AMS. O autor propõe um metamodelo que representa a sintaxe abstrata de um linguagem formal de modelagem. Em outras palavras, ele define os

conceitos suportados pela linguagem e seus relacionamentos. Os principais componentes do metamodelo são: *External Context*, *Bounded Context*, *Domain Model*, *Attribute* e *Association*. O objetivo principal desse modelo é minimizar os impactos dos desafios semânticos da utilização dessas tecnologias (DIEPENBROCK et al., 2017).

### 3.5 Anti-padrões a serem evitados

Farsi et al. (2021), Vural e Koyuncu (2021), Singjai et al. (2021b), Özkan et al. (2023) e Singjai et al. (2021a) identificaram uma série de anti-padrões no contexto da AMS e DDD. Um compilado pode ser observado no Quadro 7.

Os anti-padrões mencionados são importantes para o desenvolvimento do estudo de caso, pois devem ser evitados para que os benefícios da utilização conjunta de DDD e AMS sejam alcançados.

### 3.6 Discussões

O mapeamento da literatura mostrou-se efetivo para responder às questões de pesquisa, na medida em que foram encontradas publicações que auxiliam na resolução das perguntas-chave. O levantamento das estratégias, desafios e anti-padrões foi realizado com êxito a partir de artigos de alta qualidade.

No entanto, publicações em outros idiomas que não foram revisadas, assim como publicações não indexadas pelas ferramentas de busca utilizadas, representam uma ameaça à validade da pesquisa. Isso se deve ao fato de que informações essenciais podem não ter sido revisadas devido a essas limitações.

Por fim, as informações obtidas nesse mapeamento contribuem tanto para atingir os objetivos gerais da pesquisa quanto para a elaboração do estudo de caso.

Quadro 7 – Anti-padrões

Nome	Descrição
<i>Chattiness of a service</i>	Refere-se a excessiva comunicação entre microserviços que gera ineficiência devido à latência de rede.
<i>Nanosevice</i>	Um serviço excessivamente granular no qual a sobrecarga de comunicação, manutenção e operação supera sua utilidade.
<i>Anemic domain model</i>	Trata-se de um modelo de domínio em que os objetos contêm pouca ou nenhuma regra de negócio. As invariantes são misturadas com outras lógicas, o que dificulta manutenção e refatoração.
<i>Data class</i>	Similar ao <i>Anemic domain model</i> , esse tipo de classe só contém, além de atributos, <i>getters</i> e <i>setters</i> . Os comportamentos são criados fora da classe, o que reduz coesão e dificulta a manutenção.
<i>Distributed monoliths</i>	Refere-se a um sistema que externamente se assemelha a <a href="#">Arquitetura de Microserviços</a> , porém possui um alto nível de acoplamento entre os componentes, diminuindo assim as vantagens dos microserviços.
<i>Start with API Design</i>	Trata-se de uma abordagem na qual o contrato de <a href="#">API</a> é definido antes do modelo de domínio. <a href="#">Singjai et al. (2021b)</a> levantou que esta estratégia costuma gerar a criação de <i>Anemic domain models</i> .
<i>Feature envy</i>	Acontece quando um método está mais interessado em uma classe diferente da que está inserido.
<i>Inappropriate intimacy</i>	Descreve um par de classes não relacionadas conceitualmente, mas que possuem grande acoplamento entre si.
<i>Message chain</i>	Uma cadeia de mensagem ocorre quando um cliente envia uma mensagem a outro objeto que, por sua vez, a envia outro o objeto e assim por diante.

Fonte: [Farsi et al. \(2021\)](#), [Vural e Koyuncu \(2021\)](#), [Singjai et al. \(2021b\)](#), [Özkan et al. \(2023\)](#) e [Singjai et al. \(2021a\)](#)

## 4 Estudo de Caso: Requisitos, Organização e Métodos

Este capítulo apresenta o contexto, os requisitos, a organização e os métodos utilizados no desenvolvimento do estudo de caso em uma locadora de veículos chamada *CarroFacil*.

### 4.1 Contexto

O estudo de caso é realizado em uma empresa fictícia chamada *CarroFacil*. Essa empresa é uma locadora de veículos que atua em todo o território nacional. Ela possui uma frota de veículos própria e uma grande quantidade de clientes. Eles podem alugar veículos por períodos de tempo variados, desde horas até semanas. A *CarroFacil* possui um sistema de locação de veículos que foi desenvolvido há alguns anos e está apresentando problemas de escalabilidade, desempenho e manutenibilidade. Por esse motivo, a empresa decidiu desenvolver um novo sistema de locação de veículos utilizando microserviços e [DDD](#).

A [Figura 5](#) apresenta o diagrama de caso de uso do sistema de locação de veículos. As seções a seguir apresentam um detalhamento dos casos de uso do sistema.

### 4.2 Processo de Desenvolvimento

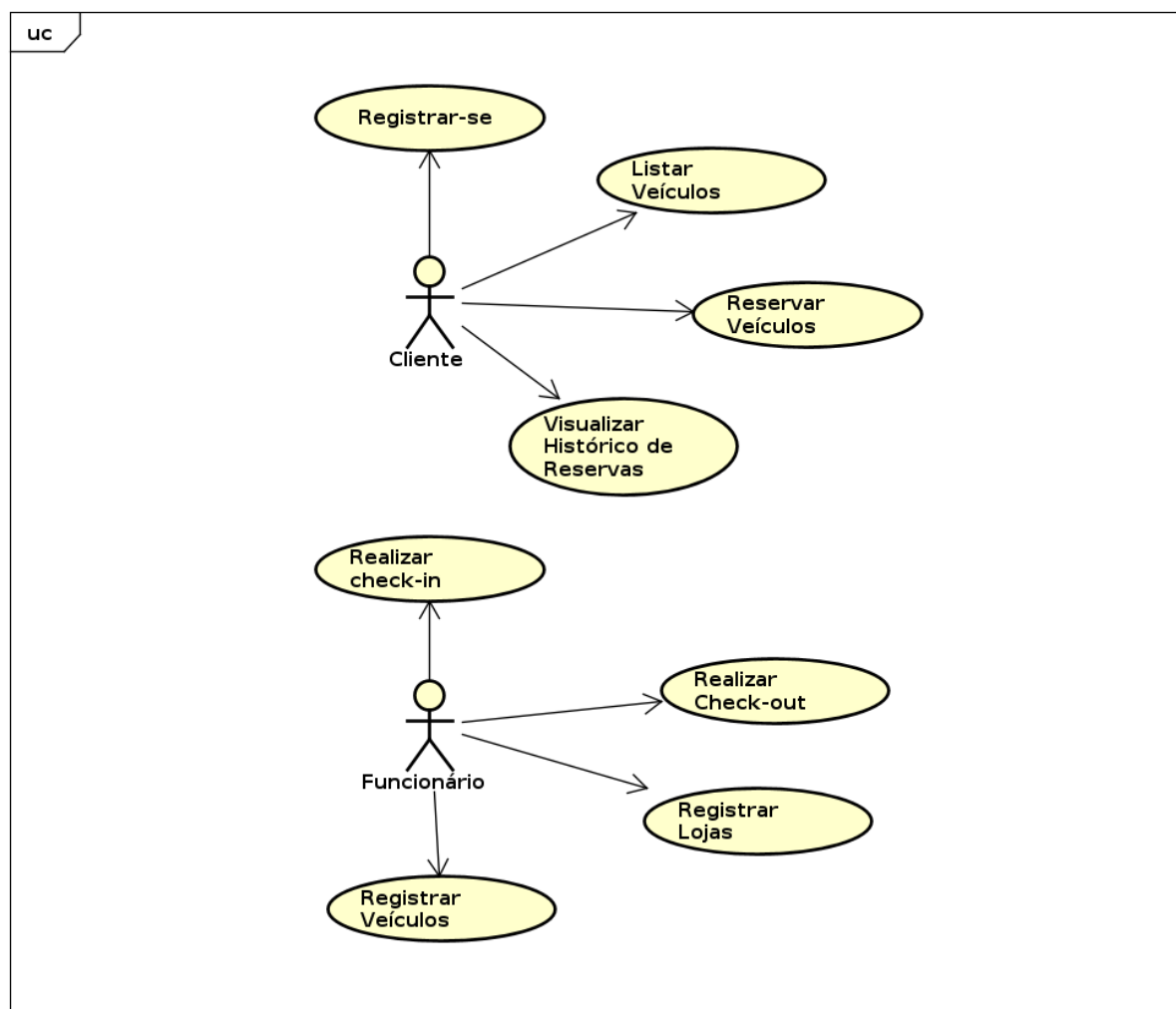
O processo de desenvolvimento utilizado para construir o sistema de locação de veículos é o *Kanban*. Essa metodologia de desenvolvimento ágil é baseada em um quadro de tarefas, no qual cada tarefa é representada por um cartão ([GOMES, 2014](#)). O quadro é dividido em colunas que representam o estado atual de cada tarefa. As colunas mais comuns são: *To Do*, *Doing* e *Done*. O quadro é atualizado conforme as tarefas são realizadas. A [Figura 6](#) apresenta um exemplo de quadro *Kanban* que é utilizado para mapear as tarefas e medir o progresso do desenvolvimento deste sistema.

### 4.3 Requisitos

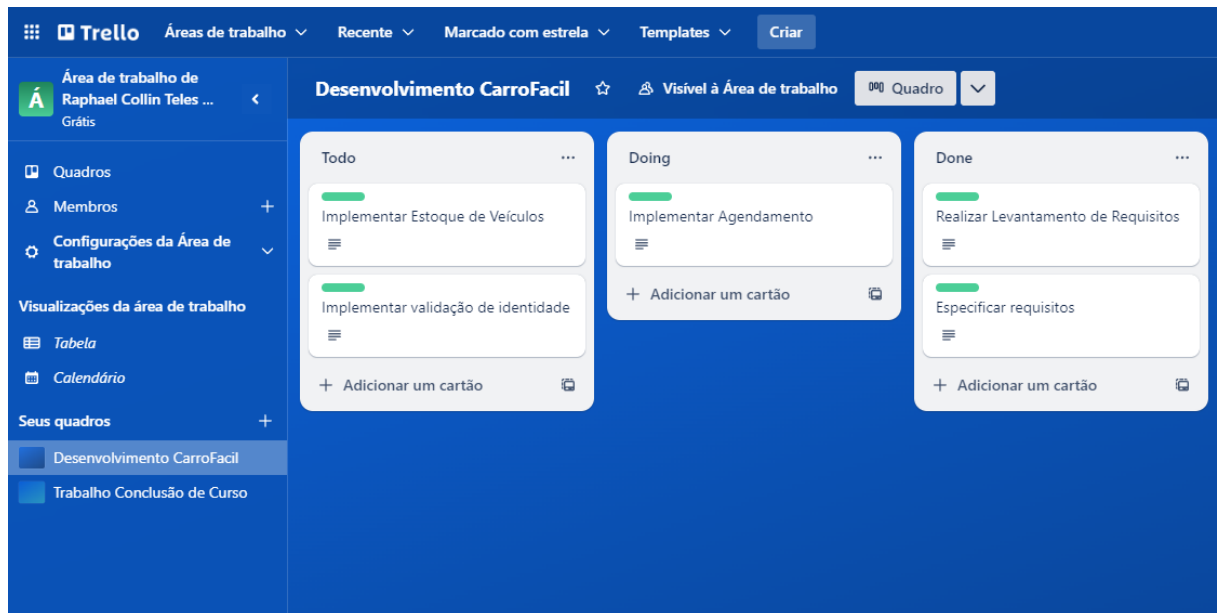
A seguir são apresentados os requisitos do sistema de locação de veículos.



Figura 5 – Diagrama de caso de uso



Fonte: o autor

Figura 6 – Exemplo de quadro *Kanban*

Fonte: o autor

#### 4.3.1 Requisitos Funcionais

Quadro 8 – Registrar cliente

<b>Caso de uso</b>	Registrar cliente
<b>Descrição</b>	Permite que o cliente se cadastre na plataforma.
<b>Ator</b>	Cliente
<b>Pre-condições</b>	O cliente ainda não está cadastrado.
<b>Pós-condições</b>	O cliente é registrado na plataforma.
<b>Cenário Principal</b>	<ol style="list-style-type: none"> <li>1. O cliente informa o nome, e-mail e senha.</li> <li>2. O cliente recebe uma confirmação do cadastro.</li> </ol>
<b>Fluxo de Exceção</b>	<ol style="list-style-type: none"> <li>1. Dados inválidos são informados. Nesse caso, o sistema exibe uma mensagem de erro.</li> <li>2. Usuário já cadastrado. O sistema também exibe uma mensagem de erro customizada para esse caso.</li> </ol>

Fonte: o autor.

O caso de uso do [Quadro 8](#) especifica o registro de um cliente. Esse é o primeiro passo para que um cliente possa realizar uma reserva de veículo.

Quadro 9 – Listar veículos

<b>Caso de uso</b>	Listar veículos
<b>Descrição</b>	Permite que o cliente visualize os veículos disponíveis para locação.
<b>Ator</b>	Cliente
<b>Pre-condições</b>	O cliente está autenticado na plataforma.
<b>Pós-condições</b>	Uma listagem de veículos disponíveis é exibida.
<b>Cenário Principal</b>	O cliente acessa a página de veículos, onde é possível visualizar os veículos disponíveis.

Fonte: o autor.

O [Quadro 9](#) especifica o processo de listagem de veículos. Esse processo é realizado pelo cliente e permite que ele visualize os veículos disponíveis para aluguel.

Quadro 10 – Reservar um veículo

<b>Caso de uso</b>	Reservar um veículo
<b>Descrição</b>	Permite que um cliente reserve um veículo por um período de tempo.
<b>Ator</b>	Cliente
<b>Pre-condições</b>	O cliente está autenticado na plataforma.
<b>Pós-condições</b>	Veículo reservado fica indisponível para outros clientes.
<b>Cenário Principal</b>	<ol style="list-style-type: none"><li>1. O usuário seleciona o veículo, a data de início e a data de término da locação.</li><li>2. É exibida uma confirmação para o usuário.</li></ol>
<b>Fluxo de Exceção</b>	<ol style="list-style-type: none"><li>1. O veículo não está mais disponível.</li><li>2. O cliente seleciona um período de tempo indisponível.</li><li>3. Em ambos os casos, o cliente é alertado sobre o problema.</li></ol>

Fonte: o autor.

O [Quadro 10](#) especifica o processo de reserva de um veículo. Esse processo é realizado pelo cliente e permite que ele alugue um veículo por um período de tempo.

Quadro 11 – Realizar *check-in*

<b>Caso de uso</b>	Realizar <i>check-in</i>
<b>Descrição</b>	Permite a realização do check-in da reserva entre o cliente e o funcionário.
<b>Ator</b>	Funcionário
<b>Pre-condições</b>	<ol style="list-style-type: none"><li>1. O cliente está autenticado na plataforma.</li><li>2. O cliente possui uma reserva criada e seu identificador.</li></ol>
<b>Pós-condições</b>	O status da reserva é alterado para em progresso.
<b>Cenário Principal</b>	O cliente apresenta o identificador da reserva ao funcionário que o insere no sistema para realizar o check-in.
<b>Fluxo de Exceção</b>	<ol style="list-style-type: none"><li>1. O check-in para aquela reserva já foi realizado.</li><li>2. O usuário é alertado sobre o problema.</li></ol>

Fonte: o autor.

O caso de uso Realizar *check-in* é apresentado no [Quadro 11](#). Esse caso de uso é realizado pelo funcionário e permite que o cliente retire o veículo reservado.

Quadro 12 – Realizar *check-out*

<b>Caso de uso</b>	Realizar <i>check-out</i>
<b>Descrição</b>	Permite a realização do check-out da reserva entre o cliente e o funcionário.
<b>Ator</b>	Funcionário
<b>Pre-condições</b>	<ol style="list-style-type: none"> <li>1. O cliente possui uma reserva em progresso e seu identificador.</li> <li>2. O cliente está autenticado na plataforma.</li> </ol>
<b>Pós-condições</b>	O status da reserva é alterado para concluída.
<b>Cenário Principal</b>	O cliente retorna o veículo, o funcionário verifica o estado do veículo, e obtém informações sobre a reserva a partir da placa. Por fim, o funcionário finaliza a reserva.
<b>Fluxo de Exceção</b>	<ol style="list-style-type: none"> <li>1. O <i>check-in</i> não foi realizado para reserva.</li> <li>2. O usuário é alertado sobre o problema.</li> </ol>

Fonte: o autor.

O [Quadro 12](#) especifica o processo de realização do *check-out*. Esse processo é realizado pelo funcionário e permite que o cliente devolva o veículo alugado.

Quadro 13 – Listar reservas

<b>Caso de uso</b>	Listar reservas
<b>Descrição</b>	Permite que um cliente visualize as reservas que realizou.
<b>Ator</b>	Cliente
<b>Pre-condições</b>	Cliente está autenticado na plataforma.
<b>Pós-condições</b>	Uma listagem de reservas é exibida.
<b>Cenário Principal</b>	O cliente acessa a página de reservas, onde é possível visualizar as reservas realizadas.

Fonte: o autor.

O [Quadro 13](#) especifica o processo de listagem de reservas. Esse processo é realizado pelo cliente e permite que ele visualize as reservas que realizou.

Quadro 14 – Registrar veículo

<b>Caso de uso</b>	Registrar veículo
<b>Descrição</b>	Permite que um funcionário registre um novo veículo na plataforma.
<b>Ator</b>	Funcionário
<b>Pre-condições</b>	O funcionário está autenticado.
<b>Pós-condições</b>	O veículo é registrado na plataforma.
<b>Cenário Principal</b>	<ol style="list-style-type: none"><li>1. O funcionário informa o tipo, marca, modelo, quilometragem, ano, placa, chassi e a cor do veículo.</li><li>2. O funcionário envia a requisição.</li></ol>
<b>Fluxo de Exceção</b>	<ol style="list-style-type: none"><li>1. Dados inválidos são informados. Nesse caso, o sistema exibe uma mensagem de erro.</li></ol>

Fonte: o autor.

O [Quadro 14](#) especifica o processo de registro de um veículo. Esse processo é realizado pelo funcionário e permite que ele registre um novo veículo na plataforma.

#### 4.3.2 Requisitos não Funcionais

Com base na quantidade de clientes atuais e a estimativa de crescimento, bem como suas expectativas quanto ao desempenho do sistema, são definidos os seguintes requisitos não funcionais:

1. Os serviços devem garantir a segurança dos dados dos clientes.
2. O tempo de resposta dos serviços deve ser menor que 2 segundos em 90% das requisições.
3. O sistema como um todo deve suportar 100 usuários simultâneos.
4. As funcionalidades dos serviços devem ser fornecidas através de [APIs](#) acessíveis com o protocolo HTTP.

### 4.4 Design

Para construção de cada microsserviço, é feito uso da [Arquitetura Hexagonal](#). Essa, por sua vez, separa a aplicação em duas partes principais: núcleo da aplicação

e adaptadores. O núcleo da aplicação contém as regras de negócio e os adaptadores são responsáveis por adaptar a aplicação para o mundo externo. Para modelagem do domínio, é utilizado o *Domain-Driven Design (DDD)*. O DDD fornece uma estratégia para modelagem do domínio de negócio, diversos padrões para resolver problemas de modelagem recorrentes e facilidade de entendimento e manutenção de código (EVANS, 2004).

Para realização do *design* deste projeto, é utilizado a *UML*. Especificamente, são criados diagramas de classes com propósito de modelar os tipos de objeto, seus relacionamentos e os serviços que fornecem. Adicionalmente, para detalhamento do fluxo de execução de operações chave, alguns diagramas de sequência são empregados. Além disso, visando fornecer uma visão geral da arquitetura do sistema, o diagrama de contexto de sistema do modelo C4 é utilizado. Esse modelo engloba um conjunto de abstrações hierárquicas e um conjunto de diagramas que fornecem diferentes perspectivas do sistema e que são extremamente úteis para representar a arquitetura de um sistema (BROWN, 2023).

## 4.5 Implementação

A implementação desse projeto é feita utilizando a linguagem de programação *Java* na versão 17 (ORACLE, 2024). Além disso, o *framework Spring Boot* é empregado na construção dos microsserviços. Essa ferramenta fornece uma série de recursos para construção de microsserviços como: injeção de dependências, configuração de banco de dados, configuração de *logs*, entre outros (PIVOTAL SOFTWARE, 2023).

São utilizados dois banco de dados para armazenamento dos dados do sistema. O primeiro é um banco de dados relacional, que armazena dados de reservas e veículos. O segundo é um banco de dados não relacional, que armazena dados de usuários. Para o banco de dados relacional, é utilizado o *PostgreSQL* (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2023). Para o banco de dados não relacional, foi escolhido o *Amazon DynamoDB* (AMAZON WEB SERVICES, 2023e).

O *PostgreSQL* é um sistema de gerenciamento de banco de dados relacional de código aberto. Esse banco de dados é um dos mais populares do mundo, sendo utilizado por diversas empresas, como: *Apple*, *Spotify*, *Netflix*, entre outras (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2023). Trata-se de um banco de dados escalável, confiável, fácil de usar e com suporte a transações ACID.

Por outro lado, o *Amazon DynamoDB* é um banco de dados não relacional, que fornece desempenho de milissegundos de um dígito a qualquer escala. Esse banco de dados é totalmente gerenciado, ou seja, não é necessário realizar a configuração de servidores,

escalabilidade, replicação, entre outros. Além disso, o *Amazon DynamoDB* é um banco de dados sem servidor, ou seja, o usuário paga apenas pelo que utiliza ([AMAZON WEB SERVICES, 2023e](#)).

Para a intercomunicação entre os microsserviços de maneira assíncrona, são utilizados o *Amazon Simple Notification System (SNS)* e o *Amazon Simple Queue System (SQS)*. O SNS é um serviço de mensagens que permite a publicação e a entrega de mensagens para assinantes ou pontos de extremidade ([AMAZON WEB SERVICES, 2023b](#)). Por outro lado, o SQS é um *broker* permite que os microsserviços se comuniquem de maneira assíncrona, sem que um microsserviço precise conhecer o outro ([AMAZON WEB SERVICES, 2023c](#)). Além disso, o *Amazon SQS* permite que as mensagens sejam armazenadas em uma fila, caso o microsserviço destinatário da mensagem esteja fora do ar. Utilizados em conjunto, essas ferramentas permitem a comunicação assíncrona na forma "um para muitos" entre os microsserviços. Um microsserviço publica uma mensagem em um tópico do SNS e os microsserviços interessados utilizam uma fila do SQS para consumir essa mensagem.

## 4.6 Validação

O projeto é desenvolvido com as estratégias [Test-driven development \(TDD\)](#) e [Behavior Driven Development \(BDD\)](#). Essas abordagens de desenvolvimento de *software* permitem que o código seja desenvolvido de maneira mais confiável e com maior qualidade ([BARAUNA; HARDARDT, 2020](#)). Além disso, o [TDD](#) e o [BDD](#) permitem que o código seja desenvolvido de maneira mais rápida, pois os testes são escritos antes do código.

Três tipos de testes são escritos para validar os requisitos funcionais do sistema: testes unitários, testes de integração e testes de sistema. Os testes unitários são escritos para validar métodos e classes do sistema. Por outro lado, os testes de integração validam a integração entre dois microsserviços e integrações com componentes externos, como banco de dados. Por fim, os testes de sistema são implementados para validar os requisitos do sistema. É importante ressaltar que todos os testes são automatizados utilizando o *JUnit* ([JUNIT, 2024](#)). O framework *Mockito* é utilizado para simular o comportamento de objetos reais ([MOCKITO, 2024](#)). Além disso, o *Testcontainers* é utilizado para criar contêineres *Docker* para os testes de sistema ([TESTCONTAINERS, 2024](#)).

Por outro lado, para validação dos requisitos não funcionais, testes de carga são empregados. O ambiente de execução desses testes é o mesmo de implantação, definido na seção 4.7. Através da ferramenta Gatling ([GATLING, 2024](#)), a bateria de testes consiste de 100 usuários executando operações simultâneas por um período de 30 minutos. São avaliados o tempo de resposta, a utilização da CPU, o consumo de memória e a quantidade de erros. Os resultados obtidos são comparados com o [Quadro 15](#). Cada operação possui as



seguintes etapas: registrar um novo veículo, cadastrar um cliente, realizar uma reserva, realizar o *check-in* e realizar o *check-out*.

Após a execução dos testes de carga, os resultados obtidos são avaliados e comparados com os requisitos não funcionais. Dessa forma, um relatório é produzido com uma análise gráfica e numérica.

Quadro 15 – Métricas de comparação

Métrica	Alvo
Tempo de resposta	$\leq 2$ s em 90% das requisições.
Utilização da CPU	$\leq 70\%$ durante toda a execução.
Consumo de memória	$\leq 70\%$ durante toda a execução.
Quantidade de erros	$\leq 1\%$ das requisições.

Fonte: o autor.

Os parâmetros descritos no [Quadro 15](#) foram definidos pelo autor com base em experiências anteriores de desenvolvimento de sistemas similares no ambiente corporativo, juntamente com a recomendação do orientador.

## 4.7 Implantação

O sistema é implantado na nuvem da *Amazon Web Services (AWS)*. A *AWS* é uma plataforma de computação em nuvem que oferece mais de 200 serviços completos de *data centers* em todo o mundo ([AMAZON WEB SERVICES, 2023d](#)). Esses serviços incluem computação, armazenamento, banco de dados, *networking*, *analytics*, *machine learning*, inteligência artificial, *Internet of Things*, segurança, entre outros.

Cada microsserviço é implantado em um *container Docker*. Essa tecnologia permite que os microsserviços sejam executados de maneira isolada, sem que um microsserviço interfira no outro. Além disso, o *Docker* permite que os microsserviços sejam executados em qualquer ambiente, sem que seja necessário realizar alterações no código ([DOCKER, 2023](#)). O serviço *Amazon ECS* é utilizado para orquestrar os *containers Docker*. Esse serviço permite que os *containers* sejam executados em um ambiente de produção de maneira escalável e confiável ([AMAZON WEB SERVICES, 2023a](#)). Ao todo, serão utilizadas 5 *tasks* (uma para cada microsserviço). Cada *task* possui 2 CPUs e 4 GB de memória RAM.

Dois *clusters* do *PostgreSQL* com o auxílio do serviço *Amazon RDS* são utilizados ([POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2023](#)). Um para o serviço de

estoque e outro para o serviço de reservas. Com a arquitetura de microsserviços, é essencial o isolamento das bases de dados. Além disso, três tabelas do *Amazon DynamoDB* são utilizadas para armazenamento de informações de usuários, clientes e funcionários. Da mesma forma, um tópico do SNS para reservas e um fila SQS que consome informações de reservas e mantém um contador de reservas por cliente são utilizados.

Para a orquestração e automatização do provisionamento e configuração da infraestrutura, é feito uso da ferramenta *Terraform*. Essa ferramenta permite que a infraestrutura seja definida como código, ou seja, é possível definir a infraestrutura utilizando uma linguagem de programação ([HASHICORP, 2023](#)). Além disso, o *Terraform* permite que a infraestrutura seja provisionada e configurada de maneira automatizada.

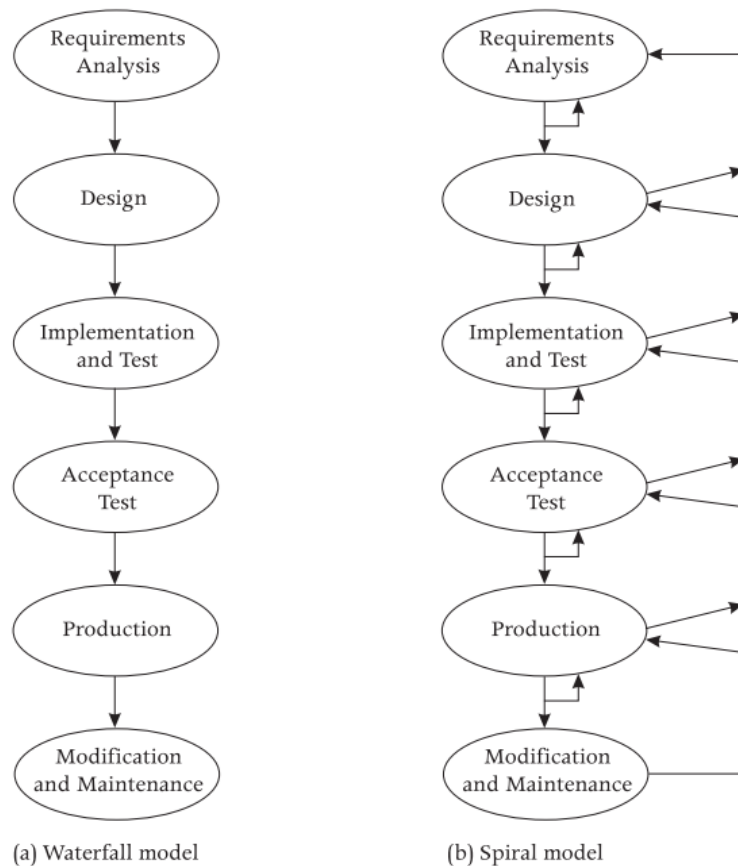
## 5 Estudo de Caso: Design, Implementação e Testes

Este capítulo apresenta o desenvolvimento do estudo de caso com Microserviços e DDD. Ele descreve as etapas para construção do sistema a partir de requisitos funcionais e não-funcionais.

### 5.1 Ciclo de Vida do Desenvolvimento de Software

Liskov e Guttag (2000) descreve 6 etapas no ciclo de vida de um software: Análise de requisitos, *Design*, Implementação e Teste, Teste de aceitação, Produção e Manutenção. A Figura 7 ilustra essas etapas.

Figura 7 – Ciclo de Vida do Desenvolvimento de Software



Fonte: Liskov e Guttag (2000)

Para este trabalho, os requisitos foram definidos no Capítulo 4. Da mesma forma,

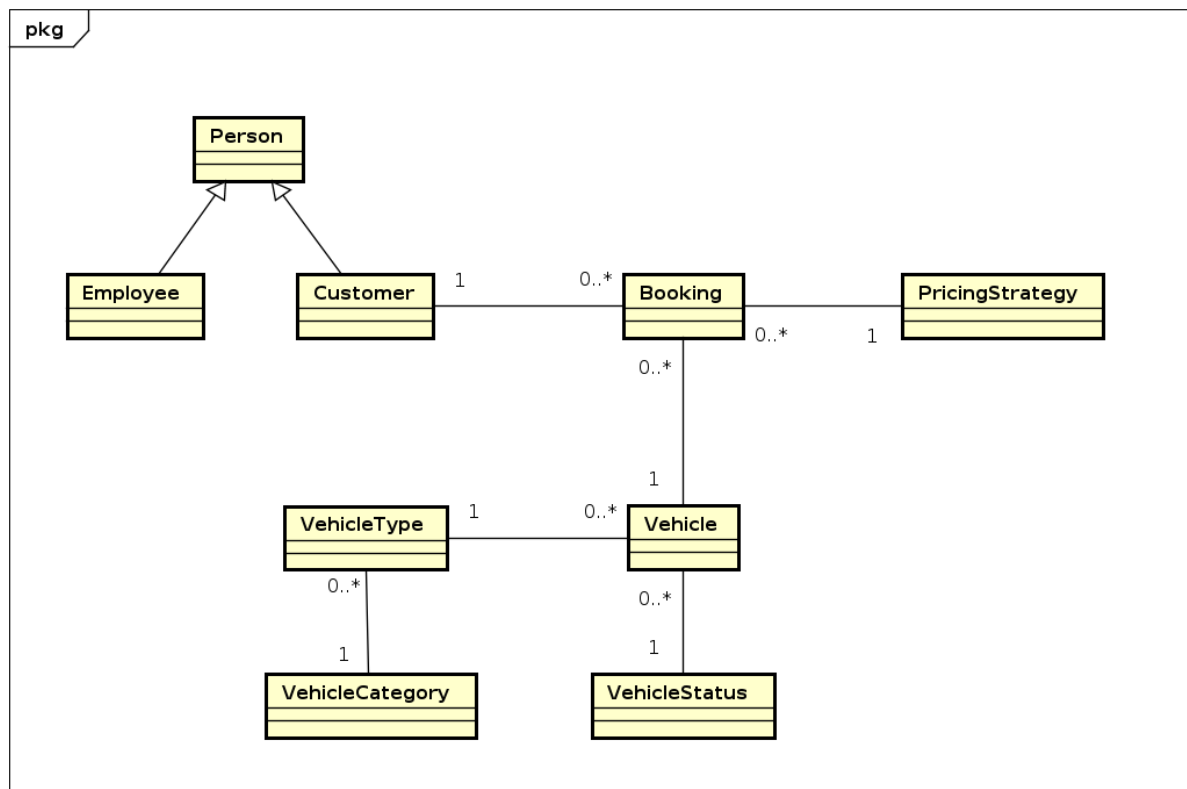
a fase de manutenção não será abordada, pois o foco é a construção do sistema.

## 5.2 Design

Esta seção utiliza como entrada os requisitos funcionais e não-funcionais definidos no [Capítulo 4](#) para descrever o design do sistema utilizando [Arquitetura de Microserviços \(AMS\)](#) e [Domain-Driven Design \(DDD\)](#).

Inicialmente, é importante definir os conceitos chave do sistema, bem como seus relacionamentos. A [Figura 8](#) ilustra as abstrações do sistema.

Figura 8 – Modelo conceitual do sistema



Fonte: o autor

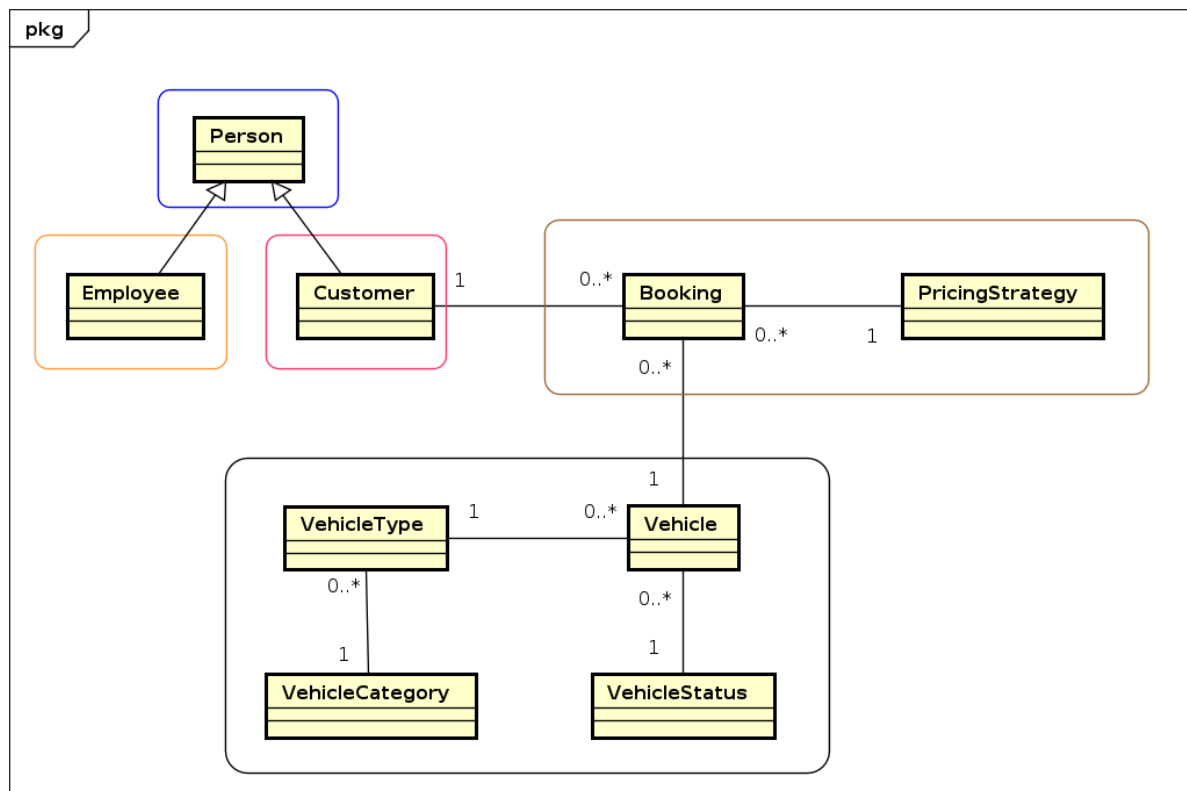
O sistema tem como componente principal a entidade *Booking* que está relacionada com um *Vehicle* e um *Customer*. Cada *Vehicle* possui um *VehicleType* que tem como função agrupar veículos com características similares. O *VehicleType*, por sua vez, está associado a um *VehicleCategory* como *Hatch* ou *SUV*. O veículo também tem um *VehicleStatus* que define o estado atual de um veículo específico. Além disso, cada reserva contém uma *PricingStrategy* que é responsável por calcular o preço da reserva baseado em diferentes critérios. Um *Customer* é um subtipo de *Person*. Da mesma forma, o *Employee* é um subtipo de *Person*.

No contexto deste caso de uso, as abstrações *Booking*, *Vehicle*, *Customer*, *Person* e *Employee* possuem identidade e ciclo de vida próprios. Portanto, são entidades. Por outro lado, *VehicleType*, *VehicleCategory*, *VehicleStatus* e *PricingStrategy* são **Value Object (VO)**, pois não possuem identidade própria e são imutáveis.

### 5.2.1 Bounded Contexts

Após ter identificado as entidades e **VO**, é necessário definir os **BC** do sistema. **Evans** (2004) define **BC** como um limite lógico dentro do qual um modelo é consistente. Além disso, cada **BC** possui um ou mais *Aggregates*. Nesse sentido, a **Figura 9** ilustra os **BC** do sistema.

Figura 9 – Bounded Contexts do sistema



Fonte: o autor

Ao todo, o sistema possui 5 **Bounded Context (BC)**:

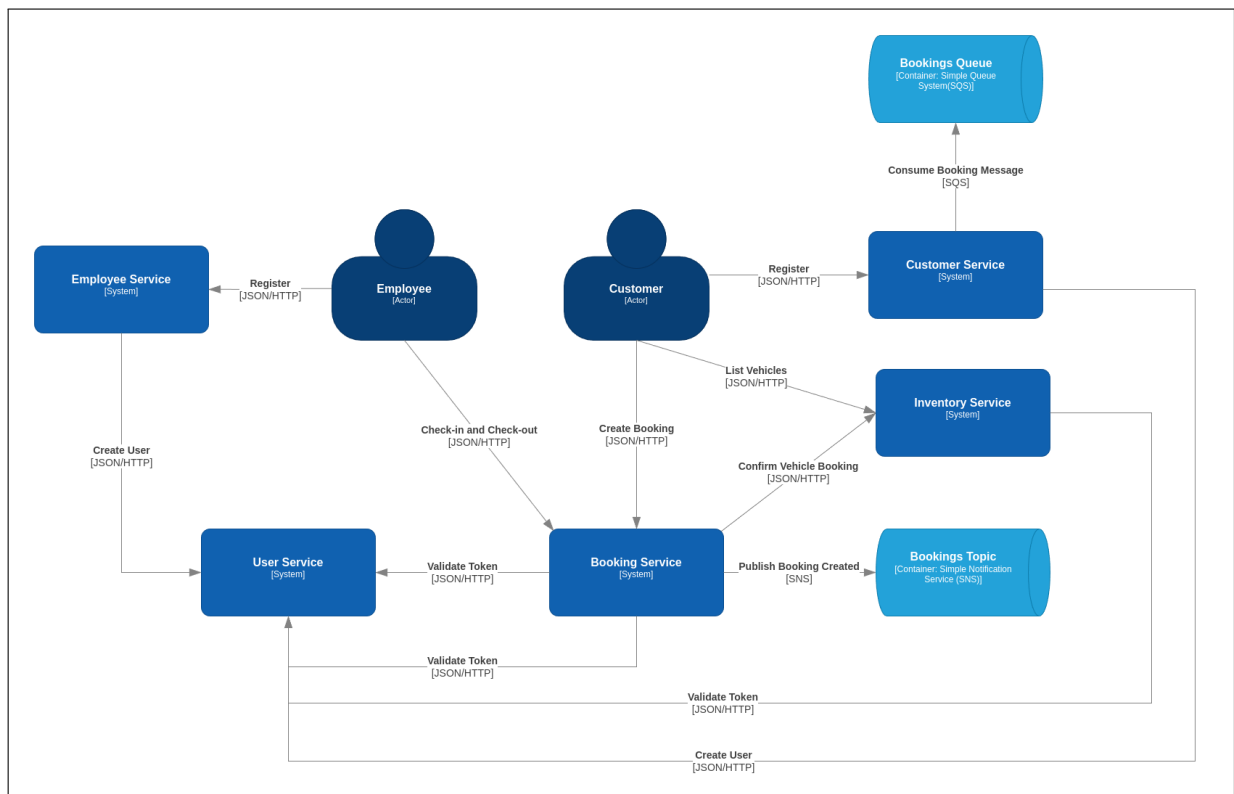
- *Person*: Responsável por gerenciar informações de pessoas e autenticação.
- *Customer*: Realiza o gerenciamento informações de clientes.
- *Employee*: Responsável por gerenciar informações de funcionários.
- *Vehicle*: Está relacionado a estoque de veículos e tipos de veículo.

- *Booking*: Responsável por gerenciar informações de reservas e estratégias de precificação.

### 5.2.2 Microserviços

Com os *BC* definidos, é possível definir os limites de cada microserviço. Seguindo as melhores práticas revisadas no [Capítulo 3](#), cada *BC* é mapeado em um microserviço. A [Figura 10](#) ilustra a divisão dos microserviços.

Figura 10 – Microserviços do sistema



Fonte: o autor

Cada microserviço é responsável por um *BC* e possui sua própria base de dados. O *User Service* realiza o cadastro e login de todos usuários do sistema, além de validar *tokens* de autenticação. O *Customer Service* lida com informações de clientes. Esse serviço também consome uma fila de mensagens de reservas e atualiza um contador de reservas para cada usuário. O *Employee Service* gerencia informações de funcionários. O *Inventory Service* trata informações de veículos como quantidade em estoque, marca, modelo, cor e tipo. Por fim, o *Booking Service* gerencia informações de reservas e estratégias de precificação.

### 5.2.3 Interações entre Microserviços

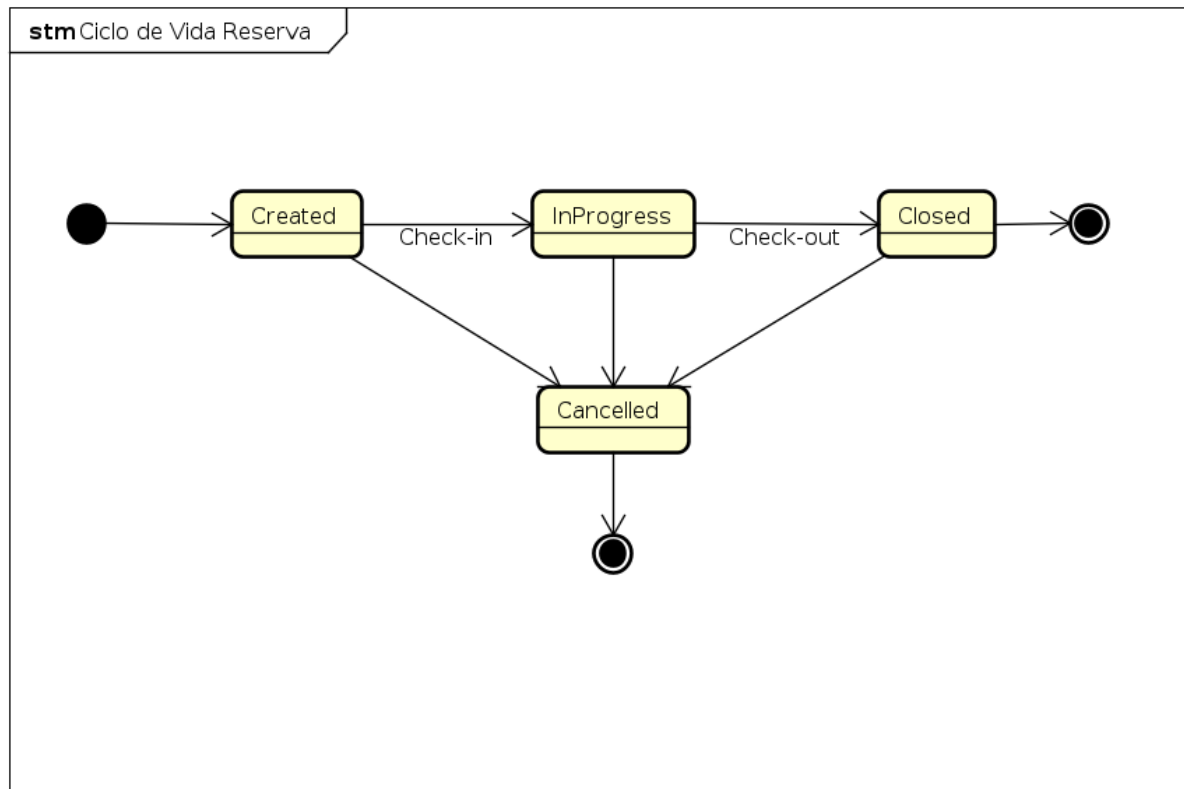
Como mencionado no [Capítulo 2](#), há duas formas principais para estabelecer a comunicação entre microserviços: síncrona e assíncrona. Para este caso de uso, é utilizada a comunicação assíncrona com a utilização de tópicos e filas de mensagens. A comunicação assíncrona é preferível, pois permite que os microserviços sejam escalados independentemente, não bloqueia o fluxo de execução e possui maior resiliência. O *Booking Service* publica uma mensagem em um tópico de reservas toda vez que uma nova reserva é criada. O *Customer Service* consome essa mensagem e atualiza o contador de reservas do usuário.

No entanto, há muitas situações em que a comunicação síncrona é necessária, pois o cliente precisa de uma resposta mediata. Uma opção muito comum é utilizar *API Restful* com o protocolo HTTP. Por exemplo, no momento de criação de uma nova reserva o *Booking Service* precisa verificar a disponibilidade de veículos no *Inventory Service*. Nesse caso, o *Booking Service* envia uma requisição HTTP para o *Inventory Service* e aguarda a resposta.

### 5.2.4 Principais casos de uso

A seguir são detalhadas as interações entre os microserviços para os principais casos de uso do sistema. Especificamente, são descritos os casos de uso que compõem o ciclo de vida de uma reserva ilustrado no [Figura 11](#).

Figura 11 – Ciclo de vida de uma reserva



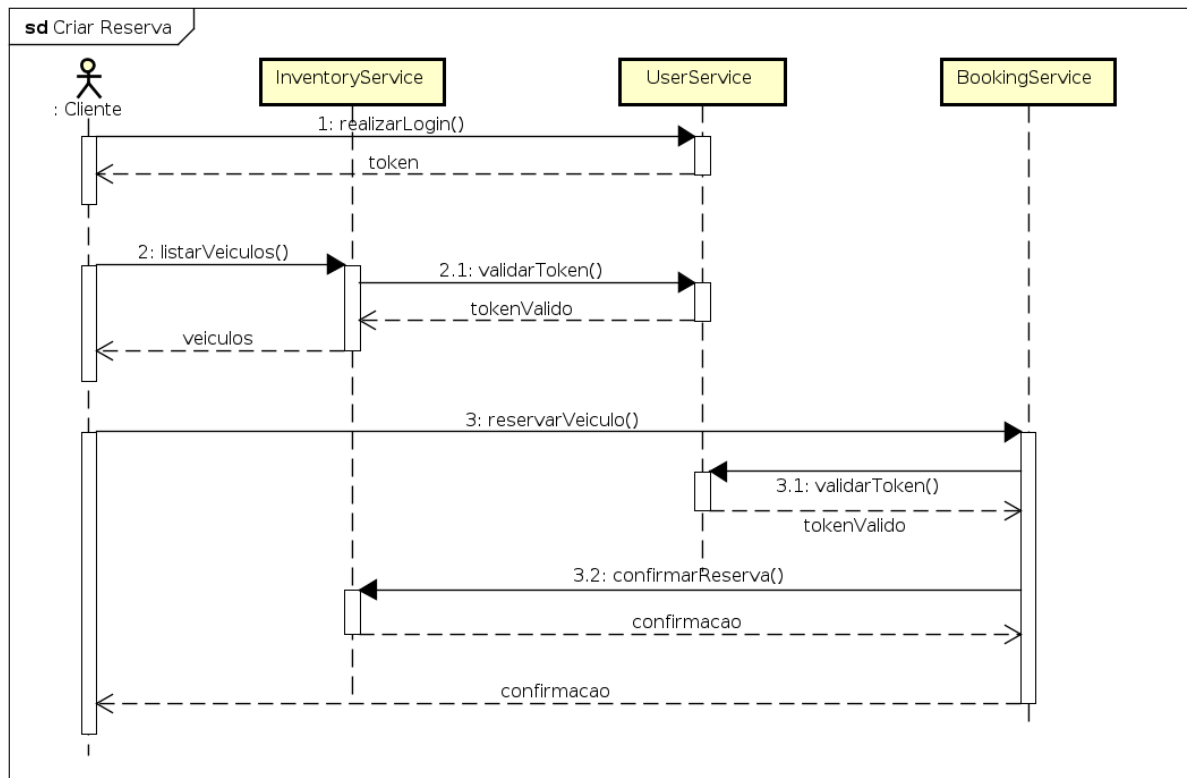
Fonte: o autor

#### 5.2.4.1 Criar reserva

A [Figura 12](#) ilustra o fluxo de criação de uma reserva. Inicialmente, o cliente realiza o login no *User Service* e recebe um token de autenticação como resposta. Em seguida, ele acessa o *Inventory Service* para verificar a disponibilidade de veículos. Esse verifica se o usuário está autenticado realizando uma chamada para o *User Service*. Caso o usuário esteja autenticado, o *Inventory Service* retorna a lista de veículos disponíveis. Posteriormente, O cliente seleciona um veículo e realiza a reserva no *Booking Service*. Esse serviço envia uma requisição para o *Inventory Service* para decrementar a quantidade de veículos disponíveis e confirmar a reserva. Por fim, o *Booking Service* retorna a confirmação da reserva.



Figura 12 – Criar reserva

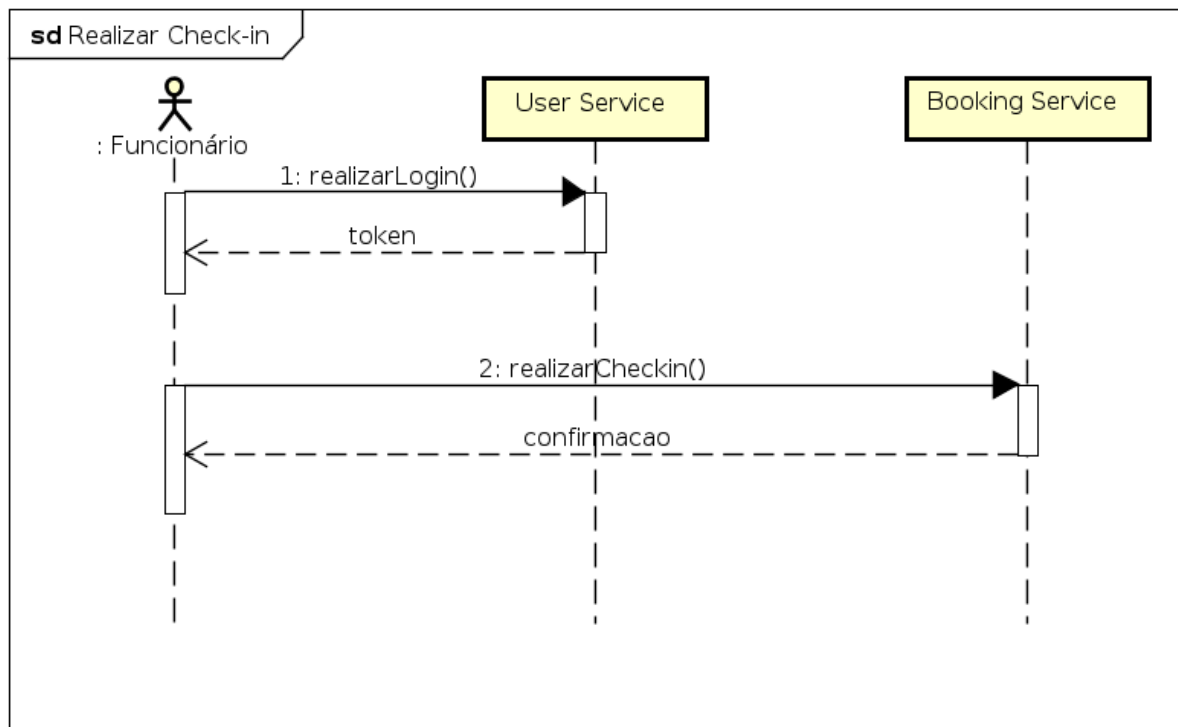


Fonte: o autor

#### 5.2.4.2 Realizar check-in

A [Figura 13](#) ilustra o fluxo de realizar um check-in. Primeiramente, o funcionário realiza o login no *User Service* e recebe um token de autenticação como resposta. Em seguida, ele acessa o *Booking Service* para realizar o check-in. Esse serviço verifica se o usuário está autenticado realizando uma chamada para o *User Service*. Caso o usuário esteja autenticado, o *Booking Service* realiza o check-in e retorna a confirmação.

Figura 13 – Realizar check-in

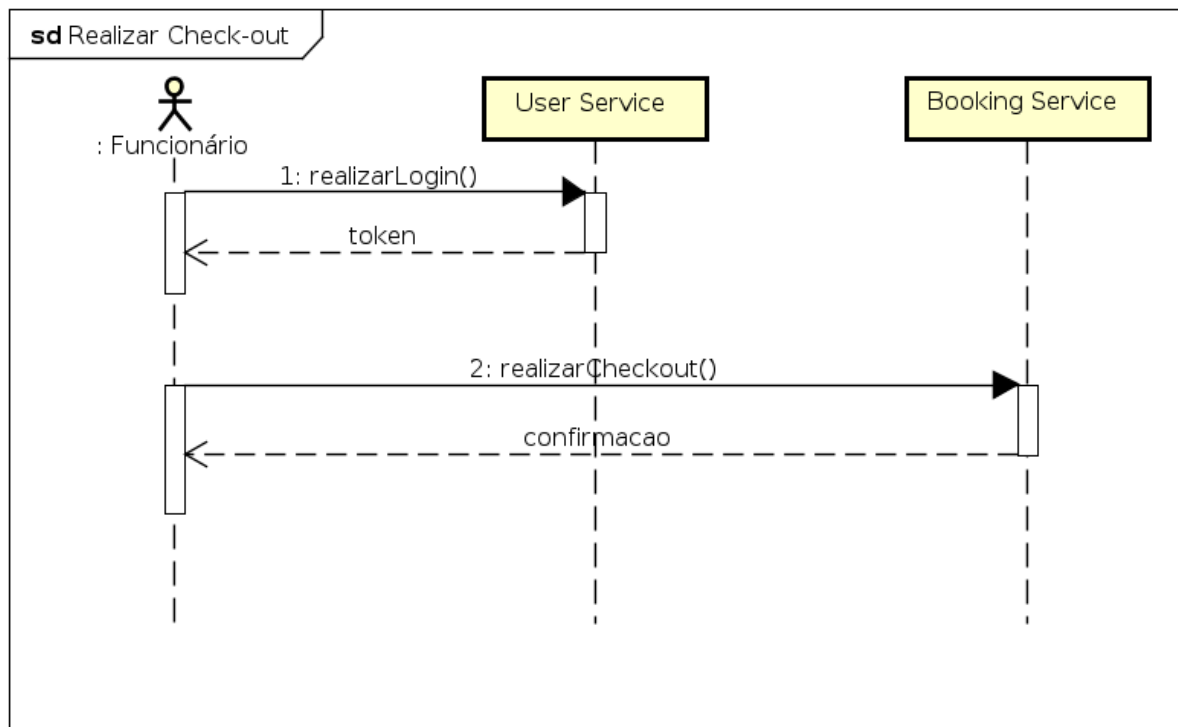


Fonte: o autor

#### 5.2.4.3 Realizar check-out

A [Figura 14](#) ilustra o fluxo de realizar um check-out. Inicialmente, o funcionário realiza a autenticação no *User Service* e recebe um token de autenticação como resposta. Posteriormente, ele acessa o *Booking Service* para realizar o check-out. Esse serviço verifica se o usuário está autenticado realizando uma chamada para o *User Service*. Caso o usuário esteja autenticado, o *Booking Service* realiza o check-out e retorna a confirmação.

Figura 14 – Realizar check-out

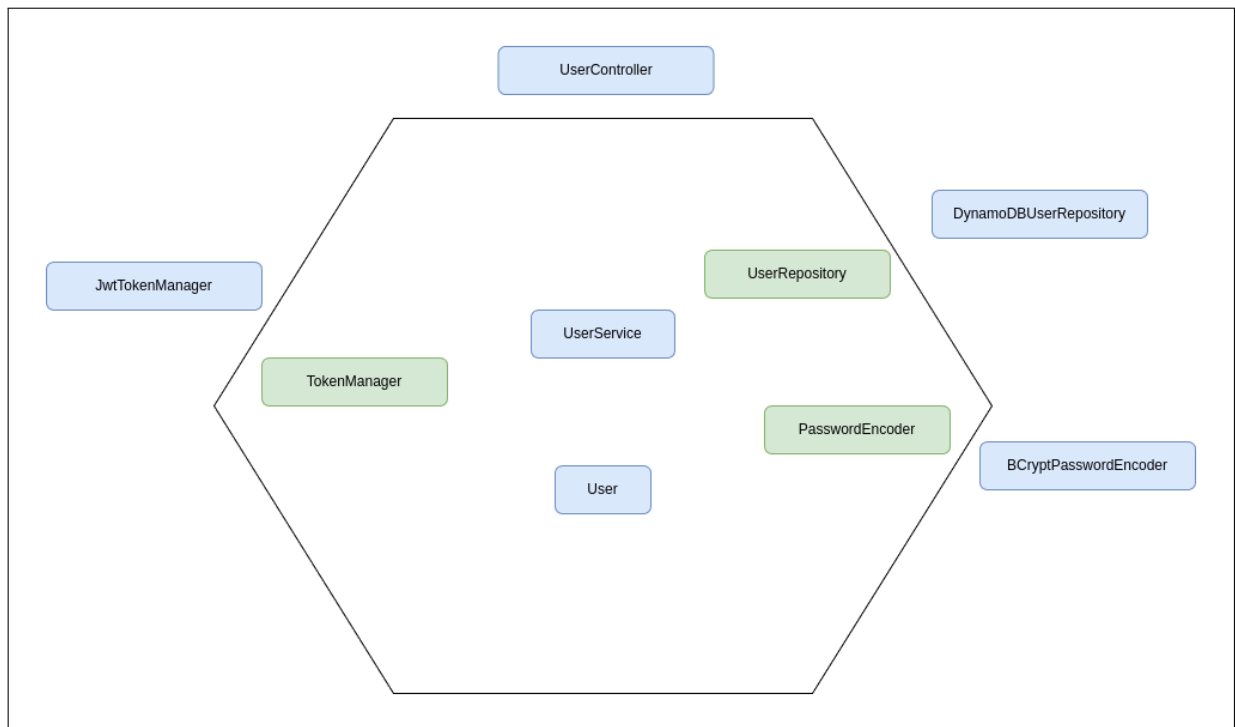


Fonte: o autor

### 5.2.5 User Service

A [Figura 15](#) apresenta os principais componentes do *User Service* na Arquitetura Hexagonal.

Figura 15 – User Service



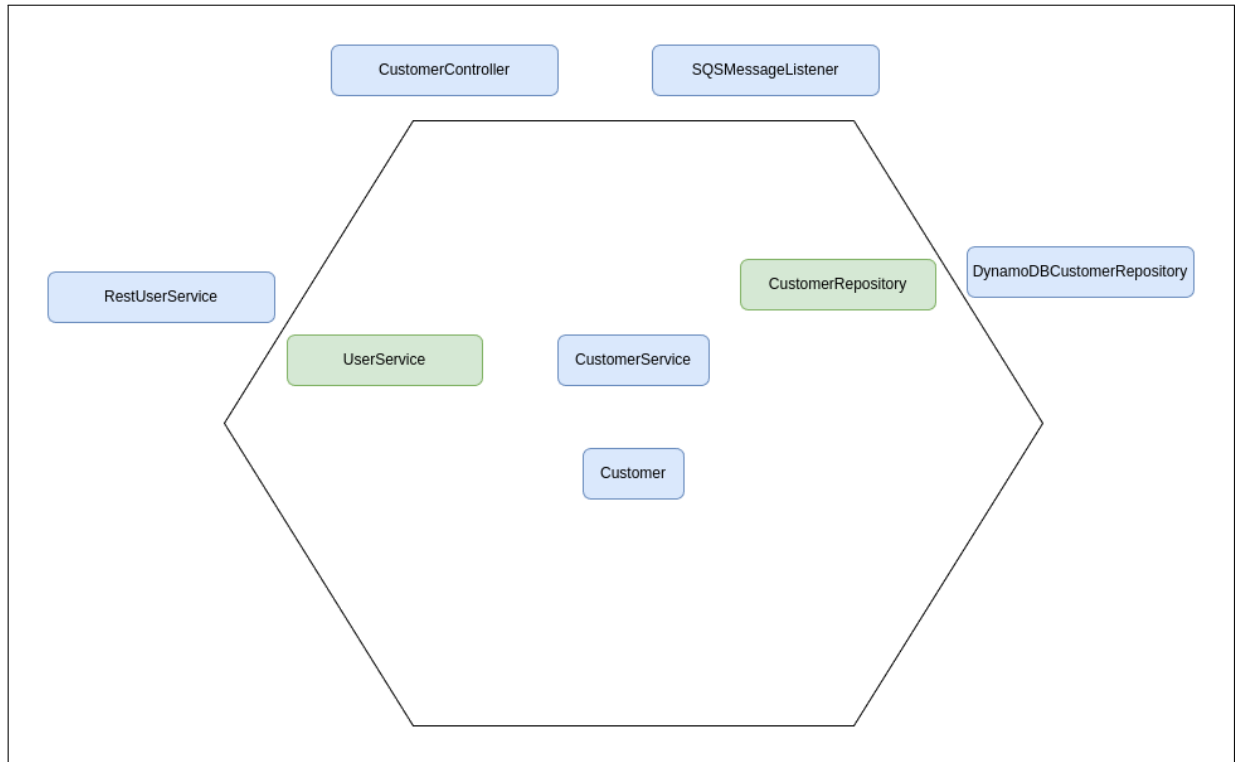
Fonte: o autor

Os componentes do *Application core* são: *User* e *UserService*. A aplicação possui as seguintes portas: *UserRepository*, *PasswordEncoder* e *TokenManager*. Por fim, os adaptadores *UserController*, *DynamoDBUserRepository*, *BCryptPasswordEncoder* e *JwtTokenManager* são responsáveis por integrar a aplicação com o mundo externo.

### 5.2.6 Customer Service

A [Figura 16](#) apresenta os principais componentes do *Customer Service* na Arquitetura Hexagonal.

Figura 16 – Customer Service



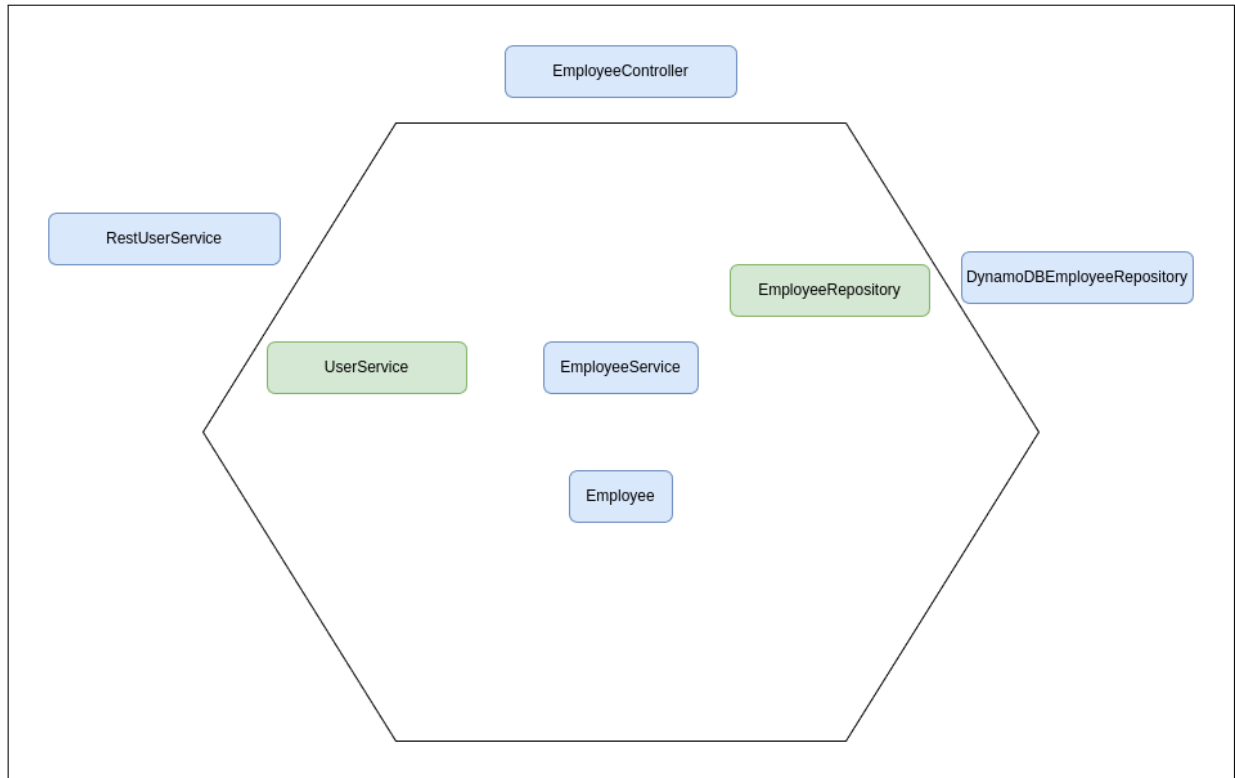
Fonte: o autor

Nesse serviço, o *Application core* é composto por *Customer* e *CustomerService*. As portas são: *CustomerRepository* e *UserService*. Por fim, os adaptadores são: *CustomerController*, *RestUserService*, *DynamoDBCustomerRepository* e *SQSMessageListener*.

### 5.2.7 Employee Service

A [Figura 17](#) apresenta os principais componentes do *Employee Service* na Arquitetura Hexagonal.

Figura 17 – Employee Service



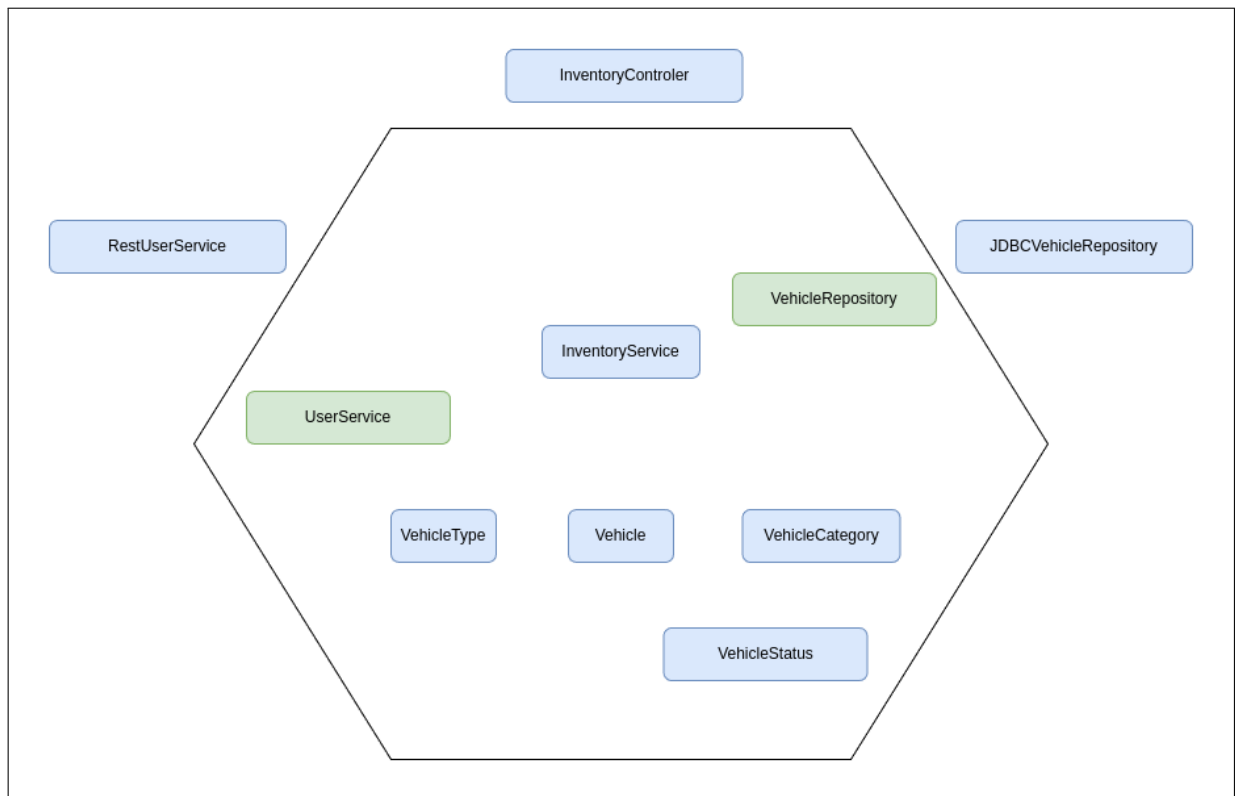
Fonte: o autor

O *Application core* é composto por *Employee* e *EmployeeService*. As portas são: *EmployeeRepository* e *UserService*. Por fim, os adaptadores são: *EmployeeController*, *DynamoDBEmployeeRepository* e *RestUserService*.

### 5.2.8 Inventory Service

Essa seção apresenta os principais componentes do *Inventory Service* na Arquitetura Hexagonal, ilustrados na [Figura 18](#).

Figura 18 – Inventory Service



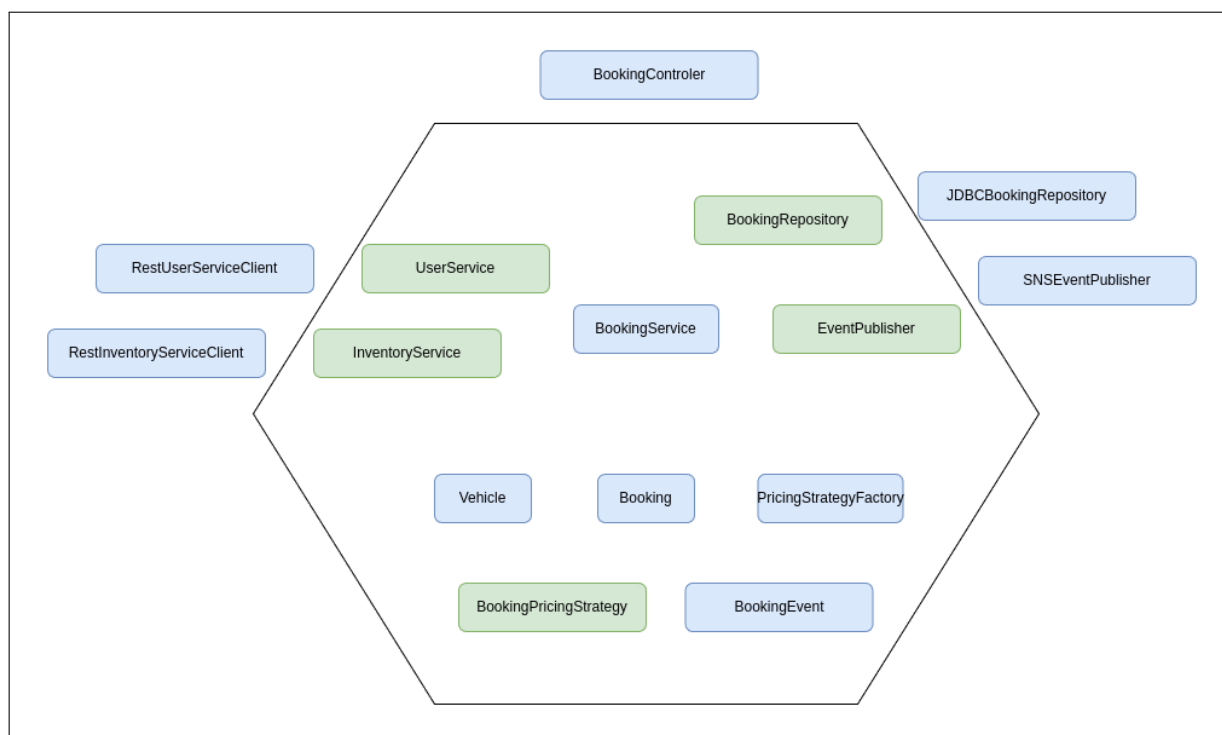
Fonte: o autor

Nessa aplicação, o *Application core* é composto por *Vehicle*, *VehicleType*, *VehicleCategory*, *VehicleStatus* e *InventoryService*. As portas são: *VehicleRepository* e *UserService*. Por fim, os adaptadores são: *InventoryController*, *JDBCInventoryRepository* e *RestUserService*.

### 5.2.9 Booking Service

A [Figura 19](#) apresenta os principais componentes do *Booking Service* na Arquitetura Hexagonal.

Figura 19 – Booking Service



Fonte: o autor

O *Application core* é composto por *Booking*, *BookingPricingStrategy*, *Vehicle*, *PricingStrategyFactory*, *BookingEvent* e *BookingService*. As portas são: *BookingRepository*, *EventPublisher*, *UserService* e *InventoryService*. Por fim, os adaptadores são: *BookingController*, *JDBCBookingRepository*, *SNSEventPublisher*, *RestUserServiceClient* e *RestInventoryServiceClient*.

## 5.3 Implementação

Esta seção apresenta os trechos de código chave para a implementação do caso de uso. O código completo pode ser encontrado no repositório do projeto <sup>1</sup>.

### 5.3.1 Login de usuário

O [Codigo 5.1](#) apresenta o método para realizar login de um usuário. Esse método recebe um *LoginRequest* contendo o email e senha do usuário. Em seguida, ele busca o usuário no repositório e compara a senha informada com a senha armazenada no banco de dados. Caso a senha seja válida, o método retorna um *UserResponse* contendo o token de autenticação. O procedimento lança exceções caso o email não seja encontrado ou a senha seja inválida.

<sup>1</sup> <<https://github.com/C0lliNN/CarroFacil>>



---

```
1 public class UserService {
2     private final UserRepository repository;
3     private final PasswordEncoder passwordEncoder;
4     private final TokenManager tokenManager;
5
6     public UserResponse login(LoginRequest request) {
7         User user = repository.findByEmail(request.email()).
8             orElseThrow(() -> new EmailNotFoundException("The email '%s'
9                 could not be found.", request.email()));
10
11         if (!passwordEncoder.comparePasswordAndHash(request.password(), user.
12             getPassword())) {
13             throw new IncorrectPasswordException("The provided password is
14                 incorrect.");
15         }
16
17         return createUserResponseWithToken(user);
18     }
19
20     // Other methods
21 }
```

---

Codigo 5.1 – Método para realizar login

### 5.3.2 Cadastro de Cliente

O [Codigo 5.2](#) apresenta o método para realizar o cadastro de um cliente. Esse método recebe um *RegisterRequest* contendo o nome, email e senha do cliente. Em seguida, ele realiza o cadastro do usuário no *User Service* e cria um cliente no *Customer Service*. Por fim, o método retorna um *CustomerResponse* contendo as informações do cliente.

---

```
1 public class CustomerService {
2     private final UserServiceClient userServiceClient;
3     private final CustomerRepository customerRepository;
4
5     public CustomerResponse register(RegisterRequest request) {
6         User user = userServiceClient.register(request.name(), request.email(),
7             request.password());
8
9         Customer customer = Customer.builder()
10             .name(request.name())
11             .userId(user.getId())
12             .build();
13
14         return CustomerResponse.fromCustomer(customerRepository.save(customer),
15             user);
16     }
17
18     // Other methods
19 }
```

---

Codigo 5.2 – Método para realizar cadastro de cliente

### 5.3.3 Realizar Reserva

O [Codigo 5.3](#) apresenta o método para realizar uma reserva. Esse método recebe um *BookingRequest* contendo as informações da reserva. Em seguida, ele busca o veículo no *Inventory Service* e realiza a reserva no *Booking Service*. Além disso, o método publica um evento que poderá ser consumido por outros serviços. Por fim, o método retorna um *BookingResponse* contendo as informações da reserva.

---

```
1 public class BookingService {
2     private BookingRepository bookingRepository;
3     private InventoryClient inventoryClient;
4     private BookingEventPublisher bookingEventPublisher;
5
6     public BookingResponse createBooking(BookingRequest bookingRequest) {
7         Booking booking = bookingRequest.createBooking();
8         Vehicle vehicle = inventoryClient.getVehicle(booking.getVehicleId());
9         booking.setVehicle(vehicle);
10
11         booking = bookingRepository.save(booking);
12
13         bookingEventPublisher.publishBookingEvent(
14             new BookingEvent(booking.getId(), booking.getUserId(), vehicle.
15                             getId())
16         );
17         return BookingResponse.from(booking);
18     }
19
20     // Other methods
21 }
```

---

Codigo 5.3 – Método para criar reserva

### 5.3.4 Realizar Check-in

O [Codigo 5.4](#) contém o código necessário para realizar um check-in. No *BookingService* é possível ver o método *checkin* que recebe o id da reserva e realiza o check-in. Esse método busca a reserva no repositório, realiza o check-in e salva a reserva. Toda a operação é executada em uma transação ACID com nível de isolamento *serializable* para prevenir *check-ins* duplicados. O método *checkIn* da classe *Booking* é responsável por validar o estado da reserva, atualizar o status e a data de *check-in*.

---

```
1 public class BookingService {
2     private BookingRepository bookingRepository;
3     private InventoryClient inventoryClient;
4     private BookingEventPublisher bookingEventPublisher;
5
6     @Transactional(isolation = Isolation.SERIALIZABLE)
7     public void checkin(int bookingId) {
8         Booking booking = bookingRepository.findById(bookingId)
9             .orElseThrow(() -> new EntityNotFoundException("Booking not
10                 found"));
11
12         booking.checkIn();
13         bookingRepository.save(booking);
14     }
15     // Other methods
16 }
17
18 public class Booking {
19     // other fields and methods
20
21     public void checkIn() {
22         if (status != Status.CREATED) {
23             throw new InvalidBookingStateException("Booking is not in CREATED
24                 state");
25         }
26
27         status = Status.IN_PROGRESS;
28         checkedInAt = LocalDateTime.now();
29     }
30 }
```

---

Codigo 5.4 – Métodos para realizar check-in

### 5.3.5 Realizar Check-out

O [Codigo 5.5](#) contém o código necessário para realizar um check-out. No *BookingService* é possível ver o método *checkout* que recebe o id da reserva e realiza o check-out. Esse método busca a reserva no repositório, realiza o check-out e salva a reserva. Toda a operação é executada em uma transação ACID com nível de isolamento de *serializable* para prevenir *check-outs* duplicados. O método *checkOut* da classe *Booking* é responsável por validar o estado da reserva, atualizar o status e a data de *check-out*.

---

```
1 public class BookingService {
2     private BookingRepository bookingRepository;
3     private InventoryClient inventoryClient;
4     private BookingEventPublisher bookingEventPublisher;
5
6     @Transactional(isolation = Isolation.SERIALIZABLE)
7     public void checkout(int bookingId) {
8         Booking booking = bookingRepository.findById(bookingId)
9             .orElseThrow(() -> new EntityNotFoundException("Booking not
10                 found"));
11
12         booking.checkOut();
13         bookingRepository.save(booking);
14     }
15     // Other methods
16 }
17
18 public class Booking {
19     // other fields and methods
20
21     public void checkOut() {
22         if (status != Status.IN_PROGRESS) {
23             throw new InvalidBookingStateException("Booking is not in
24                 IN_PROGRESS state");
25         }
26
27         status = Status.CLOSED;
28         checkedOutAt = LocalDateTime.now();
29     }
30 }
```

---

Codigo 5.5 – Métodos para realizar check-out

### 5.3.6 Comunicação síncrona entre Microserviços

O [Codigo 5.6](#) demonstra como a comunicação síncrona entre microserviços é realizada. O método *getVehicle* da classe *RestInventoryClient* é responsável por obter um veículo do *Inventory Service*. Esse método utiliza o *RestTemplate* para realizar uma requisição para *Inventory Service*. É importante notar que a classe *RestInventoryClient* implementa a interface de domínio *InventoryClient*. Dessa forma o domínio não está acoplado ao mecanismo de comunicação.

---

```
1 public class RestInventoryClient implements InventoryClient {
2     private RestTemplate restTemplate;
3     private String baseUrl;
4
5     @Override
6     public Vehicle getVehicle(int id) {
7         HttpHeaders headers = new HttpHeaders();
8         headers.set("Authorization", "Bearer " + getToken());
9         HttpEntity<VehicleResponse> entity = new HttpEntity<>(headers);
10
11         ResponseEntity<VehicleResponse> response = restTemplate.exchange(baseUrl
12             + "/vehicles/{id}", HttpMethod.GET, entity, VehicleResponse.class,
13             id);
14         if (response.getStatusCode().value() == 404) {
15             throw new EntityNotFoundException("Vehicle not found");
16         }
17         if (response.getStatusCode().isError()) {
18             throw new RuntimeException("Error while fetching vehicle");
19         }
20         return response.getBody().toVehicle();
21     }
22 }
```

---

Codigo 5.6 – Método para obter um veículo do *Inventory Service*

### 5.3.7 Comunicação assíncrona entre Microserviços

O [Codigo 5.7](#) apresenta como a comunicação assíncrona entre microserviços é realizada. A classe *SNSEventPublisher* é responsável por publicar eventos de reserva no *Booking Service*. O método *publishBookingEvent* recebe um evento de reserva, serializa o evento e publica no tópico . Por outro lado, a classe *SQSListener* é responsável por ouvir eventos de reserva no *Customer Service*. O método *listen* recebe uma mensagem do , converte a mensagem e incrementa o contador de reservas do usuário. Da mesma forma que na comunicação síncrona, o domínio não está acoplado ao mecanismo de comunicação.

---

```

1 // class in Booking Service
2 public class SNSEventPublisher implements BookingEventPublisher {
3     private final SnsClient snsClient;
4     private final ObjectMapper objectMapper;
5
6     private String topicArn;
7
8     public void publishBookingEvent(BookingEvent bookingEvent) {
9         String message = objectMapper.writeValueAsString(bookingEvent);
10
11         PublishRequest request = PublishRequest.builder()
12             .topicArn(topicArn)
13             .message(message)
14             .build();
15
16         PublishResponse response = snsClient.publish(request);
17     }
18 }
19
20 // class in Customer service
21 public class SQSListener {
22     private final CustomerService service;
23     private final ObjectMapper objectMapper;
24
25     @SqsListener(queueNames = "${aws.queues.bookings}")
26     public void listen(Message message) {
27         try {
28             MessageBody body = objectMapper.readValue(message.body(),
29                 MessageBody.class);
30             BookingMessage bookingMessage = objectMapper.readValue(body.
31                 getMessage(), BookingMessage.class);
32
33             log.info("Received message: {}", bookingMessage);
34
35             service.incrementBookingsCount(bookingMessage.getUserId());
36         } catch (Exception e) {
37             throw new RuntimeException("Error while processing message", e);
38         }
39     }
40 }

```

---

Código 5.7 – Código para realizar comunicação assíncrona entre microsserviços

## 5.4 Testes

Esta seção apresenta os testes realizados para validar os requisitos funcionais e não funcionais do sistema.

### 5.4.1 Testes Unitários

Os testes unitários foram realizados utilizando o *framework* JUnit. O [Código 5.8](#) apresenta um teste unitário simples para o método *checkIn* da classe *Booking*. Esse teste verifica se o status da reserva é alterado para *IN\_PROGRESS* e se a data de check-in é definida quando o status é *CREATED*. Além disso, o teste verifica se uma exceção é lançada quando o status não é *CREATED*.

---

```
1  @Nested
2  @DisplayName("checkIn method")
3  class CheckInMethod {
4
5      @Test
6      @DisplayName("should change status to IN_PROGRESS and set checkedInAt to
7          current time when status is CREATED")
8      void
9          shouldChangeStatusToInProgressAndSetCheckedInAtToCurrentTimeWhenStatusIsCreated
10         () {
11         Booking booking = new Booking();
12         booking.setStatus(Booking.Status.CREATED);
13
14         booking.checkIn();
15
16         assertEquals(Booking.Status.IN_PROGRESS, booking.getStatus());
17         assertNotNull(booking.getCheckedInAt());
18     }
19
20     @Test
21     @DisplayName("should throw InvalidBookingStateException when status is not
22         CREATED")
23     void shouldThrowInvalidBookingStateExceptionWhenStatusIsNotCreated() {
24         Booking booking = new Booking();
25         booking.setStatus(Booking.Status.CANCELLED);
26
27         assertThrows(InvalidBookingStateException.class, booking::checkIn);
28     }
29 }
```

---

Codigo 5.8 – Teste unitário simples

### 5.4.2 Testes de Integração

O framework Mockito é utilizado para execução dos testes de integração de maneira isolada. O [Codigo 5.9](#) apresenta um teste de integração para o método *checkin* da classe *BookingService*. Esse teste verifica se uma exceção é lançada quando a reserva não é encontrada e se o status da reserva é alterado para *IN\_PROGRESS* quando a reserva é encontrada. Além disso, o teste verifica se o método *save* do repositório é chamado uma vez.

---

```
1 @Nested
2 @DisplayName("method: checkin")
3 class Checkin {
4
5     @Mock
6     private BookingRepository bookingRepository;
7
8     @Test
9     @DisplayName("when booking is not found, then it should throw a
10         EntityNotFoundException")
11     void whenBookingIsNotFound_thenItShouldThrowAEntityNotFoundException() {
12         assertThrows(EntityNotFoundException.class, () -> bookingService.checkin
13             (1));
14     }
15
16     @Test
17     @DisplayName("when all operations are successful, then it should checkin the
18         booking")
19     void whenAllOperationsAreSuccessful_thenItShouldCheckinTheBooking() {
20         when(bookingRepository.findById(1)).thenReturn(Optional.of(booking));
21         when(bookingRepository.save(booking)).thenReturn(booking);
22
23         bookingService.checkin(1);
24
25         assertEquals(Booking.Status.IN_PROGRESS, booking.getStatus());
26         verify(bookingRepository, times(1)).save(booking);
27     }
28 }
```

---

Codigo 5.9 – Teste de integração

### 5.4.3 Testes de Sistema

Os testes de sistema são realizados com auxílio do Testcontainers, uma ferramenta que permite a execução de testes de sistema com alto nível de confiabilidade através da criação de containers Docker. O [Codigo 5.10](#) apresenta um teste de sistema para a rota *POST /bookings*. Esse teste verifica se uma requisição inválida retorna um código de status 400 e se uma requisição válida retorna um código de status 201.



---

```

1  @Nested
2  @DisplayName("POST /bookings")
3  class PostBookings {
4
5      @Test
6      @DisplayName("When request is not valid, then it should return 400 Bad
7      Request")
8      void whenRequestIsValid_shouldReturn400() throws Exception {
9          mockMvc.perform(post("/bookings")
10                      .contentType(MediaType.APPLICATION_JSON)
11                      .content("{\"" +
12                          "\"vehicleId\": 1,\" +
13                          "\"userId\": \"\",\" +
14                          "\"startTime\": \"2023-01-01T00:00:00\",\" +
15                          "\"endTime\": \"2023-01-02T00:00:00\"\" +
16                          \"}\"))
17                      .andExpect(status().isBadRequest());
18
19      @Test
20      @WithMockEmployee
21      @DisplayName("when request is valid, then it should return 201 Created")
22      void whenRequestIsValid_shouldReturn201() throws Exception {
23          when(inventoryClient.getVehicle(1)).thenReturn(new Vehicle(1, "vehicle",
24                          "model", 2023));
25
26          mockMvc.perform(post("/bookings")
27                      .contentType(MediaType.APPLICATION_JSON)
28                      .content("{\"" +
29                          "\"vehicleId\": 1,\" +
30                          "\"userId\": \"user-id\",\" +
31                          "\"startTime\": \"2026-01-01T00:00:00\",\" +
32                          "\"endTime\": \"2026-01-02T00:00:00\"\" +
33                          \"}\"))
34                      .andExpect(status().isCreated());
35      }

```

---

Codigo 5.10 – Teste de sistema

#### 5.4.4 Testes de Carga

Para os testes de carga, o framework *Gatling* é utilizado. O [Codigo 5.11](#) apresenta um teste de carga para o cenário de reserva. Esse teste simula o cadastro de 100 usuários, a criação de um veículo, a reserva do veículo e o check-in e check-out da reserva. O teste é realizado com 100 usuários simultâneos durante 30 minutos como especificado no [Capítulo 4](#).

---

```

1 private final int NUM_USERS = 100;
2 private final int RAMP_UP_TIME = 30;
3
4 ScenarioBuilder registerScenario = scenario("register")
5     .feed(feeder)
6     .exec(
7         http("Register Customer")
8             .post(CUSTOMER_BASE_URL + "/register")
9             .header("Content-Type", "application/json")
10            .body(StringBody("{\"name\": \"#{name}\", \"email\": \"#{email}\", \"password\": \"#{password}\"}"))
11            .check(status().is(200))
12            .check(jsonPath("$.token").saveAs("token")))
13    ).exec(http("Create Vehicle")
14        .put(INVENTORY_BASE_URL + "/vehicles")
15        .header("Content-Type", "application/json")
16        .header("Authorization", "Bearer " + EMPLOYEE_TOKEN)
17        .body(StringBody("{\"typeId\":1,\"make\":\"Hyundai\",\"model\":\"Creta\",\"year\":2024,\"mileage\":10000,\"licensePlate\":\"23525\",\"chassisNumber\":\"252355125\",\"engineNumber\":\"234242\",\"color\":\"white\"}"))
18        .check(status().is(200))
19        .check(jsonPath("$.id").saveAs("vehicleId")))
20    ).exec(http("Book Vehicle")
21        .post(BOOKING_BASE_URL + "/bookings")
22        .header("Content-Type", "application/json")
23        .header("Authorization", "Bearer #{token}")
24        .body(StringBody("{\"vehicleId\":#{vehicleId},\"startTime\":\"2024-12-03T10:15:30\",\"endTime\":\"2024-12-04T10:15:30\"}"))
25        .check(status().is(201))
26        .check(jsonPath("$.id").saveAs("bookingId")))
27    ).exec(http("Check in Booking")
28        .patch(BOOKING_BASE_URL + "/bookings/#{bookingId}/check-in")
29        .header("Content-Type", "application/json")
30        .header("Authorization", "Bearer " + EMPLOYEE_TOKEN)
31        .check(status().is(200)))
32    ).exec(http("Check out Booking")
33        .patch(BOOKING_BASE_URL + "/bookings/#{bookingId}/check-out")
34        .header("Content-Type", "application/json")
35        .header("Authorization", "Bearer " + EMPLOYEE_TOKEN)
36        .check(status().is(200)));
37
38
39 HttpProtocolBuilder httpProtocol =
40     http.enableHttp2();
41
42 {
43     setUp(
44         registerScenario.injectClosed(
45             rampConcurrentUsers(0).to(NUM_USERS).during(Duration.ofMinutes(RAMP_UP_TIME))
46         )
47     ).protocols(httpProtocol);
48 }

```

---

Codigo 5.11 – Teste de carga

## 6 Resultados e Discussões

Este capítulo apresenta os resultados obtidos com a execução dos testes de carga descritos no [Capítulo 4](#). Além disso, há uma discussão geral sobre o processo de desenvolvimento do caso de uso com a utilização da [Arquitetura de Microserviços \(AMS\)](#) e [Domain-Driven Design \(DDD\)](#).

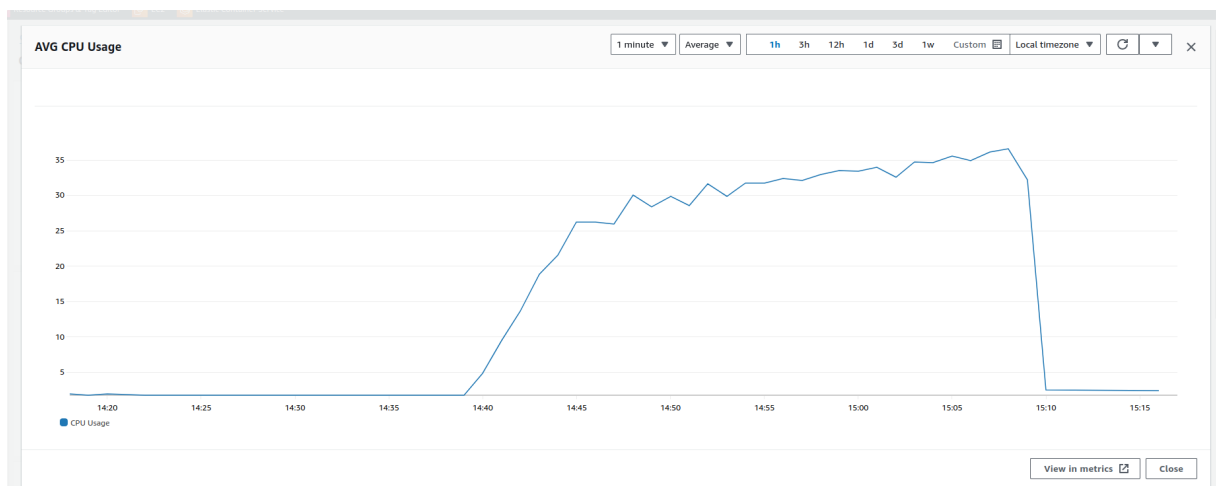
### 6.1 Resultados

Essa seção apresenta uma análise gráfica dos resultados obtidos com a execução dos testes de carga. Os gráficos foram gerados pelo *AWS CloudWatch* a partir das métricas coletadas dos serviços de computação e banco de dados da *Amazon Web Services* durante a execução dos testes. Os dados apresentados apresentam a média dos serviços.

#### 6.1.1 Utilização da CPU

Na [Figura 20](#) é possível observar a utilização da CPU durante a execução dos testes de carga. Durante toda a simulação, a utilização de CPU se manteve em torno de 35%, um valor considerado baixo. Isso indica que o sistema é capaz de suportar um número maior de requisições sem que haja um aumento significativo na utilização de CPU. Esse valor também está abaixo do alvo de 70% estabelecido anteriormente.

Figura 20 – Utilização da CPU

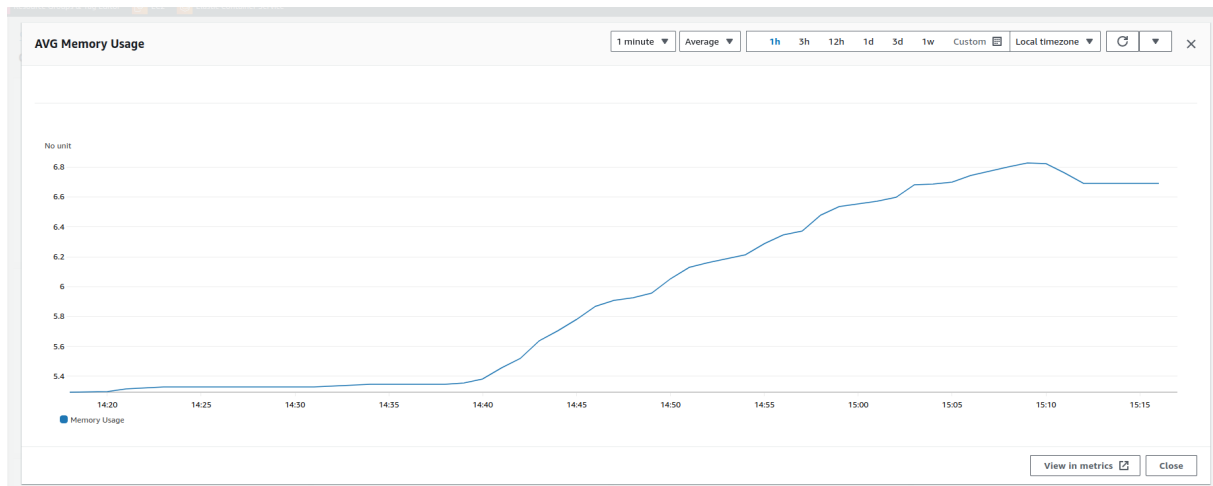


Fonte: o autor

### 6.1.2 Utilização da Memória RAM

Na Figura 21 é possível observar a utilização da memória RAM durante a execução dos testes de carga. Houve um uso muito baixo de memória, com uma média em torno de 7%. Isso indica que o sistema como um todo é *CPU-bound*, termo utilizado para descrever sistemas que são limitados pela CPU e não pela memória. Essa métrica também ficou abaixo do alvo.

Figura 21 – Utilização da Memória RAM

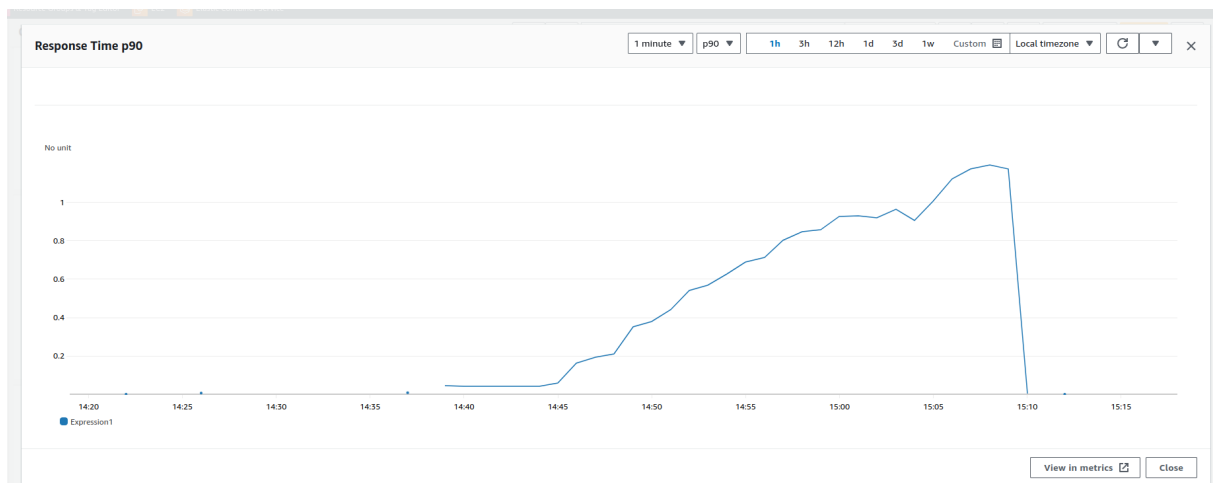


Fonte: o autor

### 6.1.3 Tempo de Resposta

Na Figura 22 é possível observar o tempo de resposta das requisições durante a simulação. Utilizando como base o P90, o tempo de resposta se manteve em torno de 1 segundo, um valor considerado aceitável. Esse valor também está abaixo do alvo de 2 segundos estabelecido anteriormente.

Figura 22 – Tempo de Resposta

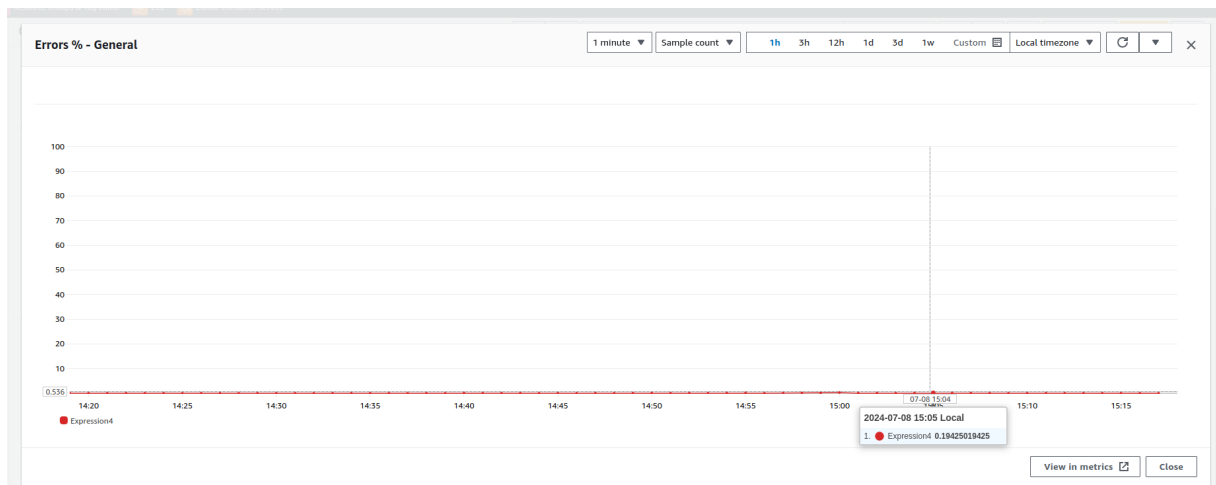


Fonte: o autor

### 6.1.4 Porcentagem de erros

Na Figura 23 é possível observar a porcentagem de erros durante a execução dos testes de carga. Durante toda a simulação, a porcentagem de erros se manteve em 0.2% (média), indicando que o sistema é capaz de suportar um grande número de requisições sem impactar os usuários. Esse valor também está abaixo do alvo de 1% estabelecido anteriormente.

Figura 23 – Porcentagem de Erros



Fonte: o autor

## 6.2 Trabalhos Futuros

Como trabalhos futuros, é possível destacar a construção de novos estudos de caso com a utilização de DDD e AMS em outros contextos complexos como, por exemplo, sistemas financeiros, sistemas de logística e sistemas de saúde. Além disso, um trabalho futuro interessante seria a comparação de entre dois sistemas de domínios iguais, um desenvolvido com as técnicas apresentadas nesse trabalho e um outro somente com DDD ou AMS. Assim, seria possível quantificar os benefícios e desvantagens da utilização dessas tecnologias em conjunto.

## 6.3 Discussões

Essa seção apresenta uma discussão geral sobre o processo de desenvolvimento do caso de uso com a utilização da AMS e DDD. A utilização dessas tecnologias trouxe diversos desafios e benefícios ao desenvolvimento do sistema.

A separação de serviços em microsserviços permitiu que cada parte do sistema fosse desenvolvida de forma independente, facilitando a manutenção e evolução do sistema.

Além disso, a utilização de [DDD](#) permitiu que o domínio do negócio fosse modelado de forma mais clara e eficiente, facilitando o entendimento do sistema como um todo.

Da mesma forma, como cada microsserviço tem um foco específico, se torna mais fácil o entendimento de como cada parte do sistema funciona. Isso também facilita a escalabilidade do sistema, uma vez que é possível escalar apenas os serviços que estão sobrecarregados.

A utilização de [AMS](#) e [DDD](#) trouxe diversos desafios ao desenvolvimento do sistema. O maior deles foi a necessidade de um maior esforço de *design upfront* para garantir a correta separação de serviços e a definição do domínio do negócio. Isso se deve ao fato de que a separação de serviços em microsserviços e a utilização de [DDD](#) requerem um maior entendimento do negócio e da arquitetura do sistema.

Além disso, a complexidade aumentada para criação e execução de testes também foi um desafio. Como cada microsserviço é um sistema independente, é necessário criar testes para cada parte do sistema, o que aumenta a complexidade dos testes.

Outro desafio foi o maior custo computacional para executar o sistema localmente. Como cada microsserviço é uma aplicação independente, possui seu próprio processo em nível de sistema operacional, o que aumenta o custo computacional para executar o sistema localmente. Assim, os desenvolvedores precisam de máquinas mais potentes.

Por fim, a maior complexidade de deploy também foi um desafio. É necessário criar mais serviços, configurar máquinas virtuais, *load balancers* e banco de dados. Além disso, é importante configurar corretamente a rede para garantir a comunicação entre os serviços. Inicialmente, se tem um custo maior para manter o sistema em produção. Porém, com o crescimento do tráfego, a escalabilidade do sistema se torna mais fácil.

## 7 Conclusão

Este trabalho apresentou um estudo de caso sobre a utilização de [Domain-Driven Design \(DDD\)](#) e [Arquitetura de Microsserviços \(AMS\)](#) no desenvolvimento de um sistema para uma locadora de veículos. O objetivo foi avaliar como essas tecnologias podem ser utilizadas para aumentar a escalabilidade e resiliência de sistemas complexos. Com base dos resultados obtidos com a execução dos testes de carga, é possível concluir que essa abordagem é eficaz para a construção de sistemas distribuídos. Além disso, nota-se como diferentes partes do sistema são expostas a tráfegos distintos, o que permite a escalabilidade independente de cada serviço.

Com a modelagem do sistema através de [DDD](#), foi possível definir limites claros entre os diferentes contextos do negócio com a utilização de [Bounded Context \(BC\)](#). Além disso, a utilização de [DDD](#) permitiu a definição de um modelo de domínio rico e expressivo, que reflete de forma fiel as regras de negócio da locadora de veículos.

Por outro lado, o desenvolvimento deste estudo de caso trouxe diversos desafios como a necessidade de um maior esforço de *design upfront* para garantir a correta separação de serviços e a definição do domínio do negócio. Além disso, percebe-se uma maior complexidade para realização de testes e depuração de problemas, uma vez que o sistema é composto por diversos serviços independentes.

Este trabalho cumpriu com os objetivos propostos, na medida que apresentou uma estratégia para transformar requisitos funcionais em um *design* com [AMS](#) e [DDD](#), forneceu informações relevantes para definição do estilo de comunicação adequado entre microsserviços e demonstrou desempenho e escalabilidade do sistema desenvolvido através de testes de carga.

# Referências

- AMAZON WEB SERVICES. *Amazon ECS*. 2023. Disponível em: <<https://aws.amazon.com/pt/ecs/>>. Citado na página 48.
- AMAZON WEB SERVICES. *Amazon SNS*. 2023. Disponível em: <<https://aws.amazon.com/pt/sns/>>. Citado na página 47.
- AMAZON WEB SERVICES. *Amazon SQS*. 2023. Disponível em: <<https://aws.amazon.com/pt/sqs/>>. Citado na página 47.
- AMAZON WEB SERVICES. *Amazon Web Services*. 2023. Disponível em: <<https://aws.amazon.com/pt/>>. Citado na página 48.
- AMAZON WEB SERVICES. *DynamoDB*. 2023. Disponível em: <<https://aws.amazon.com/pt/dynamodb/>>. Citado 2 vezes nas páginas 46 e 47.
- BARAUNA, H.; HARDARDT, P. *TDD e BDD na prática: Construa aplicações Ruby usando RSpec e Cucumber*. Casa do Código, 2020. ISBN 9786586110302. Disponível em: <<https://books.google.com.br/books?id=lsb2DwAAQBAJ>>. Citado na página 47.
- BROWN, S. *C4 Model*. 2023. Disponível em: <<https://c4model.com/>>. Citado na página 46.
- COCKBURN, A. Hexagonal architecture. Netflix Tech Blog, 2005. Citado na página 27.
- DIEPENBROCK, A.; RADEMACHER, F.; SACHWEH, S. An ontology-based approach for domain-driven design of microservice architectures. In: . [s.n.], 2017. Disponível em: <[https://www.scopus.com/inward/record.uri?eid=2-s2.0-85083245566&doi=10.18420%2f2017\\_177&partnerID=40&md5=011546174a451e1af30742079d4bee6b](https://www.scopus.com/inward/record.uri?eid=2-s2.0-85083245566&doi=10.18420%2f2017_177&partnerID=40&md5=011546174a451e1af30742079d4bee6b)>. Citado 4 vezes nas páginas 34, 35, 36 e 37.
- DOCKER. *Docker*. 2023. Disponível em: <<https://www.docker.com/>>. Citado na página 48.
- EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley, 2004. Citado 6 vezes nas páginas 17, 18, 19, 33, 46 e 52.
- FARSI, H. et al. Following domain driven design principles for microservices decomposition: Is it enough? In: . [s.n.], 2021. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85125661319&doi=10.1109%2fAICCSA53542.2021.9686947&partnerID=40&md5=722f2e36999e79af62e01002a0251206>>. Citado 3 vezes nas páginas 34, 37 e 38.
- FOWLER, M. *Domaindrivendesign*. 2020. Citado 2 vezes nas páginas 13 e 17.
- FOWLER, M.; LEWIS, J. *Microservice*. 2014. Citado na página 13.
- GATLING. *Gatling*. 2024. Disponível em: <<https://gatling.io/>>. Citado na página 47.
- GOMES, A. *Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio*. Casa do Código, 2014. ISBN 9788566250992. Disponível em: <<https://books.google.com.br/books?id=zHCCCwAAQBAJ>>. Citado na página 39.



- HASHICORP. *Terraform*. 2023. Disponível em: <<https://www.terraform.io/>>. Citado na página 49.
- IZRAILEVSKY, S. V. Y.; MESHENBERG, R. Completing the netflix cloud migration. Netflix Tech Blog, 2014. Citado 2 vezes nas páginas 13 e 14.
- JOSELYNE, M. I.; BAJPAI, G.; NZANYWAYINGOMA, F. A systematic framework of application modernization to microservice based architecture. In: . [s.n.], 2021. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85124674237&doi=10.1109%2fICEET53442.2021.9659783&partnerID=40&md5=e1f9d6287b6b78cf93f96286a4a7f402>>. Citado na página 34.
- JOSÉLYNE, M. I. et al. Partitioning microservices: A domain engineering approach. In: . [s.n.], 2018. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85051527248&doi=10.1145%2f3195528.3195535&partnerID=40&md5=c34c5ca0054ee0eb41a5d11cbc6b22ff>>. Citado na página 34.
- JUNIT. *JUnit*. 2024. Disponível em: <<https://junit.org/junit5/>>. Citado na página 47.
- KAPFERER, S.; ZIMMERMANN, O. Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: . [s.n.], 2020. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85082984325&partnerID=40&md5=364b5b1e485f3396de84ac9a1c86e435>>. Citado na página 34.
- KOLNY, M. Scaling up the prime video audio/video monitoring service and reducing costs by 90Prime Video Tech, 2023. Citado na página 14.
- LISKOV, B.; GUTTAG, J. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. [S.l.]: Addison-Wesley, 2000. ISBN 9780201657685. Citado na página 50.
- LOUKIDES, M.; SWOYER, S. *Microservice Adoption*. [S.l.], 2020. Citado na página 13.
- MA, S.-P. et al. Microservice migration using strangler fig pattern and domain-driven design. *Journal of Information Science and Engineering*, 2022. Disponível em: <[https://www.scopus.com/inward/record.uri?eid=2-s2.0-85144142091&doi=10.6688%2fJISE.202211\\_38%286%29.0010&partnerID=40&md5=42dc871f0666914cdc97a24036f96ce5](https://www.scopus.com/inward/record.uri?eid=2-s2.0-85144142091&doi=10.6688%2fJISE.202211_38%286%29.0010&partnerID=40&md5=42dc871f0666914cdc97a24036f96ce5)>. Citado 2 vezes nas páginas 33 e 34.
- MERSON, P.; YODER, J. Modeling microservices with ddd. In: . [s.n.], 2020. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85085748453&doi=10.1109%2fICSA-C50368.2020.00010&partnerID=40&md5=0653c03a1e47ca80481ed652890b314c>>. Citado na página 34.
- MOCKITO. *Mockito*. 2024. Disponível em: <<https://site.mockito.org/>>. Citado na página 47.
- NEWMAN, S. *S. Building Microservices: Designing Fine-Grained Systems*. [S.l.]: O'Reilly, 2021. Citado 8 vezes nas páginas 14, 21, 22, 23, 24, 25, 26 e 27.
- ORACLE. *Java*. 2024. Disponível em: <<https://www.java.com/pt-BR/>>. Citado na página 46.

- PARNAS, D. L. Information distribution aspects of design methodology. 3 2012. Disponível em: <[https://kilthub.cmu.edu/articles/journal\\_contribution/Information\\_distribution\\_aspects\\_of\\_design\\_methodology/6606470](https://kilthub.cmu.edu/articles/journal_contribution/Information_distribution_aspects_of_design_methodology/6606470)>. Citado na página 24.
- PETRASCH, R. Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication. In: . [s.n.], 2017. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85031757791&doi=10.1109%2fJCSSE.2017.8025912&partnerID=40&md5=aea02d9b1ca05f568a51982748d9bb3e>>. Citado na página 34.
- PIVOTAL SOFTWARE. *Spring Boot*. 2023. Disponível em: <<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started.introducing-spring-boot>>. Citado na página 46.
- POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL*. 2023. Disponível em: <<https://www.postgresql.org/>>. Citado 2 vezes nas páginas 46 e 48.
- PROTT, D.; WIKES, L. Understanding service-oriented architecture. In: . [S.l.]: CDBI Forum, 2009. Citado na página 21.
- RADEMACHER, F.; SORGALLA, J.; SACHWEH, S. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 2018. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85046890847&doi=10.1109%2fMS.2018.2141028&partnerID=40&md5=1cc57dbf003187e5680636f9c56a779a>>. Citado 3 vezes nas páginas 34, 35 e 36.
- RADEMACHER, F. et al. Microservice architecture and model-driven development: Yet singles, soon married (?). In: . [s.n.], 2018. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85058570602&doi=10.1145%2f3234152.3234193&partnerID=40&md5=088f5f1637910b22bb2e80c4a0035195>>. Citado na página 34.
- RICHARDSON, C. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN 9781617294549. Disponível em: <<https://books.google.com.br/books?id=UeK1swEACAAJ>>. Citado 5 vezes nas páginas 21, 22, 23, 26 e 27.
- SINGJAI, A.; ZDUN, U.; ZIMMERMANN, O. Practitioner views on the interrelation of microservice apis and domain-driven design: A grey literature study based on grounded theory. In: . [s.n.], 2021. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85106968487&doi=10.1109%2fICSA51549.2021.00011&partnerID=40&md5=476a80ff6c833f8671930c4a98147cea>>. Citado 5 vezes nas páginas 33, 34, 35, 37 e 38.
- SINGJAI, A. et al. Patterns on deriving apis and their endpoints from domain models. In: . [s.n.], 2021. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85123784790&doi=10.1145%2f3489449.3489976&partnerID=40&md5=2eaabb15a62d4093d97f7f8b0cdad0c0>>. Citado 4 vezes nas páginas 33, 34, 37 e 38.
- TANENBAUM, A. *Sistemas operacionais modernos*. Prentice-Hall do Brasil, 2010. ISBN 9788576052371. Disponível em: <<https://books.google.com.br/books?id=nDatQwAACAAJ>>. Citado na página 20.
- TESTCONTAINERS. *Testcontainers*. 2024. Disponível em: <<https://www.testcontainers.org/>>. Citado na página 47.

- VURAL, H.; KOYUNCU, M. Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access*, 2021. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85111946709&doi=10.1109%2fACCESS.2021.3060895&partnerID=40&md5=351dd180169e50554e74049eb7d71c18>. Citado 4 vezes nas páginas 33, 34, 37 e 38.
- YALE, Y.; SILVEIRA, H.; SUNDARAM, M. A microservice based reference architecture model in the context of enterprise architecture. In: . [s.n.], 2017. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85016732570&doi=10.1109%2fIMCEC.2016.7867539&partnerID=40&md5=ff4d39cf614f3b0c27af9cda1cad7404>. Citado na página 34.
- YOURDON, E.; CONSTANTINE, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. 1st. ed. USA: Prentice-Hall, Inc., 1979. ISBN 0138544719. Citado na página 24.
- ZHANG, K. et al. Design of domain-driven microservices-based software talent evaluation and recommendation system. In: . [s.n.], 2022. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85147898540&doi=10.1109%2fICEKIM55072.2022.00076&partnerID=40&md5=f8f586b54951b8590e148a289f1125b6>. Citado na página 34.
- ÖZKAN, O.; BABUR, ; BRAND, M. van den. Refactoring with domain-driven design in an industrial context: An action research report. *Empirical Software Engineering*, 2023. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161920562&doi=10.1007%2fs10664-023-10310-1&partnerID=40&md5=fa5a16613281a3184bf1dfe64f434d5e>. Citado 3 vezes nas páginas 34, 37 e 38.