

Gustavo Bonifácio Conceição

# **ANÁLISE COMPARATIVA ENTRE ARQUITETURA DE SISTEMA MONOLÍTICO E MICROSERVIÇO: DESEMPENHO E CONSUMO DE RECURSO**

Trabalho de conclusão de curso apresentado  
como requisito parcial para a obtenção do  
grau de Bacharel em Sistemas de Informação  
Instituto Federal Catarinense.

Instituto Federal Catarinense – IFC

Câmpus Araquari

Sistemas de Informação

Orientador: Prof. Dr. Marco Antonio Torrez Rojas

Araquari – SC

DEZEMBRO de 2023

Gustavo Bonifácio Conceição

ANÁLISE COMPARATIVA ENTRE ARQUITETURA DE SISTEMA MONOLÍTICO E MICROSERVIÇO: DESEMPENHO E CONSUMO DE RECURSO/ Gustavo Bonifácio Conceição. – Araquari – SC, DEZEMBRO de 2023-

82 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Marco Antonio Torrez Rojas

Monografia (Graduação) – Instituto Federal Catarinense – IFC  
Câmpus Araquari

Sistemas de Informação, DEZEMBRO de 2023.

I. *Spring Boot*. II. Microserviços. III. *Amazon Web Service*. IV. Arquitetura de *Software*. V. Teste de Desempenho. VI. Utilização de Recursos.

Gustavo Bonifácio Conceição

# **ANÁLISE COMPARATIVA ENTRE ARQUITETURA DE SISTEMA MONOLÍTICO E MICROSERVIÇO: DESEMPENHO E CONSUMO DE RECURSO**

Trabalho de conclusão de curso apresentado  
como requisito parcial para a obtenção do  
grau de Bacharel em Sistemas de Informação  
Instituto Federal Catarinense.

Araquari – SC, 18 de dezembro de 2023:

---

**Prof. Dr. Marco Antonio Torrez Rojas**  
Orientador

---

**Prof. Dr. Eduardo Silva**  
Convidado 1

---

**Prof. Dr. Mehran Misaghi**  
Convidado 2

Araquari – SC  
DEZEMBRO de 2023

*Dedico esse trabalho a minha família. A minha mãe, pelas várias vezes que ela abriu mão de algo para ela em prol da educação e do futuro dos seus filhos. Ao meu irmão, por ser meu melhor amigo e companheiro de aventuras, brincadeiras e séries ruins. E por último, mas não menos importante, eu dedico esse trabalho ao meu pai que não viveu para ver os filhos completarem os estudos, mas que esteve comigo em pensamentos durante todos esse anos.*

# Agradecimentos

Sou grato a minha mãe, Luci, por me apoiar nas minhas decisões que me trouxeram até aqui, algo que só foi possível pois aprendi com ela a coragem e persistência para encarar desafios, como este trabalho. Também agradeço ao meu irmão, Rodrigo, pelo seu jeito divertido de ser, e por estar sempre presente durante os momentos que precisei de opinião crítica sobre o desenvolvimento desse projeto.

Além disso, agradeço a minha namorada Maria, por seu companheirismo e sua confiança, e pelos nos momentos que compartilhamos juntos, que impulsionam minhas decisões pensando no futuro.

Gratidão também ao Instituto Federal Catarinense, por me apresentar um ambiente inspirador, com professores que me instigaram a buscar e compartilhar conhecimento, e proporcionar amizades com pessoas que tenho como referência profissional e pessoal.

E por fim, deixo meu agradecimento a Adson Vieira, ex-colega de profissão e que prestou grande ajuda no início deste trabalho, sanando dúvidas e apresentando ajuda com minhas várias incertezas sobre como abordar os assunto aqui expressados. A Vinícius Acordi Soethe, agradeço pela consultoria a cerca de modelos de implementação de *software*. Também agradeço a Henry Peters Couto, estagiário da fábrica de *software* do Instituto Federal Catarinense, que me auxiliou com conhecimentos técnicos e boa vontade para me impulsionar a concluir este trabalho.

*“programming gives me this amazing power, to build my hole little universe, with its own rules, paradigms and practices, and create something out of nothing with the pure power of logic .”*

*- Linda Liukas*

# Resumo

Com o intuito de apresentar uma análise comparativa entre um sistema desenvolvido utilizando arquitetura monolítica e outro com arquitetura de microsserviço para auxiliar na tomada de decisão sobre qual arquitetura empregar em um projeto de software, foram desenvolvidos dois sistemas em Java com utilização do *framework Spring Boot*, com banco de dados MySQL, empregados as duas arquiteturas. Foram realizados o *baseline test*, ou teste de linha de base, realizado com 1000 usuários virtuais realizando requisições aos sistemas, para definir um ponto de referência para comparação com o resultado do *load test*, ou testes de carga, realizado com 3000 usuários virtuais realizando requisições aos sistemas, sendo este um teste realizado nos sistemas com um número superior de requisições para avaliar o seu funcionamento com um acesso maior de usuários, coletando dados sobre o funcionamento em duas métricas: tempo de resposta de requisições HTTP e utilização de CPU do ambiente AWS onde os sistemas foram hospedados. O sistema com microsserviços apresentou tempo de resposta das requisições superior ao sistema monolítico, assim como maior consumo de CPU, devido aos mecanismos necessários para a comunicação entre os diversos microsserviços presentes no sistema, revelando um ponto negativo desta arquitetura, o que torna a sua escolha não recomendada dentro dos parâmetros avaliados neste trabalho.

**Palavras-chave:** Arquitetura de *software*. Microsserviços. Monolítico. Desempenho. Consumo de Recurso.

# Abstract

*With the intent to present a comparative analysis between a system developed using monolithic architecture and another with microservice architecture to assist in decision making on which architecture to employ in a software project, two systems were developed in Java using the Spring Boot framework, with MySQL database, with both architectures. The baseline test carried out with 1000 virtual users making requests to the systems, to define a reference point for comparison with the result of the load test, carried out with 3000 virtual users making requests to the systems, this being a test carried out on systems with a higher number of requests to evaluate their operation with greater user access, collecting data on operation in two analyses: response time for HTTP requests and use of CPU of the AWS environment where the systems were hosted. The system with microservices presented a higher response time for requests than the monolithic system, as well as greater CPU consumption, due to the mechanisms necessary for communication between the different microservices present in the system, revealing a negative point of this architecture, which makes its choice not recommended within the parameters evaluated in this work..*

**Keywords:** *Software architecture. Microservices. Monolithic. Performance. Resource Consumption.*



# Lista de ilustrações

Figura 1 – Diferentes Arquiteturas de software. . . . .	18
Figura 2 – Arquitetura Monolítica. . . . .	19
Figura 3 – Arquitetura de Microserviços. . . . .	20
Figura 4 – Aplicação com Microserviços e API Gateway. . . . .	21
Figura 5 – Padrão de Projeto DTO. . . . .	24
Figura 6 – Arquitetura monolítica aplicada ao experimento. . . . .	27
Figura 7 – Banco de dados da aplicação monolítica. . . . .	28
Figura 8 – Estrutura de diretórios a partir do diretório app do sistema com arquitetura monolítica. . . . .	29
Figura 9 – Instância EC2 da aplicação monolítica. . . . .	30
Figura 10 – RDS com o banco de dados utilizado na instância EC2 da aplicação monolítica. . . . .	30
Figura 11 – Diagrama de funcionamento da requisição Criar Cliente, no sistema com arquitetura monolítica. . . . .	31
Figura 12 – Diagrama de funcionamento da requisição Criar Produto, no sistema com arquitetura monolítico. . . . .	32
Figura 13 – Diagrama de funcionamento da requisição Adiciona Produto ao Carrinho, no sistema com arquitetura monolítica. . . . .	33
Figura 14 – Diagrama de funcionamento da requisição Pagar Pedido em Aberto, no sistema com arquitetura monolítica. Fonte: O autor . . . . .	34
Figura 15 – Arquitetura de microserviços aplicada ao experimento. . . . .	35
Figura 16 – Banco de dados da aplicação com microserviços. . . . .	36
Figura 17 – Servidor Eureka funcionando em um ambiente local. . . . .	37
Figura 18 – Estrutura de diretórios a partir do diretório app do microserviço User-Registry no sistema com arquitetura de microserviços. . . . .	38
Figura 19 – Estrutura de diretórios a partir do diretório app do microserviço Catalog no sistema com arquitetura de microserviços. . . . .	38
Figura 20 – Estrutura de diretórios a partir do diretório app do microserviço Warehouse no sistema com arquitetura de microserviços. . . . .	39
Figura 21 – Estrutura de diretórios a partir do diretório app do microserviço Catalog no sistema com arquitetura de microserviços. . . . .	40
Figura 22 – Instâncias EC2 da aplicação com microserviços. . . . .	41
Figura 23 – RDS de cada instância EC2 na aplicação de arquitetura com microserviços. . . . .	42
Figura 24 – Diagrama de funcionamento da requisição Criar Cliente, no sistema com arquitetura de microserviços. . . . .	43

Figura 25 – Diagrama de funcionamento da requisição Criar Produto, no sistema com arquitetura de microsserviços. . . . .	44
Figura 26 – Diagrama de funcionamento da requisição Adiciona Produto ao Carrinho, no sistema com arquitetura de microsserviços. Fonte: O autor . . . . .	45
Figura 27 – Diagrama de funcionamento da requisição Pagar Pedido em Aberto, no sistema com arquitetura de microsserviços . . . . .	46
Figura 28 – Gráfico comparativo entre as arquiteturas monolítica e com microsserviço durante o baseline test para teste de desempenho. . . . .	49
Figura 29 – Gráfico comparativo entre as arquiteturas monolítica e com microsserviço durante o load test para teste de desempenho. . . . .	50
Figura 30 – Gráfico comparativo do sistema desenvolvido com arquitetura monolítica durante os testes de desempenho baseline e load. . . . .	51
Figura 31 – Gráfico comparativo do sistema desenvolvido com arquitetura de microsserviços durante os testes de desempenho baseline e load. . . . .	52
Figura 32 – Tabela do consumo da CPU das arquiteturas monolítica e com microsserviços, durante o load test. . . . .	54
Figura 33 – Tabela do consumo da CPU das arquiteturas monolítica e com microsserviços, durante o load test. . . . .	55
Figura 34 – Gráfico comparativo do consumo da CPU no sistema de arquitetura monolítica durante o baseline test e o load test. . . . .	57
Figura 35 – Gráfico comparativo do consumo de CPU no microsserviço Gateway durante o baseline test e o load test. . . . .	58
Figura 36 – Gráficos comparativos do consumo de CPU nos microsserviços User-Registry, Catalog, Warehouse e Cart, durante o baseline test e o load test. Fonte: O autor. . . . .	59
Figura 37 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema monolítico durante o baseline test. . . . .	79
Figura 38 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema monolítico durante o load test. . . . .	80
Figura 39 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema com microsserviços durante o baseline test. . . . .	81
Figura 40 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema com microsserviços durante o load test. . . . .	82

# Lista de abreviaturas e siglas

API	<i>Application Programming Interface</i>
HTTP	<i>Hypertext Transfer Protocol</i>
REST	<i>Representational State Transfer</i>
CRUD	<i>Create, Read, Update, Delete</i>
ORM	<i>Object Relational Mapper</i>
AWS	<i>Amazon Web Service</i>
EC2	<i>Elastic Compute Cloud</i>
RDS	<i>Relational Database Service</i>
SGBD	<i>Sistema de Gerenciamento de Banco de Dados</i>
DTO	<i>Data Transfer Object</i>
UI	<i>User Interface</i>
DB	<i>Data Base</i>
SOA	<i>Software Oriented Architecture</i>
SQL	<i>Structured Query Language</i>

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>14</b>
<b>1.1</b>	<b>JUSTIFICATIVA</b>	<b>15</b>
<b>1.2</b>	<b>PROBLEMA</b>	<b>15</b>
<b>1.3</b>	<b>OBJETIVO</b>	<b>16</b>
1.3.1	OBJETIVOS ESPECÍFICOS	16
<b>1.4</b>	<b>METODOLOGIA</b>	<b>16</b>
<b>2</b>	<b>REFERENCIAL TEÓRICO</b>	<b>17</b>
<b>2.1</b>	<b>ARQUITETURA DE SOFTWARE</b>	<b>17</b>
2.1.1	MONOLÍTICO	18
2.1.2	MICROSSERVIÇOS	20
<b>2.2</b>	<b>TESTES DE DESEMPENHO DE SISTEMA</b>	<b>22</b>
2.2.1	BASELINE TEST	22
2.2.2	LOAD TEST	22
<b>3</b>	<b>CENÁRIO DO EXPERIMENTO</b>	<b>23</b>
<b>3.1</b>	<b>MATERIAIS</b>	<b>25</b>
3.1.1	JAVA	25
3.1.2	SPRING BOOT	25
3.1.3	MySQL	25
3.1.4	POSTMAN	26
3.1.5	APACHE JMETER	26
3.1.6	AMAZON WEB SERVICES	26
<b>3.2</b>	<b>CENÁRIO DA ARQUITETURA MONOLÍTICA</b>	<b>27</b>
3.2.1	ESTRUTURA DE DIRETÓRIOS	28
3.2.2	HOSPEDAGEM	29
3.2.3	FUNCIONALIDADES	30
<b>3.3</b>	<b>CENÁRIO DA ARQUITETURA DE MICROSSERVIÇOS</b>	<b>34</b>
3.3.1	ESTRUTURA DE DIRETÓRIOS	37
3.3.2	HOSPEDAGEM	41
3.3.3	FUNCIONALIDADES	42
<b>4</b>	<b>TESTES</b>	<b>47</b>
<b>4.1</b>	<b>TESTES DE DESEMPENHO</b>	<b>47</b>
4.1.1	BASELINE TEST	48
4.1.2	LOAD TEST	50

4.1.3	RESULTADOS . . . . .	51
4.2	TESTE DE CONSUMO DE RECURSOS . . . . .	53
4.2.1	BASILINE TEST . . . . .	54
4.2.2	LOAD TEST . . . . .	55
4.2.3	RESULTADOS . . . . .	56
4.3	ANÁLISE DOS RESULTADOS . . . . .	60
5	CONCLUSÃO . . . . .	62
5.1	TRABALHOS FUTUROS . . . . .	62
	REFERÊNCIAS . . . . .	64
	APÊNDICES . . . . .	67
.1	Código do Controlador da entidade Client no sistema com arquitetura monolítica. . . . .	68
.2	Código do Controlador da entidade Product no sistema com arquitetura monolítica. . . . .	69
.3	Código do Controlador da entidade Stock no sistema com arquitetura monolítica. . . . .	70
.4	Código do Controlador da entidade ItemOrder no sistema com arquitetura monolítica. . . . .	71
.5	Código do Controlador da entidade Orders no sistema com arquitetura monolítica. . . . .	72
.6	Código Arquivo Application.yaml do Microserviço Gateway, Contendo as Rotas de Direcionamento para Cada Microserviço. . . . .	73
.7	Código do Controlador da Entidade Client, no Microserviço User-Registry, no Sistema com Arquitetura de Microserviços. . . . .	74
.8	Código do Controlador da Entidade Product, no Microserviço Catalog, no Sistema com Arquitetura de Microserviços. . . . .	75
.9	Código do Controlador da Entidade Stock, no Microserviço Warehouse, no Sistema com Arquitetura de Microserviços. . . . .	76
.10	Código do Controlador da Entidade ItemOrder, no Microserviço Cart, no Sistema com Arquitetura de Microserviços. . . . .	77
.11	Código do Controlador da Entidade Orders, no Microserviço Cart, no Sistema com Arquitetura de Microserviços. . . . .	78

# 1 INTRODUÇÃO

Por muitos anos, o desenvolvimento de software só conhecia uma forma de estruturar projetos: sistemas monolíticos, que é uma abordagem onde o software se encontra em um único projeto que contém as suas diferentes funcionalidades, além de banco de dados único, desenvolvido em uma única linguagem de programação em uma abordagem coesa. Como primeiro modo conhecido de estrutura, sua adesão foi rápida e por décadas a sua base continuou a mesma, como é explorado em [Blinowski, Ojdowska e Przybyłek \(2022\)](#).

Porém, os software cresceram em suas funcionalidades, com mais recursos sendo explorados, e devido a dependência de uma única instância do projeto sendo utilizada, conflitos ao realizar manutenção apareciam com cada vez mais frequência ([MARTENS et al., 2011](#)). Tanto para a melhoria no desenvolvimento quanto manutenção, novos conceitos surgiram, como entrega contínua, virtualização por demanda, modularização de sistemas, infraestrutura automatizada e sistemas escaláveis se tornaram conhecidos, trazendo novas abordagens para o desenvolvimento de software ([FOWLER, 2014](#)).

A abordagem do desenvolvimento modularizado começou com a arquitetura orientada a serviços, do inglês *Service Oriented Architecture* (SOA), em 1996. A abordagem SOA consiste em modularizar a aplicação em partes menores, possibilitando trabalhar com diferentes regras de negócio, tecnologias e técnicas de implementação ([VALIPOUR et al., 2009](#)). Ainda assim, houve uma evolução na abordagem SOA, movido pelo avanço de tecnologias de hospedagem em nuvem sob demanda, além da crescente necessidade de transformar um software em blocos menores além de serviços convencionais, mais especializados em suas funções ([BAŠKARADA; NGUYEN; KORONIOS, 2018](#)). Nesse cenário, surgiram os microsserviços.

Microsserviços são pequenos sistemas autônomos, com funções específicas, que se comunicam entre si, e grandes vantagens surgem com o seu uso, ligado à modularização do sistema baseada em funcionalidades, como um ambiente de implementação no serviço de hospedagem dedicado para cada microsserviço, à possibilidade de se utilizar diferentes tecnologias a cada microsserviço, além de bancos de dados dedicados para cada um. Por conta destas vantagens, a implementação de microsserviços ganharam grande espaço no mercado desde a última década ([DRAGONI et al., 2017](#)).

Porém, ainda assim os sistemas monolíticos não foram extintos, mesmo que tenham perdido espaço. Muitos projetos nascem sendo desenvolvidos para serem monólitos, e empresas se beneficiam do uso misto dessas arquiteturas ([HELLE et al., 2020](#)). Microsserviços são uma arquitetura útil, mas mesmo os seus defensores concordam que eles só são poderosos em sistemas mais complexos ([FOWLER, 2015](#)).

Algo que ocorre com relativa frequência é quando um software, profissional ou não, nascem como sistemas com arquitetura monolítica e conforme o software cresce e novas implementações precisam ser adicionadas, é feita uma migração para microsserviços que, se não planejada corretamente, pode causar grandes gastos, perda de tempo e aumento de complexidade para quem está fazendo essa implementação (PONCE; MÁRQUEZ; ASTUDILLO, 2019).

Este trabalho se propõe a apresentar uma análise comparativa entre dois sistemas desenvolvidos com as duas arquiteturas, avaliando o desempenho por meio de testes de tempo de resposta de requisições, e o consumo de recurso, medindo o consumo de CPU durante a realização das mesmas requisições, para que seja possível contribuir na tomada de decisões, auxiliar em estudos e embasar interessados na migração de uma arquitetura para outra.

## 1.1 JUSTIFICATIVA

Por conta de avanços na área de arquitetura de software e o surgimento de tecnologias que levaram ao surgimento da arquitetura baseada em microsserviços, houve uma grande adesão do mercado para essa abordagem de desenvolvimento. Porém, é necessário uma análise avaliando ambas arquiteturas de software, monolítica e baseada em microsserviços, para auxiliar na tomada de decisão sobre a utilização de cada abordagem.

## 1.2 PROBLEMA

Por conta da popularização da arquitetura de microsserviços após uma discussão sobre o tema em uma conferência sobre arquitetura de software em maior de 2011 (FOOTE, 2021), ocorreu um fenômeno onde várias empresas passaram por uma migração de uma arquitetura monolítica para esta nova tendência de mercado, com a crescente implementação desta arquitetura em seus projetos.

A arquitetura de um projeto de software é essencial para que o seu ciclo de vida seja constante, sem que haja impedimentos que tornem o sistema incapaz de acompanhar as mudanças que a maioria dos software estão sujeitos a sofrerem. Mesmo com isso em mente, ainda encontramos sistemas que são mal escritos ou projetados. Construir sistemas monolíticos, que são estruturados como um único bloco de desenvolvimento em que mais diversas funcionalidades são projetadas juntas, é uma abordagem mais simples de projeto, de fácil compreensão e estruturação.

Porém, conforme o sistema cresce e diferentes funcionalidades são adicionadas, a aplicação de conceitos de engenharia de software para escalonamento se vê necessária. Mesmo que o ideal seja já desenvolver o software voltado para o escalonamento, não é

incomum que o que aconteça seja a migração de um sistema monolítico para microsserviços. Ainda assim, uma análise de se uma arquitetura deve ser empregada é o que muitas empresas vêm fazendo, pensando no futuro de seus projetos.

## 1.3 OBJETIVO

O objetivo geral deste trabalho é realizar uma análise comparativa entre as arquiteturas de software monolítica e com microsserviços, por meio de testes de desempenho e testes de consumo de recurso de hardware.

### 1.3.1 OBJETIVOS ESPECÍFICOS

- Apresentar embasamento teórico referente a arquitetura monolítica e a arquitetura de microsserviços, a fim de nivelar conhecimentos;
- Desenvolver aplicações com o uso das duas arquiteturas de software;
- Realizar testes de desempenho, analisando o tempo de resposta de requisição da aplicação desenvolvida com ambas as arquiteturas;
- Realizar testes de consumo de recursos da máquina onde o software está hospedado, em ambas as arquiteturas.

## 1.4 METODOLOGIA

Este trabalho foi feito com a metodologia de pesquisa experimental, onde o cenário a ser estudado é a implementação de dois sistemas, um em arquitetura monolítica e o outro utilizando arquitetura baseada em microsserviços. Para o desenvolvimento dos software utilizados nos testes, foi utilizado a linguagem Java com o seu *framework Spring Boot*, e banco de dados MySQL. A checagem de funcionalidades via consulta de *Application Programming Interface* (API) foi realizada utilizando a ferramenta *Postman*, os testes de desempenho empregou-se o Apache JMeter e na hospedagem dos software, foi adotada a plataforma em nuvem *Amazon Web Service* (AWS).



## 2 REFERENCIAL TEÓRICO

Este capítulo apresentará conceitos sobre as definições das duas arquiteturas já citadas - arquitetura monolítica e baseada em microsserviços - de forma a nivelar conhecimentos sobre o que será apresentado na análise comparativa.

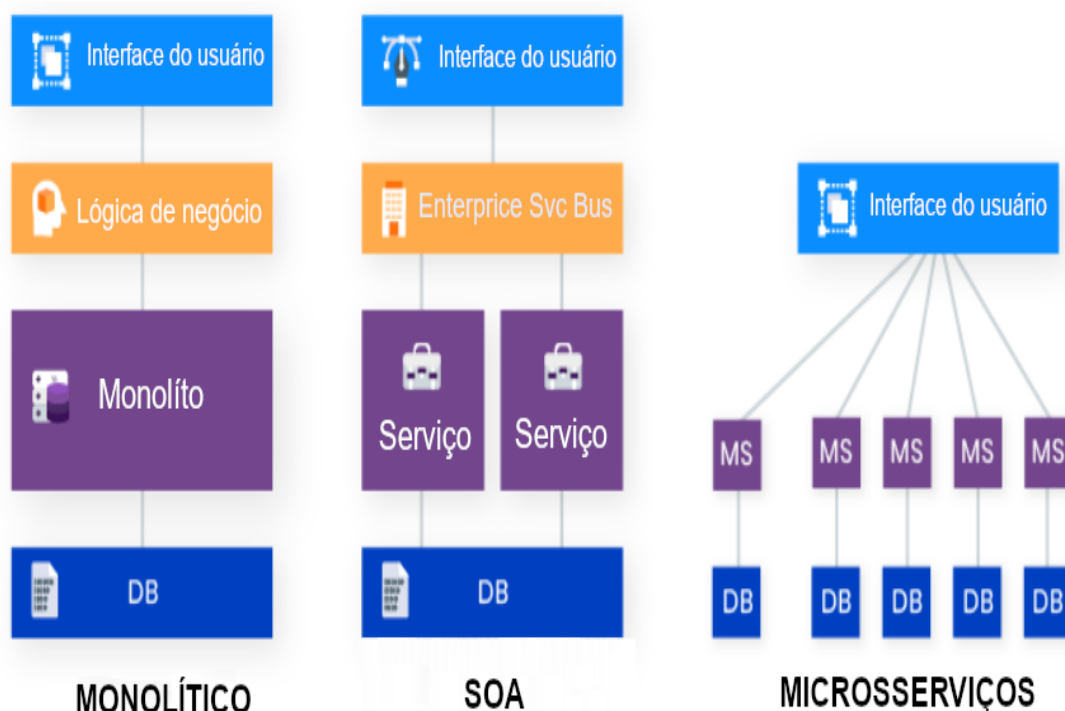
### 2.1 ARQUITETURA DE SOFTWARE

A arquitetura de software refere-se a forma como um sistema será estruturado, considerando a comunicação entre elementos, as responsabilidades de cada sessão da aplicação e uma eventual modularização, ou seja, separar o sistema em blocos de acordo com as suas funcionalidades e contexto de atuação ([GONÇALVES, 2021](#)). O entendimento de conceitos sobre arquitetura de software empregada em um projeto são essenciais para que o desenvolvimento e manutenção da aplicação seja coeso, cumprindo as responsabilidades que a arquitetura se propõe a atender.

O planejamento de qual arquitetura utilizar em um projeto é um trabalho feito considerando todas as especificações de requisitos do projeto, partindo de uma representação do software final com alto nível de abstração, de forma a percorrer todo o ciclo de vida de cada funcionalidade e/ou componente do software, considerando fatores como reusabilidade, eficiência e confiabilidade

Salientamos que a apresentação da arquitetura SOA está presente apenas para contextualizar as arquiteturas, mostrando o que existiu entre a arquitetura monolítica e a arquitetura com microsserviços. Este trabalho não aprofundará no seu estudo. A Figura [1](#) ilustra três arquiteturas de software (monolítica, SOA e microsserviços).

Figura 1 – Diferentes Arquiteturas de software.



Fonte: adaptado de (WOBETO, 2022).

A Figura 1 apresenta como são estruturadas um software em três arquiteturas de software distintas, com e o seu funcionamento ocorre da seguinte forma:

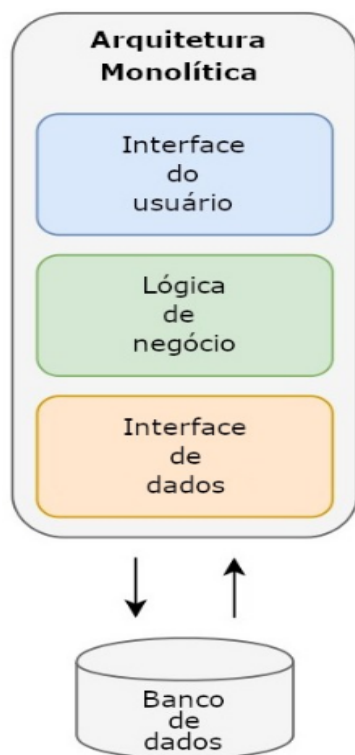
- **Monolítico:** Nesta abordagem o sistema está contido em um único projeto, com suas diversas funcionalidades dentro de um mesmo projeto, e compartilhando um único banco de dados;
- **SOA:** A orientação a serviços permite que o sistema tenha o as suas funcionalidades divididas por contexto, permitindo um gerenciamento e aplicação de diferentes regras de negócio para cada um. O banco de dados ainda é compartilhado;
- **Microserviços:** A utilização dos microserviços permite isolar completamente as funcionalidades, encapsulando as suas funcionalidades e garantindo um banco de dados para cada um.

### 2.1.1 MONOLÍTICO

A arquitetura de software monolítico é uma abordagem de desenvolvimento de sistemas onde todas as funcionalidades da aplicação estão em um único projeto, e sendo autônomo, não dependendo de requisição de dados externos, o que significa que todas as camadas, como interface do usuário, lógica de negócios e acesso ao banco de dados, estão presentes nesta mesma aplicação (GOS; ZABIEROWSKI, 2020). A Figura 2 apresenta

como é organizada as diferentes partes de um sistema nesta arquitetura, com a interface do usuário, lógica de negócio e interface de acesso a dados todos contidos em um único projeto, e acessando um único banco de dados.

Figura 2 – Arquitetura Monolítica.



Fonte: adaptado de (LENGA, 2019).

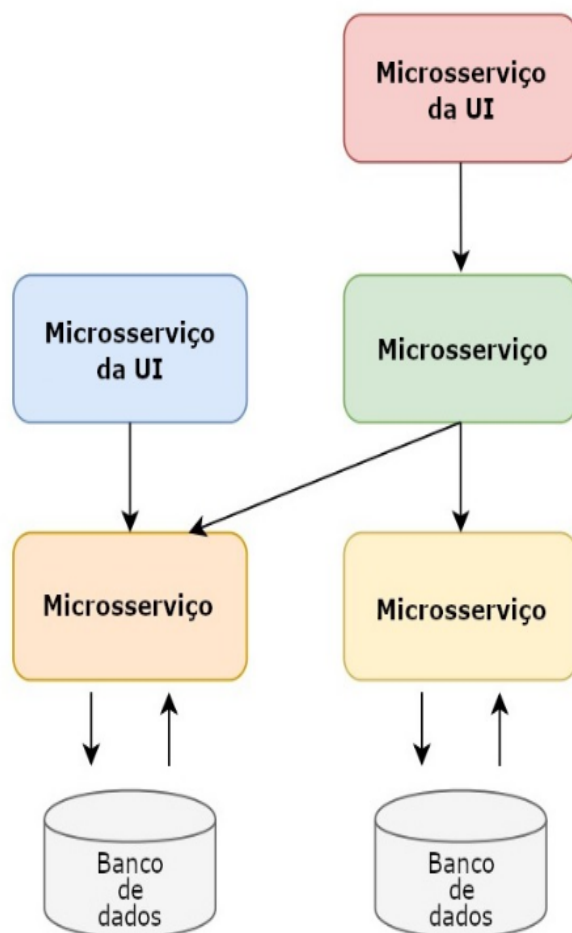
O surgimento da arquitetura monolítica está ligado ao início do desenvolvimento de software e é relacionado com a era da computação baseada em *mainframe* (grandes computadores que processava os dados em um único bloco), na metade do século XX (BADDULA, 2023). Desde então, sua abordagem vem sendo utilizada por ser uma forma simples de se desenvolver software.

Devido a sua estrutura, aplicações monolíticas tem a sua implementação em uma única unidade lógica, ou seja, uma vez pronta a aplicação irá se conter em um único projeto, que será hospedado em um ambiente onde a mesma será disponibilizada, e isso torna o sua implantação simplificada e de fácil abordagem (KALSKE; MÄKITALO; MIKKONEN, 2018). Porém, isso vem com algumas limitações. Escalar um sistema, seja em recursos do ambiente onde ele foi disponibilizado, ou expandir o software com mais funcionalidades ou adotar novas tecnologias e técnicas para dar continuidade a longo prazo no projeto, sem atrapalhar o restante do sistema, se torna uma tarefa complicada (RICHARDSON, 2023).

### 2.1.2 MICROSERVIÇOS

A arquitetura de microserviços é uma abordagem de desenvolvimento de uma única aplicação dividida em pequenos serviços especialistas, cada um com seus próprios processos e mecanismos de comunicação (FOWLER, 2014). Este modelo de desenvolvimento permite que esses serviços sejam criados para funcionalidades específicas, acessando seus bancos de dados próprios, com suas regras de negócio de acordo com o que for considerado na hora de construir o software.

Figura 3 – Arquitetura de Microserviços.



Fonte: adaptado de (LENGA, 2019).

A Figura 3 apresenta a utilização de diferentes microserviços para funcionalidades distintas, como interface do usuário, do inglês *user interface* (UI), e microserviços atuando no *backend*, com acesso aos seus próprios bancos de dados.

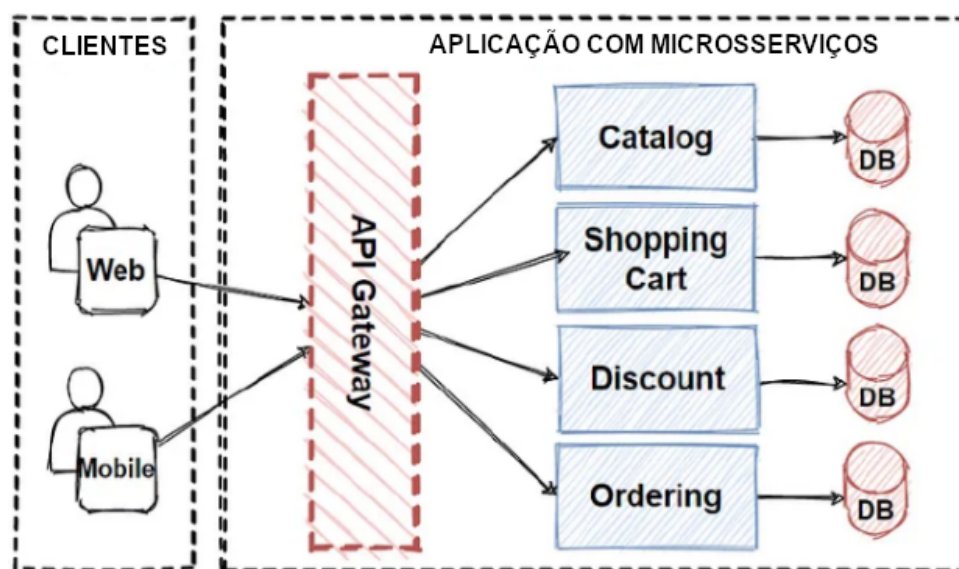
Como esta arquitetura se baseia em desenvolver os sistemas de forma modular separados, é necessário que ocorra a troca de dados entre eles quando necessário. Isso ocorre por meio de uma camada de abstração que utiliza algum protocolo de comunicação, como APIs com requisições HTTP (*Hypertext Transfer Protocol*) ou APIs REST (*Representational State Transfer*).

Como explica (NEWMAN, 2015), é possível desenvolver microsserviços com linguagens de programação distintas, utilizando o que for mais conveniente para o que a funcionalidade daquele serviço busca oferecer, e isso também é válido para banco de dados, uma vez serviços precisam ter seus bancos de dados próprios. Essa flexibilidade é utilizada como benefício ao manter um sistema, e ao considerar implementar novas tecnologias, com microsserviços é possível alterar somente o que é necessário, contribuindo com a fragmentação do sistema.

Também é característica desta arquitetura a utilização de recursos de disponibilidade. Como os microsserviços são implementados separadamente e, com a utilização de recursos de hospedagem, podem ter mais de uma instância disponível, garantindo experiência do usuário com um sistema que apresenta constante disponibilidade, apresentando tolerância a falhas e tratando os serviços de forma isolada para atualizações ou correções (NADAREISHVILI et al., 2016).

A gerência de microsserviços acontece por meio de uma série de ferramentas, serviços e técnicas que garantem o funcionamento do sistema, como na implementação de um *API Gateway* que é utilizado para gerenciar as requisições enviadas por diferentes clientes, e faz o direcionamento de acordo com o que foi solicitado, servindo como um ponto de entrada para a aplicação (ALSHUQAYRAN; ALI; EVANS, 2016).

Figura 4 – Aplicação com Microsserviços e API Gateway.



Fonte: adaptado de (OZKAYA, 2021).

A Figura 4 ilustra o *API Gateway* realizando o direcionamento de requisições vindas de um cliente *web* e outro cliente de dispositivo *mobile* (móvel), em um sistema desenvolvido com microsserviços, com cada microsserviço acessando o seu próprio banco de dados.

## 2.2 TESTES DE DESEMPENHO DE SISTEMA

Testes de desempenho são utilizados para avaliar como o sistema se comporta com diferentes cenários. Para este trabalho será utilizado o teste de linha de base e o teste de carga.

### 2.2.1 BASELINE TEST

O *baseline test*, ou teste de base de linha, é um teste que visa definir um ponto de referência para testes posteriores (no caso deste trabalho, o *load test*), e pode ser utilizado para diversos contextos ([MALIK; ADAMS; HASSAN, 2010](#)). No ambiente teste trabalho, como a métrica avaliada o desempenho de sistemas por meio de do registro de tempo de resposta de funcionalidades, o papel do *baseline test* será de apontar um valor médio do uso do sistema, para que o *load test* demonstre o seu funcionamento sob um volume maior de requisições para as funcionalidades avaliadas.

### 2.2.2 LOAD TEST

O *load test*, ou teste de carga, consiste em testar o sistema com um grande volume de requisições, simulando uma quantidade significativa de usuários virtuais, forçando o seu funcionamento além do convencional, com o objetivo de identificar pontos frágeis do sistema, possibilitando realizar melhorias para garantir o suporte a um grande número de usuários mantendo o seu desempenho ([VOKOLOS; WEYUKER, 1998](#)).

A utilização de um teste de desempenho, como o teste de carga, acontece por meio de planos de testes que contém as especificações sobre qual o escopo do teste, quais métricas serão avaliadas, quais parâmetros serão utilizados para a realização dos testes e como os dados gerados pelos testes serão tratados ([NEVEDROV, 2006](#)).

### 3 CENÁRIO DO EXPERIMENTO

Este capítulo irá apresentar como o experimento foi estruturado e realizado, expondo o como as arquiteturas avaliadas (monolítico e microsserviço) foram empregadas em cada aplicação desenvolvida. Para realização dos testes, foram desenvolvidos duas aplicações idênticas, chamadas de *Dining Room* (Sala de Jantar) com a temática de um sistema de comércio digital (*ecommerce*) voltada comercialização de eletrodomésticos.

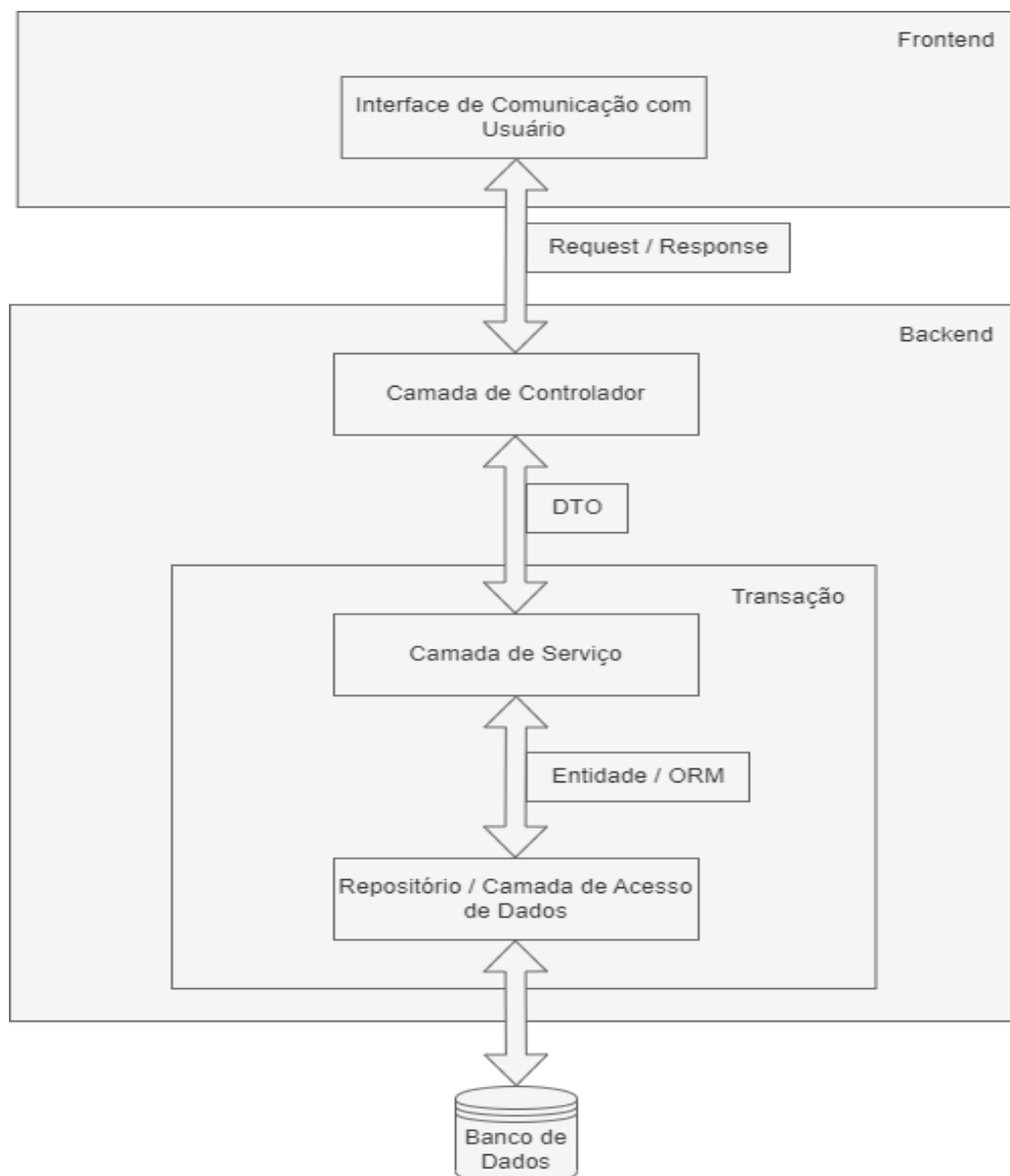
As aplicações contém seis entidades: “*Client*” contém o registro de usuários considerados clientes do sistema; “*Brand*” contém o registro de marcas dos produtos registrados no sistema; “*Product*” contém os produtos comercializados; “*Stock*” registra a quantidade que existe disponível em estoque para cada produto; “*ItemOrder*” possui um registro de produtos selecionados para uma compra; “*Orders*” contém uma compra, que possui um ou mais itens selecionados para compra dentro de si, além de uma entidade chamada “*PaymentMethod*” do tipo enumeração que contém métodos de pagamentos.

Todas as entidades possuem os métodos *Create*, *Read*, *Update* e *Delete* (CRUD), que são as ações de criação e manutenção básicas no desenvolvimento de uma aplicação. Além disso, a aplicação de regras de negócio a cada entidade determinam como o sistema deve funcionar, de acordo com cada funcionalidade sendo explorada.

Os sistemas desenvolvidos não possuem interface gráfica, de forma que a interação com as suas funcionalidades ocorre via ponto de chamada da aplicação (os chamados *endpoints*), com a ferramenta *Postman* sendo utilizada para este fim.

A seguir, a Figura 5 apresenta o padrão de projeto Objeto de Transferência de Dados (*Data Transfer Object*, sigla DTO), utilizado neste trabalho.

Figura 5 – Padrão de Projeto DTO.



Fonte: O autor.

A Figura 5 ilustra o DTO, com a sua utilidade de isolar o a camada de transação e o acesso ao banco de dados, por meio de um objeto do tipo entidade (*entity*) que está restrito apenas ao *backend*, e ao interagir com a interface de requisições do usuário, o objeto deve ser convertido em um DTO, para isolar o que chega do controlador e está em contato com o banco de dados.

O desenvolvimento da aplicação se utilizou um ambiente sem *frontend*, utilizando chamada via *endpoints* presentes no arquivo controlador (*Controller*) de cada entidade para realizar a comunicação entre o cliente e a aplicação, por meio da ferramenta *Postman* durante o desenvolvimento, e o mesmo endereço de acesso aos *endpoints* foi, com a conclusão do desenvolvimento do sistema, utilizado no Apache JMeter para a realização dos testes utilizados na análise deste trabalho.



Na camada de serviço (*Service*) está a aplicação de regra de negócio de entidade do sistema, recebendo um objeto do tipo DTO do controlador e, em caso de comunicação com o banco de dados, objetos do tipo entidade (*Entity*) são instanciados e, por meio de uma transação no banco de dados, para garantir a integridade dos dados (SOUSA, 2020). A interação com o banco de dados acontece por meio da camada de acesso a dados, na forma de uma interface de repositório (*Repository*).

## 3.1 MATERIAIS

Nesta seção serão apresentadas as tecnologias e ferramentas empregadas neste trabalho.

### 3.1.1 JAVA

Java é uma linguagem de programação de alto nível, orientada a objetos e amplamente utilizada em desenvolvimento de software. Java é notável por sua robustez, segurança e facilidade de manutenção, sendo comumente empregado no desenvolvimento de aplicativos de *desktop*, aplicativos móveis (*Android*), aplicativos *web*, sistemas distribuídos e servidores de aplicativos. Além disso, a plataforma Java oferece uma ampla gama de bibliotecas e *frameworks* que simplificam o desenvolvimento de software, tornando-a uma escolha popular em diversas áreas da computação (ORACLE, 2023a). Para este trabalho, foi utilizado o Java versão 17.

### 3.1.2 SPRING BOOT

O *Spring Boot* é um *framework* robusto e abrangente de desenvolvimento de aplicativos Java que oferece um conjunto de soluções para simplificar o desenvolvimento de software empresarial. Ele se baseia em princípios de inversão de controle e injeção de dependência para promover a modularidade, flexibilidade e capacidade de testes de aplicativos Java. O *Spring Boot* fornece recursos para gerenciamento de ciclo de vida de objetos, segurança, acesso a dados, integração com serviços *web* e muito mais, tornando-o uma escolha popular para desenvolvedores que desejam criar aplicativos escaláveis, eficientes e de fácil manutenção (IBM, 2023). Para este trabalho, foi utilizada a versão 3.2.

### 3.1.3 MySQL

O MySQL é um sistema de gerenciamento de banco de dados relacional de código aberto amplamente utilizado em todo o mundo. Desenvolvido originalmente pela MySQL AB (agora parte da *Oracle Corporation*), o MySQL é conhecido por sua confiabilidade, desempenho e facilidade de uso. Ele permite que os usuários armazenem, gerenciem e

recuperam dados de maneira eficiente, seguindo o modelo de banco de dados relacional, com suporte a SQL (*Structured Query Language*). O MySQL é utilizado em uma variedade de aplicativos, desde pequenos sites pessoais até grandes sistemas empresariais, tornando-se uma das opções mais populares para gerenciamento de dados em muitos cenários de desenvolvimento de software ([ORACLE, 2023b](#)). Para este trabalho, Foi utilizado o MySQL na versão 8.0.29.

### 3.1.4 POSTMAN

O Postman é uma plataforma de colaboração e desenvolvimento de API que oferece um conjunto de ferramentas para simplificar a criação, teste e documentação de APIs. Com uma interface de usuário amigável, o Postman permite que desenvolvedores criem solicitações HTTP, testem o comportamento das APIs, automatizem fluxos de trabalho de teste e compartilhe facilmente suas coleções de API com outros membros da equipe. É amplamente utilizado para depuração, monitoramento e validação de APIs, tornando-se uma ferramenta valiosa para desenvolvedores, testadores e equipes de desenvolvimento que trabalham com serviços *web* e APIs RESTful ([POSTMAN, 2023](#)). Para este trabalho, o Postman será utilizado na versão 10.20 para testar as funcionalidades do software, uma vez que não terá telas interativas no para uma usabilidade interativa.

### 3.1.5 APACHE JMETER

O Apache *JMeter* é uma ferramenta de código aberto amplamente utilizada para testar o desempenho de aplicativos da *web* e avaliar sua capacidade de lidar com carga e estresse sob diversas condições. O *JMeter* permite a criação de testes de carga, testes de estresse, e testes de desempenho, simulando o comportamento de múltiplos usuários simultâneos, monitorando métricas de desempenho e identificando gargalos ou problemas de escalabilidade em aplicações *web*, serviços *web* e servidores. Essa ferramenta é valiosa para desenvolvedores e engenheiros de qualidade que desejam garantir que seus aplicativos funcionem de forma confiável e eficiente em situações de alto tráfego ([APACHE... , 2023](#)). A versão utilizada neste trabalho é a 5.6.2.

### 3.1.6 AMAZON WEB SERVICES

A *Amazon Web Services* (AWS) é uma plataforma de computação em nuvem líder mundialmente, oferecida pela Amazon. Ela disponibiliza uma ampla gama de serviços de computação, armazenamento, banco de dados, análise, aprendizado de máquina e muito mais, que permitem às empresas e desenvolvedores executar suas aplicações e infraestrutura de TI na nuvem. A AWS é conhecida por sua escalabilidade, confiabilidade e flexibilidade, tornando-a uma escolha popular para organizações de todos os tamanhos, desde *startups*

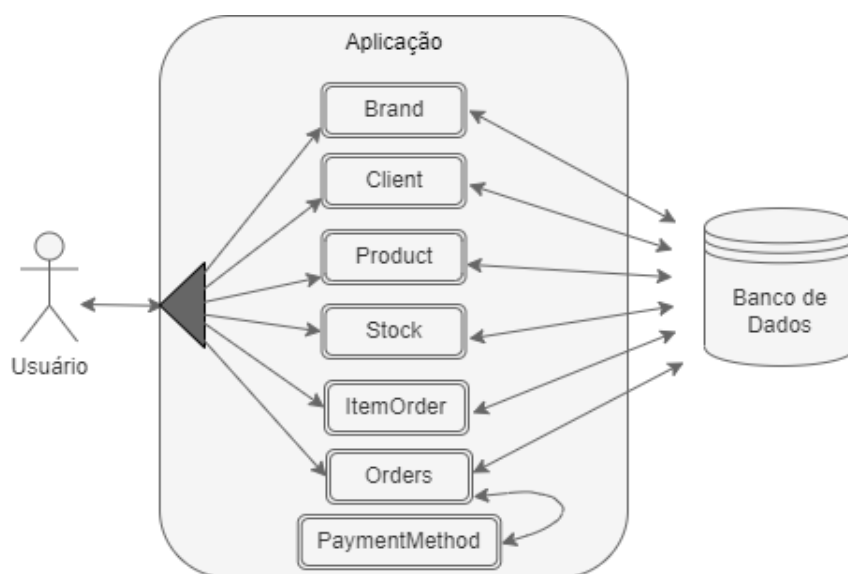
até corporações multinacionais, que desejam aproveitar os benefícios da computação em nuvem em seus negócios. (AMAZON..., 2023)

Para este trabalho, os serviços empregados foram o *Amazon Elastic Compute Cloud* (EC2) e o *Amazon Relational Database Service* (RDS). O EC2 oferece uma plataforma para realizar a implementação de software por meio de uma máquina virtual, e neste trabalho foi escolhido o sistema operacional *Amazon Linux AMI*. O RDS é um serviço de gerenciamento de banco de dados que pode ser relacionado a uma instância EC2, tornando possível a implementação de um software com banco de dados, e o Sistema de Gerenciamento de Banco de Dados (SGBD) foi o MySQL.

## 3.2 CENÁRIO DA ARQUITETURA MONOLÍTICA

Nesta seção será apresentado a aplicação desenvolvida com a arquitetura monolítica. A Figura 6 apresenta as entidades presentes no sistema.

Figura 6 – Arquitetura monolítica aplicada ao experimento.

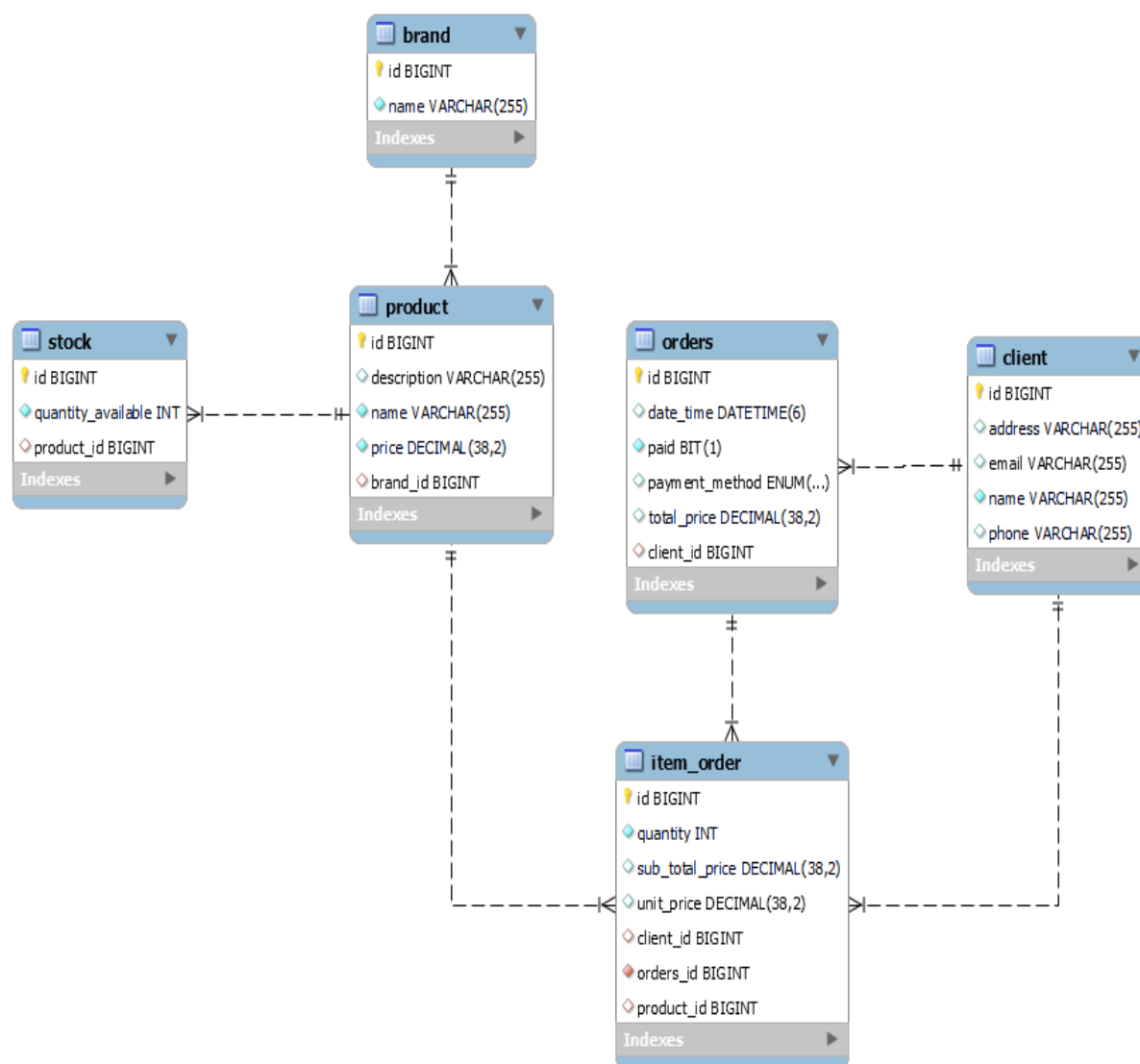


Fonte: O autor.

A Figura 6 apresenta as entidades "*Brand*", "*Client*", "*Product*", "*Stock*", "*ItemOrder*", "*Orders*" e "*PaymentMethod*" estão unidas em única uma aplicação, com apenas um banco de dados ligado a mesma. O usuário realiza as requisições diretamente para a aplicação, que está disponível em um único endereço, e acesso às entidades é direto, por meio de chamadas via *endpoints*.

O banco de dados, desenvolvido com o SGDB relacional MySQL, está apresentado na figura 7.

Figura 7 – Banco de dados da aplicação monolítica.



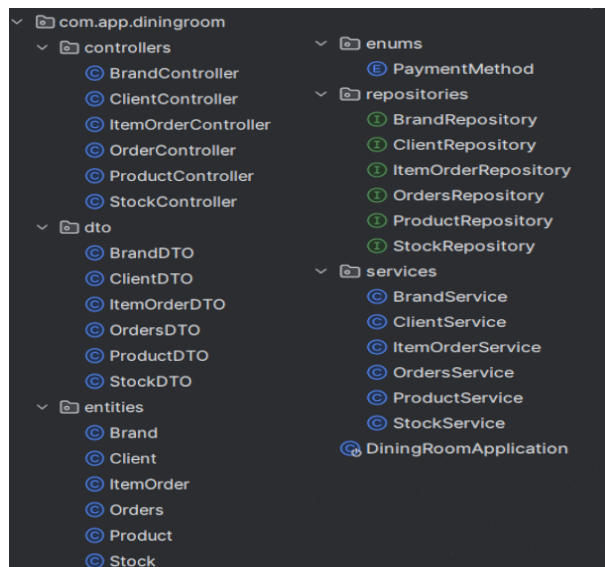
Fonte: O autor.

A Figura 7 apresenta o diagrama Entidade Relacionamento (ER) do banco de dados do sistema monolítico, com as suas ligações e colunas e cada tabela, apresentando a relação de chave primária e estrangeira de cada tabela. A entidade "PaymentMethod" não possui uma tabela no banco de dados pois é do tipo enumeração.

### 3.2.1 ESTRUTURA DE DIRETÓRIOS

A arquitetura utilizada no projeto define como a estrutura de diretórios será definida. A seguir, a Figura 8 ilustra a estrutura de diretórios do sistema monolítico.

Figura 8 – Estrutura de diretórios a partir do diretório app do sistema com arquitetura monolítica.



Fonte: O autor.

A Figura 8 apresenta a estrutura de diretórios do projeto do sistema com arquitetura monolítica, dentro do diretório *diningroom*, que contém o código-fonte da aplicação Java.

O código de cada *Controller*, com os *endpoints* de cada funcionalidade utilizada durante a análise deste trabalho estão disponíveis nos apêndices. O Código 1 apresenta o *Controller* da entidade *Client*, com as funcionalidades os *endpoints* necessários para criar um registro e atualizar o mesmo.

O Código 2 apresenta o *Controller* da entidade *Product*, também com os *endpoints* para criar e atualizar um registro. Os *endpoints* para as mesmas funcionalidades na entidade *Stock* estão apresentados no Código 3, e também para a entidade *ItemOrder*, como apresenta o Código 4.

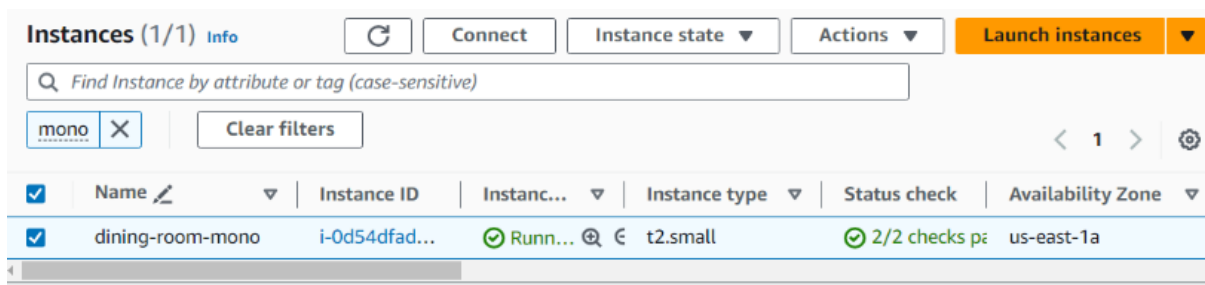
Por último, é apresentado no Código 5, com os *endpoints* necessários para criar um registro de pedido e para pagar um pedido, alterando o seu estado de "pedido em aberto" para "pago".

O sistema completo está disponível em um repositório na plataforma GitHub, no endereço <https://github.com/GustavoBonif/dining-room-mono>.

### 3.2.2 HOSPEDAGEM

Para a hospedagem da aplicação foi utilizada a plataforma em ambiente nuvem AWS, A Figura 9 apresenta a instância do sistema.

Figura 9 – Instância EC2 da aplicação monolítica.

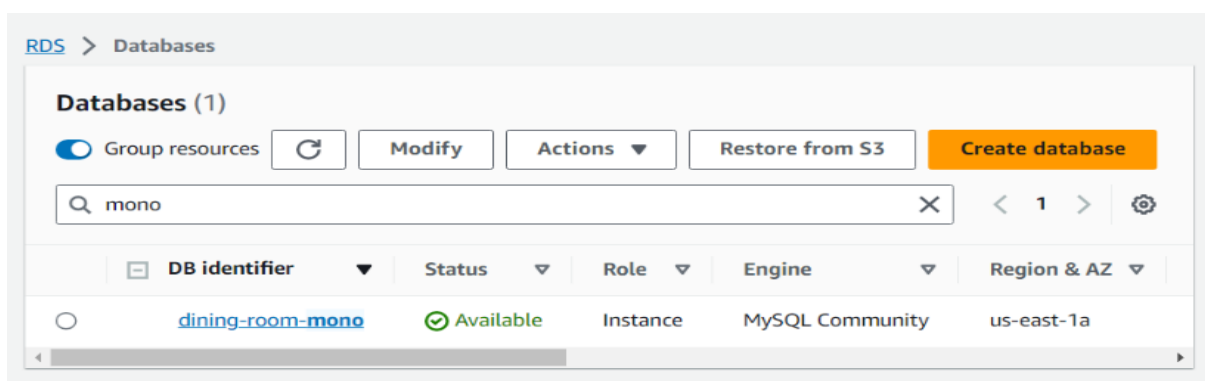


Fonte: O autor.

A Figura 9 apresenta instância *Elastic Compute Cloud* (EC2), com uma máquina do tipo *t2.small* criada para a hospedagem do sistema. A instância possui 1 CPU (3.3 GHz Intel Xeon) e com 2 GB de Memória RAM (AMAZON, 2023).

Para o banco de dados, é utilizado uma instância RDS, conforme é apresentado na Figura 10.

Figura 10 – RDS com o banco de dados utilizado na instância EC2 da aplicação monolítica.



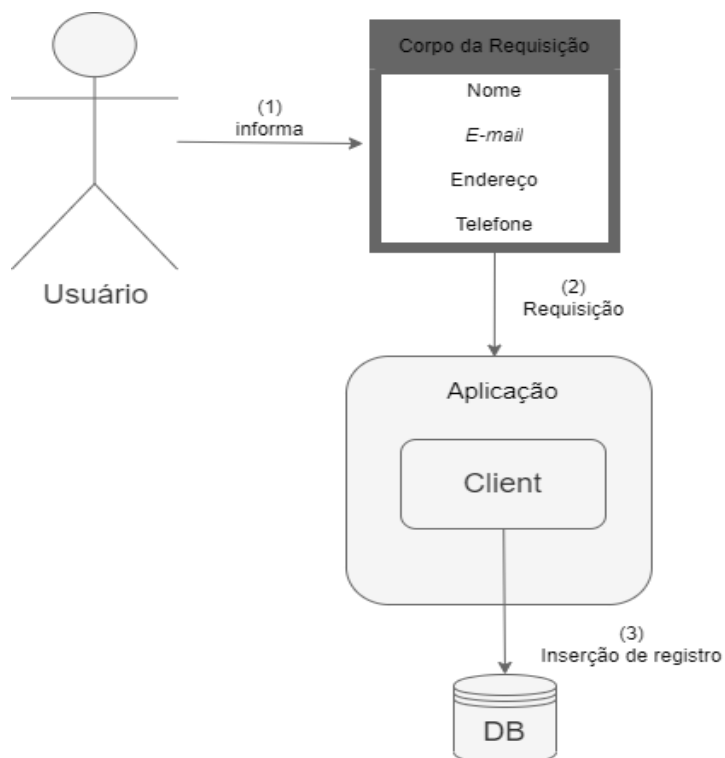
Fonte: O autor.

A Figura 10 apresenta uma instância de um banco de dados na AWS, disponível por meio do serviço *Relational Database Service* (RDS). A instância possui com 20 GB de armazenamento. Ambas a instância EC2 e o RDS estão na zona disponível pela AWS *us-east-1a*, localizada na zona leste dos Estados Unidos.

### 3.2.3 FUNCIONALIDADES

Para a realização dos testes, foram utilizadas as funcionalidades "criar cliente", "criar produto", "adicionar pedido ao carrinho" e "pagar pedido em aberto". A Figura 11 apresenta o funcionamento da ação "criar cliente".

Figura 11 – Diagrama de funcionamento da requisição Criar Cliente, no sistema com arquitetura monolítica.

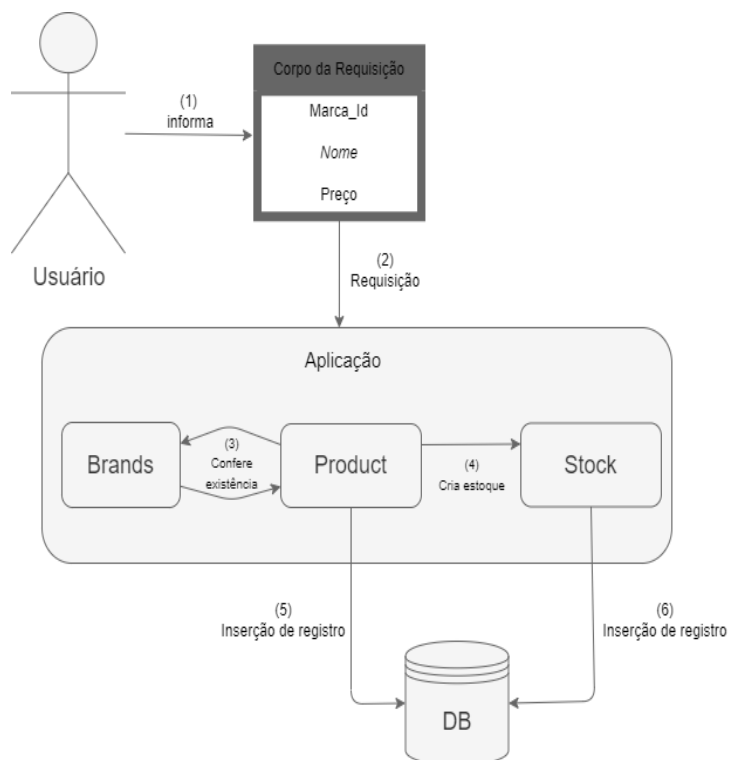


Fonte: O autor.

A funcionalidade "criar cliente" apresentada na Figura 11 consiste em uma ação restrita apenas a entidade “*Client*”, onde é informado um nome, *e-mail*, endereço e telefone (passo 1). A requisição é enviada direto para a aplicação (passo 2), acessando o serviço solicitado. Uma vez verificado a integridade dos dados no sistema monolítico, ocorre a criação do registro no banco de dados (passo 3), na tabela "Client".

A Figura 12 apresenta a ação "criar produto".

Figura 12 – Diagrama de funcionamento da requisição Criar Produto, no sistema com arquitetura monolítico.



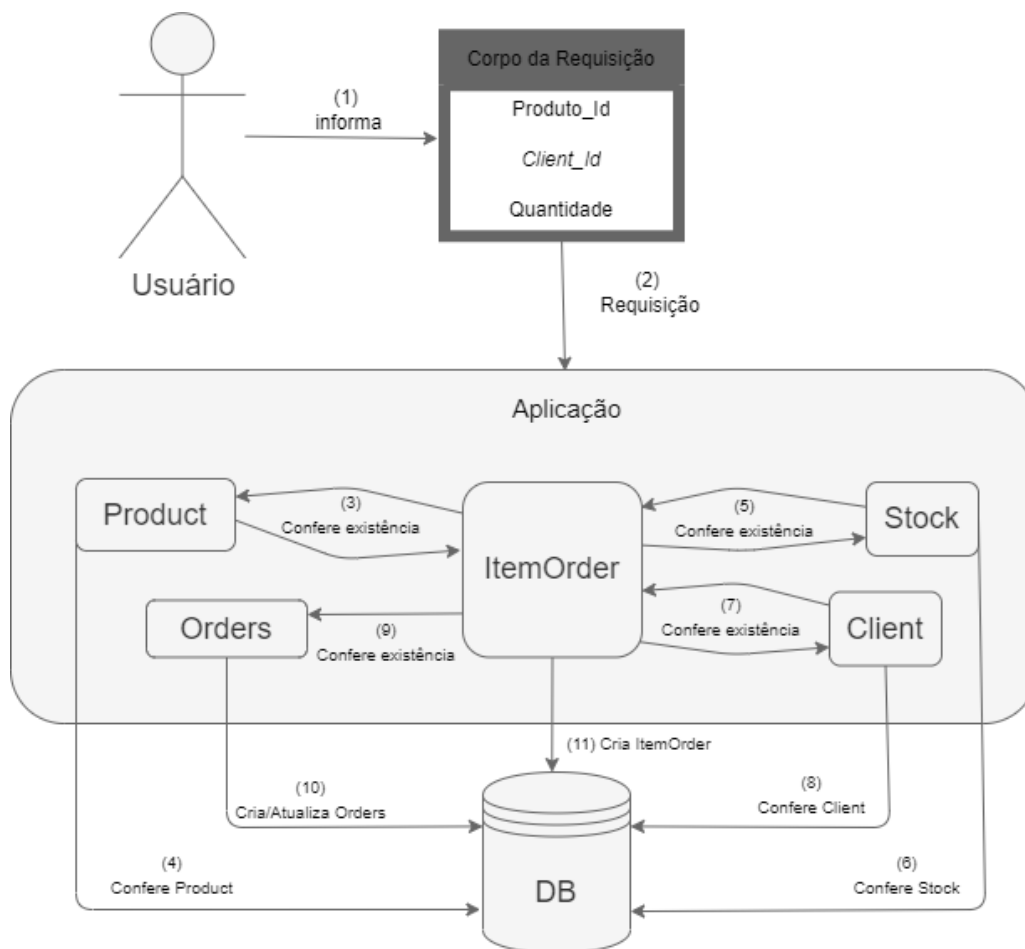
Fonte: O autor.

Ilustrado na Figura 12, a funcionalidade "criar produto" consiste em informar a marca a qual aquele produto pertence, um preço e um nome para o produto (passo 1). Os dados vão para a entidade "Product" (passo 2). Durante processo de criação, é verificado em "Brands" se a marca informada existe (passo 3), em seguida é realizada a inserção de um registro na entidade "Stock" (passo 4), que contém a quantidade disponível em estoque. Depois, o sistema irá salvar o novo registro de produto (passo 5), seguido da criação de registro do estoque (passo 6). Para salvar os registros, são utilizadas as tabelas "Product" e "Stock" do banco de dados.

A próxima funcionalidade é a "adicionar produto ao carrinho", apresentado na Figura 13.



Figura 13 – Diagrama de funcionamento da requisição Adiciona Produto ao Carrinho, no sistema com arquitetura monolítica.

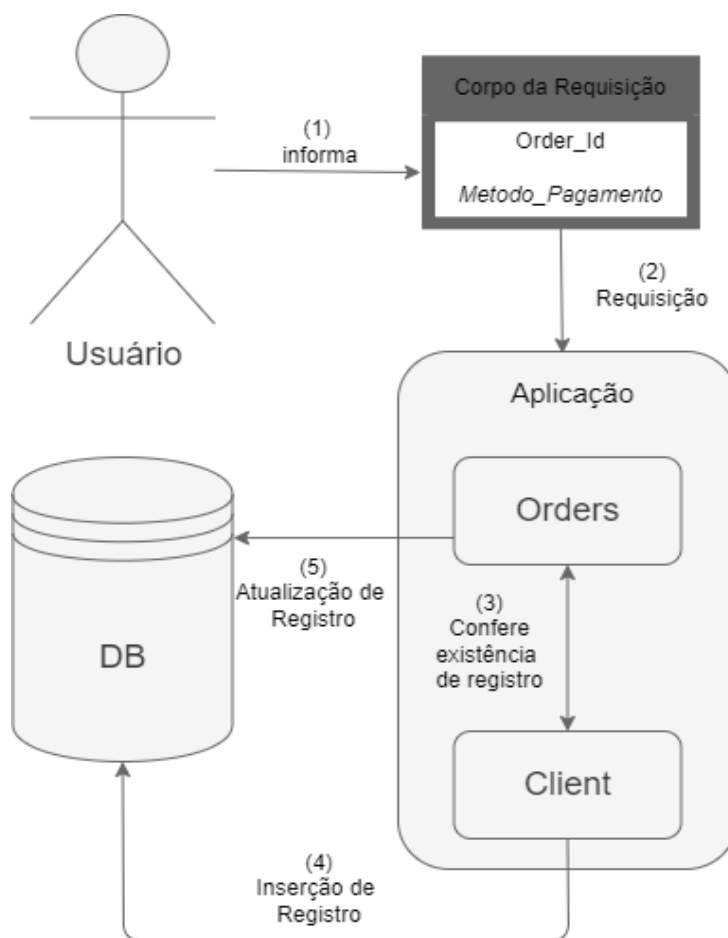


Fonte: O autor.

A Figura 13 apresenta a ação "adicionar produto ao carrinho", onde é informado um produto, um cliente e uma quantidade requerida para a compra (passo 1). A requisição é enviada para "ItemOrder" (passo 2) onde é tratada. É realizada uma verificação na entidade "Product" para atestar que o produto informado existe no sistema (passo 3) conferindo no banco de dados (passo 4), depois é feita uma verificação na entidade "Stock", para atestar que a quantidade requerida existe no estoque (passo 5) com conferência no banco de dados (passo 6), seguido de uma verificação em "Client" para atestar se o cliente informado existe (passo 7) conferindo no banco de dados (passo 8). O sistema realiza então uma verificação na entidade "Orders" a procura de um registro com o status de não pago para o cliente informado (passo 9), o que significa que está em aberto, e nesse caso adiciona o produto informado a este pedido. Caso não haja um pedido em aberto, será criado um novo pedido para o cliente informado (passo 10) e o produto é adicionado ao mesmo (passo 11).

Por último, a ação "pagar pedido em aberto", apresentada na Figura 14.

Figura 14 – Diagrama de funcionamento da requisição Pagar Pedido em Aberto, no sistema com arquitetura monolítica. Fonte: O autor



Fonte: O autor.

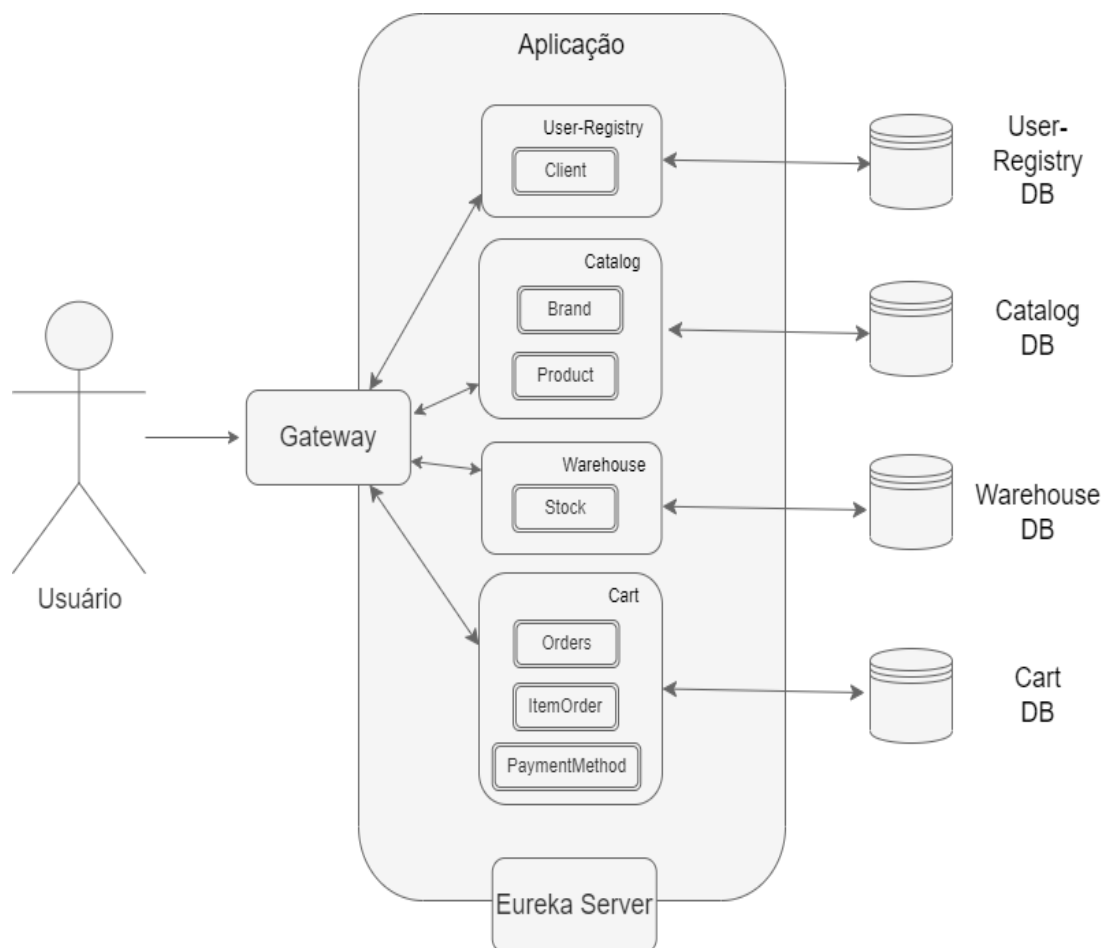
A funcionalidade "pagar pedido em aberto", apresentada na Figura 14 consiste em informar um pedido (registro de "Orders") e informar um método de pagamento (passo 1), por meio da entidade do tipo enumeração "PaymentMethod". O sistema verifica se o pedido informado está em aberto existe para o registro de "Client" presente na registro de "Orders" (passo 3), informado no corpo da requisição, e caso seja verdadeiro, registra o pedido pago em "Client" (passo 4), e depois retira o estado de em aberto do pedido e salva o registro em "Orders" (passo 5).

### 3.3 CENÁRIO DA ARQUITETURA DE MICROSERVIÇOS

Para a aplicação empregando a arquitetura de microserviços, foram criados quatro microserviços para as entidades: "Dining-Room-User-Registry", que contém a entidade "Client"; "Dining-Room-Catalog", com as entidade "Brand" e "Product"; "Dining-Room-Warehouse", que possui a entidade "Stock"; "Dining-Room-Cart", que contém a entidade "Orders" e "ItemOrder", além de duas entidades para a gestão dos microserviços descritos: "Dining-Room-Eureka-Server" que contém o servidor de mapeamento e descobrimento dos

microserviços, e “*Dining-Room-Gateway*”, que possui o *API Gateway* configurado para tratar e direcionar o acesso aos microserviços, de acordo com as requisições, conforme é apresentado na Figura 15.

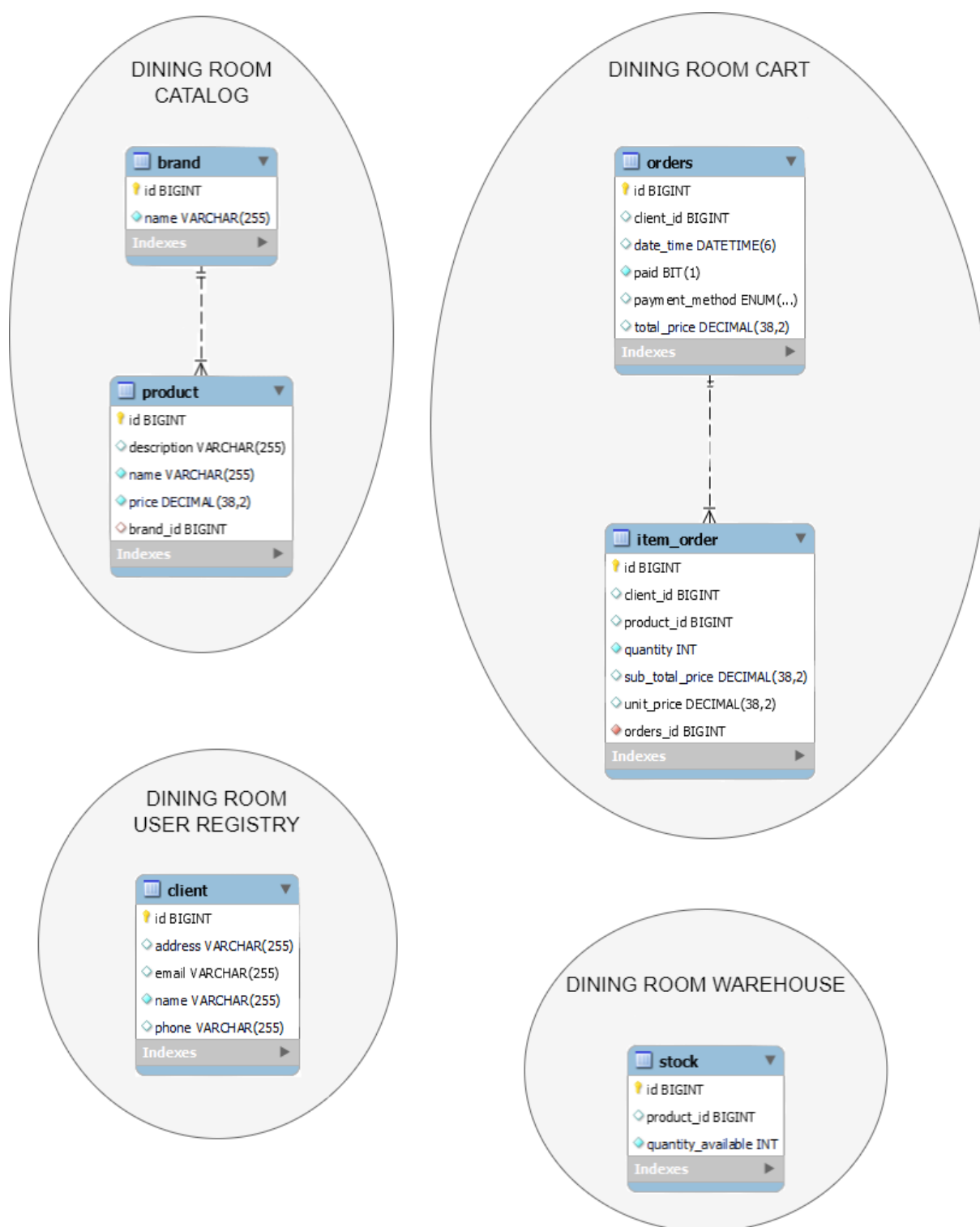
Figura 15 – Arquitetura de microserviços aplicada ao experimento.



Fonte: O autor.

A Figura 15 apresenta a aplicação da arquitetura de microserviços, apresentando como funciona a comunicação entre as requisições vindas do usuário, sendo gerenciadas pelo *gateway* de acordo com cada funcionalidade e, conseqüentemente, realizando a chamada para cada microserviço, que por sua vez possuem seus próprios banco de dados, apresentado no diagrama Entidade Relacionamento (ER) na Figura 16.

Figura 16 – Banco de dados da aplicação com microserviços.



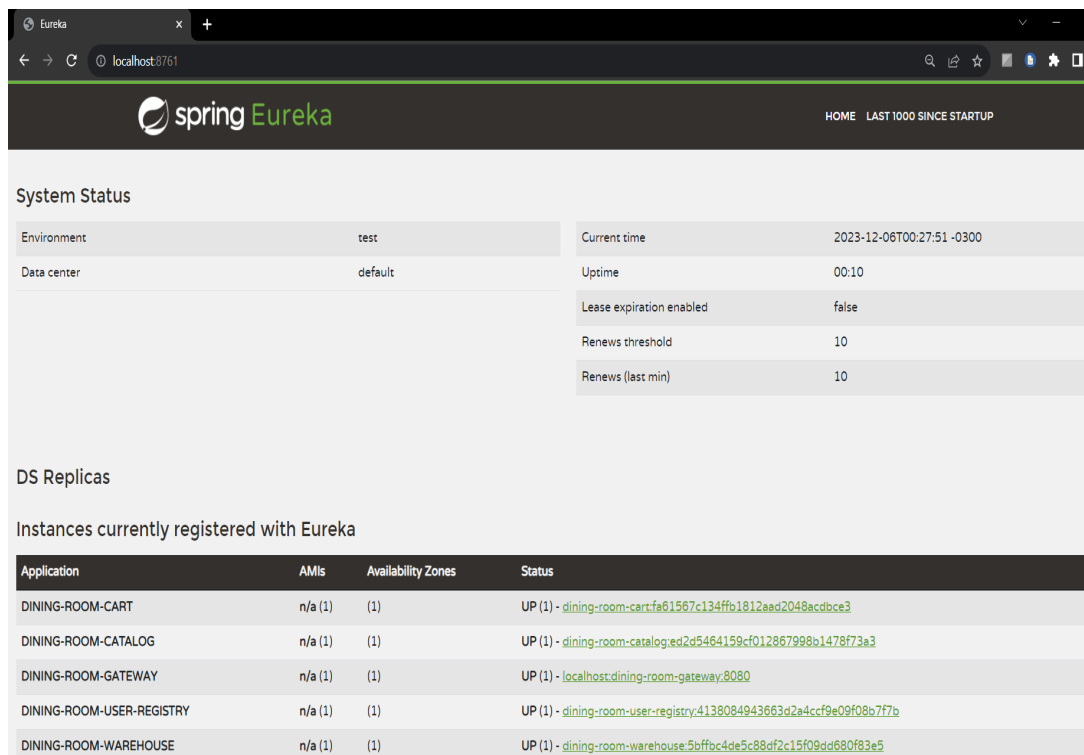
Fonte: O autor.

A Figura 16 apresenta banco de dados da aplicação, que foi dividido em um para cada microserviço, com exceção dos microserviços que não precisam de banco de dados, seguindo os princípios da arquitetura de microserviços.

Os microserviços foram criados com base no contexto de cada funcionalidade,

dedicando um banco de dados para cada um, utilizando o mesmo nome. Para que os microsserviços possam se comunicar, atuando como partes de um mesmo projeto, é necessário realizar o mapeamento de cada um. Para este trabalho, foi utilizado o serviço de mapeamento Eureka, apresentado na Figura 17.

Figura 17 – Servidor Eureka funcionando em um ambiente local.



The screenshot shows the Spring Eureka web interface. The browser address bar indicates the URL is localhost:8761. The page title is 'spring Eureka'. The navigation bar includes 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into sections: 'System Status', 'DS Replicas', and 'Instances currently registered with Eureka'.

**System Status**

Environment	test	Current time	2023-12-06T00:27:51 -0300
Data center	default	Uptime	00:10
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	10

**DS Replicas**

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status
DINING-ROOM-CART	n/a (1)	(1)	UP (1) - dining-room-cart:fa61567c134ffb1812aad2048acdbce3
DINING-ROOM-CATALOG	n/a (1)	(1)	UP (1) - dining-room-catalog:ed2d5464159cf012867998b1478f73a3
DINING-ROOM-GATEWAY	n/a (1)	(1)	UP (1) - localhost:dining-room-gateway:8080
DINING-ROOM-USER-REGISTRY	n/a (1)	(1)	UP (1) - dining-room-user-registry:4138084943663d2a4ccf9e09f08b7f7b
DINING-ROOM-WAREHOUSE	n/a (1)	(1)	UP (1) - dining-room-warehouse:5bfff0c4de5c88df2c15f09dd680f83e5

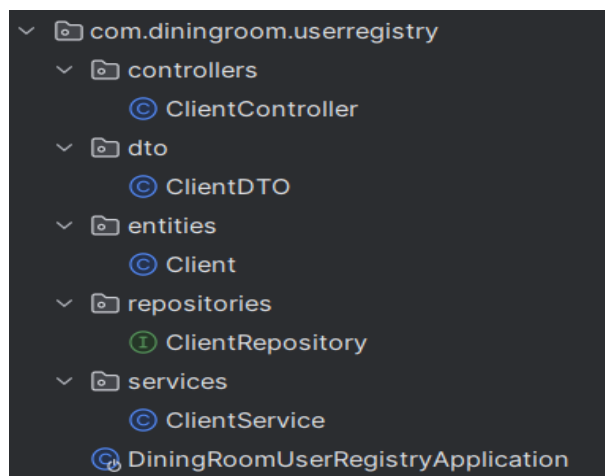
Fonte: O autor.

A Figura 17 apresenta a interface do Servidor *Eureka* em funcionamento em um computador local, fora do ambiente AWS, pois no ambiente AWS a sua instância, “*Dining-Room-Eureka-Server*”, está com acesso privado, seguindo boas práticas de configurações de segurança. O microsserviço “*Dining-Room-Gateway*” utiliza o *API Gateway* disponível também no *spring.cloud*. Um servidor Eureka, que faz parte do ecossistema *spring.cloud*, que é um conjunto de dependências para se utilizar microsserviços com o *spring*. O Eureka é um servidor utilizado para mapear os microsserviços que fazem parte da aplicação, que são registrados como *eureka-clients* (clientes-eureka).

### 3.3.1 ESTRUTURA DE DIRETÓRIOS

Como o uso da arquitetura de microsserviços consiste em projetos, a estrutura de diretórios será apresentada para cada microsserviço da aplicação. A Figura 18 apresenta a estrutura de diretórios do microsserviço *User-Registry*.

Figura 18 – Estrutura de diretórios a partir do diretório app do microserviço User-Registry no sistema com arquitetura de microserviços.



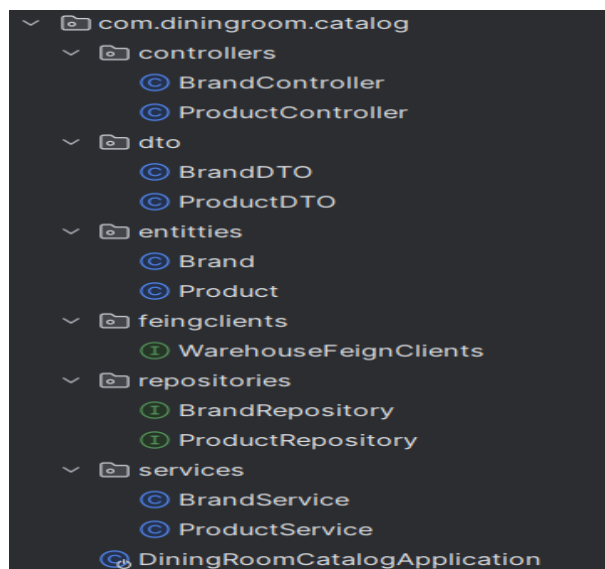
Fonte: O autor.

A Figura 18 apresenta a estrutura de diretórios do microserviço *User-Registry*, dentro do diretório da aplicação, *diningroom.userregistry*.

O código do arquivo controlador (*Controller*) está disponível nos apêndices, no Código 7, que apresenta os *endpoints* de criação e atualização do registro.

Para o microserviço *Catalog*, a sua estrutura de diretórios está apresentada na Figura 19.

Figura 19 – Estrutura de diretórios a partir do diretório app do microserviço Catalog no sistema com arquitetura de microserviços.



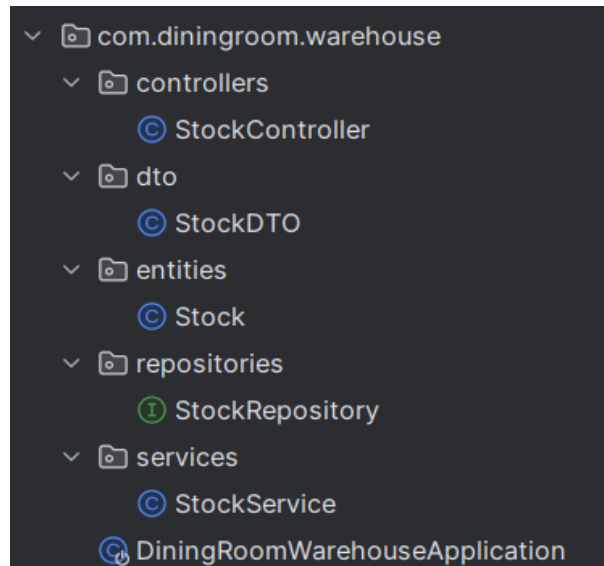
Fonte: O autor.

A Figura 19 apresenta a estrutura de diretórios do microserviço *Catalog*, dentro do diretório da aplicação, *diningroom.catalog*. o diretório *feingclients*, que contém a interface *WarehouseFeignClients*, utilizada para realizar requisições com o microserviço *Warehouse*.

O Código 8 apresenta o *Controller* da entidade *Product*, no microserviço *Catalog*, que contém os *endpoints* para a criação e atualização de produtos.

A Figura 20 apresenta a estrutura de diretórios do microserviço *Warehouse*.

Figura 20 – Estrutura de diretórios a partir do diretório app do microserviço Warehouse no sistema com arquitetura de microserviços.



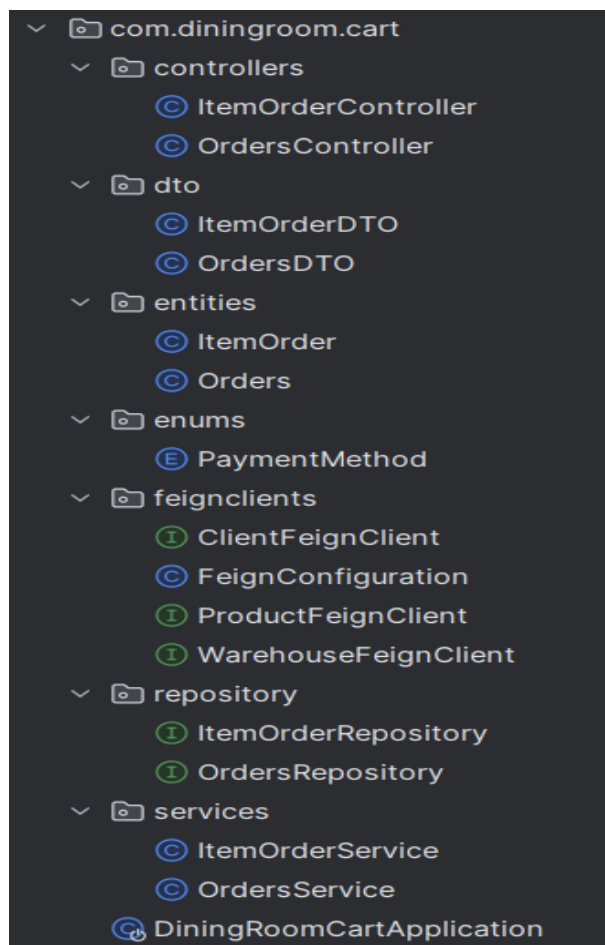
Fonte: O autor.

A Figura 20 apresenta a estrutura de diretórios do microserviço *User-Registry*, dentro do diretório da aplicação, *diningroom.warehouse*;

O Código 9 apresenta o *Controller* da entidade *Product*, no microserviço *Catalog*, que contém os *endpoints* para a criação e atualização de registros no estoque.

Por fim, a Figura 21 apresenta a estrutura de diretórios do microserviço *Cart*, dentro do diretório da aplicação, *diningroom.cart*.

Figura 21 – Estrutura de diretórios a partir do diretório app do microserviço Catalog no sistema com arquitetura de microserviços.



Fonte: O autor.

A estrutura do microserviço *Cart*, apresentado na Figura 21, contém duas entidades, *ItemOrder* e *Orders*, e um diretório *feignclients*, que contém as interfaces utilizadas para realizar as requisições com os demais microserviços do sistema, além do arquivo *FeignConfiguration*, onde são criadas configurações de tratamento das requisições.

O Código 10 apresenta o *Controller* da entidade *ItemProduct*, que contém os *endpoints* para a criação e atualização de item adicionados a um pedido. Os *endpoints* do *Controller* de *Orders* está apresentado no Código 11, com os *endpoints* de criação pedidos e mudança de estado, alterando a situação de "pedido em aberto" para "pago".

As aplicações desenvolvidas com a arquitetura de microserviços estão disponíveis na plataforma GitHub. Para isso, foram criados repositórios para cada um dos microserviços da aplicação, disponíveis em:

- Eureka-Server: <https://github.com/GustavoBonif/dining-room-eureka-server>
- Gateway: <https://github.com/GustavoBonif/dining-room-gateway>

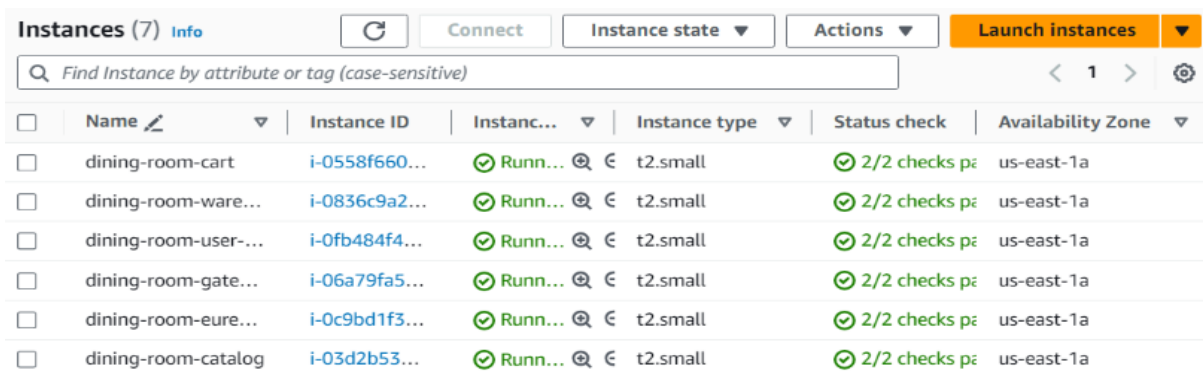


- User-Registry: <https://github.com/GustavoBonif/dining-room-user-registry>
- Catalog: <https://github.com/GustavoBonif/dining-room-catalog>
- Warehouse: <https://github.com/GustavoBonif/dining-room-warehouse>
- Cart: <https://github.com/GustavoBonif/dining-room-cart>

### 3.3.2 HOSPEDAGEM

A hospedagem a aplicação ocorreu em um ambiente de computação em nuvem na plataforma AWS. A Figura 22 apresenta as instâncias utilizadas no sistema.

Figura 22 – Instâncias EC2 da aplicação com microsserviços.



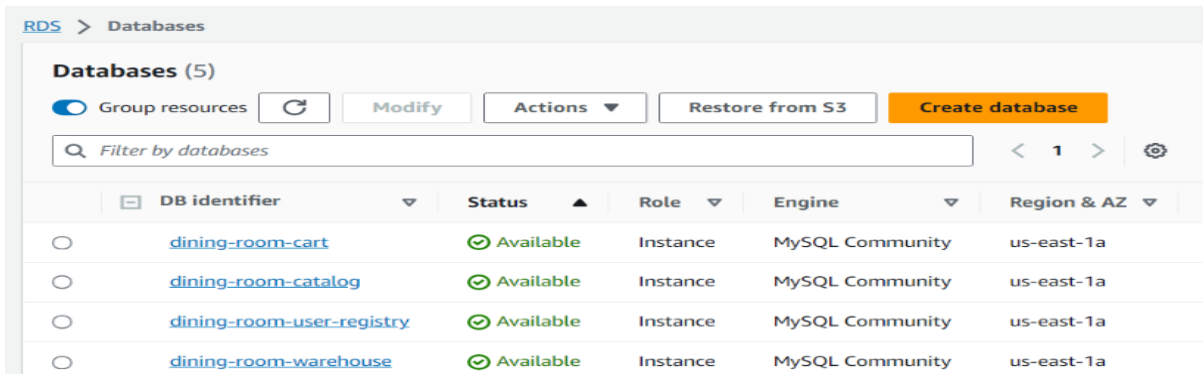
	Name	Instance ID	Instance state	Instance type	Status check	Availability Zone
<input type="checkbox"/>	dining-room-cart	i-0558f660...	Running	t2.small	2/2 checks passed	us-east-1a
<input type="checkbox"/>	dining-room-warehouse	i-0836c9a2...	Running	t2.small	2/2 checks passed	us-east-1a
<input type="checkbox"/>	dining-room-user-registry	i-0fb484f4...	Running	t2.small	2/2 checks passed	us-east-1a
<input type="checkbox"/>	dining-room-gateway	i-06a79fa5...	Running	t2.small	2/2 checks passed	us-east-1a
<input type="checkbox"/>	dining-room-eureka-server	i-0c9bd1f3...	Running	t2.small	2/2 checks passed	us-east-1a
<input type="checkbox"/>	dining-room-catalog	i-03d2b53...	Running	t2.small	2/2 checks passed	us-east-1a

Fonte: O autor.

A Figura 22 apresenta seis instâncias EC2 utilizadas na aplicação, cada uma com a mesma configuração que a instância utilizada na hospedagem da aplicação de arquitetura monolítica, ou seja, cada instância consiste em uma máquina do tipo *t2.small* criada para a hospedagem do sistema, com 1 CPU (3.3 GHz Intel Xeon) e com 2 GB de Memória RAM. Seguindo os conceitos da arquitetura, cada instância contém um microsserviço do sistema, sendo eles *Cart*, *Warehouse*, *User-Registry*, *Gateway*, *Eureka-Server* e *Catalog*.

Para o banco de dados, foi utilizado o serviço RDS, apresentado na Figura 23.

Figura 23 – RDS de cada instância EC2 na aplicação de arquitetura com microsserviços.



	DB identifier	Status	Role	Engine	Region & AZ
<input type="radio"/>	dining-room-cart	Available	Instance	MySQL Community	us-east-1a
<input type="radio"/>	dining-room-catalog	Available	Instance	MySQL Community	us-east-1a
<input type="radio"/>	dining-room-user-registry	Available	Instance	MySQL Community	us-east-1a
<input type="radio"/>	dining-room-warehouse	Available	Instance	MySQL Community	us-east-1a

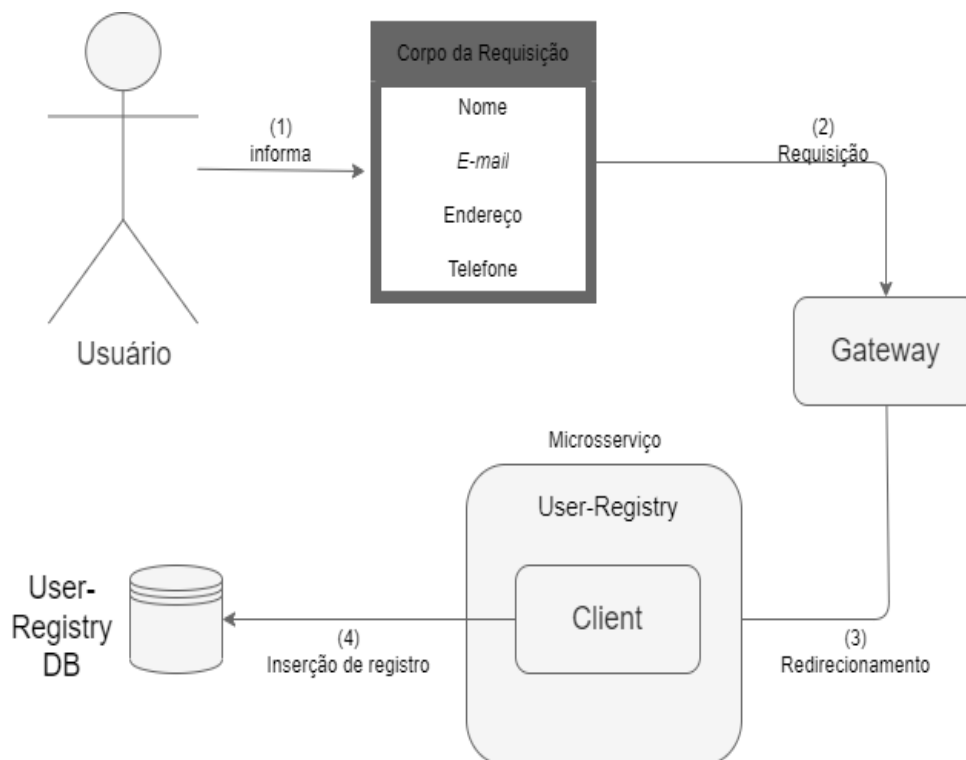
Fonte: O autor.

A Figura 23 apresenta quatro instâncias do serviço Relational Database Service (RDS) com 20 GB de armazenamento. As instâncias “*Dining-Room-Eureka-Server*” e “*Dining-Room-Gateway*” não precisam de banco de dados. Assim como no ambiente configurado para a aplicação monolítica, todas as instâncias EC2 e RDS estão na zona disponível pela AWS *us-east-1a*, localizada na zona leste dos Estados Unidos.

### 3.3.3 FUNCIONALIDADES

As funcionalidades estudadas durante os testes desta análise são; "criar cliente"; "criar produto"; "adicionar um produto ao carrinho"; "pagar pedido em aberto". A Figura 24 apresenta a primeira funcionalidade.

Figura 24 – Diagrama de funcionamento da requisição Criar Cliente, no sistema com arquitetura de microsserviços.

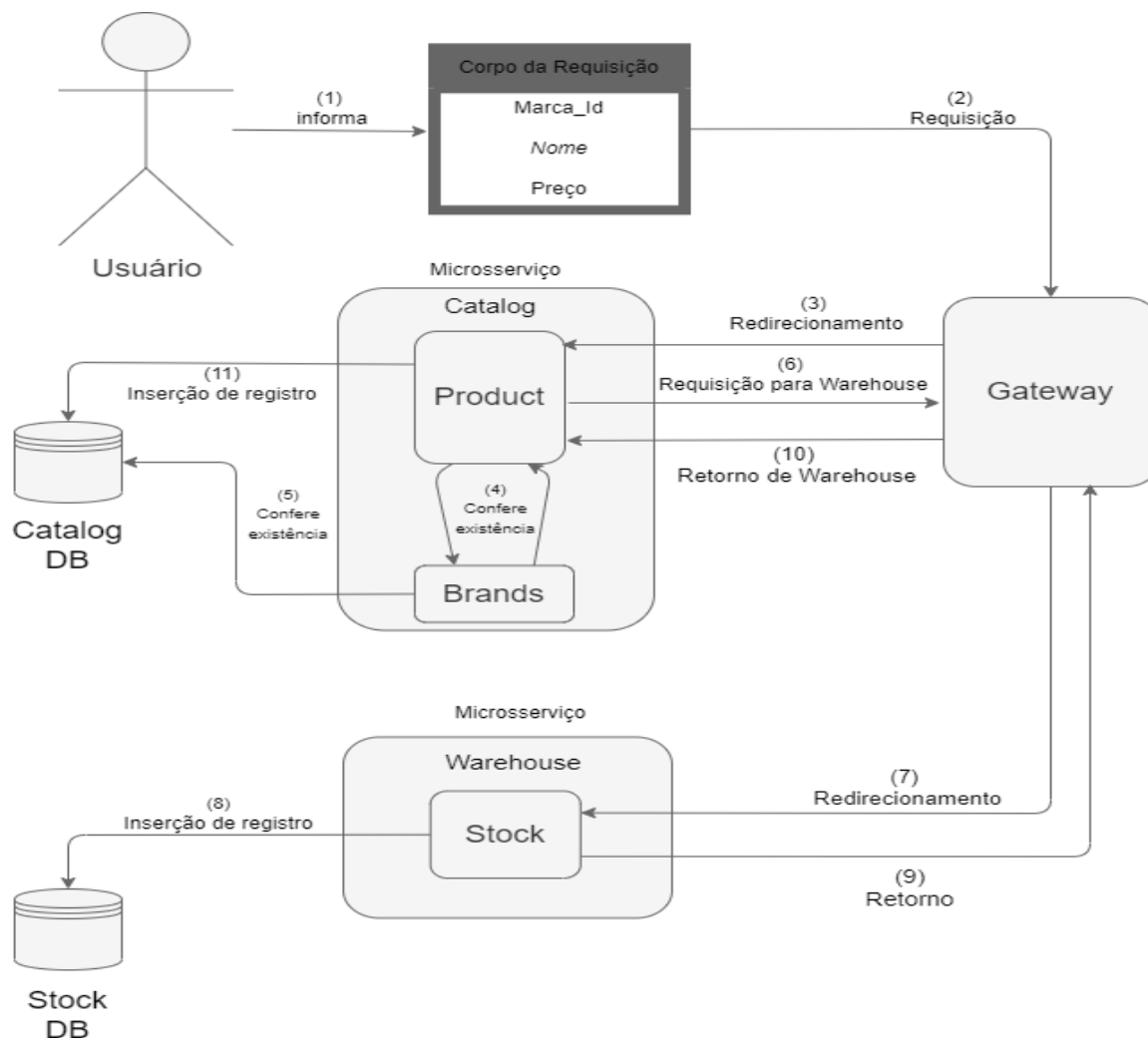


Fonte: O autor.

A Figura 24 ilustra a funcionalidade "criar cliente", ação restrita apenas a entidade "*Client*". Nela é informado um nome, *e-mail*, endereço e telefone (passo 1). O *gateway* recebe a requisição (passo 2) e realiza o redirecionamento da requisição vinda do usuário para o microsserviço "*User-Registry*" (passo 3), que processa a informação e realiza a inserção do registro no banco de dados do microsserviço (passo 4).

A próxima funcionalidade é a "cria produto", apresentado na Figura 25.

Figura 25 – Diagrama de funcionamento da requisição Criar Produto, no sistema com arquitetura de microsserviços.

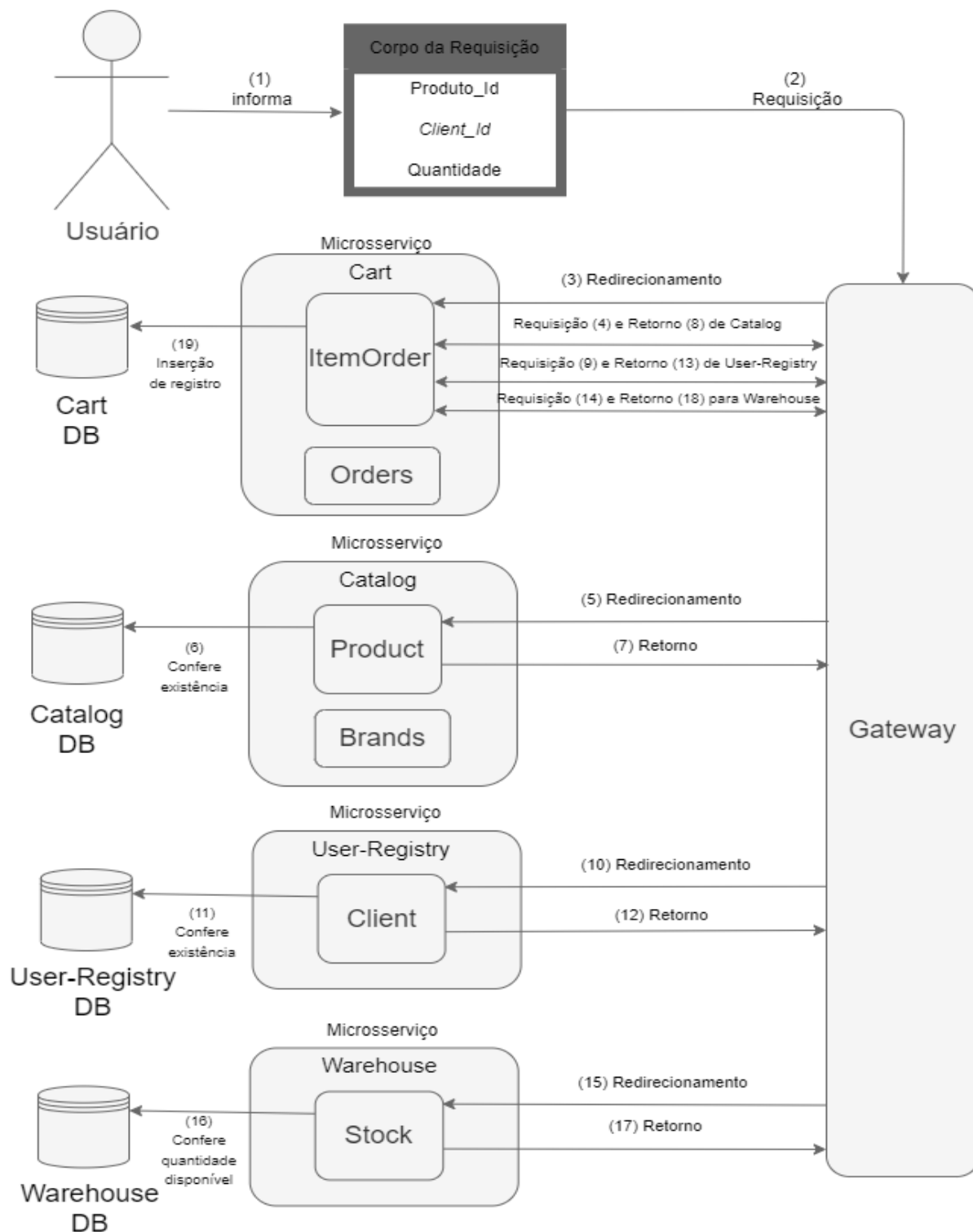


Fonte: O autor.

A Figura 25 apresenta a ação "criar produto", que consiste em informar a marca a qual aquele produto pertence, um preço e um nome para o produto (passo 1). O *gateway* recebe a requisição (passo 2) e redireciona a requisição para o microsserviço "*Catalog*" que possui os registros de produtos (passo 3). É realizada a verificação na entidade "*Brands*" para atestar registro de marca exites (passo 4) com conferência no banco de dados (passo 5). Depois, é realizada uma requisição para o microsserviço "*Warehouse*" (passo 6), passando pelo *gateway* que redireciona para "*Warehouse*" (passo 7) onde ocorre a inserção de um registro na entidade "*Stock*" (passo 8), que contém a quantidade disponível em estoque. Mediante a inserção de registro no estoque, ocorre o retorno (passo 9) para o *gateway*, que mais uma vez redireciona para "*Catalog*" (passo 10), onde o sistema irá salvar o novo registro de produto (passo 11).

A próxima funcionalidade é "adicionar produto ao carrinho", que é apresentada na Figura 26.

Figura 26 – Diagrama de funcionamento da requisição Adiciona Produto ao Carrinho, no sistema com arquitetura de microsserviços. Fonte: O autor



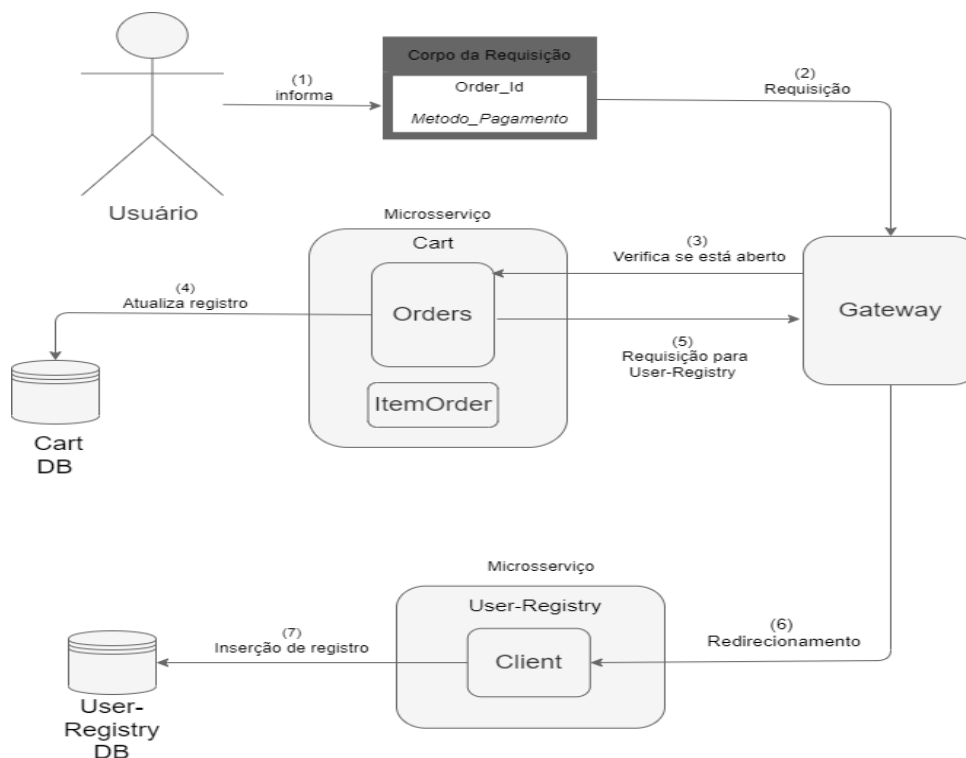
Fonte: O autor.

A Figura 26 apresenta a funcionalidade "adicionar um produto ao carrinho", sendo esta a ação que mais utiliza a comunicação entre os microsserviços. Para que a ação seja sucedida, é informado um produto, um cliente e uma quantidade requerida para a compra (passo 1) e encaminhada ao *gateway* (passo 2). O *gateway* redireciona para "Cart" (passo 3)

onde os dados são conferidos. É realizada uma requisição para "*Catalog*" (passo 4), onde após o redirecionamento pelo *gateway* (passo 5) e conferência se o produto existe no banco de dados de "*Catalog*" (passo 6), retorna para "*Cart*" (passo 7 e 8). Após isso é realizada uma nova requisição, dessa vez para "*User-Registry*" (passo 9 e 10), que verifica se o cliente informado existe no banco de dados (passo 11) e retorna a informação para "*Cart*" (passo 12 e 13). A última verificação acontece em "*Warehouse*", que depois de receber a requisição de "*Cart*" através do *gateway* (passo 14 e 15), confere no banco de dados se a quantidade requerida está disponível em estoque (passo 16), retornando a resposta para "*Cart*" (passo 17 e 18). Por fim, é realizada a criação do registro em "*ItemOrder*", no banco de dados de "*Cart*", (passo 19).

Por fim, a Figura 27 apresenta a ação "pagar pedido em aberto".

Figura 27 – Diagrama de funcionamento da requisição Pagar Pedido em Aberto, no sistema com arquitetura de microsserviços



Fonte: O autor.

A Figura 27 apresenta a funcionalidade "pagar pedido em aberto" consiste em informar um pedido, registro de "*Orders*", e informar um método de pagamento, por meio da entidade do tipo enumeração "*PaymentMethod*" (passo 1) que é redirecionado pelo *gateway* para "*Cart*" (passo 2 e 3), que confere os dados enviados e realiza uma atualização no pedido, tirando o *status* de pedido aberto (passo 4). Depois, é realizada uma requisição para "*User-Registry*" (passo 5 e 6), que insere registra o pedido pago em "*Client*" (passo 7).

## 4 TESTES

Nesta seção, será descrita a análise comparativa que será realizada com base em testes realizados entre os dois sistemas já apresentados, dentro do ambiente de hospedagem na plataforma AWS. Para a análise serão avaliadas duas métricas nos sistemas: testes de desempenho e consumo de recursos no ambiente em que os sistemas foram hospedados.

Para ambos os testes explorados nesta análise, foi realizada uma bateria de testes preliminares, chamado *baseline test*, ou teste de linha de base, que apresenta um ponto de referência da aplicação antes do *load test* ser realizado. Os dados coletados foram organizados de acordo com o contexto de cada teste, por meio da utilização de gráficos e tabelas de dados.

O modelo utilizado para realização dos testes segue o que é apresentado em (LUCIO, 2017), que serviu de inspiração para este trabalho e também (VOKOLOS; WEYUKER, 1998), que demonstra uma análise voltada para o desempenho de *software* de maneira mais detalhada.

### 4.1 TESTES DE DESEMPENHO

Durante a realização dos testes de desempenho, foi realizado um *baseline test* com cada funcionalidade, em cada aplicação, para determinar definir um ponto de referência do desempenho do sistema com as funcionalidades escolhidas, e depois foi realizado o *load test* para analisar o funcionamento do sistema durante uma quantidade excessiva de requisições. Os testes de desempenho aconteceram com o uso da ferramenta Apache Jmeter, com a utilização do sistema de planos de testes para realizar as medições necessárias. Um plano de testes consiste em definir um grupo de acesso, que pode conter um ou mais usuários virtuais, que realizam requisições HTTP, durante um determinado tempo.

O *baseline test* foi realizado com um plano de testes com 1000 usuários virtuais realizando uma única requisição por usuário. Para o *load test* foi utilizado um plano de testes com 3000 usuários virtuais realizando também uma única requisição cada. As requisições acontecem de forma síncrona.

Para registrar o tempo de resposta das requisições feitas durante o teste, foi utilizado o relatório agregado, que é um recurso do Apache Jmeter para apresentar alguns indicadores das requisições, como tempo de resposta médio da aplicação, taxa de erro e tempo mínimo e máximo de durante uma série de requisições. Para a análise demonstrada em gráficos, foram utilizados o tempo médio de resposta das requisições realizadas, com o seu resultado sendo apresentado em milissegundos.

Salientamos que nem todos os testes tiveram totalidade de sucesso durante suas execuções, porém esta informação faz parte dos testes, já que o motivo desta taxa de falha é o funcionamento de regras de negócio do próprio sistema.

O uso de gráficos foi escolhido para a melhor visualização dos dados recolhidos durante os testes, porém a visualização dos resultados adquiridos diretamente do Apache JMeter para o teste de desempenho estão disponíveis nos apêndices, apresentados nas Figuras 37, 38, 39 e 40.

A Figura 37 apresenta o relatório agregado, gerado pelo Apache JMeter para as requisições "Criar Produto", "Criar Cliente", "Adicionar Pedido ao Carrinho" e "Pagar Pedido" durante o *baseline test* realizados no sistema de arquitetura monolítica. A Figura 38 apresenta o relatório agregado, gerado pelo Apache JMeter para as requisições "Criar Produto", "Criar Cliente", "Adicionar Pedido ao Carrinho" e "Pagar Pedido" durante o *load test* realizados no sistema de arquitetura monolítica.

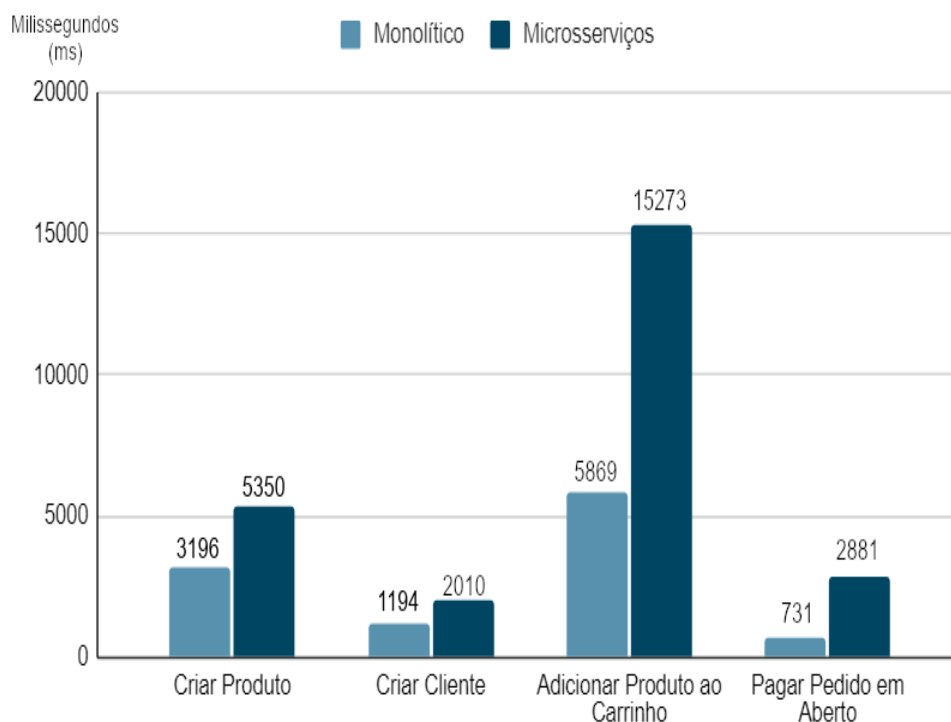
A Figura 39 apresenta o relatório agregado, gerado pelo Apache JMeter para as requisições "Criar Produto", "Criar Cliente", "Adicionar Pedido ao Carrinho" e "Pagar Pedido" durante o *baseline test* realizados no sistema de arquitetura de microsserviços. E por fim, a Figura 40 apresenta o relatório agregado, gerado pelo Apache JMeter para as requisições "Criar Produto", "Criar Cliente", "Adicionar Pedido ao Carrinho" e "Pagar Pedido" durante o *load test* realizados no sistema de arquitetura de microsserviços.

#### 4.1.1 BASELINE TEST

O teste de desempenho consiste em duas partes. A primeira parte é a realização do *baseline test*, que os resultados estão presentes no gráfico da Figura 28. O teste demonstrou uma diferença onde o tempo de resposta de todos os resultados do sistema que emprega a arquitetura de microsserviços foi superior.



Figura 28 – Gráfico comparativo entre as arquiteturas monolítica e com microserviço durante o baseline test para teste de desempenho.



Fonte: O autor.

A Figura 28 apresenta os seguintes resultados:

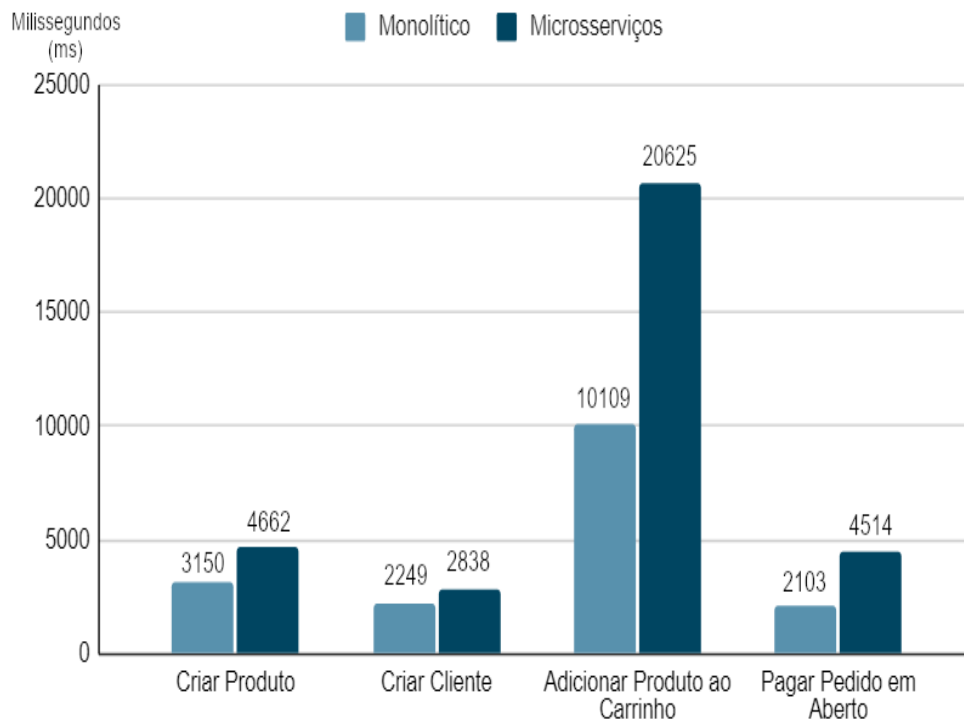
- Pagar Pedido em Aberto: houve uma diferença expressiva entre as arquiteturas, onde a aplicação com microserviços apresentou resultados 294% superior à aplicação monolítica;
- Adicionar Produto ao Carrinho: a funcionalidade apresentou também uma diferença significativa entre as aplicações, onde o sistema com arquitetura baseada em microserviços apresentou uma o tempo de resposta da requisição HTTP 160% superior a mesma funcionalidade no sistema com arquitetura monolítica;
- Criar Produto: o sistema com microserviços apresentou uma diferença de 67% a mais no tempo de resposta em comparação ao sistema monolítico;
- Criar cliente: apresentou uma diferença de 68% a mais no sistema com microserviços em comparação ao sistema monolítico.

Neste teste, foi possível observar que todos os resultados no sistema com arquitetura de microserviços foi superior ao de arquitetura monolítica, motivado pelo tempo de resposta que é necessário entre as comunicações dos microserviços, e quanto mais comunicações, maior o tempo de espera para a finalização da funcionalidade, como é evidenciado em "Adicionar Produto ao Carrinho".

### 4.1.2 LOAD TEST

Após definir pontos de referência com o *baseline test*, foi realizado o *load test*. O gráfico da Figura 29 apresenta os resultados obtidos. Apesar de ainda ter o tempo de resposta da aplicação com a arquitetura de microsserviços superior, como presente no *baseline test*, a diferença entre as arquiteturas é reduzida nas duas mais expressivas.

Figura 29 – Gráfico comparativo entre as arquiteturas monolítica e com microsserviço durante o load test para teste de desempenho.



Fonte: O autor.

Para a Figura 29 obtivemos os seguintes resultados:

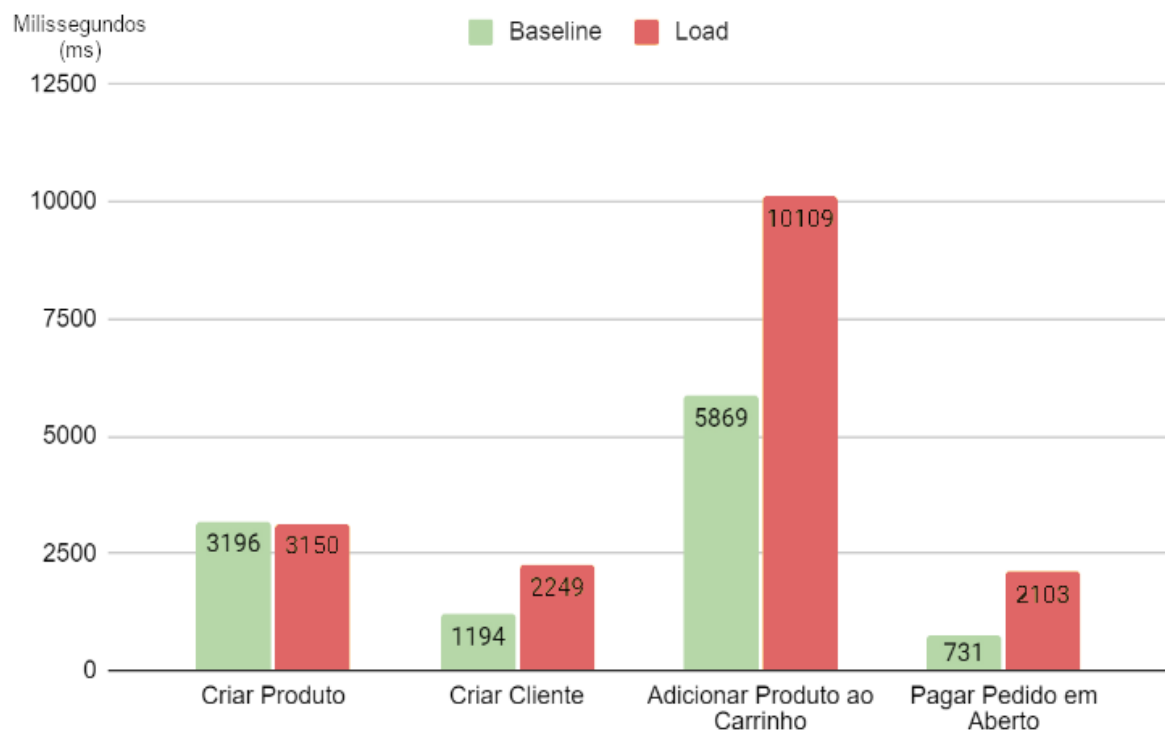
- Pagar Pedido em Aberto: no sistema com microsserviços apresentou uma diferença de 115% a mais do que no sistema monolítico;
- Adicionar Produto ao Carrinho: apresentou uma diferença de 104% a mais no tempo de resposta da requisição HTTP;
- Criar Produto: nesta funcionalidade, o sistema com microsserviços apresentou um resultado de 148% maior do que o sistema monolítico;
- Criar cliente: obteve 26% de diferença entre o tempo de resposta, com a arquitetura de microsserviços sendo a com o maior tempo.

Para o *load test*, o resultado apresenta o tempo de resposta sendo superior em todas as funcionalidades no sistema com microsserviços, assim como no *baseline test*. Porém a diferença de tempo de resposta entre as duas arquiteturas sendo reduzida, o que abre margem para ser explorado em trabalhos futuros que podem apontar se existe um ponto onde as duas arquiteturas apresentam resultados semelhantes.

### 4.1.3 RESULTADOS

Nesta sessão os dados obtidos durante o *baseline test* e o *load test* serão tratados de forma isolada para cada arquitetura, permitindo uma melhor visualização dos resultados obtidos. A Figura 30 apresenta os resultados para o sistema desenvolvido com arquitetura monolítica.

Figura 30 – Gráfico comparativo do sistema desenvolvido com arquitetura monolítica durante os testes de desempenho baseline e load.



Fonte: O autor.

Na Figura 30 podemos observar a seguinte diferença entre os testes:

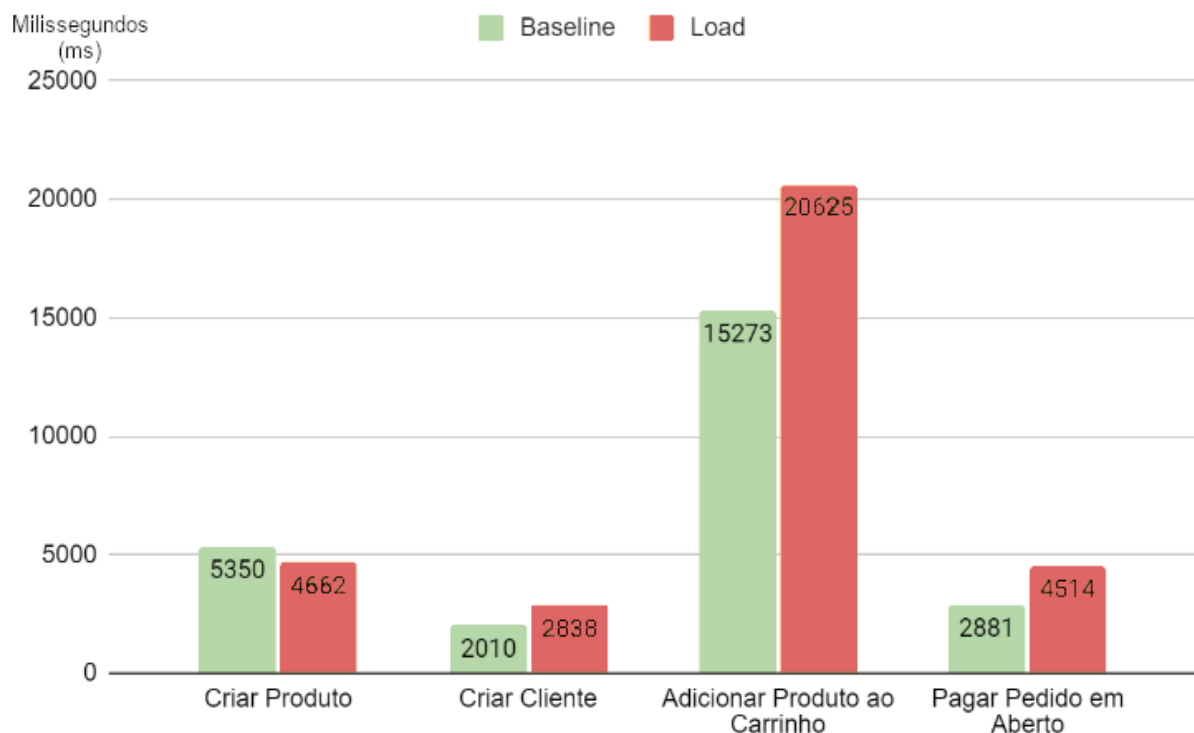
- Criar Produto: o resultado obtido durante o *baseline test* foi um 1% superior ao resultado do *load test*, sendo o único onde o *baseline test* foi superior;
- Criar Cliente: o valor durante o *load test* foi 1,88 vezes o valor obtido durante o *baseline test*;

- Adicionar Produto ao Carrinho: o tempo de resposta do *load balance* foi 1,72 vezes o tempo de resposta do *baseline test*;
- Pagar Pedido em Aberto: durante o *load test* foi apresentado um valor 2,88 vezes o resultado do *baseline test*, sendo a maior diferença proporcional entre os testes.

A diferença obtida entre o *baseline test* e o *load test*, nos mostra como o sistema se comporta conforme o número de requisições aumenta, ilustrando como a interações entre as entidades, mesmo em um sistema monolítico, requer maior tempo para que a requisição seja concluída. Com exceção de "Criar Produto", onde os valores foram similares, as demais funcionalidades tiveram os seus resultados elevados durante o *load test*, o que serve para apontar onde o sistema apresenta possíveis atrasos no funcionamento. Em um eventual trabalho futuro utilizando este trabalho como base, é possível utilizar estes dados para testar pontos onde o sistema falha, contribuindo para a qualidade do software.

O mesmo modo de apresentar os dados pode ser visto para o sistema com arquitetura baseada em microsserviços, com apresentar a Figura 31.

Figura 31 – Gráfico comparativo do sistema desenvolvido com arquitetura de microsserviços durante os testes de desempenho baseline e load.



Fonte: O autor.

De acordo com o gráfico da Figura 31, é possível apresentar as seguintes afirmações:

- Criar Produto: o *baseline test*, assim como no teste do sistema monolítico, foi o único que apresentou o resultado superior ao *load test*, com 15% a mais no valor;

- Criar Cliente: o valor durante o *load test* foi 1,41 vezes o valor obtido durante o *baseline test*;
- Adicionar Produto ao Carrinho: o tempo de resposta do *load balance* foi 1,35 vezes o tempo de resposta do *baseline test*;
- Pagar Pedido em Aberto: durante o *load test* foi apresentado um valor 1,57 vezes o resultado do *baseline test*, também apresentando a maior diferença proporcional entre os testes, assim como nos testes do sistema monolítico.

O sistema com arquitetura de microsserviços apresentou, assim como no sistema monolítico, resultado superior durante o *load test* em todas as funcionalidades com exceção de "Criar Produto". Porém, a diferença entre os testes é menor do que na arquitetura monolítica e, assim como comentado na análise do *load test* em ambas as arquiteturas juntas, esta informação é possível ser explorada em trabalhos futuros para determinar um eventual ponto onde a arquitetura de microsserviços apresenta estabilidade, independente do número de usuários realizando requisições ao sistema. Porém, no modelo estudado neste trabalho, os resultados em ambos os testes foram superiores aos mesmos testes com o sistema de arquitetura monolítica, devido ao tempo de resposta necessário nas comunicações entre os microsserviços presentes nas funcionalidades exploradas.

## 4.2 TESTE DE CONSUMO DE RECURSOS

Para a análise de consumo de recursos, foi realizado o monitoramento de utilização da CPU em cada instância EC2 na plataforma AWS, durante as requisições para as funcionalidades analisadas no durante o teste de desempenho. A interface de monitoramento de instâncias EC2 proporciona gráficos para acompanhar o consumo de recursos, que estão disponíveis nas figuras dos apêndices deste trabalho. Para uma melhor visualização e organização das informações, os dados serão apresentados neste capítulo no formato de tabelas. Vale lembrar que para o *baseline test* foram utilizados 1000 usuários virtuais realizando uma requisição cada um, e no *load test* são utilizados 3000 usuários virtuais, também realizando uma requisição cada um.

É importante salientar que nas tabelas existem células que não possuem dados, sendo estas preenchidas com um hífen (-). Isso acontece pois nem todas as funcionalidade interagem com todos os microsserviços do sistema.

Também é necessário evidenciar que o microsserviço “*Dining-Room-Eureka-Server*” não foi utilizado na análise, pois o seu funcionamento é para mapear os diferentes microsserviços presentes na aplicação, mas não participa da utilização direta da mesma.

### 4.2.1 BASELINE TEST

Durante o *baseline test*, apresentado na tabela da Figura 32, o sistema monolítico demonstrou uma constante no consumo da CPU, variando entre 7,98% e 13,7%. enquanto que no sistema com microsserviços, houve uma variação maior, justificado pelo tempo de resposta necessário para ocorrer a troca de dados entre das diferentes entidades. Isto é visível ao compararmos as diferentes funcionalidades.

Figura 32 – Tabela do consumo da CPU das arquiteturas monolítica e com microsserviços, durante o load test.

	Monolítico	Gateway	User Registry	Catalog	Warehouse	Cart
Criar Produto	13,7%	9,13%	-	15,5%	10,9%	-
Criar Cliente	7,98%	6,68%	8,45%	-	-	-
Adicionar Produto ao Carrinho	11,5%	8,14%	14,1%	23,4%	13%	41,8%
Pagar Pedido	11,1%	7,52%	-	-	-	11%

Fonte: O autor.

Observando a tabela da Figura 32, é obtemos as seguintes afirmações:

- Criar Produto: A ação apresentou uma variação pequena, e vale lembrar que ela utiliza mais de uma entidade, pois ao criar um produto, o sistema cria um registro no estoque, na entidade “*Stock*”, para o produto recém criado. A requisição consumiu 13,7% no sistema monolítico e, no sistema com microsserviços, 9,13% no *gateway*, 15,5% em “*Catalog*”, onde está a entidade “*Product*”, e 10,9% em “*Warehouse*”, onde está a entidade “*Stock*”;
- Na funcionalidade, em que a ação é restrita apenas a entidade “*Client*”, no sistema monolítico obteve um 7,98% de consumo da CPU, enquanto que no sistema com microsserviços, o *gateway* registrou 6,68%, para realizar a comunicação com a vinda via requisição HTTP e a entidade, esta que está presente no microsserviço “*User-Registry*”, e apresentou 8,45% de consumo da CPU;
- Pagar Pedido em Aberto: outra ação restrita a uma só entidade, que neste caso é “*Cart*”, e obteve 11,1% no sistema monolítico e no sistema com microsserviços, 7,52% no *gateway* e 11% em “*Cart*”;
- Adicionar Produto ao Carrinho: nesta funcionalidade é apresentada a maior diferença, pois nela ocorre a comunicação com todas as outras entidades. No sistema monolítico, a variação foi de 11,5%, se mantendo dentro do padrão das demais. Porém, no sistema com arquitetura de microsserviços, ocorreu o consumo elevado em diferentes microsserviços. O *gateway* apresentou um consumo de 8,14%, “*Warehouse*” apresentou

13% e “*User-Registry*”, 14,1%. O destaque está para os outros microserviços, que como precisam esperar os demais serviços responderem, ficaram com o consumo mais elevado. “*Catalog*” apresentou 23,4% de consumo, e “*Cart*” apresentou 41,8%, devido a sua ação principal na funcionalidade, onde a chamada dos outros microserviços ocorre por ele.

Os resultados obtidos são justificados devido ao tempo de resposta dos diferentes microserviços presentes em cada funcionalidade, o que necessita de uma comunicação via requisições HTTP, o que exige que cada processo fique em funcionamento por um tempo a mais do que a arquitetura monolítica, que não depende destes mecanismos de comunicação para consultar outras entidades no sistema.

#### 4.2.2 LOAD TEST

Durante o *load test*, o sistema monolítico não apresentou a mesma constante nos resultados como apresentou o *baseline test*, porém a forma como os microserviços responderam às diferentes funcionalidades foi a mesma, como demonstra o gráfico da Figura 33.

Figura 33 – Tabela do consumo da CPU das arquiteturas monolítica e com microserviços, durante o load test.

	Monolítico	Gateway	User Registry	Catalog	Warehouse	Cart
Criar Produto	12,6%	7,81%	-	17,9%	12,4%	-
Criar Cliente	10,5%	8,2%	10,5%	-	-	-
Adicionar Produto ao Carrinho	35,3%	12,6%	31,1%	49%	32,9%	73,1%
Pagar Pedido	7,9%	12,1%	-	-	-	16,3%

Fonte: O autor.

De acordo com a tabela da Figura 32, temos os seguintes resultados:

- Criar Produto: aqui o o sistema monolítico apresentou 12,6% de consumo da CPU, menor do que os 13,7% durante o *baseline test*. No sistema com microserviços, na mesma funcionalidade, o *gateway* apresentou 7,81% de uso da CPU, também menor que a mesma análise durante o *baseline test*, que foi de 9,13%, Porém, o consumo nos microserviços “*Catalog*” e “*Warehouse*” foram superior, com os valores de 17,9% e 12,4% respectivamente;
- Criar Cliente: Para esta ação, o sistema monolítico apresentou 10,5% de consumo da CPU, sendo 2,6% menor do que na análise anterior. Porém no sistema com

microsserviços, foram obtidos os valores 8,2% para o *gateway*, e 10,5% para o “*User-Registry*”, ambos superiores aos dados obtidos no *baseline test*.

- Em “pagar pedido” a aplicação monolítica apresentou 7,9%, valor menor do que os 11,1% presentes no *baseline test*. Enquanto que na aplicação com microsserviços, o *gateway* apresentou 12,1% e “*Cart*”, 16,3%, ambos valores maiores do que no teste anterior.
- Adicionar Produto ao Carrinho: apresentou no sistema monolítico 35,3% de consumo da CPU, valor 207% maior do que no *baseline test*. Também ocorreu o aumento no sistema com microsserviços, com o *gateway* apresentando 12,6%, “*User-Registry*” apresentou 31,1%, “*Catalog*” apresentou 49%, “*Warehouse*” obteve 49% e “*Cart*”, registrou 74,1% de consumo da CPU.

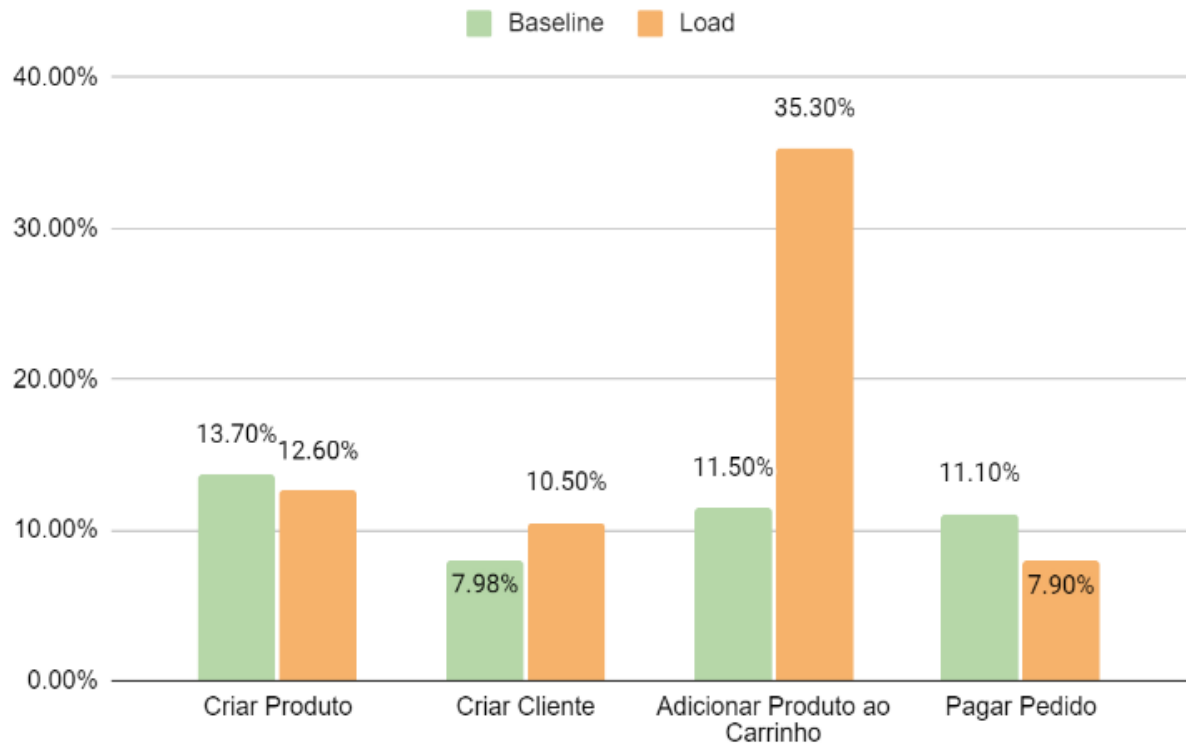
Com os resultados obtidos durante o *load test*, é fica ainda mais evidente o que é apontado na análise dos resultados do *baseline test*, onde é pontado que na arquitetura de microsserviços é necessário um tempo para que os diferentes microsserviços se comuniquem, via requisições HTTP, causando consumo elevado de processamento da CPU, algo que não acontece na arquitetura monolítica. É possível observar também como esse consumo de recurso é elevado com o *load test* por conta da quantidade elevada de usuários realizando requisições, o que abre espaço para, em um eventual trabalho futuro em cima dos dados apresentados aqui, tratar de informações como escalabilidade e disponibilidade do sistema.

### 4.2.3 RESULTADOS

Assim como na apresentação de resultados dos testes de desempenho, aqui serão apresentados os dados obtidos durante o *baseline test* e o *load test* de forma isolada para cada arquitetura. A Figura 34 apresenta um gráfico comparativo do consumo da CPU no *baseline test* e no *load test*, no sistema de arquitetura monolítica.



Figura 34 – Gráfico comparativo do consumo da CPU no sistema de arquitetura monolítica durante o baseline test e o load test.



Fonte: O autor.

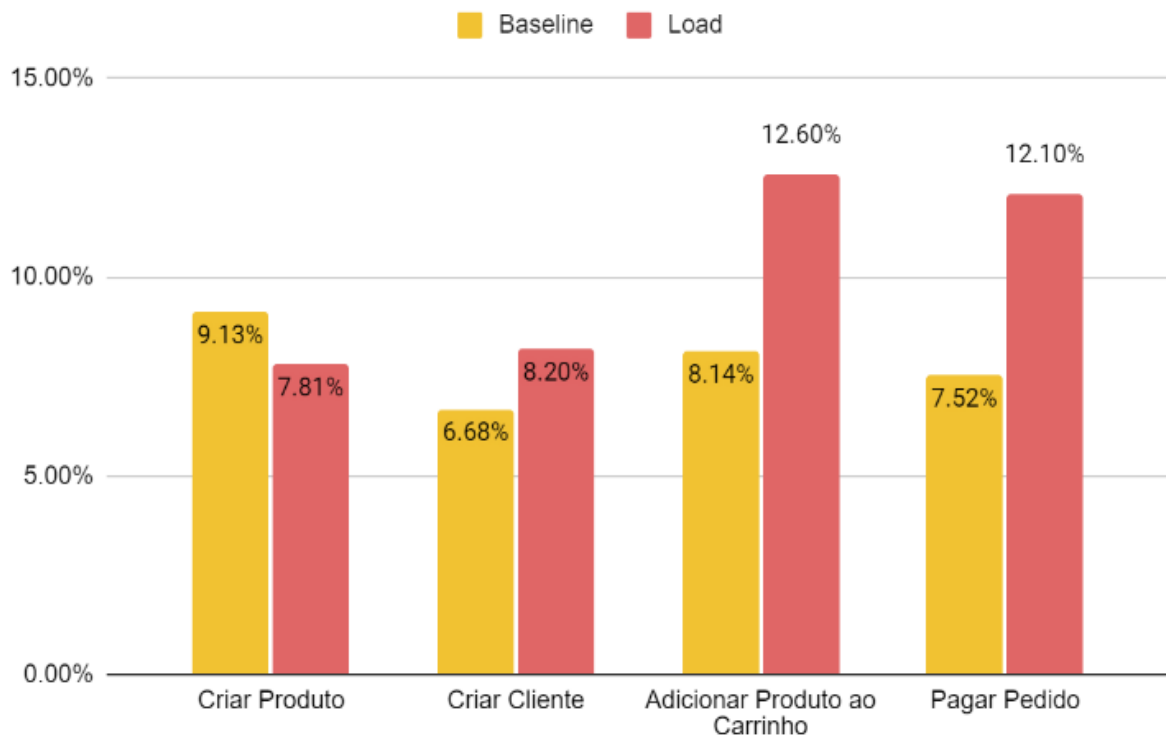
De acordo com o gráfico da Figura 34, temos os seguintes resultados:

- Criar Produto: o *baseline test* apresentou valor 9% superior ao *load test*;
- Criar Cliente: no *load test* o valor foi 1,32 vezes o resultado do *baseline test*;
- Adicionar Produto ao Carrinho: o *load test* obteve resultado 3,07 o valor do *baseline test*, sendo a maior diferença entre as funcionalidades testadas;
- Pagar Pedido: o *baseline test* apresentou 1,41 vezes o valor do *load test*.

A diferença entre os testes, na arquitetura monolítica, apresenta similaridade nos resultados, com exceção da funcionalidade "Adicionar Produto ao Carrinho". Isso demonstra como o consumo de recurso ocorre em uma funcionalidade que demande maior interação entre entidades, mesmo que essas entidades estejam no mesmo sistema.

Agora serão apresentados os resultados para os testes no sistema desenvolvido com a arquitetura de microsserviços. Primeiro, será apresentado o microsserviço do "gateway" que está presente em todas as funcionalidades devido ao seu papel de redirecionamento de requisições. Os dados estão na Figura 35.

Figura 35 – Gráfico comparativo do consumo de CPU no microserviço Gateway durante o baseline test e o load test.



Fonte: O autor.

Com o gráfico apresentado na Figura 35, obtemos as seguintes conclusões:

- Criar Produto: o *baseline test* apresentou valor 17% superior ao *load test*;
- Criar Cliente: no *load test* o valor foi 1,23 vezes o resultado do *baseline test*;
- Adicionar Produto ao Carrinho: o *load test* obteve resultado 1,55 vezes o valor do *baseline test*, sendo a maior diferença entre as funcionalidades testadas;
- Pagar Pedido: o *load test* apresentou 1,61 vezes o valor do *baseline test*.

Como microserviços com o *API gateway* está presente em todas as funcionalidades, o seu consumo pode ser registrado em todos os testes, porém, como a sua função é realizar o redirecionamento de cada requisição, sem necessitar de consulta com banco de dados e processamento de regras de negócio específicas de cada microserviço, seu consumo de CPU é baixo, com 9,13% sendo o maior resultado durante o *baseline test* e 12,60% durante o *load test*.

Para a apresentação dos resultados dos demais microserviços, foi realizado o agrupamento dos gráficos na mesma figura, para facilitar a visualização dos resultados. Os gráficos estão expostos na Figura 36.

Figura 36 – Gráficos comparativos do consumo de CPU nos microsserviços User-Registry, Catalog, Warehouse e Cart, durante o baseline test e o load test. Fonte: O autor.



Fonte: O autor.

Seguindo o agrupamento por microsserviços apresentado na Figura 36, temos os seguintes resultados:

- *User-Registry*:
  - Criar Cliente: o *load test* apresentou valor 24% superior ao *baseline test*;
  - Adicionar Produto ao Carrinho: o *load test* obteve resultado 2,25 vezes o valor do *baseline test*.
- *Warehouse*:
  - Criar Produto: *load test* apresentou 14% a mais do que o *baseline test*
  - Adicionar Produto ao Carrinho: o *load test* apresentou valor 2,58 vezes o resultado do *baseline test*.
- *Cart*:
  - Adicionar Produto ao Carrinho: o *load test* apresentou valor 1,75 vezes o resultado do *baseline test*, com 73,10% do consumo da CPU, o maior resultado em todos os testes;
  - Pagar Pedido: o *load test* excedeu o *baseline test* em 0,45% do seu valor.

- *Catalog*:
  - Criar Produto: o *load test* seu resultado foi 1,15 vezes o valor do *baseline test*;
  - Adicionar Produto ao Carrinho: *load test* apresentou 205% do resultado do *baseline test*.

Com as quatro funcionalidades sendo apresentadas juntas, é possível visualizar bem como que ocorre a evolução do consumo de CPU do *baseline test* para o *load test*, com algumas funcionalidade que não necessitam de interações com outra entidade, como "Criar Cliente" apresentando uma constante, exemplificado no microserviço "*User-Registry*". Enquanto isso, funcionalidades que necessitam da comunicação entre entidades, e, por consequência da arquitetura, precisam se comunicar com outros microserviços, o consumo de CPU é elevado, por conta do tempo em que cada mecanismo de comunicação precisa para realizar as requisições HTTP entre esses microserviços, como é apresentado no "*Cart*", com a funcionalidade "Adicionar Produto ao Carrinho".

## 4.3 ANÁLISE DOS RESULTADOS

Com base nos dados apresentados, ficou evidente como a comunicação entre os microserviços pode vir a ser prejudicial na utilização do sistema. Como existe a troca constante de dados constante, um microserviço precisa que todas as chamadas dele para outras entidades retornem valor, e o tempo de resposta que existe nessas chamadas geram grandes períodos no aguardo de uma ação, como foi exemplificado em na funcionalidade “adicionar produto ao carrinho”, que obteve 2,60 vezes o tempo de resposta do sistema com arquitetura de microserviços durante o *baseline test*, e 2,04 vezes o valor durante o *load test*, e esta diferença continua em todos os outras ações apresentadas.

Para o consumo da CPU também foram registrados valores maiores no sistema utilizando arquitetura com microserviços em comparação ao sistema de arquitetura monolítica. Isso acontece por conta do tempo necessário para que a comunicação entre os microserviços ocorra, sempre passando pelo *gateway* antes de acessar um microserviço, e o mesmo retorna os dados da requisição para o *gateway* que redireciona para o microserviço que fez a requisição, fato que foi apresentado com o *load test* na funcionalidade “adicionar produto ao carrinho”, onde a aplicação monolítica obteve 35,5% de consumo de CPU, enquanto as instâncias hospedando os diferentes microserviços da segunda aplicação obtiveram valores pequenos, como os 12,6% registrados pelo *gateway*, até 73,1%, como foi o caso do microserviço “*Cart*”.

Por meio desta análise, podemos dizer que o uso da arquitetura de microserviços em questões de desempenho e utilização de CPU na instância em que a aplicação está localizada tem o seu fluxo de funcionamento afetado, devido ao fato de que esta arquitetura

depende da comunicação entre as entidades presentes no sistema, e o tempo necessário para que isso aconteça, além dos recursos utilizados para este fim criam obstáculos não existentes na arquitetura monolítica. Em alguns testes, como na comparação entre o *baseline test* e o *load test* na sessão de desempenho do sistema com arquitetura monolítica, houveram situações com resultados que possibilitam a criação de trabalhos futuros para determinar, por exemplo, se mediante a um teste com um número superior de usuários acessando o sistema, se a arquitetura de microsserviços apresenta estabilidade em relação a arquitetura monolítica, revelando assim uma grande vantagem para sistemas que recebem um grande número de acessos, contribuindo com fatores como escalabilidade e disponibilidade do software.

## 5 CONCLUSÃO

É possível compreender o que é a arquitetura monolítica e como surgiu a arquitetura de microsserviços, e sua rápida adesão nos projetos de software a partir dos anos 2010. Pensando nisto, foram realizados testes para demonstrar como cada uma dessas arquiteturas se comporta sob efeitos de testes de desempenho e consumo de recursos: o testes de linha de base (*baseline test*), com 1000 usuários virtuais realizando uma requisição cada, servindo como um ponto de referência para o teste de carga, e o teste de carga (*load test*), com 3000 usuários virtuais realizando uma requisição cada.

Com os testes de desempenho, foi possível observar que a arquitetura de microsserviços apresenta, na maioria dos casos, um tempo de resposta superior a arquitetura monolítica, devido ao tempo necessário para que os microsserviços se comuniquem, por intermédio do *gateway*, que atua no redirecionamento de requisições, cenário este que não ocorre no sistema monolítico. Também observamos como a arquitetura de microsserviços apresenta mais consumo de CPU, devido ao envio de dados para ponto de chamada da aplicação (*endpoints*) e a utilização que é retornado para o microsserviço que realizou esta chamada, também passando pelo *gateway* na chamada e retorno dos dados solicitados. Sendo este funcionamento necessário exigido para a comunicação entre os microsserviços, e ocasionando no maior uso de recursos do ambiente em que o sistema está hospedado.

Com base na análise realizada, podemos concluir que a escolha da utilização da arquitetura de microsserviços não é recomendada se a métrica avaliada para o uso desta arquitetura for o que foi apresentado neste trabalho (desempenho e consumo de recurso), devido ao seu tempo de espera das requisições apresentando até 294% de uma mesma requisição realizada em uma arquitetura monolítica, e também o consumo de recursos no ambiente de hospedagem, com até microsserviços apresentando até 2,07 vezes o uso da CPU em comparação com um sistema monolítico. Esses dados devem ser considerados ao estudar a utilização das arquiteturas de software em um projeto, pois são fatores-chave na criação de uma sistema que apresente robusto, confiável e apresentando uma disponibilidade ao usuário mais constante.

### 5.1 TRABALHOS FUTUROS

É possível analisar outros tópicos importantes para compreender e avaliar de maneira mais densa a arquitetura monolítica e com microsserviços, como avaliando a escalabilidade do sistema por meio de ferramentas de balanceamento de carga e a utilização de múltiplas instâncias, como o que é oferecido pelo serviço *Auto Scalling* do AWS, para

evitar que um ambiente fique sobrecarregado com o número de usuários elevado realizando requisições. Também é possível avaliar a disponibilidade de suas instâncias, realizando propositalmente uma falha em algum microserviço para que, quando o mesmo for utilizado, ele pare de funcionar e assim medir como os outros microserviços da aplicação tratam esse problema, e para isso existem ferramentas como o *circuit breaker resilience4j*, disponível na dependência *spring.cloud* do *Spring Boot*. Outros objetos de trabalhos futuros é a a flexibilidade ao incluir novas tecnologias ao sistema, com a possibilidade de deixar um microserviço utilizando tecnologias em uma versão mais recente do que os outros microserviços do sistema, e também a complexidade de realizar uma migração de uma arquitetura para outra, desenvolvendo um sistema com uma arquitetura, como a monolítica, e realizar a migração para a arquitetura de microserviços, mantendo as funcionalidades de acordo, porém com as metodologias existentes na nova arquitetura adotada..

# Referências

- ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: IEEE. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. [S.l.], 2016.
- AMAZON. *Instâncias T2 do Amazon EC2*. 2023. Acessado em 28 de novembro de 2023. Disponível em: <<https://aws.amazon.com/pt/ec2/instance-types/t2/>>.
- AMAZON Web Services (AWS). 2023. Acessado em 8 de novembro de 2023. Disponível em: <<https://aws.amazon.com/pt/what-is-aws/>>.
- APACHE JMeter. 2023. Acessado em 8 de novembro de 2023. Disponível em: <<https://jmeter.apache.org>>.
- BADDULA, P. *The Evolution of Software Architecture: Monolithic to Microservices*. 2023. Acessado em 12 de novembro de 2023. Disponível em: <<https://medium.com/@phanindra208/the-evolution-of-software-architecture-monolithic-to-microservices-cb62fcd7aa94>>.
- BAŠKARADA, S.; NGUYEN, V.; KORONIOS, A. Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems*, Taylor & Francis, 2018.
- BLINOWSKI, G.; OJDOWSKA, A.; PRZYBYŁEK, A. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, IEEE, v. 10, 2022.
- DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, Springer, 2017.
- FOOTE, K. D. *A Brief History of Microservices*. 2021. Acessado em 28 de novembro de 2023. Disponível em: <<https://www.dataversity.net/a-brief-history-of-microservices/>>.
- FOWLER, J. L. M. *Microservices: a definition of this new architectural term*. 2014. Disponível em: <<https://martinfowler.com/articles/microservices.html#footnote-etymology>>.
- FOWLER, M. *MonolithFirst*. 2015. Accessed on 2023-11-10. Disponível em: <<https://martinfowler.com/bliki/MonolithFirst.html>>.
- GONÇALVES, M. M. *Arquitetura de Software: Estilos e Padrões de Design*. 2021. Acessado em 24 de novembro de 2023. Disponível em: <<https://medium.com/@marcelomg21/arquitetura-de-software-estilos-e-padr%C3%B5es-de-design-50d62d684ef2>>.
- GOS, K.; ZABIEROWSKI, W. The comparison of microservice and monolithic architecture. In: IEEE. *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. [S.l.], 2020.
- HELLE, P. et al. Coping with technological diversity by mixing different architecture and deployment paradigms. *International Journal On Advances in Software*, 2020.



IBM. *Spring Boot Framework*. 2023. Acessado em 8 de novembro de 2023. Disponível em: <<https://www.ibm.com/br-pt/topics/java-spring-boot>>.

KALSKE, M.; MÄKITALO, N.; MIKKONEN, T. Challenges when moving from monolith to microservice architecture. In: SPRINGER. *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17*. [S.l.], 2018.

LENGA, S. *Modernization of Monolithic Legacy Applications towards a Microservice Architecture with ExplorViz*. Dissertação (Mestrado) — Kiel University. Department of Computer Science. Software Engineering Group, 2019.

LUCIO, J. P. D.

*Análise Comparativa Entre Arquitetura Monolítica e de Microserviços* — Universidade Federal de Santa Catarina, Departamento de Informática e Estatística, Florianópolis, Brasil, 2017.

MALIK, H.; ADAMS, B.; HASSAN, A. E. Pinpointing the subsystems responsible for the performance deviations in a load test. In: IEEE. *2010 IEEE 21st international symposium on software reliability engineering*. [S.l.], 2010.

MARTENS, A. et al. From monolithic to component-based performance evaluation of software architectures: a series of experiments analysing accuracy and effort. *Empirical Software Engineering*, Springer, 2011.

NADAREISHVILI, I. et al. *Microservice architecture: aligning principles, practices, and culture*. [S.l.]: "O'Reilly Media, Inc.", 2016.

NEVEDROV, D. Using jmeter to performance test web services. *Published on dev2dev*, Citeseer, p. 1–11, 2006.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. [S.l.]: O'Reilly Media, 2015.

ORACLE. *Java Programming Language*. 2023. Acessado em 8 de novembro de 2023. Disponível em: <<https://www.oracle.com/java/>>.

ORACLE. *MySQL Database*. 2023. Acessado em 8 de novembro de 2023. Disponível em: <<https://www.oracle.com/mysql/what-is-mysql/>>.

OZKAYA, M. *API Gateway Pattern*. 2021. Acessado em 20 de novembro de 2023. Disponível em: <<https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df>>.

PONCE, F.; MÁRQUEZ, G.; ASTUDILLO, H. Migrating from monolithic architecture to microservices: A rapid review. In: IEEE. *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. [S.l.], 2019. p. 1–7.

POSTMAN. *Postman - What is Postman?* 2023. Acessado em 8 de novembro de 2023. Disponível em: <<https://www.postman.com/product/what-is-postman/>>.

RICHARDSON, C. *Microservices: Decomposing Applications for Deployability and Scalability*. 2023. Acessado em 12 de novembro de 2023. Disponível em: <<https://www.infoq.com/articles/microservices-intro/>>.

- SOUSA, J. *Transações no SQL: Mantendo os dados íntegros e consistentes*. 2020. Acessado em 10 de dezembro de 2023. Disponível em: <<https://www.alura.com.br/artigos/transacoes-no-sql-mantendo-os-dados-integros>>.
- VALIPOUR, M. H. et al. A brief survey of software architecture concepts and service oriented architecture. In: IEEE. *2009 2nd IEEE International Conference on Computer Science and Information Technology*. [S.l.], 2009.
- VOKOLOS, F. I.; WEYUKER, E. J. Performance testing of software systems. In: *Proceedings of the 1st International Workshop on Software and Performance*. [S.l.: s.n.], 1998.
- WOBETO, R. *O que é um padrão na arquitetura de software?* 2022. Acessado em 24 de novembro de 2023. Disponível em: <<https://www.dio.me/articles/parte-2-o-que-e-um-padrao-na-arquitetura-de-software>>.

## Apêndices

## .1 Código do Controlador da entidade Client no sistema com arquitetura monolítica.

```
1
2 @RestController
3 @RequestMapping(value = "/clients")
4 public class ClientController {
5
6     @Autowired
7     private ClientService service;
8
9     @PostMapping()
10    public ResponseEntity<String> createClient(@RequestBody ClientDTO
11    clientDTO) {
12        try {
13            Client clientCreated = service.create(clientDTO);
14            return new ResponseEntity<>("Sucesso ao criar o cliente: " +
15            clientCreated.getName() + " - " + clientCreated.getId(), HttpStatus.
16            CREATED);
17        } catch (Exception e) {
18            return new ResponseEntity<>("Erro ao criar o cliente: " + e.
19            getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
20        }
21    }
22
23    @PatchMapping(value =("/{clientId}")
24    public ResponseEntity<String> updateClient(@PathVariable Long clientId ,
25    @RequestBody ClientDTO clientDTO) {
26        try {
27            ClientDTO clientUpdatedDTO = service.updateClient(clientId ,
28            clientDTO);
29            return new ResponseEntity<>("Cliente atualizada com sucesso. ID
30            : " + clientUpdatedDTO.getId(), HttpStatus.OK);
31        } catch (EntityNotFoundException e) {
32            return new ResponseEntity<>("Erro ao atualizar a cliente: " + e
33            .getMessage(), HttpStatus.NOT_FOUND);
34        } catch (Exception e) {
35            return new ResponseEntity<>("Erro ao atualizar a cliente: " + e
36            .getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
37        }
38    }
39 }
```

Código 1 – Fonte: O autor

## .2 Código do Controlador da entidade Product no sistema com arquitetura monolítica.

```
1 @RestController
2 @RequestMapping(value = "/products")
3 public class ProductController {
4
5     @Autowired
6     private ProductService service;
7
8     @PostMapping()
9     public ResponseEntity<String> createProduct(@RequestBody ProductDTO
productDTO) {
10         try {
11             ProductDTO productCreatedDTO = service.create(productDTO);
12             return new ResponseEntity<>("Sucesso ao criar produto: " +
productCreatedDTO.getName() + " - " + productCreatedDTO.getId(),
HttpStatus.CREATED);
13         } catch (Exception e) {
14             return new ResponseEntity<>("Erro ao criar produto: " + e.
getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
15         }
16     }
17
18     @PatchMapping(value = "/{productId}")
19     public ResponseEntity<String> updateProduct(@PathVariable Long
productId, @RequestBody ProductDTO productDTO) {
20         try {
21             ProductDTO productUpdatedDTO = service.updateProduct(productId,
productDTO);
22             return new ResponseEntity<>("Produto atualizado com sucesso. ID
: " + productUpdatedDTO.getId(), HttpStatus.OK);
23         } catch (EntityNotFoundException e) {
24             return new ResponseEntity<>("Erro ao atualizar produto: " + e.
getMessage(), HttpStatus.NOT_FOUND);
25         } catch (Exception e) {
26             return new ResponseEntity<>("Erro ao atualizar produto: " + e.
getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
27         }
28     }
29 }
```

Código 2 – Fonte: O autor

### .3 Código do Controlador da entidade Stock no sistema com arquitetura monolítica.

```
1 @RestController
2 @RequestMapping("/stocks")
3 public class StockController {
4     private StockService service;
5
6     @Autowired
7     public StockController(StockService stockService) {
8         this.service = stockService;
9     }
10
11     @PostMapping
12     public ResponseEntity<String> createStock(@RequestBody StockDTO
stockDTO) {
13         return service.mountNewStock(stockDTO);
14     }
15
16     @PatchMapping("/{id}")
17     public ResponseEntity<String> updateStock(@PathVariable Long id ,
@RequestBody StockDTO stockDTO) {
18         return service.updateStock(id , stockDTO);
19     }
20 }
```

Código 3 – Fonte: O autor

## .4 Código do Controlador da entidade ItemOrder no sistema com arquitetura monolítica.

```
1 @RestController
2 @RequestMapping("/itemsOrder")
3 public class ItemOrderController {
4     private ItemOrderService service;
5
6     @Autowired
7     public ItemOrderController(ItemOrderService service) {
8         this.service = service;
9     }
10
11     @PostMapping
12     public ResponseEntity<String> create(@RequestBody ItemOrderDTO
13     itemOrderDTO) {
14         return service.mountNewItemOrder(itemOrderDTO);
15     }
16
17     @PatchMapping("/{id}")
18     public ResponseEntity<String> update(@PathVariable Long id ,
19     @RequestBody ItemOrderDTO itemOrderDTO) {
20         return service.update(id , itemOrderDTO);
21     }
22 }
```

Código 4 – Fonte: O autor

## .5 Código do Controlador da entidade Orders no sistema com arquitetura monolítica.

```
1 @RestController
2 @RequestMapping("/orders")
3 public class OrderController {
4     private OrdersService service;
5
6     @Autowired
7     public OrderController(OrdersService service) {
8         this.service = service;
9     }
10
11     @PostMapping
12     public ResponseEntity<String> create(@RequestBody OrdersDTO ordersDTO)
13     {
14         Long newOrderId = service.create(ordersDTO);
15
16         return new ResponseEntity<>("Sucesso ao criar o pedido de ID: " +
17             newOrderId, HttpStatus.CREATED);
18     }
19
20     @PutMapping("/pay/{orderId}")
21     public ResponseEntity<String> payOrder(@PathVariable Long orderId,
22         @RequestBody Map<String, String> requestBody) {
23         String paymentMethodStr = requestBody.get("paymentMethod");
24
25         PaymentMethod paymentMethod = PaymentMethod.valueOf(
26             paymentMethodStr);
27
28         return service.payOrder(orderId, paymentMethod);
29     }
30 }
```

Código 5 – Fonte: O autor



## .6 Código Arquivo Application.yaml do Microserviço Gateway, Contendo as Rotas de Direcionamento para Cada Microserviço.

```
1 spring:
2   application:
3     name: "dining-room-gateway"
4   cloud:
5     gateway:
6       routes:
7         - id: dining-room-warehouse
8           uri: lb://dining-room-warehouse
9           predicates:
10            - Path=/stocks/**
11
12         - id: dining-room-catalog
13           uri: lb://dining-room-catalog
14           predicates:
15            - Path=/brands/**
16
17         - id: dining-room-catalog
18           uri: lb://dining-room-catalog
19           predicates:
20            - Path=/products/**
21
22         - id: dining-room-cart
23           uri: lb://dining-room-cart
24           predicates:
25            - Path=/orders/**
26
27         - id: dining-room-cart
28           uri: lb://dining-room-cart
29           predicates:
30            - Path=/itemsOrder/**
31
32         - id: dining-room-user-registry
33           uri: lb://dining-room-user-registry
34           predicates:
35            - Path=/clients/**
36 server:
37   port: 8080
38
39 eureka:
40   client:
41     serviceUrl:
42       defaultZone: http://{GATEWAY_HOST}:8761/eureka/
```

## .7 Código do Controlador da Entidade Client, no Microserviço User-Registry, no Sistema com Arquitetura de Microserviços.

```
1
2 @RestController
3 @RequestMapping(value = "/clients")
4 public class ClientController {
5
6     @Autowired
7     private ClientService service;
8
9     @PostMapping()
10    public ResponseEntity<String> createClient(@RequestBody ClientDTO
11    clientDTO) {
12        try {
13            Client clientCreated = service.create(clientDTO);
14            return new ResponseEntity<>("Sucesso ao criar o cliente: " +
15            clientCreated.getName() + " - " + clientCreated.getId(), HttpStatus.
16            CREATED);
17        } catch (Exception e) {
18            return new ResponseEntity<>("Erro ao criar o cliente: " + e.
19            getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
20        }
21    }
22
23    @PatchMapping(value = "/{clientId}")
24    public ResponseEntity<String> updateClient(@PathVariable Long clientId,
25    @RequestBody ClientDTO clientDTO) {
26        try {
27            ClientDTO clientUpdatedDTO = service.updateClient(clientId,
28            clientDTO);
29            return new ResponseEntity<>("Cliente atualizada com sucesso. ID
30            : " + clientUpdatedDTO.getId(), HttpStatus.OK);
31        } catch (EntityNotFoundException e) {
32            return new ResponseEntity<>("Erro ao atualizar a cliente: " + e.
33            getMessage(), HttpStatus.NOT_FOUND);
34        } catch (Exception e) {
35            return new ResponseEntity<>("Erro ao atualizar a cliente: " + e.
36            getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
37        }
38    }
39 }
```

## .8 Código do Controlador da Entidade Product, no Microserviço Catalog, no Sistema com Arquitetura de Microserviços.

```
1
2 @RestController
3 @RequestMapping(value = "/products")
4 public class ProductController {
5
6     @Autowired
7     private ProductService service;
8
9     @PostMapping()
10    public ResponseEntity<String> createProduct(@RequestBody ProductDTO
productDTO) {
11        try {
12            ProductDTO productCreatedDTO = service.create(productDTO);
13            return new ResponseEntity<>("Sucesso ao criar produto: " +
productCreatedDTO.getName() + " - " + productCreatedDTO.getId(),
HttpStatus.CREATED);
14        } catch (Exception e) {
15            return new ResponseEntity<>("Erro ao criar produto: " + e.
getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
16        }
17    }
18
19    @PatchMapping(value = "/{productId}")
20    public ResponseEntity<String> updateProduct(@PathVariable Long
productId, @RequestBody ProductDTO productDTO) {
21        try {
22            ProductDTO productUpdatedDTO = service.updateProduct(productId,
productDTO);
23            return new ResponseEntity<>("Produto atualizado com sucesso. ID
: " + productUpdatedDTO.getId(), HttpStatus.OK);
24        } catch (EntityNotFoundException e) {
25            return new ResponseEntity<>("Erro ao atualizar produto: " + e.
getMessage(), HttpStatus.NOT_FOUND);
26        } catch (Exception e) {
27            return new ResponseEntity<>("Erro ao atualizar produto: " + e.
getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
28        }
29    }
30 }
```

## .9 Código do Controlador da Entidade Stock, no Microserviço Warehouse, no Sistema com Arquitetura de Microserviços.

```
1
2 @RestController
3 @RequestMapping("/stocks")
4 public class StockController {
5     private StockService service;
6
7     @Autowired
8     public StockController(StockService stockService) {
9         this.service = stockService;
10    }
11
12    @PostMapping("/createFromProduct")
13    public ResponseEntity<String> createFromProduct(@RequestParam("
productId") Long productId) {
14        return service.mountNewStockFromNewProduct(productId);
15    }
16
17    @RequestMapping(value = "/updateStockByProduct", method = RequestMethod
.PUT)
18    public ResponseEntity<String> updateStockByProduct(@RequestParam("
productId") Long productId, @RequestParam("quantityOrdered") int
quantityOrdered) {
19        return service.updateStockByProduct(productId, quantityOrdered);
20    }
21 }
```

Código 9 – Fonte: O autor

## .10 Código do Controlador da Entidade ItemOrder, no Microserviço Cart, no Sistema com Arquitetura de Microserviços.

```
1
2 @RestController
3 @RequestMapping("/itemsOrder")
4 public class ItemOrderController {
5     private ItemOrderService service;
6
7     @Autowired
8     public ItemOrderController(ItemOrderService service) {
9         this.service = service;
10    }
11
12    @PostMapping
13    public ResponseEntity<String> create(@RequestBody ItemOrderDTO
14    itemOrderDTO) {
15        return service.mountNewItemOrder(itemOrderDTO);
16    }
17
18    @PatchMapping("/{id}")
19    public ResponseEntity<String> update(@PathVariable Long id ,
20    @RequestBody ItemOrderDTO itemOrderDTO) {
21        return service.update(id , itemOrderDTO);
22    }
23 }
```

Código 10 – Fonte: O autor

## .11 Código do Controlador da Entidade Orders, no Microserviço Cart, no Sistema com Arquitetura de Microserviços.

```
1
2 @RestController
3 @RequestMapping("/orders")
4 public class OrdersController {
5
6     private OrdersService service;
7
8     @Autowired
9     public OrdersController(OrdersService service) {
10         this.service = service;
11     }
12
13     @PostMapping
14     public ResponseEntity<String> create(@RequestBody OrdersDTO ordersDTO)
15     {
16         Long newOrderId = service.create(ordersDTO);
17
18         return new ResponseEntity<>("Sucesso ao criar o pedido de ID: " +
19             newOrderId, HttpStatus.CREATED);
20     }
21
22     @PutMapping("/pay/{orderId}")
23     public ResponseEntity<String> payOrder(@PathVariable Long orderId,
24         @RequestBody Map<String, String> requestBody) {
25         String paymentMethodStr = requestBody.get("paymentMethod");
26
27         PaymentMethod paymentMethod = PaymentMethod.valueOf(
28             paymentMethodStr);
29
30         return service.payOrder(orderId, paymentMethod);
31     }
32 }
```

Código 11 – Fonte: O autor

Figura 37 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema monolítico durante o baseline test.

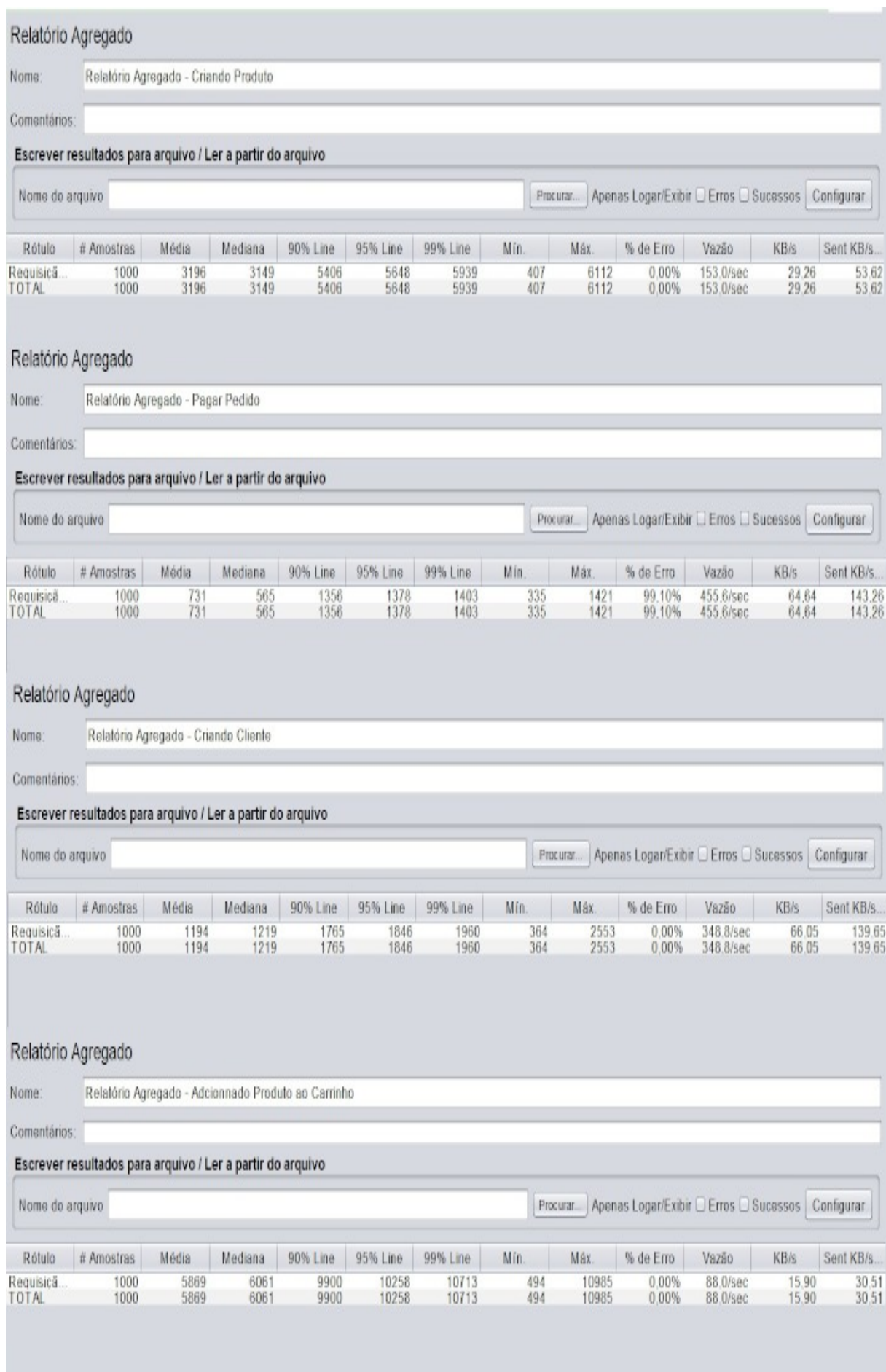


Figura 38 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema monolítico durante o load test.

### Relatório Agregado

Nome:

Comentários:

**Escrever resultados para arquivo / Ler a partir do arquivo**

Nome do arquivo    ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição...	3000	3150	3030	5603	6204	6560	560	6867	0.00%	424.7/sec	79.63	146.82
TOTAL	3000	3150	3030	5603	6204	6560	560	6867	0.00%	424.7/sec	79.63	146.82

### Relatório Agregado

Nome:

Comentários:

**Escrever resultados para arquivo / Ler a partir do arquivo**

Nome do arquivo    ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição...	3000	2249	2364	3116	3643	3968	536	4023	0.00%	597.7/sec	113.83	239.32
TOTAL	3000	2249	2364	3116	3643	3968	536	4023	0.00%	597.7/sec	113.83	239.32

### Relatório Agregado

Nome:

Comentários:

**Escrever resultados para arquivo / Ler a partir do arquivo**

Nome do arquivo    ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição...	3000	2103	1998	3396	3550	3750	423	4219	99.67%	653.3/sec	92.58	205.44
TOTAL	3000	2103	1998	3396	3550	3750	423	4219	99.67%	653.3/sec	92.58	205.44

### Relatório Agregado

Nome:

Comentários:

**Escrever resultados para arquivo / Ler a partir do arquivo**

Nome do arquivo    ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição...	3000	10109	10352	16961	18073	18890	551	20038	0.00%	148.3/sec	26.80	52.30
TOTAL	3000	10109	10352	16961	18073	18890	551	20038	0.00%	148.3/sec	26.80	52.30



Figura 39 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema com microserviços durante o baseline test.

Relatório Agregado

Nome: Relatório Agregado - Criando Produto

Comentários: Baseline Test - Microserviços

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo
Procurar...
Apenas Logar/Exibir
☐ Erros
☐ Sucessos
Configurar

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Min	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição - ...	1000	5350	5275	7752	8043	8253	2442	8472	0.00%	117.5/sec	17.78	41.32
TOTAL	1000	5350	5275	7752	8043	8253	2442	8472	0.00%	117.5/sec	17.78	41.32

Relatório Agregado

Nome: Relatório Agregado - Criando Cliente

Comentários: Baseline Test - Microserviços

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo
Procurar...
Apenas Logar/Exibir
☐ Erros
☐ Sucessos
Configurar

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Min	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição - ...	1000	2010	1892	3026	3288	3458	484	3840	4.50%	231.9/sec	34.07	93.97
TOTAL	1000	2010	1892	3026	3288	3458	484	3840	4.50%	231.9/sec	34.07	93.97

Relatório Agregado

Nome: Relatório Agregado - Pagar Pedido

Comentários: Baseline Test - Microserviços

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo
Procurar...
Apenas Logar/Exibir
☐ Erros
☐ Sucessos
Configurar

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Min	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição ...	1000	2881	2919	4090	4318	4520	1252	4734	99.00%	193.9/sec	25.93	59.83
TOTAL	1000	2881	2919	4090	4318	4520	1252	4734	99.00%	193.9/sec	25.93	59.83

Relatório Agregado

Nome: Relatório Agregado - Adicionando Produto ao Carrinho

Comentários: Baseline Test - Microserviços

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo
Procurar...
Apenas Logar/Exibir
☐ Erros
☐ Sucessos
Configurar

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Min	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição - ...	1000	15273	15758	20365	20770	21005	7051	21618	0.00%	45.7/sec	6.43	15.89
TOTAL	1000	15273	15758	20365	20770	21005	7051	21618	0.00%	45.7/sec	6.43	15.89

Figura 40 – Relatórios Agregado do tempo de resposta das requisições HTTP para o sistema com microserviços durante o load test.

### Relatório Agregado

Nome:

Comentários:

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo   ☐ Apenas Logar/Exibir ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição - ...	3000	4662	4687	7442	7757	8131	602	21157	3.53%	127.6/sec	18.62	44.20
TOTAL	3000	4662	4687	7442	7757	8131	602	21157	3.53%	127.6/sec	18.62	44.20

### Relatório Agregado

Nome:

Comentários:

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo   ☐ Apenas Logar/Exibir ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição - ...	3000	2838	2877	3971	4231	4833	498	5042	4.87%	514.3/sec	74.97	206.43
TOTAL	3000	2838	2877	3971	4231	4833	498	5042	4.87%	514.3/sec	74.97	206.43

### Relatório Agregado

Nome:

Comentários:

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo   ☐ Apenas Logar/Exibir ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição ...	3000	4514	4922	5750	6035	6989	805	7117	99.87%	388.0/sec	49.23	113.55
TOTAL	3000	4514	4922	5750	6035	6989	805	7117	99.87%	388.0/sec	49.23	113.55

### Relatório Agregado

Nome:

Comentários:

Escrever resultados para arquivo / Ler a partir do arquivo

Nome do arquivo   ☐ Apenas Logar/Exibir ☐ Erros ☐ Sucessos

Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Requisição ...	3000	20625	21741	28242	29154	29958	929	30331	0.57%	85.9/sec	12.04	29.83
TOTAL	3000	20625	21741	28242	29154	29958	929	30331	0.57%	85.9/sec	12.04	29.83