UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS VII - PATOS
CENTRO DE CIÊNCIAS EXATAS E SOCIAIS APLICADAS
CURSO DE GRADUAÇÃO EM BACHARELADO EM COMPUTAÇÃO

HARLLEM ALVES DO NASCIMENTO

ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE ARQUITETURAS MONOLÍTICAS E DE MICROSSERVIÇOS EM UMA APLICAÇÃO WEB

PATOS - PB
2023

HARLLEM ALVES DO NASCIMENTO

ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE ARQUITETURAS MONOLÍTICAS E DE MICROSSERVIÇOS EM UMA APLICAÇÃO WEB

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual da Paraíba, em cumprimento à exigência para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Demetrio Gomes Mestre

PATOS - PB
2023

N244a Nascimento, Harllem Alves do.

Análise Comparativa de Desempenho entre Arquiteturas Monolíticas e de Microsserviços em uma Aplicação Web [manuscrito] / Harllem Alves do Nascimento. - 2023.

73 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências Exatas e Sociais Aplicadas, 2023.

"Orientação : Prof. Dr. Demetrio Gomes Mestre, Coordenação do Curso de Computação - CCEA. "

1. Arquitetura Monolítica. 2. Microsserviços. 3. Testes de Carga. I. Título

21. ed. CDD 005.13

Elaborada por Edson M. A. Monteiro - CRB - CRB 15/743 BSC7/UEPB

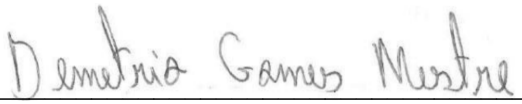
HARLLEM ALVES DO NASCIMENTO

ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE ARQUITETURAS MONOLÍTICAS E DE MICROSSERVIÇOS EM UMA APLICAÇÃO WEB.

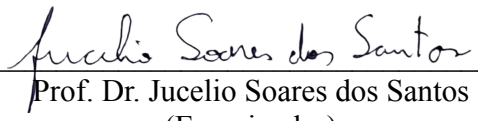
Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual da Paraíba, em cumprimento à exigência para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 30 de novembro de 2023

BANCA EXAMINADORA



Prof. Dr. Demétrio Gomes Mestre
(Orientador)



Prof. Dr. Jucelio Soares dos Santos
(Examinador)

Prof. Me. Pablo Ribeiro Suárez
(Examinador)

Dedico este trabalho a minha família, amigos, professores e todos que me apoiaram nesta jornada. Este trabalho só foi possível graças a vocês.

AGRADECIMENTOS

À Deus, pela minha vida, e por me permitir ultrapassar todos os obstáculos encontrados ao longo da vida.

À minha família, que sempre me apoiou e incentivou ao longo desta jornada acadêmica. Seu amor e apoio incondicional foram fundamentais para minha motivação e sucesso.

Aos meus amigos e colegas de curso, pelos momentos de estudo em grupo, pelas trocas de conhecimento e pelas amizades que levarei para a vida.

À Universidade Estadual da Paraíba e seus professores, que proporcionaram um ambiente de aprendizado e pesquisa inspirador.

Aos participantes da pesquisa e todas as pessoas que colaboraram de alguma forma com este trabalho, tornando-o possível.

Aos autores e pesquisadores cujas obras foram essenciais para a construção deste trabalho.

Àqueles que, de alguma forma, contribuíram para o meu crescimento acadêmico e pessoal, meu sincero agradecimento.

Este trabalho é dedicado a todos vocês.

É isso. Esta é a sua história.
Tudo começa aqui.
Auron

RESUMO

Este estudo consiste em uma análise comparativa entre as arquiteturas de software monolítica e de microsserviços em uma aplicação de chat, com o propósito de identificar diferenças nas estratégias de escalabilidade, eficiência no uso de recursos e impacto no tempo de resposta. A metodologia incluiu o desenvolvimento de uma aplicação monolítica e sua subsequente decomposição em microsserviços. Foram realizados testes de carga simulando cenários realistas, e os resultados destacaram que as estratégias de escalabilidade variam consideravelmente entre as arquiteturas. A arquitetura de microsserviços possibilitou o escalonamento automático de setores e funcionalidades específicas, conforme a demanda, demonstrando flexibilidade na gestão dos recursos. A eficiência de uso de recursos foi influenciada pela natureza das tarefas e características da carga de trabalho. Em relação à latência e ao tempo de resposta, a arquitetura monolítica superou a de microsserviços em algumas funcionalidades, enquanto a última se destacou em outras. Observou-se que o número de chamadas de APIs internas nas aplicações de microsserviços impactou diretamente a latência e o desempenho global. A pesquisa foi fundamentada principalmente nos estudos de Microsserviços de Fowler e Lewis (2014), e, espera-se que a mesma possa contribuir para desenvolvedores e tomadores de decisão que buscam selecionar a arquitetura mais apropriada para suas aplicações, destacando áreas para futuras pesquisas.

Palavras-chave: Arquitetura Monolítica; Microsserviços; Testes de Carga.

ABSTRACT

This study consists of a comparative analysis between monolithic and microservices software architectures in a chat application, to identify differences in scalability strategies, resource efficiency, and impact on response time. The methodology included the development of a monolithic application and its subsequent decomposition into microservices. Realistic scenario load tests were conducted, and the results highlighted that scalability strategies vary considerably between the architectures. The microservices architecture enabled automatic scaling of specific sectors and functionalities based on demand, demonstrating flexibility in resource management. The nature of tasks and workload characteristics influenced resource efficiency. Regarding latency and response time, the monolithic architecture outperformed the microservices in some functionalities, while the latter excelled in others. It was observed that the number of internal API calls in microservices applications directly impacted latency and overall performance. The research was primarily grounded in the studies on Microservices by Fowler e Lewis (2014), and it is expected that it can provide insights for developers and decision-makers seeking to select the most suitable architecture for their applications, while also highlighting areas for future research.

Keywords: Monolithic Architecture; Microservices; Load Testing.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
DDD	*Domain-Driven Design*
DOM	*Document Object Model*
DTO	*Data Transfer Object*
IP	*Internet Protocol*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
JSON	*JavaScript Object Notation*
JWT	*JSON Web Token*
MVC	*Model-View-Controller*
REST	*Representational State Transfer*
STOMP	*Simple Text Oriented Messaging Protocol*
TCP	*Transmission Control Protocol*
URI	*Uniform Resource Identifiers*
URL	*Uniform Resource Locator*
UUID	*Universally Unique Identifier*
VMM	*Virtual Machine Monitor*
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Com o crescimento explosivo da *internet*, as aplicações *web* desempenham um papel cada vez mais relevante na era digital, fornecendo serviços e informações essenciais para usuários em todo o mundo (Rosen; Shklar, 2009a). Essas aplicações variam desde plataformas de mídia social e comércio eletrônico até serviços bancários e sistemas de gerenciamento de conteúdo.

Segundo Statista (2021)[1] a quantidade de dados gerados anualmente tem crescido de forma contínua desde 2010. Esse aumento é notável, considerando que em 2010 havia apenas 2 *zettabytes* de dados, e em 2020, já foi atingida a marca de 64,2 *zettabytes*, tendo como previsão um aumento para 120 *zettabytes* até 2023 e 181 *zettabytes* até 2025. Diante do crescimento constante e acelerado do tráfego de dados ao longo dos anos, torna- se imperativa a necessidade de uma adaptação contínua das plataformas *on-line* encarregadas de processar esses dados. Da mesma forma, surge a necessidade de considerar a migração de sistemas para novas arquiteturas, pois segundo Richards e Ford (2020), a compreensão de um *software* não se limita apenas à sua estrutura, mas também abrange as razões subjacentes às decisões de design tomadas durante o seu desenvolvimento.

Diante deste cenário desafiador, a seleção da arquitetura para o desenvolvimento de uma aplicação *web* torna-se um elemento crucial para a eficaz disponibilização de um serviço *on-line*. Conforme mencionado por (Bosch, 1999), a relevância da arquitetura de software se manifesta na sua influência sobre o desempenho, a robustez, a distribuição e a manutenção de um sistema.

Neste cenário dinâmico e cada vez mais digital, a comparação entre arquiteturas monolíticas e de microsserviços desempenha um papel crucial nas decisões estratégicas de desenvolvedores e líderes de projeto. Essa análise não só impacta a eficiência do desenvolvimento de *software*, mas também desempenha um papel significativo na avaliação prática de cenários de alta carga em ambas as arquiteturas.

Na busca por uma compreensão aprofundada das arquiteturas monolíticas e de microsserviços, foram encontrados três estudos relacionados que lançam luz sobre as complexidades inerentes a essa escolha arquitetural. O presente trabalho, almeja contribuir para a compreensão do desempenho e das características distintivas dessas abordagens em uma aplicação *web*. Ao mergulharmos nas análises comparativas conduzidas por Al-Debagy e Martinek (2018), Duarte (2017), e Amaral e Carvalho (2017), notamos as divergências de desempenho e os cenários propícios para cada arquitetura com diferentes critérios de análise.

Para realizar uma avaliação abrangente e fundamentada, este estudo adotará uma abordagem baseada em testes de carga. Esses testes serão conduzidos por meio da simulação de fluxos de ações típicas realizadas por usuários fictícios, criados com o auxílio da ferramenta de código aberto *JMeter* (Apache, 2023). Essa ferramenta é reconhecida por sua capacidade de avaliar tanto a funcionalidade quanto o desempenho de aplicações *web*, tornando-a ideal para coletar dados objetivos e métricas confiáveis. Este estudo se baseou nas contribuições de Fowler e Lewis

[1] Plataforma global de dados e inteligência de negócios com uma ampla coleção de estatísticas

(2014), autoridades em desenvolvimento de *software*, notadamente em microsserviços. Suas pesquisas forneceram uma base sólida para a análise comparativa entre as arquiteturas monolítica e de microsserviços.

Os resultados obtidos a partir dos testes de carga foram registrados, analisados e comparados. Essa análise não se limitou apenas ao desempenho puro, mas também abordou aspectos como escalabilidade e consumo de recursos. A combinação dessas métricas proporcionou uma visão abrangente e equilibrada das vantagens e desvantagens de ambas as arquiteturas, permitindo a identificação de cenários ideais para a adoção de cada uma delas.

Diante do amplo debate acerca da escolha entre as arquiteturas de microsserviços e a abordagem monolítica para aplicações *web*, este estudo se concentra na análise comparativa desses modelos. A comparação foi realizada por meio da avaliação de uma aplicação *web*, considerando os cenários, vantagens e desvantagens presentes na seleção das arquiteturas. Com objetivos específicos delineados, o estudo engloba uma revisão bibliográfica, o desenvolvimento inicial de uma aplicação monolítica, sua subsequente decomposição em microsserviços, a preparação da infraestrutura utilizando o Kubernetes (2023) e Docker (2023a), culminando na realização de testes de carga para uma análise abrangente do desempenho.

1.1 Objetivos

A análise conduzida neste estudo foi centrada na avaliação de um aplicativo *web* desenvolvido sob duas arquiteturas distintas: a arquitetura de microsserviços e a arquitetura monolítica. Essa abordagem permitiu uma minuciosa comparação do desempenho entre esses dois contextos de desenvolvimento de sistemas *web*, permitindo-nos identificar e destacar as diversas vantagens e desvantagens associadas a cada uma dessas estruturas arquiteturais.

1.1.1 Objetivo geral

Comparar o desempenho de duas arquiteturas de desenvolvimento distintas, microsserviços e monolítica, por meio da avaliação de um aplicativo *web*. O objetivo é fornecer percepções para decisões informadas no cenário do desenvolvimento de sistemas.

1.1.2 Objetivos específicos

A fim alcançar o objetivo geral deste trabalho, foram definidos os seguintes objetivos específicos:

- Realizar uma revisão bibliográfica sobre os conceitos e trabalhos relacionados à análise de aplicações *web* baseadas em microsserviços e arquitetura monolítica;
- Desenvolver uma aplicação completa seguindo os padrões de arquitetura monolítica focada em uma sala de bate-papo em tempo real;
- Decompor a aplicação monolítica em uma aplicação baseada em microsserviços;

- Preparar a infraestrutura e ambiente de execução da aplicação utilizando *Kubernetes* e *Docker*;
- Utilizar testes de carga para avaliar a eficiência de cada uma das arquiteturas e, com base nos dados obtidos, realizar uma análise comparativa entre elas.

1.2 Justificativa

Primeiramente, o crescente volume de dados das aplicações *web* (Statista, 2021), cria naturalmente a necessidade de demanda por alto desempenho e escalabilidade, tornando essencial a escolha de uma arquitetura apropriada. A arquitetura monolítica, que tradicionalmente tem sido amplamente utilizada, apresenta características que a tornam fácil de desenvolver e manter em estágios iniciais. No entanto, com o aumento da complexidade e do tráfego de informações geradas pelos usuários, pode tornar-se um gargalo em termos de escalabilidade e flexibilidade.

Por outro lado, a arquitetura de microsserviços, uma abordagem mais recente e modular, oferece a flexibilidade de desenvolver e implantar serviços independentes, o que pode facilitar a escalabilidade e a manutenção de aplicações em crescimento. No entanto, essa abordagem não está isenta de desafios, como maior complexidade na gestão de vários serviços e maior sobrecarga de comunicação.

Portanto, a análise comparativa entre essas duas arquiteturas em um contexto prático, como uma sala de bate-papo em tempo real, é relevante, já que pode oferecer contribuições significativas a desenvolvedores e empresas que buscam decidir qual abordagem adotar. Através da realização de testes de carga, é possível quantificar o desempenho de ambas as arquiteturas em condições de carga realista, o que se mostra essencial para tomar decisões informadas. Além disso, esta análise pode contribuir para o avanço do conhecimento na área de arquitetura de software, ao revisitar informações empíricas sobre o desempenho dessas arquiteturas em um cenário específico.

Em resumo, dada a importância crítica da escolha de arquiteturas de *software* eficazes, a realização deste estudo é justificável como um esforço para fornecer informações valiosas e orientadas por dados, a fim de auxiliar desenvolvedores e organizações na tomada de decisões informadas quanto à arquitetura mais apropriada para o desenvolvimento de sistemas de sala de debate em tempo real.

1.3 Estrutura do trabalho

Este Trabalho de Conclusão de curso está estruturado em cinco capítulos, o primeiro consiste na introdução, apresentando uma visão geral da pesquisa, o que inclui a contextualização do problema abordado, a definição dos objetivos, a justificativa que fundamenta a pesquisa e uma visão geral da estrutura do trabalho.

O segundo capítulo é destinado à Fundamentação Teórica, na qual são apresentados os conceitos fundamentais e revisados dos trabalhos relacionados, contribuindo para estabelecer o

contexto e a relevância da pesquisa.

No terceiro capítulo, a abordagem se aprofunda na estratégia de implementação do sistema *web*. Isso envolve uma descrição detalhada do ambiente e dos testes executados. No referido capítulo são explorados os métodos, as técnicas e as ferramentas empregadas na construção do trabalho, além dos procedimentos adotados para atingir os objetivos estabelecidos.

O quarto capítulo é voltado para a análise crítica dos resultados obtidos durante a condução do trabalho, o que envolve uma discussão profunda das descobertas e uma análise das implicações dos resultados em relação aos objetivos.

No quinto e último capítulo são apresentados, de forma resumida, os principais resultados e conclusões obtidas através das respostas dadas as questões de pesquisa, assim como são oferecidas sugestões e recomendações para trabalhos futuros que possam ser elaborados com base neste trabalho.

Neste capítulo introdutório, destacamos a relevância da escolha arquitetural em aplicações web, enfatizando sua influência crucial na eficiência e escalabilidade dos sistemas. No próximo capítulo, aprofundaremos nossa compreensão ao explorar a literatura e trabalhos relacionados, fornecendo uma base teórica para a análise comparativa entre as arquiteturas monolítica e de microsserviços.

2 REFERENCIAL TEÓRICO

Neste capítulo, serão abordados os principais tópicos essenciais para a compreensão deste trabalho, fundamentados no referencial teórico. Esta seção desempenha um papel fundamental ao estabelecer as bases conceituais que sustentam esta pesquisa, fornecendo uma visão abrangente das teorias e conceitos relevantes.

2.1 Modelo cliente–servidor

O modelo cliente-servidor foi concebido durante a década de 1970 pela divisão de pesquisa Xerox PARC da empresa Xerox. Conforme afirmado por Rosen e Shklar (2009b), é comum que aplicações baseadas em *TCP/IP* adotem o modelo cliente/servidor, nesse arranjo, os servidores, frequentemente referidos como serviços, permanecem em um estado de espera para receber solicitações originadas por programas clientes, e, ao recebê-las, procedem à sua devida execução.

No contexto desse modelo, os navegadores *web* são exemplos notáveis de programas clientes, mas a aplicabilidade não se limita a essa categoria, uma vez que os servidores também podem realizar solicitações a outros servidores.

2.2 Protocolos de rede

Um protocolo de rede pode ser visto como uma linguagem que os dispositivos de rede utilizam para se comunicar entre si. Kurose e Ross (2014, p. 7) definem um protocolo como "[...] o formato e a ordem das mensagens trocadas entre duas ou mais entidades comunicantes, bem como as ações realizadas na transmissão e/ou no recebimento de uma mensagem ou outro evento". Dessa forma, protocolos de rede estabelecem regras e convenções que regulam como os dados são formatados, transmitidos e recebidos em uma rede, garantindo uma comunicação fluida e confiável entre dispositivos. Nas subseções subsequentes, serão apresentados os dois protocolos que são empregados neste trabalho.

2.2.1 Protocolo HTTP

O Protocolo de Transferência de Hipertexto (*HTTP*) é um protocolo essencial para a comunicação entre clientes e servidores na Internet. Ele facilita a transferência de dados e desempenha um papel fundamental na interconexão de máquinas em rede. Segundo Berners-Lee, Fielding e Frystyk (1996), *HTTP* é um protocolo ágil e genérico para sistemas colaborativos de hipermídia, permitindo transferência de dados independente de sua natureza.

O protocolo *HTTP* se baseia principalmente no paradigma de solicitação/resposta, onde clientes enviam solicitações aos servidores, que, por sua vez, geram e retornam respostas utilizando *HTTP*. Esta descrição está alinhada com a abordagem de Rosen e Shklar (2009c), que enfatiza a essencial dinâmica de comunicação entre clientes e servidores, onde clientes podem

ser aplicativos ou servidores, frequentemente em máquinas separadas, se comunicando por meio de uma rede.

Recursos fornecidos por um servidor podem ser acessados por meio de *URI's* (*Uniform Resource Identifiers*), que segundo Berners-Lee, Fielding e Frystyk (1996) é uma cadeia de caracteres usadas para identificar recursos de rede com base em seu nome, localização ou qualquer outra característica.

Além de fornecer a *URI*, é imperativo especificar um método de acesso ao recurso, Berners-Lee, Fielding e Frystyk (1996) explicam que Métodos *HTTP* são os *tokens* indicativos da ação a ser realizada no recurso identificado pela *URI*. Vários métodos de identificação estão disponíveis para diversas ações, por exemplo:

- **Método GET:** utilizado para recuperar informações de um recurso na *URI*, como consultar usuários ou um usuário específico ao fornecer parâmetros.
- **Método POST:** empregado para criar ou editar recursos através da *URI*.
- **Método HEAD:** responsável por fornecer metadados sobre o recurso acessado pela *URI*.

2.2.2 Protocolo Websocket

O Protocolo *WebSocket*, conforme descrito por Fette e Melnikov (2011), é um modelo de comunicação bidirecional de baixa latência entre cliente e servidor. Este protocolo utiliza uma única conexão *TCP* (*Transmission Control Protocol*) para atender às necessidades de aplicativos da *web* que requerem comunicação bidirecional, eliminando a necessidade de múltiplas conexões *HTTP*.

O *STOMP* (*Simple Text Oriented Messaging Protocol*) é um protocolo de comunicação que usa quadros semelhantes ao *HTTP*, com comandos, cabeçalhos e corpos opcionais. Pode transmitir mensagens em texto ou binário, com codificação padrão em *UTF-8*, mas permite outras codificações (STOMP, 2012). Este protocolo é um protocolo simples implementado em conjunto com o *WebSocket* para prover funcionalidades de troca de mensagens em tempo real. Dentre as funcionalidades presentes, podemos citar, por exemplo:

- **CONNECT:** usado para estabelecer uma conexão com o servidor STOMP.
- **SUBSCRIBE:** permite que os clientes se inscrevam para receber mensagens de um destino específico (como uma fila ou tópico).
- **SEND:** Utilizado para enviar mensagens para um destino específico.
- **UNSUBSCRIBE:** permite que os clientes cancelem a inscrição de destinos aos quais se inscreveram anteriormente.
- **DISCONNECT:** permite que os clientes cancelem a inscrição de destinos aos quais se inscreveram anteriormente.

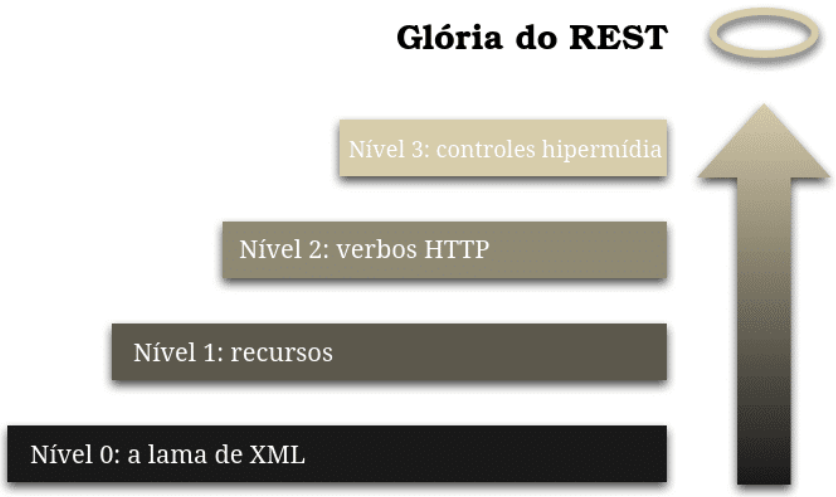
2.3 Interface de Programação de Aplicação (API) e o Padrão REST

Uma *API* é a sigla para Interface de Programação de Aplicativos, no contexto *Web*, funciona como um meio de fornecer um ponto de acesso para recursos através de um servidor. Jacobson, Brail e Woods (2012) explicam que uma *API* pode ser compreendida como um contrato entre sistemas.

A abordagem conhecida como *REST* (Transferência de Estado Representacional) para *API's* é um padrão arquitetural amplamente adotado, especialmente devido à sua simplicidade e flexibilidade. Nesse modelo, as solicitações feitas aos serviços são respondidas com uma representação do estado do recurso desejado, frequentemente no formato *JSON*[1], utilizando o protocolo *HTTP*. Nesse contexto, os cabeçalhos e parâmetros desempenham um papel fundamental nas solicitações *HTTP*, pois transportam informações cruciais para operações como armazenamento em cache e para definir o tipo de operação a ser realizada.

O Modelo de Maturidade de Richardson é um conceito proposto por Richardson (2008) para avaliar o grau de adesão e maturidade de uma *API* em relação aos princípios *REST*, que segundo Fowler (2010) se trata de uma abordagem útil para classificar uma *API*.

Figura 1 – Glória do *REST*



Fonte: (Fowler, 2010). Tradução própria.

A **Figura 1** apresenta uma representação em níveis de uma *API*, na qual os níveis são determinados pelas características presentes na *API*. Estas características, conforme explicado por Fowler (2010), incluem:

- **Nível 0 - A lama de *XML*[2]:** neste nível, a *API* não segue os princípios do *REST*. Em vez disso, as operações são realizadas por meio de solicitações *POST* para URLs fixas e retornam dados em formato *XML*.

[1] ***JSON* (*JavaScript Object Notation*)** é um formato de troca de dados leve e legível por máquinas, baseado em pares de chave-valor, amplamente utilizado na comunicação entre sistemas web.

[2] ***XML* (*eXtensible Markup Language*)** é uma linguagem de marcação extensível que cria documentos legíveis para humanos e computadores, porém, é conhecida por sua natureza detalhada e excesso de marcação.

- **Nível 1 - Recursos:** neste nível, a *API* começa a adotar o conceito de recursos. Cada recurso é identificado por uma URL única e pode ser acessado por meio de diferentes métodos *HTTP*, como *GET*, *POST*, *PUT* e *DELETE*.
- **Nível 2 - Verbos *HTTP*:** neste nível, a *API* utiliza os verbos *HTTP* de maneira apropriada. *GET* é usado para recuperar informações, *POST* para criar recursos, *PUT* para atualizá-los e *DELETE* para removê-los. Isso torna a *API* mais consistente com os princípios *REST*.
- **Nível 3 - Controles de Hipermídia:** este é o nível mais avançado de maturidade. Aqui, a *API* fornece não apenas dados, mas também links hipermídia (hiperlinks) que permitem que o cliente navegue pela *API* de forma dinâmica, descobrindo recursos e operações disponíveis sem depender de conhecimento prévio.

2.4 Arquitetura de software

Bass, Clements e Kazman (2013) destacam a função fundamental da arquitetura de *software* como uma ligação que ajuda as empresas a alcançarem seus objetivos de negócios. Esses objetivos, muitas vezes, são conceituais e difíceis de se definir claramente, e é a arquitetura de *software* que desempenha um papel crucial na transformação dessas metas abstratas em sistemas concretos. Os autores ainda destacam que arquitetura de *software* de um sistema se refere às configurações essenciais que permitem compreender o sistema, incluindo os componentes de *software*, as interações entre eles e as propriedades que definem sua funcionalidade.

Richards e Ford (2020) estabelecem duas leis para a arquitetura de *software*. A primeira delas afirma que em uma arquitetura de *software*, tudo envolve *trade-offs*. Esses *trade-offs* estão intrinsecamente ligados à análise do contexto e à escolha da abordagem mais adequada, exigindo a consideração de uma variedade de fatores, uma vez que não existe uma solução única que seja a melhor para todos os casos. Já a segunda lei, ressalta a importância não apenas entender como um sistema de *software* é estruturado, mas também as razões por trás das decisões de design tomadas durante o seu desenvolvimento. Isso ajuda a obter uma compreensão mais completa da arquitetura de *software* e facilita a manutenção e melhoria do sistema no futuro.

Com base nestas definições, a atenção será direcionada para as duas arquiteturas que compõem o cerne deste estudo: a Arquitetura Monolítica e a Arquitetura de Microsserviços. Estas arquiteturas serão exploradas nas subseções subsequentes.

2.4.1 Arquitetura Monolítica

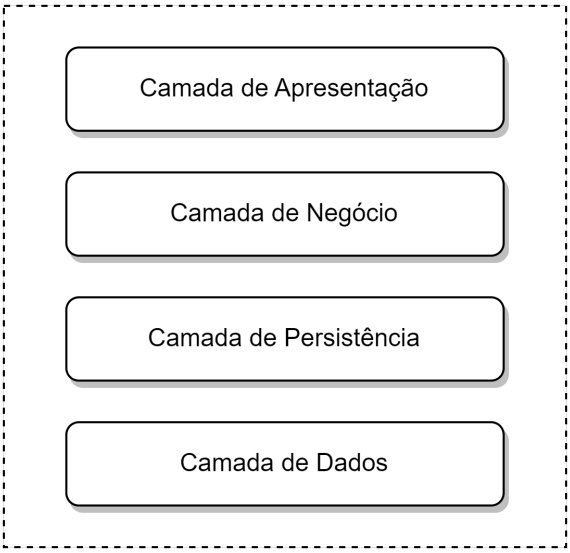
Uma aplicação que segue a arquitetura monolítica, pode ser definida como um sistema em que todos os seus componentes e módulos não podem ser executados independentemente (Dragoni *et al.*, 2017), dessa forma, todo o sistema é gerenciado por um único processo.

Apesar da arquitetura monolítica oferecer simplicidade na gestão, visto que todos os seus componentes são encapsulados em um único local, essa abordagem apresenta desafios significativos para aplicações complexas em um horizonte de longo prazo.

De acordo com Dragoni *et al.* (2017), monólitos de grande escala apresentam notória complexidade em termos de manutenção e aprimoramento, dentre os principais motivos destacados pelo autor, podemos citar:

- A identificação e resolução de falhas demandam análises extensivas através do código-fonte completo.
- Qualquer modificação em um único módulo de um monólito requer a reinicialização completa da aplicação, o que, em projetos complexos, resulta frequentemente em períodos consideráveis de inatividade.
- Os desenvolvedores estão limitados a utilizar as tecnologias inicialmente estabelecidas para o projeto, além de terem a responsabilidade de adicionar ou atualizar bibliotecas, o que pode resultar em instabilidades no sistema.
- A escalabilidade[3] é restrita, já que em monólitos é necessário criar uma nova instância da aplicação, mas o tráfego pode ser direcionado apenas a uma parcela dos módulos do sistema.

Figura 2 – Estrutura de um monólito



Fonte: Adaptado de (Richards; Ford, 2020). Tradução própria.

A **Figura 2** mostra a estrutura de uma aplicação monolítica apresentando-a em quatro camadas. A primeira delas é a camada de apresentação, que tem como finalidade a exibição dos dados ao usuário final, incluindo a interface de interação. Em seguida, temos a camada de negócios, na qual são implementadas as regras de negócio da aplicação, ou seja, a lógica central do sistema. A terceira camada é a de persistência, responsável pela comunicação com o banco

[3] **Escalabilidade** é a capacidade de um sistema de se adaptar e crescer para lidar com um aumento de demanda ou carga.

de dados, garantindo o armazenamento e recuperação dos dados. Por fim, a camada de dados é destinada ao armazenamento propriamente dito das informações, sendo o repositório físico dos dados manipulados pela aplicação.

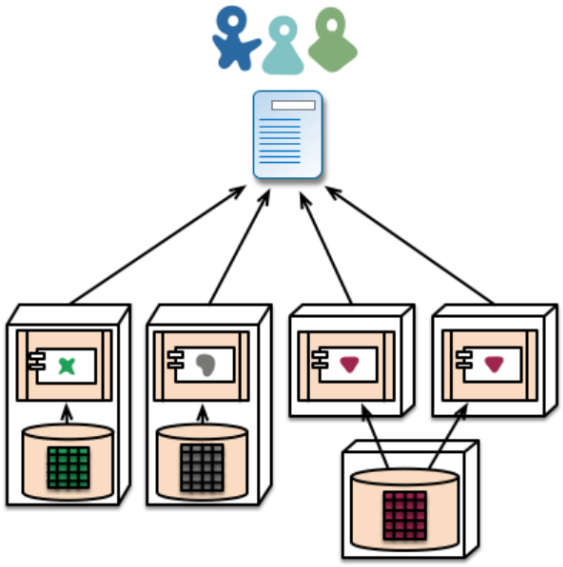
Uma aplicação monolítica, embora possa apresentar funcionalidades fortemente acopladas e impor restrições aos desenvolvedores, limitando-os a uma única linguagem de programação ou conjunto de bibliotecas específicas, oferece vantagens significativas em cenários de aplicações de pequeno ou médio porte. Isso se deve à simplificação notável no gerenciamento e desenvolvimento da infraestrutura operacional, que se traduz em benefícios substanciais, tais como, menor complexidade durante a implantação, facilidade na manutenção, especialmente em ambientes de menor complexidade, e redução de custos na gestão do sistema.

2.4.2 Arquitetura de Microsserviços

Segundo Fowler e Lewis (2014), o estilo arquitetônico de microsserviços representa uma abordagem na qual uma única aplicação é desenvolvida como um conjunto de pequenos serviços individuais, cada um operando de maneira autônoma em seu próprio processo e se comunicando por meio de mecanismos leves, frequentemente usando uma *API*.

Essa abordagem permite que a lógica de uma aplicação seja separada em serviços, sendo possível que cada serviço seja escrito em uma linguagem de programação diferente ou use um banco de dados diferente.

Figura 3 – Estrutura de um sistema baseado em microsserviços



Fonte: Fonte: Adaptado de (Fowler; Lewis, 2014).

A arquitetura de microsserviços representada na **Figura 3** é constituída por meio de componentes que, conforme definido por Fowler e Lewis (2014), são unidades de *software* independentes que podem ser substituídas e atualizadas de maneira autônoma.

Cada componente possui sua própria base de dados, embora, em algumas situações, várias instancias de um serviços possam compartilhar uma mesma base de dados. Segundo

Fowler e Lewis (2014), os microsserviços adotam a abordagem de permitir que cada serviço gerencie sua própria base de dados, que pode ser da mesma tecnologia ou de sistemas diferentes, conceito conhecido como Persistência Poliglota.

O termo Persistência Poliglota, segundo Fowler (2011), refere-se à prática de empregar uma variedade de tecnologias de armazenamento de dados para diferentes tipos de informações em uma aplicação. Isso implica a utilização de diversos sistemas de banco de dados para atender a diferentes demandas, mesmo dentro da mesma aplicação. Embora traga benefícios como melhor desempenho e escalabilidade, a adoção desse modelo implica em uma maior complexidade e demanda um amplo conhecimento técnico para sua implementação eficaz.

A descentralização dos dados, resultante da característica em que cada microsserviço possui sua própria base de dados, representa um dos principais desafios na fase de modelagem do sistema. Conforme mencionado por Fowler e Lewis (2014), a aplicação do conceito de Contexto Limitado, oriundo do *Domain-Driven Design* (DDD), oferece uma perspectiva valiosa para abordar essa complexidade, permitindo a divisão de um domínio complexo em contextos delimitados, bem definidos, e a definição das relações entre eles.

Assim sendo, a adoção da abordagem de desenvolvimento de micros serviços pode ser uma estratégia vantajosa quando se lida com *software* altamente complexo que abrange diversas funcionalidades. Nessa abordagem, cada funcionalidade pode ser alocada e gerenciada por uma equipe específica, que possui a autonomia para selecionar sua própria linguagem de programação e sistema de gerenciamento de banco de dados, visando abordar eficazmente os desafios relacionados a essa funcionalidade específica.

2.5 Virtualização

Na área da ciência da computação, o termo "virtualização" frequentemente se refere ao processo de transformar uma entidade física em uma construção lógica abstrata (Portnoy, 2012). Ao virtualizar uma entidade, é possível obter uma representação conceitual dela, facilitando assim a utilização mais abstrata e versátil dessa entidade.

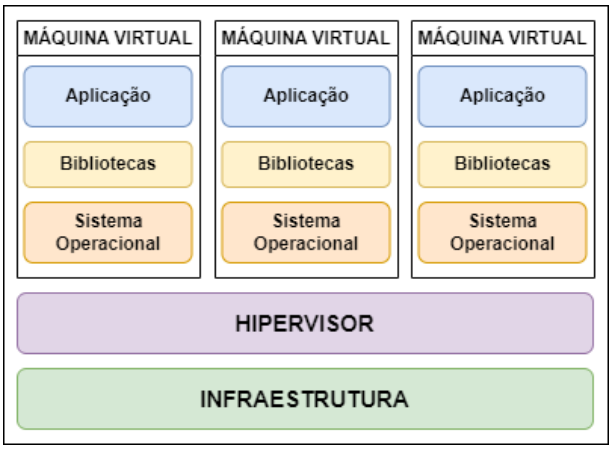
Um dos principais benefícios advindos da adoção de um ambiente virtualizado reside na capacidade de isolamento que essa abordagem proporciona. Nesse contexto, cada programa ou serviço é executado em seu próprio ambiente operacional, onde recursos de *hardware* exclusivos são alocados de maneira dedicada. Adicionalmente, esses ambientes virtuais possuem seus próprios conjuntos de bibliotecas independentes, contribuindo para uma segregação completa e eficaz dos recursos e componentes do sistema.

2.5.1 Maquinas Virtuais

Uma máquina virtual é uma réplica eficiente e isolada da máquina física real, controlada por um Monitor de Máquina Virtual (*VMM*). Segundo Popek e Goldberg (1974), a *VMM* deve oferecer um ambiente de execução de programas praticamente idêntico ao da máquina original,

com apenas pequenas reduções de velocidade, mantendo o controle total sobre os recursos do sistema.

Figura 4 – Estrutura organizacional de máquinas virtuais



Fonte: Elaboração própria (2023).

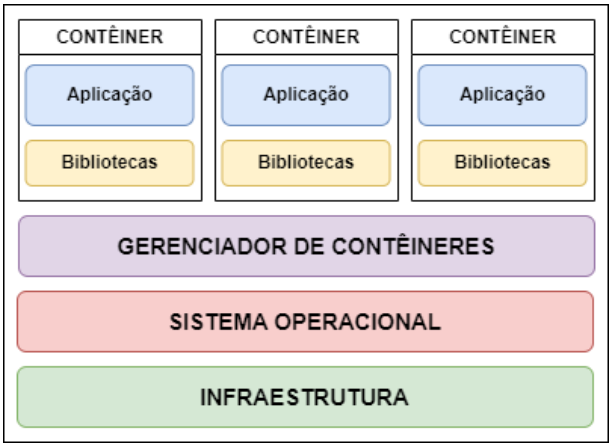
A ilustração retratada na Figura 4 delineia a estrutura fundamental ao funcionamento de uma máquina virtual. Nesse contexto, a infraestrutura serve como base para a configuração da *VMM* ou *hipervisor*, cuja principal atribuição reside na gestão das instâncias individuais de máquinas virtuais. É importante destacar que cada máquina virtual age como uma entidade isolada e autônoma, possuindo seu próprio sistema operacional, bibliotecas e aplicativos. Esse isolamento é uma característica fundamental da virtualização, permitindo que múltiplas *VMs* compartilhem a mesma infraestrutura física sem interferir umas nas outras.

2.5.2 Contêineres

Um contêiner é uma unidade de *software* que empacota aplicativos com todas as suas dependências, garantindo que funcionem de maneira consistente em diversos ambientes de computação (Docker, 2023b). Os contêineres isolam o *software* de seu ambiente e garantem que ele opere de maneira uniforme, independentemente da infraestrutura subjacente.

A representação contida na **Figura 5** expõe a estruturação de um ambiente voltado à utilização de contêineres, em que cada contêiner hospeda uma aplicação em execução. Neste contexto, a responsabilidade primordial recai sobre o Gerenciador de Contêineres, encarregado de administrar as instâncias individuais com o propósito de assegurar seu funcionamento em um ambiente isolado e estável. Uma distinção crucial entre essa abordagem e a utilização de máquinas virtuais reside no compartilhamento do *kernel* do Sistema Operacional. Isso se traduz em um processo de inicialização significativamente mais ágil para cada contêiner, que demanda apenas segundos em comparação com as tradicionais Máquinas Virtuais.

Figura 5 – Estrutura organizacional para contêineres



Fonte: Elaboração própria (2023).

2.6 Trabalhos relacionados

Este trabalho foi desenvolvido para empregar tecnologias em alta no mercado de software *web*, justificando a escolha de ambientes configurados via *Kubernetes* e *Docker*. Além disso, foi criada uma aplicação de complexidade considerável, incorporando recursos de comunicação em tempo real, gerenciamento de banco de dados e processamento e armazenamento de imagens. O intuito foi simular um cenário próximo à realidade. A análise comparativa do tempo de resposta e uso de recursos de *hardware* em diferentes arquiteturas destacou as vantagens de cada funcionalidade do sistema. Esta abordagem difere dos estudos relacionados mencionados nesta seção, os quais adotam metodologias detalhadas nas subseções subsequentes.

2.6.1 A Comparative Review of Microservices and Monolithic Architectures (2018)[4]

A pesquisa conduzida por Al-Debagy e Martinek (2018) aborda o crescente interesse que a arquitetura de microsserviços tem despertado tanto na comunidade acadêmica quanto na indústria. Especificamente, o artigo enfatiza as vantagens proporcionadas pela arquitetura de microsserviços em comparação com a abordagem monolítica e realiza uma análise comparativa de desempenho entre essas duas abordagens.

A metodologia empregada para avaliar o desempenho das duas arquiteturas envolveu a criação de uma aplicação de teste composta por três serviços, seguida da realização de testes de carga e concorrência utilizando a ferramenta *Apache JMeter*. Os resultados obtidos foram analisados com foco no conceito de *throughput*, que representa a capacidade de processamento de um sistema, isto é, a quantidade de trabalho ou dados que o sistema consegue processar em um determinado intervalo de tempo. Além disso, também foram considerados os tempos de resposta da aplicação.

Os resultados dos testes de carga indicaram que a arquitetura monolítica teve um melhor desempenho em termos de *throughput* com um número menor de usuários. No entanto, à medida

[4] Tradução: Uma Revisão Comparativa de Arquiteturas de Microserviços e Monolíticas (2018)

que o número de usuários aumentou, as arquiteturas se aproximaram em termos de desempenho. O teste de concorrência mostrou que a arquitetura monolítica teve um desempenho melhor em termos de *throughput* em comparação com a arquitetura de microsserviços, com uma diferença de 6% em média.

Um terceiro cenário de teste comparou o desempenho de diferentes configurações de aplicativos de microsserviços, incluindo o uso de tecnologias de descoberta de serviços *Consul* e *Eureka*. Os resultados indicaram que a aplicação de microsserviços com *Consul* teve um melhor *throughput* em comparação com a aplicação com *Eureka*.

Em resumo, o artigo conclui que a escolha entre arquitetura de microsserviços e monolítica depende do tamanho e dos requisitos da aplicação.

2.6.2 Análise Comparativa entre Arquitetura Monolítica e de Microsserviços (2017)

Duarte (2017) aborda uma análise comparativa entre duas arquiteturas de *software*, a monolítica e a de microsserviços. O objetivo da pesquisa é avaliar as vantagens e desvantagens de cada uma dessas abordagens por meio da implementação de um sistema de gerenciamento de cinemas em ambas as arquiteturas, utilizando *JavaScript* como linguagem de programação.

No experimento descrito, o autor implementa um sistema de gerenciamento de cinemas em ambas as arquiteturas. Ele compara a estrutura de diretórios, a complexidade da arquitetura de microsserviços e a quantidade de código necessário para cada abordagem. É destacado que a arquitetura de microsserviços exigiu consideravelmente mais esforço de codificação devido à necessidade de criar uma estrutura de diretórios para cada serviço.

A seção de testes apresenta os resultados dos testes de carga realizados com a ferramenta *Apache JMeter*. Os resultados indicam que a arquitetura monolítica teve um desempenho superior em termos de tempo de resposta, enquanto a arquitetura de microsserviços apresentou um maior tempo de comunicação entre os serviços, devido ao uso de requisições *HTTP* internas.

A conclusão do artigo destaca que a arquitetura de microsserviços se destaca em projetos maiores, onde seus benefícios em termos de agilidade no desenvolvimento, liberações de versões mais frequentes e independência de linguagem de programação são mais evidentes. No entanto, o autor reconhece que a comunicação entre os microsserviços por meio de requisições *HTTP* internas pode afetar negativamente o desempenho.

2.6.3 Arquitetura de Micro Serviços: uma Comparação com Sistemas Monolíticos (2017)

O artigo desenvolvido por Amaral e Carvalho (2017), apresenta uma análise comparativa entre duas arquiteturas de *software*: sistemas monolíticos e microsserviços. O estudo visa fornecer informações relevantes para a tomada de decisões arquiteturais em projetos de desenvolvimento de aplicações *web*, particularmente em cenários de larga escala.

A análise realizada neste artigo compreende tanto uma comparação teórica das duas arquiteturas quanto um experimento prático com aplicações implementadas em ambas. O experimento envolveu a coleta e análise de dados de desempenho, usando métricas como vazão,

latência e consumo de recursos, a fim de avaliar qual arquitetura se mostra mais adequada em diferentes contextos.

Os recursos das aplicações foram classificados em três níveis com base na quantidade de serviços que precisam ser acionados para atender a uma requisição: nível 1 não depende de outros serviços, nível 2 requer um serviço adicional, e nível 3 requer dois serviços adicionais. Os principais resultados encontrados foram os seguintes:

- **Arquitetura Monolítica:** demonstrou bom desempenho, especialmente em aplicações de baixa complexidade, onde a escalabilidade não é uma necessidade crítica. A arquitetura monolítica mostrou ter menor latência e um uso de *hardware* mais eficiente em comparação com a arquitetura de microsserviços;
- **Arquitetura de Microsserviços:** revelou-se vantajosa em cenários que exigem escalabilidade, integração de dados e alto isolamento de funcionalidades. Por outro lado, essa arquitetura requer um maior poder de processamento, o que pode influenciar em custos de infraestrutura;

Como conclusão, este artigo fornece informações para profissionais que precisam escolher entre essas duas arquiteturas em seus projetos, no entanto, é importante destacar que o experimento foi realizado em uma aplicação de baixa complexidade.

Neste capítulo, exploramos os conceitos fundamentais essenciais para a compreensão abrangente do trabalho. Isso inclui não apenas a definição das arquiteturas em foco, mas também os princípios subjacentes à virtualização, que serve como a base para o *Docker* e *Kubernetes*. No próximo segmento, nos aprofundaremos na metodologia, detalhando as abordagens e ferramentas que guiarão a análise comparativa entre as arquiteturas monolítica e de microsserviços. O objetivo deste capítulo é fornecer uma compreensão detalhada do processo metodológico utilizado, preparando o terreno para a implementação e testes explorados nos capítulos subsequentes.

3 METODOLOGIA

Neste capítulo, descreve-se a sequência de etapas seguidas na criação da aplicação de teste e na configuração do ambiente de execução. São detalhados os procedimentos, métodos e tecnologias empregados no desenvolvimento da aplicação, na preparação do ambiente de execução e na utilização do *software* para conduzir os testes de carga. Dessa forma, é oferecida uma visão completa das abordagens metodológicas aplicadas ao longo do projeto.

A pesquisa experimental é caracterizada pela intervenção sistemática do pesquisador no ambiente de estudo, com o intuito de provocar alterações controladas e, posteriormente, observar se essas intervenções resultam nos efeitos esperados (Wazlawick, 2009). Nesse contexto, esta pesquisa adota uma abordagem experimental, na qual foram criadas situações específicas de teste e análise para investigar as vantagens e desvantagens das arquiteturas de microsserviços e monolíticas. Essas intervenções planejadas e controladas possibilitaram a coleta de dados objetivos, conforme o método experimental, permitindo uma avaliação precisa e fundamentada das dinâmicas e resultados de cada modelo arquitetural em questão.

Esta pesquisa possui elementos de uma pesquisa Quantitativa, que segundo a análise de Wainer (2007), se destaca pela utilização de técnicas estatísticas, com uma ênfase na comparação de resultados e pela mensuração, normalmente em termos numéricos, de um conjunto restrito de variáveis objetivas.

A fim de analisar as duas arquiteturas de desenvolvimento de aplicações *web* por meio dos testes de carga, foram estabelecidas as seguintes questões principais (QP) de pesquisa:

- **QP1:** a escalabilidade varia significativamente entre a arquitetura monolítica e a arquitetura de microsserviços?
- **QP2:** as arquiteturas monolíticas e de microsserviços apresentam diferenças notáveis quanto à utilização eficaz de recursos de *hardware* e ao consumo de recursos computacionais em situações de alta demanda?
- **QP3:** como a latência e o tempo de resposta de sistemas baseados em arquitetura de microsserviços se comparam aos sistemas monolíticos em situações de carga variável e como essa comparação influencia o desempenho?
- **QP4:** em termos de desempenho, quais são os impactos das comunicações entre componentes em uma arquitetura de microsserviços em comparação com a arquitetura monolítica?

Para possibilitar a implementação e análise comparativa de aplicações *web* nas arquiteturas monolítica e de microsserviços, seguiu-se uma sequência de etapas. Inicialmente, foi realizada uma revisão bibliográfica aprofundada para compreensão dos conceitos e pesquisas relacionadas a essas arquiteturas. Posteriormente, uma aplicação *web* monolítica foi desenvolvida, com foco na criação de uma sala de bate-papo em tempo real. A aplicação foi, então, decomposta em microsserviços independentes. A configuração do ambiente, com a utilização de tecnologias

como *Kubernetes* e *Docker*, desempenhou um papel fundamental para o gerenciamento eficiente da aplicação.

Vale ressaltar que, ao longo do projeto, também foi desenvolvida a camada relacionada ao *front-end* da aplicação, embora essa parte não tenha sido alvo dos testes de carga. A ênfase dos testes concentrou-se nas diferenças de desempenho e eficiência entre as arquiteturas de *Back-end* (monolítica e de microsserviços). Por fim, os testes de carga foram conduzidos para avaliar o desempenho das duas arquiteturas, analisando métricas relevantes, como tempo de resposta e escalabilidade, a fim de responder às questões de pesquisa previamente estabelecidas.

3.1 Pesquisa sobre trabalhos relacionados

Para realizar a revisão bibliográfica com o propósito de identificar artigos relevantes, adotou-se a metodologia sistêmica proposta por Wazlawick (2009). Essencialmente, essa abordagem preconiza a identificação de fontes pertinentes, a compilação de artigos recentes, a seleção dos relacionados, a categorização destes, a leitura minuciosa dos de alta relevância com a elaboração de fichas de leitura, a anotação de outros artigos mencionados e a documentação completa de todo o processo. Essa metodologia proporciona uma avaliação crítica do material coletado, determinando se este é suficiente para atender aos objetivos da pesquisa ou se é necessário expandir a busca, conforme exigido pelo estudo.

Para conduzir a pesquisa, foi inicialmente estabelecido como critério a busca por artigos redigidos em língua portuguesa do Brasil na plataforma *Google* Acadêmico, no período compreendido entre os anos de 2017 e 2023. A escolha desse espaço de tempo se deu pela indisponibilidade de artigos nos últimos cinco anos. A pesquisa foi orientada por uma *string* de busca específica, a saber: “monolítico AND (“microsserviços” OR “micro serviços”) AND comparação AND JMeter -site:ieeexplore.ieee.org -migração -IoT”.

Os resultados da pesquisa abrangem artigos que abordam a comparação de desempenho entre arquiteturas monolíticas e aquelas baseadas em microsserviços. Especificamente, esses estudos se concentram na avaliação do tempo de resposta de ambas as abordagens em um cenário de alta demanda de usuários, principalmente através da realização de testes de carga. Além disso, foram aplicadas restrições específicas, como a exclusão de fontes relacionadas ao *IEEE Xplore* nos resultados obtidos por meio do *Google* Acadêmico. Também foram excluídos artigos que tratam da migração de sistemas entre diferentes arquiteturas e projetos relacionados à Internet das Coisas (IoT).

Conforme descrito na metodologia, foram obtidos os resultados que correspondem estritamente à *string* de busca central da pesquisa, conforme evidenciado no **Quadro 1**. Nesta etapa do estudo, foram identificados doze resultados que estão em total concordância com os critérios estabelecidos anteriormente. Uma análise meticulosa dos títulos e resumos desses resultados foi realizada, visando uma avaliação criteriosa. Como resultado desse processo de seleção, foram eleitos os artigos intitulados *Arquitetura de microsserviços: uma comparação com sistemas monolíticos* e *Análise comparativa entre arquitetura monolítica e de microsserviços*.

Quadro 1 – Resultados de busca de artigos na plataforma *Google* Acadêmico

Título	Ano de Publicação
Arquitetura de microsserviços: uma comparação com sistemas monolíticos	2017
Análise comparativa entre arquitetura monolítica e de microsserviços	2017
Padrões para produção de aplicações utilizando microsserviços	2021
Análise da Eficiência da Transferência de Dados em uma Rede de Microserviços–Proposta de Comparação de Desempenho entre REST E GRPC	2023
Impacto dos padrões arquiteturais de Micro Serviço e Monolítico no desenvolvimento de softwares	2017
Servindo modelos de machine learning com uma arquitetura baseada em serverless	2020
Implementação da arquitetura de microsserviços para backend de um aplicativo de supermercado	2022
Serviço de Helpdesk automático	2020
Serviço de Helpdesk Automático	2020
Sistema de recomendação para uma plataforma de comércio eletrônico	2019
Event-driven integrado com Enterprise Service Bus	2018
Desenvolvimento de *software* usando Angular e Node para assistência social	2018

Fonte: Google (2023).

A pesquisa foi então conduzida em uma plataforma diferente, a *IEEE Xplore*, com critérios mais específicos. Optou-se por restringir os resultados àqueles em língua inglesa, dado que não foram encontrados artigos em língua portuguesa que atendessem às necessidades da pesquisa. O período de análise foi delimitado de 2017 a 2023.

Quadro 2 – Resultados de busca de artigos na plataforma *IEEE Xplore*

Título	Ano de Publicação
A Comparative Review of Microservices and Monolithic Architectures	2018
Comparison of Runtime Testing Tools for Microservices	2019
Latency and RAM Usage Comparison of Advanced and Lightweight Service Mesh	2022
Design of a Geographic Information System for Forest and Land Fires Based on a Real-Time Database on Microservices Infrastructure	2022
Comparison of Different CI/CD Tools Integrated with Cloud Platform	2019
Orchestrating the Resilience of Cloud Microservices Using Task-Based Reliability and Dynamic Costing	2022
Application of mixed distributed *software* architectures for social-productive projects management in Peru	2017

Fonte: IEEE (2023).

Nesse contexto, a pesquisa foi realizada novamente com base na *string* de busca anterior-

mente mencionada, porém, com modificações apropriadas. A nova *string* de busca utilizada foi: "monolithic AND microservices AND (comparison OR comparative) AND Test". Esta alteração se justificou pela necessidade de utilizar termos em inglês e pela necessidade de generalizar a ferramenta de teste, uma vez que o *Apache JMeter* não estava especificado em todos os resumos dos artigos disponíveis na plataforma *IEEE Xplore*.

Como apresentado no **Quadro 2**, foram identificados sete artigos, nos quais foram analisados minuciosamente os títulos e resumos. O único artigo que se mostrou relevante para o tema da comparação entre as arquiteturas foi intitulado *A Comparative Review of Microservices and Monolithic Architectures*[1].

3.2 Metodologia de desenvolvimento

Para desenvolver o sistema de bate-papo, uma sequência de passos foi inicialmente estabelecida. Essa sequência fundamentou-se no *framework Scrum*[2], com a delineação de requisitos e artefatos específicos destinados a orientar o desenvolvimento em ambos os contextos arquiteturais.

Todas as iterações durante o desenvolvimento foram documentadas em um repositório no *GitHub*, plataforma de compartilhamento *on-line* de projetos de *software*, visando o controle efetivo do versionamento da aplicação. Vale ressaltar que as duas aplicações foram organizadas em diretórios distintos dentro do mesmo repositório, juntamente com os demais arquivos desenvolvidos ao longo deste trabalho.

O processo inicia-se com a abstração conceitual da aplicação como um todo, destacando as interações fundamentais entre usuários e sistema por meio de um diagrama de caso de uso. A partir desse diagrama, o escopo da aplicação é definido com base nos requisitos, abrangendo tanto os aspectos funcionais quanto os não funcionais. Isso estabelece metas para ambas as arquiteturas: monolítica e de microsserviços do sistema.

Com os requisitos delineados, o próximo passo é a criação do modelo de dados da aplicação, que inicialmente se concentra no esquema do banco de dados para armazenar as informações do sistema. Com o modelo estabelecido, é possível definir os relacionamentos entre os dados e criar o Diagrama de Classe principal da aplicação, responsável pelo gerenciamento dos dados dos usuários e das mensagens enviadas e recebidas na aplicação. Além disso, estabeleceram-se metas para a camada de segurança da aplicação, delineando uma abstração do fluxo de autenticação que deve ser seguido por ambas as arquiteturas.

Ressalta-se a importância de observar que esses métodos são comuns ao desenvolvimento da aplicação em ambos os contextos arquiteturais. A aplicação baseada em microsserviços,

[1] Tradução: Uma Revisão Comparativa de Arquiteturas de Microserviços e Monolíticas

[2] O *Scrum* é um *framework* que capacita equipes a se auto-organizarem para alcançar metas com eficiência. Com reuniões, ferramentas e funções específicas, promove autogerenciamento, aprendizado contínuo e adaptação a mudanças, sendo amplamente utilizado na solução econômica e sustentável de problemas complexos na área de *software* (Amazon, 2023).

em particular, realiza adaptações principalmente nos módulos e entidades do banco de dados, buscando assegurar a independência de todos os serviços na infraestrutura.

3.3 Desenvolvimento da aplicação monolítica

A aplicação em foco representa uma plataforma de bate-papo (chat) concebida para facilitar a comunicação em tempo real entre os seus usuários. Nesta plataforma, os usuários têm a capacidade de realizar cadastro, configurar perfis personalizados e participar de interações em salas de bate-papo, as quais podem ser públicas ou privadas. No âmbito dessas salas, os usuários podem realizar a troca de mensagens, engajar-se em discussões em grupo, receber notificações relacionadas a novas mensagens e proteger a privacidade dos seus dados pessoais por meio de autenticação por senha. Desta forma, a aplicação pode oferecer uma experiência de bate-papo confiável para seus usuários.

Decidiu-se iniciar a implementação da aplicação que servirá como objeto de estudo nos testes com a primeira etapa de desenvolvimento de um sistema monolítico. Posteriormente, este sistema será decomposto em microsserviços. Para ser possível interações em tempo real entre os usuários, faz-se necessário a gestão de sessões *WebSocket*. Para as demais funcionalidades, é necessário a exposição de serviços via *API*, a implementação de camadas de segurança e a realização de operações de leitura e escrita de arquivos de imagem. O propósito dessa abordagem é criar uma aplicação que simule o mais fielmente possível um cenário real.

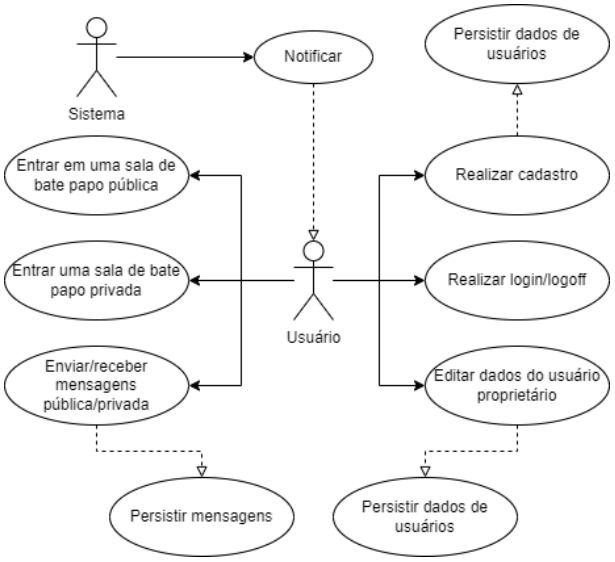
3.3.1 Requisitos do sistema

No início do desenvolvimento da aplicação, os requisitos do sistema foram estabelecidos através da criação de um diagrama de caso de uso, para delinear, em um nível mais abstrato, o fluxo principal de utilização da aplicação. Dado que se trata de uma aplicação de bate-papo, foi definido um único perfil de sistema, representado pelo usuário. Este usuário tem a capacidade de realizar diversas ações dentro do sistema, conforme ilustrado na **Figura 6**.

Na **Figura 6**, é apresentado o fluxo de ações executadas no sistema por meio do Diagrama de Caso de Uso. Inicialmente, um processo de cadastro de usuário é disponibilizado, permitindo que os dados dos usuários sejam registrados e armazenados em um banco de dados. Além disso, é viabilizada a funcionalidade de edição desses dados em um momento posterior. A ação de "*login*" e "*logout*", que se refere a entrar e sair do sistema por meio de autenticação, é também contemplada como parte das operações do usuário. Adicionalmente, o usuário tem a capacidade de ingressar em salas de bate-papo, sejam elas públicas ou privadas, bem como enviar e receber mensagens nesses canais. Essas mensagens são armazenadas pelo sistema no banco de dados. Além disso, o sistema está encarregado de notificar os usuários sobre as mensagens recebidas.

Os requisitos funcionais listados no **Quadro 3** representam as principais funcionalidades da aplicação de bate-papo, incluindo registro de usuários, edição de perfis, *login*, troca de mensagens em tempo real e notificação de mensagens recebidas. Esses requisitos formam a base

Figura 6 – Diagrama de caso de uso



Fonte: Elaboração própria (2023).

fundamental para o desenvolvimento da aplicação, garantindo uma experiência de bate-papo completa e eficaz para os usuários.

Quadro 3 – Requisitos funcionais da aplicação

Requisito	Descrição
Cadastro de Usuário	Os usuários podem se registrar na aplicação, fornecendo seus dados pessoais.
Edição de Dados do Usuário	Os usuários têm a capacidade de editar as informações do seu perfil.
Login e Logout	O sistema permite que os usuários entrem e saiam da aplicação.
Conversas privadas	Os usuários podem criar, ingressar e se comunicar em salas de bate-papo privadas.
Envio e Recebimento de Mensagens Públicas	É possível se comunicar com os usuários conectados na sala pública.
Notificação de Mensagens	O sistema notifica os usuários sobre novas mensagens recebidas.
Persistência de Dados	Todas as informações de usuário e mensagens são armazenadas de forma segura em um banco de dados.

Fonte: Elaboração própria (2023).

O **Quadro 4** destaca requisitos não funcionais essenciais da aplicação, enfocando escalabilidade, segurança, disponibilidade e desempenho. Esses requisitos garantem que a aplicação seja dimensionável para muitos usuários, mantenha a segurança dos dados, esteja sempre disponível e seja responsiva, mesmo durante picos de uso.

Quadro 4 – Requisitos não funcionais da aplicação

Requisito	Descrição
Escalabilidade	O sistema deve ser escalável para lidar com um grande número de usuários simultâneos.
Segurança dos Dados	Os dados dos usuários e mensagens devem ser protegidos para garantir a privacidade.
Disponibilidade	A aplicação deve estar disponível 24 horas por dia, 7 dias por semana, com um mínimo de tempo de inatividade planejado para manutenção.
Desempenho	A aplicação deve ser responsiva e eficiente, garantindo tempos de resposta rápidos mesmo durante picos de uso.

Fonte: Elaboração própria (2023).

3.3.2 Definição do esquema de banco de dados

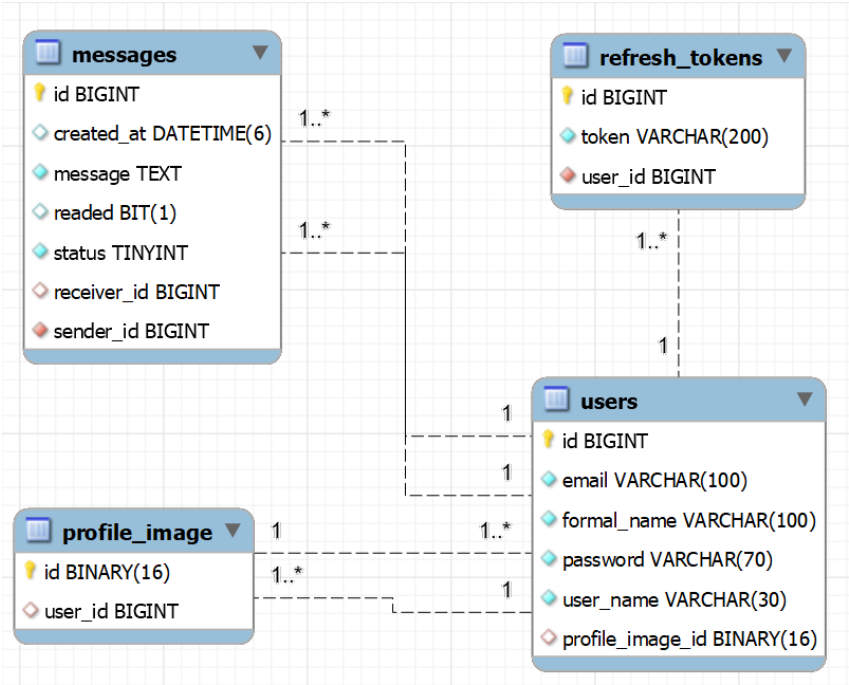
Esta seção apresenta o esquema de banco de dados do sistema, no qual o sistema de gerenciamento de banco de dados *MySQL*[3] foi escolhido como a solução para armazenar as tabelas e atributos fundamentais necessários para o funcionamento do sistema. Serão detalhadas as estruturas de armazenamento de mensagens, informações de usuário, imagens de perfil e *tokens* de autenticação.

Cada componente desempenha um papel crucial na integridade dos dados e na otimização das operações do sistema. A tabela *messages* é responsável por armazenar as mensagens trocadas entre os usuários do sistema. Cada registro nessa tabela representa uma mensagem específica. Os atributos da tabela incluem:

- *id*: um identificador único e autoincrementado para cada mensagem.
- *created_at*: uma marca de data e hora que registra o momento em que a mensagem foi criada.
- *message*: o conteúdo da mensagem, armazenado como texto.
- *readed*: um indicador que registra se a mensagem foi lida (1 para lida, 0 para não lida).
- *status*: um valor numérico que define o status da mensagem.
- *receiver_id*: uma chave estrangeira que se relaciona com a tabela *users*, indicando o destinatário da mensagem.
- *sender_id*: uma chave estrangeira que se relaciona com a tabela *users*, indicando o remetente da mensagem.

[3] O *MySQL* é um dos sistemas de gerenciamento de banco de dados SQL de código aberto mais amplamente utilizados, sendo desenvolvido, distribuído e suportado pela *Oracle Corporation*. (Oracle, 2023a)

Figura 7 – Modelo relacional do banco de dados monolítico



Fonte: Elaboração própria (2023).

A tabela *profile_image* é dedicada ao armazenamento dos metadados das imagens de perfil dos usuários. Cada registro nesta tabela representa uma imagem de perfil. Os atributos da tabela incluem:

- *id*: um identificador único para cada imagem de perfil, armazenado no formato binário, pois esta é gerada através do Identificador Único Universal (Universally Unique Identifier - UUID).
- *user_id*: uma chave estrangeira que se relaciona com a tabela *users*, indicando a qual usuário pertence à imagem de perfil.

A tabela *refresh_tokens* tem como principal finalidade o armazenamento de *tokens* de atualização utilizados na autenticação dos usuários. Nesse contexto, apenas um *token* de atualização ativo é mantido para cada usuário, o que permite a revogação da sua validade. Esse processo é viabilizado por meio da verificação na tabela de *refresh_tokens* antes de gerar um novo *token* de acesso com base no *token* de atualização. Cada registro nessa tabela representa um *token* de atualização único associado a um usuário específico. Os atributos da tabela *refresh_tokens* incluem:

- *id*: um identificador único incrementado automaticamente para cada token.
- *token*: o próprio token de atualização, armazenado como uma sequência de caracteres.
- *user_id*: uma chave estrangeira que se relaciona com a tabela ‘users‘, indicando a qual usuário o token pertence.

A tabela *users* desempenha a função de armazenar informações detalhadas referentes aos usuários registrados no sistema. Cada registro nesta tabela representa de maneira distinta um usuário específico. Dentre os atributos que compõem a tabela, destacam-se:

- *id*: um identificador único e autoincrementado para cada usuário.
- *email*: o endereço de e-mail do usuário, armazenado como texto e é exclusivo.
- *formal_name*: o nome formal do usuário, geralmente incluindo nome e sobrenome.
- *password*: A senha criptografada do usuário, armazenada de forma segura.
- *user_name*: o nome de usuário do usuário, que é exclusivo.
- *profile_image_id*: uma chave estrangeira que se relaciona com a tabela ‘profile_image‘, indicando a imagem de perfil do usuário.

Com essa abordagem, é possível assegurar as funcionalidades de persistência de dados, o que desempenha um papel crítico na integridade das informações e na operação eficaz do sistema. Isso garante que os dados permaneçam acessíveis e consistentes ao longo do tempo, mesmo em caso de reinicialização ou interrupções do sistema.

3.3.3 Estrutura do Back-end monolítico

Para a implementação da lógica do *back-end* da aplicação, incluindo o mapeamento de classes para o banco de dados e definição de rotas para as *APIs*, foi escolhido o *framework* conhecido como *Spring Boot*.

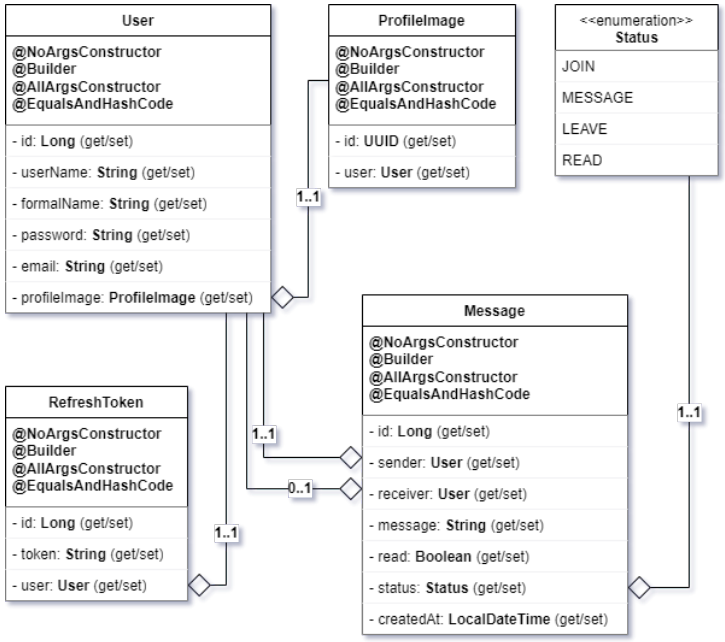
O *Spring Boot* simplifica o desenvolvimento de aplicativos independentes, baseados na estrutura *Spring*, adequados para ambientes de produção, permitindo que sejam executados sem complicações (Spring, 2023a).

O *Spring Boot* oferece um conjunto de pré-configurações que agilizam o início do desenvolvimento de aplicativos. Para utilizar diversas bibliotecas disponíveis no *Spring Starter*, uma plataforma destinada ao início de projetos com essa estrutura, muitas vezes é suficiente incluir apenas a dependência correspondente no gerenciador de dependências do projeto e, ocasionalmente, fornecer configurações simplificadas para iniciar a aplicação. Esse aspecto é uma característica distintiva do *Spring Boot* que contribui para uma produtividade aprimorada no processo de desenvolvimento.

O Diagrama de Classes, ilustrado na Figura 8, oferece uma representação visual dos relacionamentos entre as classes que compõem a camada de lógica de negócios da aplicação. No contexto do *framework Spring Boot*, a arquitetura padrão adotada é o Modelo-Visão-Controlador (MVC), que pressupõe a divisão do programa em três camadas distintas: Modelo, Visualização e Controladores.

Este diagrama, conforme apresentado na **Figura 8**, concentra-se na camada de modelo principal da aplicação. Nessa camada, são estabelecidas as relações entre os dados do usuário, as

Figura 8 – Diagrama de classe monolítico



Fonte: Elaboração própria (2023).

mensagens trocadas e os *tokens* de acesso gerados pelo sistema. Essas interações são cruciais para o funcionamento eficiente da aplicação, uma vez que formam a base dos processos de autenticação, armazenamento de informações e comunicação em tempo real. O MVC é um paradigma amplamente adotado que facilita a organização e a manutenção de sistemas, garantindo a separação de responsabilidades entre as diferentes partes do *software*. Dessa forma, o diagrama de classes fornece uma visão estrutural das entidades e como elas se relacionam, contribuindo para a compreensão do funcionamento da aplicação em um nível mais abstrato.

Para que a lógica da aplicação seja integrada com sucesso no ambiente do *Spring Boot*, foi necessário criar *Beans* personalizados que se adequassem à estrutura de dados desenvolvida. Conforme a documentação do Spring (2023b), os *Beans* são componentes registrados e controlados pelo *Spring Framework*, que podem ser configurados e injetados na aplicação.

Inicialmente, desenvolveu-se o mapeamento objeto-relacional para facilitar a comunicação entre o banco de dados MySQL e as classes *Java*. Esse mapeamento foi realizado por meio de *Beans* disponibilizados pela especificação *Java Persistence API* (JPA), que permite definir os atributos, seus tipos e os relacionamentos entre objetos usando anotações.

Os repositórios de dados foram configurados com base em *Beans* estendidos por meio da interface *JpaRepository*. Ao fornecer o tipo de dados e o tipo de chave primária da entidade definida no banco de dados, o *Spring Boot* gera dinamicamente uma implementação da interface estendida, que inclui métodos para realizar operações básicas de manipulação de dados, comumente conhecidas como CRUD (*Create, Read, Update, Delete*) – responsáveis por salvar, ler, atualizar e excluir registros no banco de dados.

Conforme o modelo da estrutura de dados estabelecido, a aplicação exigiu a criação de

cinco repositórios de dados distintos:

- *LocalImageRepository*: este repositório foi desenvolvido para armazenar imagens de usuários no armazenamento local do servidor. Utiliza o ID da entidade *ProfileImage* do banco de dados como nome do arquivo.
- *MessageRepository*: responsável pela manipulação das entidades do tipo *Message* na aplicação, este repositório gerencia as mensagens trocadas entre os usuários.
- *ProfileImageRepository*: focado na tarefa de armazenar metadados de imagens, o *ProfileImageRepository* lida especificamente com as entidades *ProfileImage* na aplicação.
- *TokenRepository*: este repositório é responsável por armazenar os tokens de atualização da aplicação, garantindo o gerenciamento dessas informações críticas.
- *UserRepository*: o *UserRepository* é encarregado de manipular as entidades do tipo *User* na aplicação, gerenciando os dados dos usuários registrados no sistema.

Durante o processo de desenvolvimento da aplicação, houve a necessidade de definir métodos adicionais para os repositórios de dados. Essas especificações foram realizadas usando a anotação "*@Query*" disponibilizada pelo JPA. Através dessa anotação, é possível formular instruções de consulta no formato *Java Persistence Query Language* (JPQL). O JPQL é uma linguagem de consulta orientada a objetos utilizada em conjunto com o JPA, permitindo a personalização flexível das consultas conforme as necessidades específicas da aplicação. Essa abordagem possibilita a criação de consultas sob medida para atender aos requisitos do sistema.

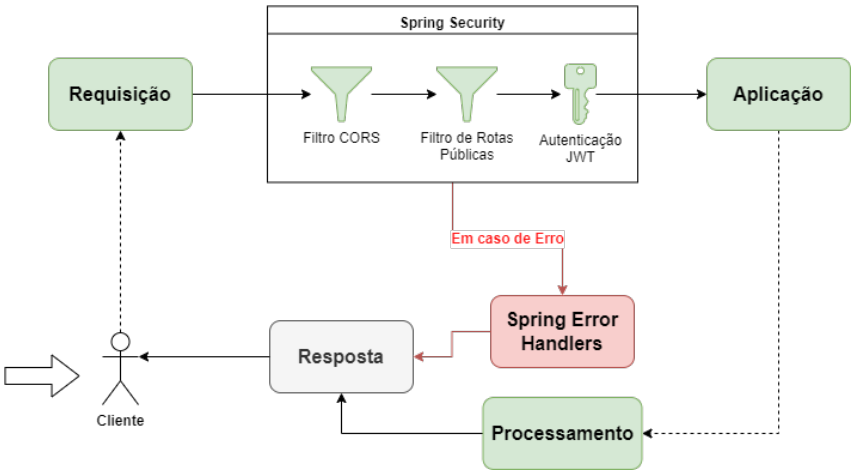
Para continuar a aderir o padrão MVC no contexto do *Spring Boot*, foram criados os *beans* relacionados aos serviços da aplicação. Esses serviços desempenham um papel crucial na execução das regras de negócios, empregando as classes da camada de modelo e interagindo com os repositórios de dados. Eles se comunicam com os controladores por meio de Objetos de Transferência de Dados (DTO), permitindo uma organização eficiente da lógica da aplicação. Os *Beans* relacionados aos serviços criados são:

- *AuthService*: este serviço é encarregado de gerenciar a autenticação inicial dos usuários. Ele realiza uma verificação das credenciais do usuário, incluindo o nome de usuário e a senha, a fim de conceder ou negar o acesso. Para que um usuário seja autenticado com sucesso, é necessário que ele esteja ativo e que as credenciais fornecidas sejam válidas.
- *ImageService*: encarregado de executar operações relativas ao armazenamento e gerenciamento de imagens, suas responsabilidades abrangem a validação da integridade e do formato das imagens antes de proceder ao armazenamento.
- *JwtService*: responsável pela geração e gerenciamento dos *tokens* de acesso *JSON Web Token* (JWT) usados na autorização de solicitações subsequentes do usuário. Após a

autenticação bem-sucedida com o *AuthService*, o *JwtService* emite um *token* de acesso, que contém informações sobre a identidade do usuário. Esse *token* é assinado digitalmente e, assim, pode ser usado para verificar a identidade do usuário em cada solicitação subsequente, sem a necessidade de consulta do banco de dados a cada vez.

- *MessageService*: responsável por tratar operações referentes ao processamento das mensagens trocadas entre os usuários.
- *RefreshTokenService*: lida com a renovação dos *token* de acesso. Os *token* de acesso costumam ter um período de validade limitado por motivos de segurança. Quando um *token* de acesso expira, o *RefreshTokenService* pode ser usado para solicitar um novo *token* de acesso sem a necessidade de fazer login novamente. Os *token* de atualização, geralmente mais seguros, podem ser usados para obter um novo *token* de acesso. Isso é útil para manter a sessão do usuário ativa e melhorar a experiência do usuário, sem comprometer a segurança.
- *UserService*: responsável pela gestão de operações relacionadas aos usuários do sistema.

Figura 9 – Fluxo de segurança monolítico



Fonte: Elaboração própria (2023).

A **Figura 9** ilustra o fluxo de segurança utilizado na aplicação através do *Spring Security*, uma biblioteca encarregada da configuração da segurança em aplicações *web* dentro do *framework Spring Boot*, foi essencial criar um *bean* por meio da classe *JwtSecurityConfig*. Este *bean* é responsável por definir configurações que desempenham um papel crítico na garantia da segurança da aplicação. Tais configurações estão encarregadas de gerenciar tanto a autenticação quanto a autorização dos usuários. Elas têm como base a utilização de *tokens* JWT para proteger as rotas e recursos da aplicação.

Portanto, implementam-se políticas de segurança que abrangem a autenticação de usuários, utilização de criptografia para senhas, configuração de um filtro JWT para proteção de rotas específicas e o tratamento de exceções relacionadas à segurança. Adicionalmente, é importante

mencionar que o sistema inclui um filtro CORS para lidar com a segurança e as permissões de requisições vindas de diferentes origens, garantindo um ambiente seguro e acessível.

No contexto das rotas públicas da aplicação, essas configurações abrangem a autorização de acesso a recursos como autenticação (“/auth”), registro de usuários (“/signup”), renovação de *tokens* (“/refresh_token”), saída do sistema (“/quit”), bem como as rotas relacionadas a *WebSocket* (“/ws”, “/ws/**”). Tais rotas foram projetadas para serem acessíveis a qualquer usuário, não exigindo autenticação, de modo a garantir que a experiência de registro, autenticação e comunicação em tempo real esteja disponível de forma aberta a todos.

A definição de rotas públicas relacionadas as conexões *WebSocket* se deu pela necessidade de criar a classe *WebSocketConnectValidation*, pois não foi possível configurar o fluxo de autenticação dos *endpoints WebSocket* através do *Spring Security* durante o desenvolvimento da aplicação. Embora o *Spring Security* disponha de filtros padrão para autenticar solicitações *HTTP* convencionais, a autenticação de conexões *WebSocket* requer uma abordagem mais específica, algo que não foi possível identificar com facilidade em exemplos de implementação na documentação.

Por esse motivo, a classe *WebSocketConnectValidation* foi desenvolvida para atender às exigências de autenticação e validação de conexões *WebSocket*. Ela desempenha o papel de um interceptador personalizado, proporcionando um controle mais abrangente sobre a autenticação e a autorização das conexões *WebSocket*.

Como última camada de implementação, seguindo o padrão MVC, foi realizado o mapeamento de rotas para recursos da aplicação *web* por meio dos Controladores. No *Spring Boot*, esses controladores são definidos como *beans*, o que é alcançado por meio da criação de uma classe anotada com a anotação “*@Controller*”, mas, pela aplicação seguir os princípios das *APIs REST*, optou-se pela “*@RestController*”.

O controlador *AuthRestController* é responsável pela autenticação de usuários e pela gestão de *tokens* de autenticação na aplicação. Ele oferece rotas para criar novos usuários, autenticar usuários existentes, renovar *tokens* de autenticação e permitir que os usuários saiam do sistema.

- **/signup**: criar um novo usuário.
- **/auth**: autenticar um usuário.
- **/refresh_token**: atualizar o *token* de autenticação.
- **/quit**: sair do sistema (revogar *token* de atualização).

O controlador *ChatController* lida com a comunicação em tempo real dentro da aplicação, permitindo que os usuários enviem mensagens públicas e privadas. Ele trata o recebimento e envio de mensagens através do protocolo *WebSocket*, garantindo que as mensagens privadas alcancem seus destinatários corretos e que as mensagens públicas sejam distribuídas na sala de chat principal. Além disso, lida com exceções e erros relacionados à entrega de mensagens.

- **/ws/**: permite que os clientes, como navegadores *web*, estabeleçam uma conexão *WebSocket* com o servidor. Isso possibilita a comunicação em tempo real e a troca de mensagens eficiente.
- **/app/private-message**: tópico para enviar mensagem para um usuário específico.
- **/app/message**: tópico para enviar mensagens ao bate-papo público.
- **/chatroom/public**: tópico para receber mensagens do bate-papo público.
- **/user/{username}/private**: tópico para receber mensagens privadas.
- **/user/{username}/errors**: tópico para receber mensagens relacionadas a erros durante a conexão *Websocket*.

A rota de conexão por meio do protocolo *WebSocket* usando STOMP possui seu próprio mecanismo de segurança. Antes de permitir a conexão de um novo usuário, ocorre uma verificação da validade do *token* de autenticação. Caso o *token* seja inválido, a conexão é recusada. O mesmo procedimento se aplica à inscrição em tópicos, sejam eles públicos ou privados, conforme descrito nas rotas mencionadas anteriormente. Se um usuário não possuir as devidas permissões, a conexão será interrompida.

O controlador *MessagesRestController* é responsável pela recuperação de mensagens privadas e públicas dos usuários. Ele oferece rotas para buscar mensagens privadas por destinatário, mensagens públicas, usuários com os quais se conversou e contagem de mensagens não lidas. Além disso, fornece funcionalidades para marcar mensagens como lidas e paginar as mensagens, garantindo que os usuários possam acessar seu histórico de mensagens de forma eficiente.

- **/messages/retrieve_messages/by/{receiver}**: buscar mensagens privadas por destinatário.
- **/messages/retrieve_messages/by/{receiver}/in/page/{page}/size/{size}**: buscar mensagens privadas por destinatário com paginação.
- **/messages/retrieve_messages/by/public**: buscar mensagens públicas.
- **/messages/retrieve_messages/by/public/in/page/{page}/size/{size}**: Buscar mensagens públicas com paginação
- **/messages/retrieve_users/talked**: buscar usuários com os quais se conversou.
- **/messages/retrieve_count/by/{sender}**: contar mensagens não lidas de um remetente específico.
- **/messages/retrieve_count/total**: contar o total de mensagens não lidas.
- **/messages/mark_messages_as_read/{sender}**: marcar todas as mensagens não lidas como lidas por um remetente.

O controlador *UserRestController* lida com operações relacionadas aos perfis dos usuários. Ele oferece funcionalidades para carregar, atualizar e buscar informações de perfil de usuário, incluindo imagens de perfil, nome formal, senha e e-mail. Também permite que os usuários realizem buscas por outros usuários com base em critérios como nome de usuário ou nome formal. Esse controlador desempenha um papel fundamental na gestão de informações de perfil de usuário na aplicação.

- **/users/upload_profile_image**: enviar imagem de perfil do usuário.
- **/users/retrieve_profile_image/{username}**: obter imagem de perfil do usuário.
- **/users/retrieve_profile_info/{username}**: obter informações públicas de perfil do usuário.
- **/users/retrieve_profile_info**: obter informações de perfil do próprio usuário.
- **/users/edit/formalname**: atualizar o nome formal do usuário.
- **/users/edit/password**: atualizar a senha do usuário.
- **/users/edit/email**: atualizar o e-mail do usuário.
- **/users/find/{query}**: encontrar usuários por nome de usuário ou nome formal.

3.4 Decomposição da aplicação monolítica para microsserviços

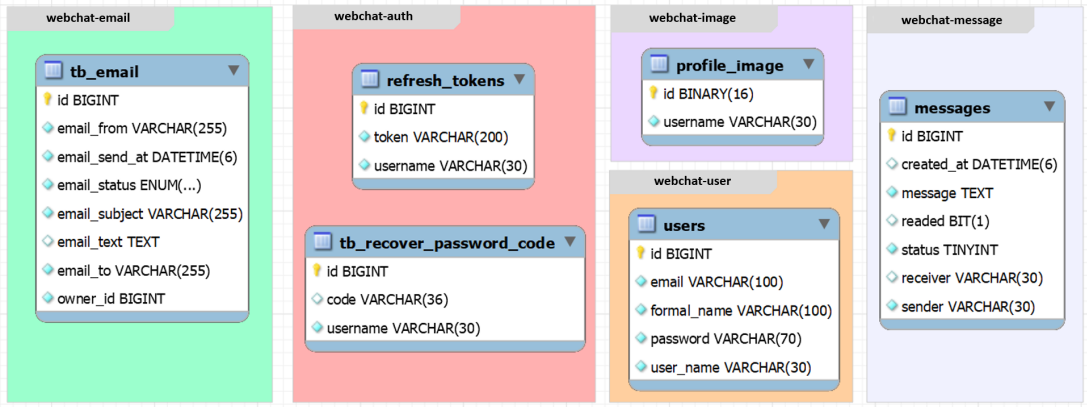
Para realizar uma análise comparativa entre as arquiteturas monolítica e de microsserviços, iniciou-se o processo de decomposição da aplicação monolítica em microsserviços. De forma congruente com a aplicação monolítica, optou-se por manter a linguagem *Java* e o *framework Spring Boot* junto do banco de dados MySQL, uma vez que esse ecossistema também oferece uma estrutura adequada para o desenvolvimento de microsserviços.

3.4.1 Reestruturação do esquema de banco de dados

A abordagem de decomposição teve como principal critério a estrutura do banco de dados, já que a decomposição deste também se fez necessária. Cada microsserviço passou a operar com seu próprio banco de dados dedicado, o que implicou na divisão dos dados. Essa decisão foi tomada com base na necessidade de isolamento e independência de dados entre os diferentes microsserviços, garantindo que cada serviço tenha controle total sobre seu próprio esquema e operações de banco de dados. Isso contribui para a escalabilidade, manutenção e implantação independentes de cada componente do sistema.

A Figura 10 apresenta os bancos de dados estabelecidos para cada serviço, cada um com suas respectivas tabelas. Inicialmente, foi desenvolvida uma funcionalidade de recuperação de senha para lidar com situações em que os usuários esquecessem suas senhas. Devido a essa implementação, um banco de dados relacionado a e-mails foi criado, e uma tabela adicional foi

Figura 10 – Estrutura dos bancos de dados para microsserviços



Fonte: Elaboração própria (2023).

associada ao banco de dados *webchat-auth* para armazenar códigos de recuperação enviados aos endereços de e-mail dos usuários.

Entretanto, é importante ressaltar que essa funcionalidade foi desenvolvida, mas posteriormente descartada durante os testes. Isso se deve ao fato de que o serviço de e-mail utilizado impunha restrições quanto ao número de mensagens que poderiam ser enviadas, o que prejudicaria a condução dos testes. Portanto, essa funcionalidade não está desenvolvida no contexto do sistema monolítico.

3.4.2 Reestruturação do Back-end

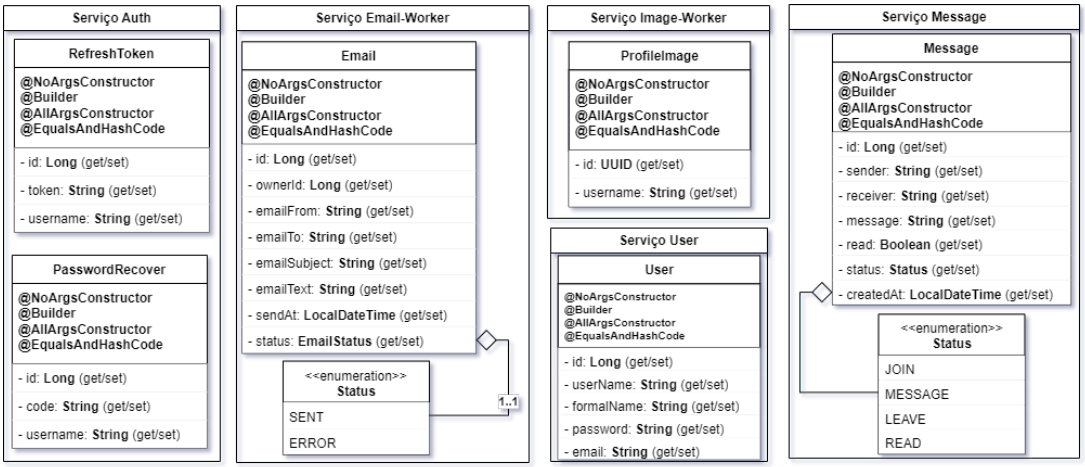
Após a separação do banco de dados para se adequar aos microsserviços, foi necessário fazer o mesmo com a estrutura de dados das regras de negócio da aplicação. Não houve mudanças na forma de funcionamento, como explicado na seção 3.3 e ilustrado na **Figura 8**, mas foi necessário adequar alguns aspectos, tendo em vista que o relacionamento entre algumas entidades foi removido para desacoplamento do sistema.

A Figura 11 apresenta a separação das entidades com base nos microsserviços, organizando-as conforme o contexto de suas funcionalidades. Essa separação implicou na remoção dos relacionamentos entre as tabelas, uma vez que cada banco de dados opera de forma independente. Para manter a integridade das regras de negócio relacionadas aos usuários, optou-se por substituir a entidade *User* nas demais por uma cadeia de caracteres que representa exclusivamente o nome do usuário (*username*), considerando que esse atributo é único e suficiente para manter a coesão.

As rotas da aplicação se mantiveram as mesmas, com exceção para duas rotas exclusivas para recuperação de senha, sendo elas:

- **/recover_password**: realizar solicitação de recuperação de senha.
- **/recover_password/validate/{code}**: link enviado para o e-mail cadastrado do usuário contendo o código de validação para criação de outra senha.

Figura 11 – Estrutura de dados dos microsserviços



Fonte: Elaboração própria (2023).

Essas rotas não foram empregadas durante os testes finais, pois dependem de um serviço de e-mail que possui limitações em sua versão gratuita, o que poderia restringir o teste adequado. No entanto, os Controladores, Serviços e Repositórios de dados definidos na aplicação monolítica permaneceram sem alterações significativas, sendo o único ajuste necessário a mudança de contexto de execução para o respectivo microsserviço correspondente.

É relevante ressaltar que alguns tópicos relacionadas ao serviço *WebSocket* foram modificadas para garantir a compatibilidade com o *RabbitMQ*. Estas incluem:

- **/topic/private.{username}**: utilizada para receber mensagens privadas.
- **/topic/chatroom.public**: responsável por receber mensagens públicas.
- **/topic/errors.{username}**: destinada a receber mensagens relacionadas a erros na conexão *WebSocket*.

Os microsserviços de Imagens (*Image-Worker*) e E-Mail (*EMail-Worker*) realizam operações assíncronas com base em mensagens do *RabbitMQ*, aprimorando o desempenho da aplicação. Conforme a descrição do RabbitMQ (2023), esse serviço atua como intermediário na comunicação entre várias partes de um sistema.

Quando um serviço necessita enviar um e-mail, o processo inicia com o seu registro no *RabbitMQ*, onde ele se conecta à fila de processamento específica e posta uma mensagem nessa fila. Após a postagem, uma instância do serviço de e-mail recebe a solicitação, processa a mensagem e envia o e-mail conforme necessário. De forma análoga, o serviço de imagem opera da mesma maneira, com as imagens aguardando em uma fila designada para posterior processamento.

Os demais serviços da aplicação operam por meio de chamadas para *APIs* internas da própria infraestrutura de microsserviços. Esse modelo arquitetônico permite que cada serviço seja responsável por funcionalidades específicas, facilitando a escalabilidade e a manutenção

da aplicação. Através das *APIs* internas, os microsserviços podem trocar informações e dados, possibilitando uma colaboração em todo o ecossistema da aplicação.

Para acesso unificado a todas as *APIs* do sistema e criação de uma camada de segurança, foi criado um serviço específico para um *Gateway*, onde este é responsável por rotear as requisições para os serviços desejados. O *Spring Boot* oferece uma biblioteca para criação de *Gateway*, sendo esta a *Spring Cloud Gateway*. Segundo Spring (2023c), o *Spring Cloud Gateway* disponibiliza um Gateway de *API* construído sobre a infraestrutura *Spring* com o objetivo principal de oferecer uma abordagem simplificada e eficaz para o roteamento de solicitações para *APIs*.

O fluxo de segurança detalhado na Seção 3.3 foi transferido para o microsserviço de *Gateway*. Este microsserviço desempenha a função crucial de controlar o acesso às rotas privadas, realizando a verificação dos *tokens* de autenticação. Isso ocorre porque o *Gateway* representa o único ponto de entrada pelo qual os clientes podem acessar a camada de *back-end* da aplicação. É importante destacar que os *tokens* são gerados exclusivamente pelo microsserviço de Autenticação (*Auth*), onde é preciso realizar *login* com nome de usuário e senha.

Em síntese, a decomposição da aplicação monolítica para a arquitetura de microsserviços foi um passo fundamental para a comparação nos testes de carga. A estratégia adotada buscou manter a coerência com a aplicação monolítica, aproveitando a linguagem *Java* e o *framework Spring Boot*, mas com uma abordagem que permitisse maior flexibilidade, escalabilidade e isolamento de dados.

A transição para microsserviços implicou uma mudança significativa na infraestrutura da aplicação, mas permitiu um melhor gerenciamento de recursos e operações, facilitando a colaboração entre diferentes partes do sistema. O uso de um *Gateway* de *API*, como o *Spring Cloud Gateway*, e a transferência do fluxo de segurança para esse microsserviço garantiram a centralização e eficácia no controle de acesso à aplicação.

3.5 Front-end

Para a criação de uma interface de usuário responsiva, optou-se por utilizar a biblioteca *ReactJS*[4]. Essa escolha se baseia na capacidade dessa biblioteca de organizar o código por meio de componentes, o que torna a implementação mais eficiente, facilita a manutenção e possibilita a construção de uma versão de distribuição, conhecida como *build*.

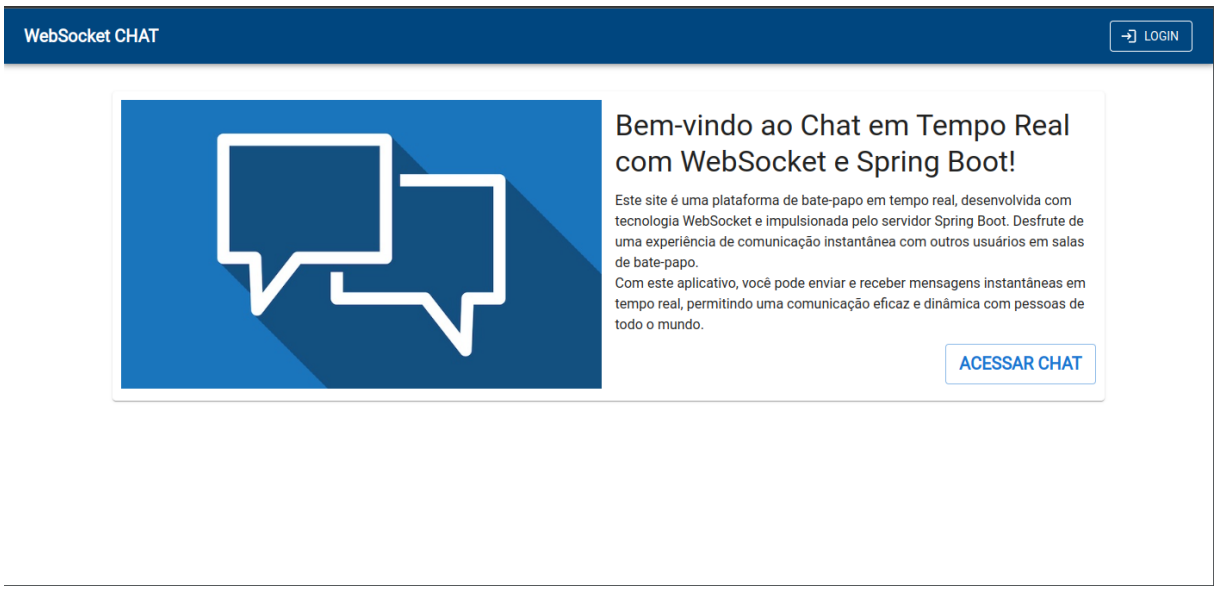
Essa eficiência de atualização é alcançada por meio do gerenciamento de uma *Document Object Model* (DOM) virtual, que representa em memória a estrutura da DOM real do navegador. O *React* compara essa DOM virtual com a DOM real para identificar as discrepâncias entre os estados antigos e novos dos componentes. Essa abordagem permite ao *React* atualizar somente as partes da DOM que efetivamente sofreram alterações, em vez de renderizar novamente a

[4] O *React* simplifica a criação de interfaces interativas, permitindo o design de visualizações para diferentes estados da aplicação e atualizando eficientemente os componentes conforme os dados mudam (React, 2023).

página inteira. Isso resulta em um desempenho mais ágil e responsivo nas interfaces de usuário desenvolvidas com *React*.

No início do desenvolvimento, foi estabelecida uma estrutura de navegação para orientar os usuários durante o uso do sistema. Esse caminho de navegação foi implementado utilizando o *React Router*, uma biblioteca que permite definir rotas de acesso na aplicação. Essas rotas proporcionam uma melhor experiência de uso ao direcionar os usuários de forma eficiente para as diferentes partes do sistema.

Figura 12 – Página inicial do sistema



Fonte: Elaboração própria (2023).

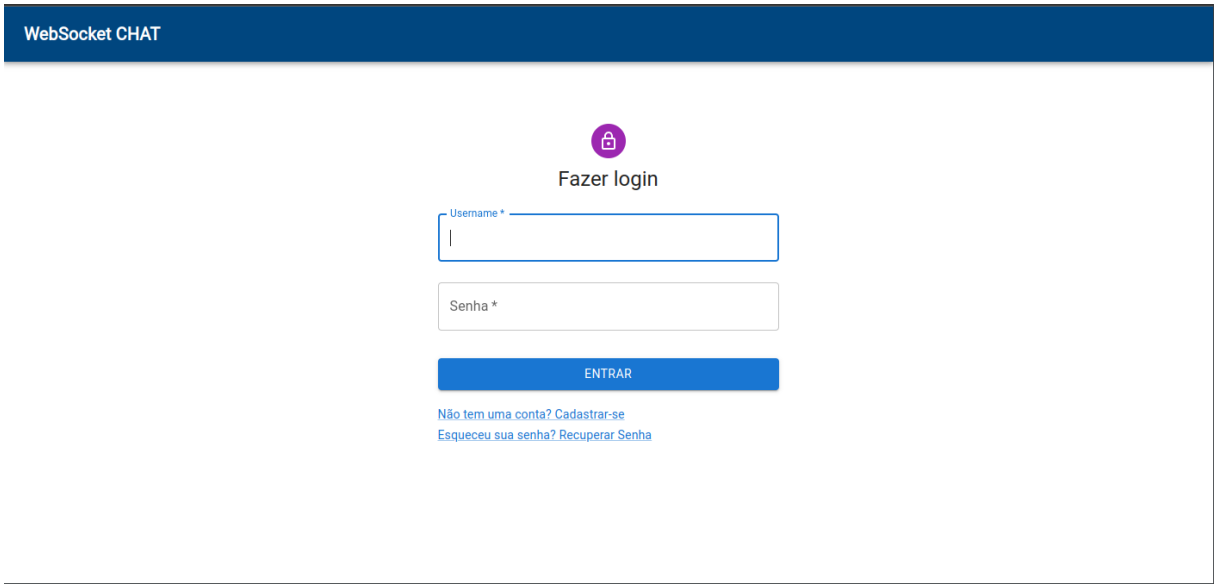
Inicialmente, foi desenvolvida a página inicial do sistema, conforme ilustrado na **Figura 12**. Essa página tem a finalidade de apresentar uma breve descrição do sistema e disponibilizar os links de acesso para os usuários.

A página subsequente é a tela de login do sistema, conforme ilustrado na **Figura 13**. Nessa página, os usuários inserem suas informações de autenticação, que são posteriormente enviadas para o *back-end* da aplicação a fim de obter um *token* de acesso para as funcionalidades do sistema. Essa etapa é fundamental para a segurança e autenticação dos usuários.

A **Figura 14** retrata a tela de cadastro de usuário, onde os usuários têm a possibilidade de criar novas contas. A página fornece avisos imediatos em caso de erros de digitação, além de apresentar informações visuais na tela quando o processo de cadastro é bem-sucedido. Isso garante uma experiência amigável e eficiente para os usuários.

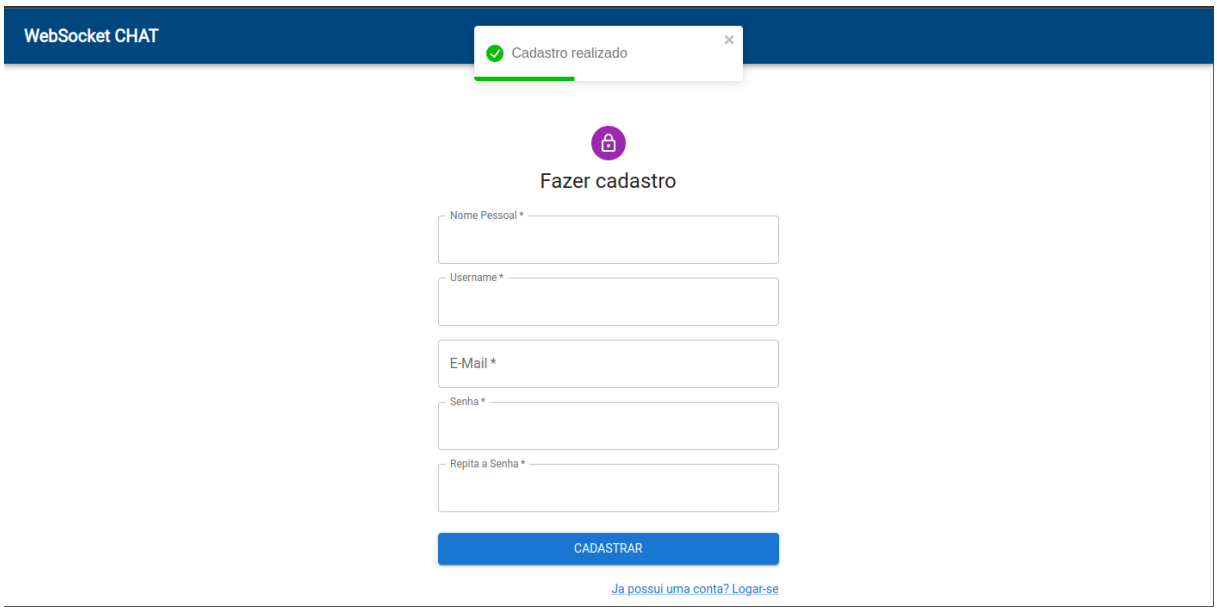
Ao acessar o sistema, o usuário é direcionado inicialmente para a tela principal, a sala de bate-papos, representada na **Figura 15**. Nesta sala, todas as conversas em andamento são registradas, e notificações em tempo real são fornecidas para outras conversas que estejam ocorrendo com o usuário logado. Essa tela centraliza a interação do usuário com as conversas, proporcionando uma experiência de chat abrangente e dinâmica.

Figura 13 – Página de *login*



Fonte: Elaboração própria (2023).

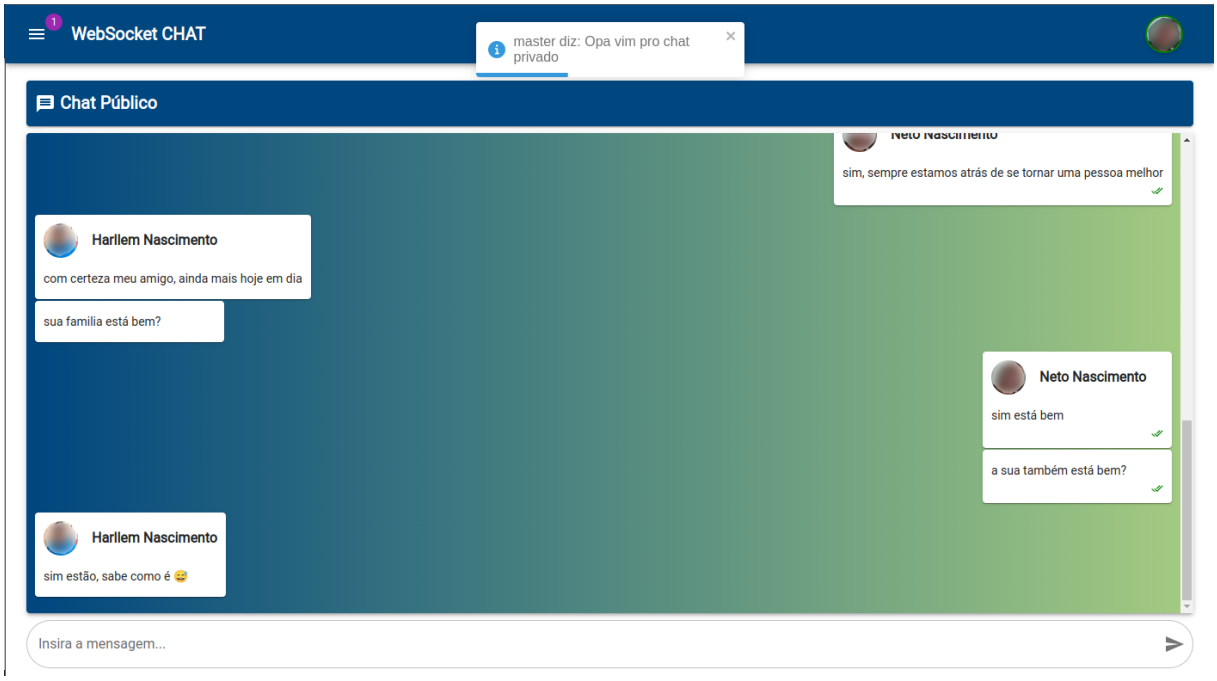
Figura 14 – Página de cadastro



Fonte: Elaboração própria (2023).

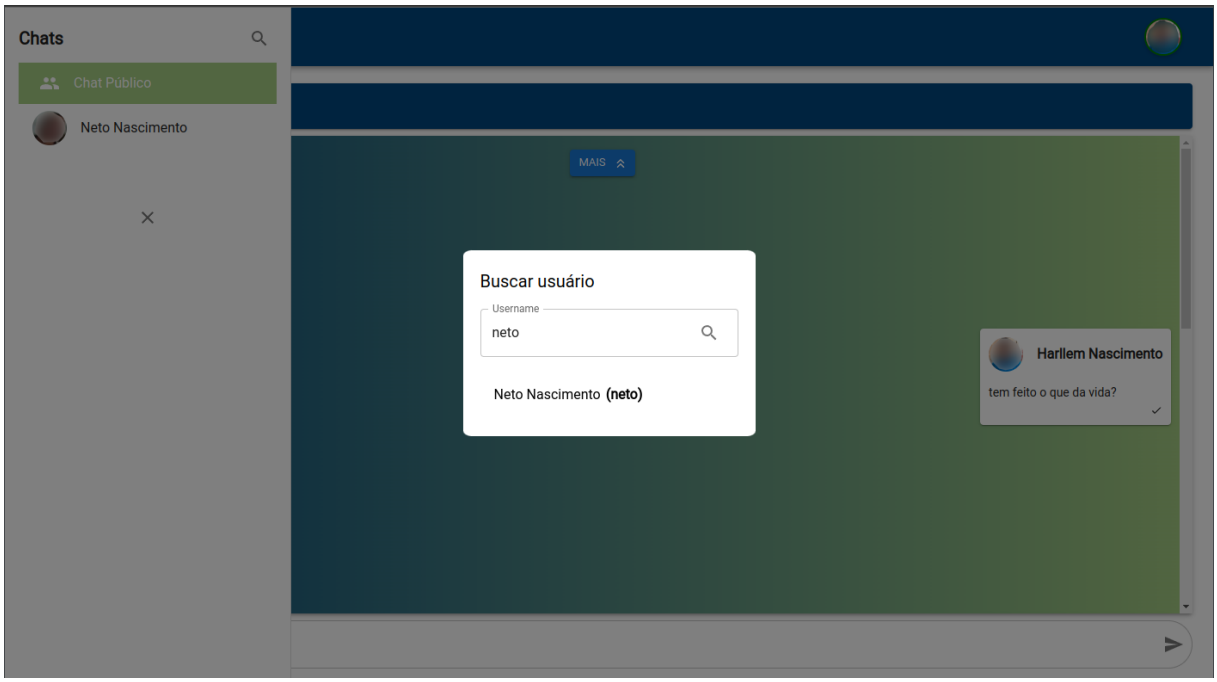
A sala de bate-papos apresenta diversas funcionalidades adicionais para aprimorar a experiência do usuário. Uma dessas funcionalidades é a capacidade de carregar mensagens paginadas, que reduz o tráfego de dados da aplicação, tornando-a mais leve para os navegadores. Isso é exemplificado na **Figura 16** por meio do botão “MAIS”. Além disso, um menu lateral oferece a opção de alternar entre o bate-papo público e salas privadas com outros usuários. Também inclui um botão de pesquisa para facilitar a localização de outros usuários. Essas características ampliam a versatilidade do ambiente de chat, proporcionando maior controle e comodidade ao usuário.

Figura 15 – Sala de bate-papo principal



Fonte: Elaboração própria (2023).

Figura 16 – Funcionalidades da sala de bate-papo principal

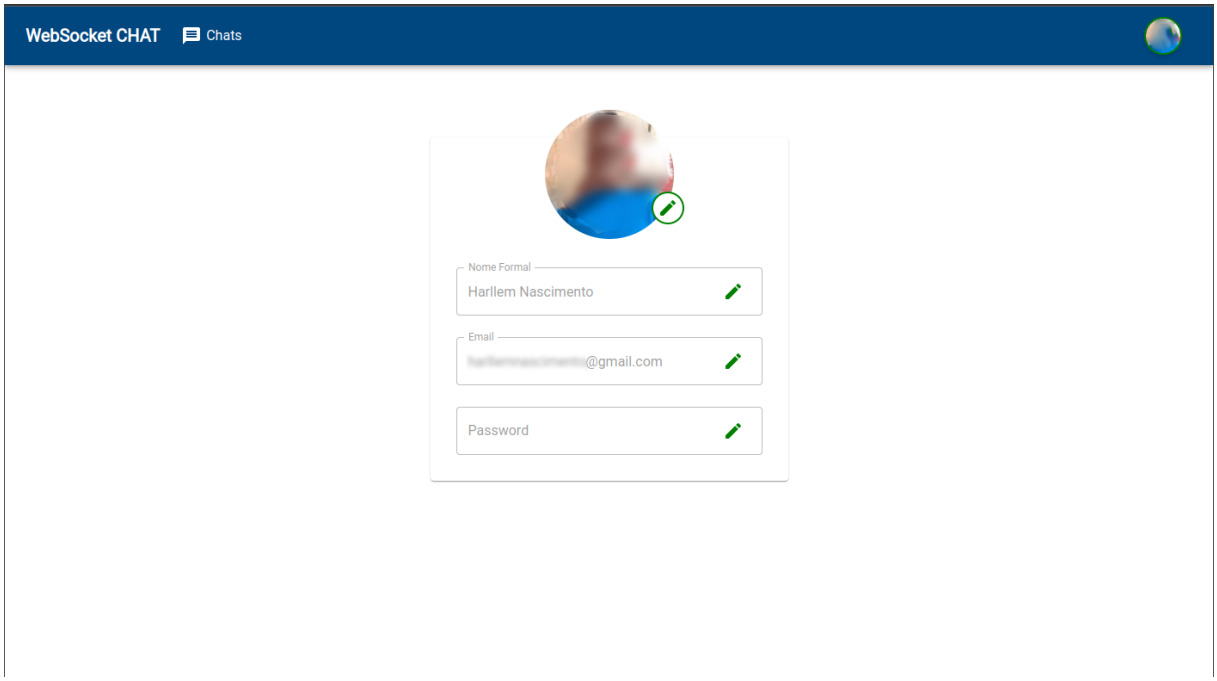


Fonte: Elaboração própria (2023).

Por último, os usuários têm a opção de editar seus dados pessoais cadastrados na conta, conforme demonstrado na **Figura 17**. Nessa página, é possível alterar a foto de perfil do usuário, seu nome de exibição nas salas de bate-papo, além de atualizar o endereço de e-mail associado à conta.

A aplicação cliente desenvolvida para o sistema de chat foi projetada com versatilidade,

Figura 17 – Página de edição de dados cadastrais



Fonte: Elaboração própria (2023).

de modo a funcionar tanto com o *back-end* monolítico quanto com a versão baseada em microsserviços. Isso se torna possível graças à manutenção da consistência nas rotas definidas durante o desenvolvimento de ambas as arquiteturas. As rotas de navegação do usuário permaneceram inalteradas, e aquelas que necessitaram de ajustes foram explicitamente declaradas no código criado no *React*, por meio de uma função dedicada para resolver essas adaptações. Esse enfoque garante uma experiência contínua e uniforme para os usuários em todas as etapas do desenvolvimento, independentemente da infraestrutura subjacente.

Essa abordagem elimina a necessidade de criar e manter duas versões distintas do cliente, o que simplifica o processo de distribuição e atualização. Os usuários podem utilizar a mesma aplicação cliente para acessar tanto a versão monolítica quanto a de microsserviços do sistema de chat, garantindo uma experiência consistente em ambas as configurações. Isso contribui para a conveniência e facilidade de uso, independentemente da escolha de arquitetura do sistema.

É relevante salientar o emprego dos componentes fornecidos pela *Material UI*[5]. Essa abordagem permite o desenvolvimento de uma interface robusta, aproveitando uma biblioteca amplamente reconhecida no contexto do *front-end* de aplicações web.

3.6 Preparo do ambiente de execução

Para criar um ambiente de teste imparcial, com condições idênticas de recursos e isolamento para ambas as arquiteturas, a escolha foi utilizar contêineres *Docker* gerenciados pelo

[5] *Material UI* é uma biblioteca de componentes de código aberto para *React* que implementa o Material Design do *Google*.

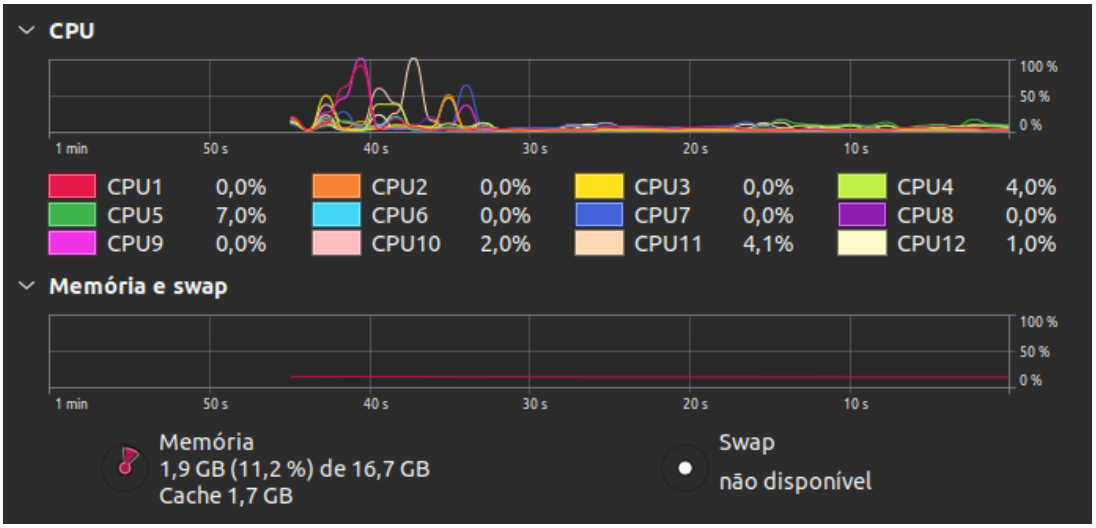
Kubernetes. Isso garante que ambos os modelos tenham acesso à mesma alocação de recursos de *hardware*, permitindo que operem sem interferências do sistema operacional.

O *Kubernetes* (K8s) é um sistema de código aberto que automatiza o gerenciamento de aplicações em contêineres, simplificando sua administração ao agrupar os contêineres de forma lógica. Ele se baseia na experiência de 15 anos da *Google* em operações de produção e adota as melhores práticas da comunidade (Kubernetes, 2023).

O *Docker* permite empacotar e executar aplicativos em ambientes isolados chamados contêineres, que são leves, autossuficientes e seguros. Isso facilita a execução de vários contêineres em uma mesma máquina e o compartilhamento consistente desses ambientes de trabalho (Docker, 2023a).

Com base nos recursos oferecidos por essas tecnologias, é viável estabelecer um ambiente de teste controlado. Nesse ambiente, serão geradas imagens de contêineres *Docker*, que serão posteriormente empregadas na infraestrutura disponibilizada pelo *Kubernetes*. O *Kubernetes* oferece funcionalidades como balanceamento de carga e monitoramento do estado de cada serviço em operação além da possibilidade de monitoramento do uso de *hardware* por cada contêiner.

Figura 18 – Máquina hospedeira dos testes



Fonte: Canonical (2023).

O *cluster*[6] implantado no ambiente do *Kubernetes* foi configurado com uma capacidade de processamento de 6000 *milicores*[7] e 10 gigabytes de memória RAM, de acordo com as especificações estabelecidas para a execução dos testes. No entanto, é importante esclarecer que esses valores não representam a capacidade máxima da máquina *host* que hospeda o *cluster*.

[6] Um *cluster* Kubernetes é o termo atribuído a um grupo de nós, como máquinas virtuais ou físicas, usados em uma implantação do *Kubernetes*. (Oracle, 2023b)

[7] Os *milicores* são representados pela unidade "m" e são uma medida que quantifica a capacidade de processamento da CPU. Essa métrica expressa a fração de um núcleo de CPU que um contêiner pode utilizar. Por exemplo, se for alocado "500m" (ou 0,5 núcleos), isso significa que o contêiner pode usar 50% da capacidade de processamento de um único núcleo da CPU.

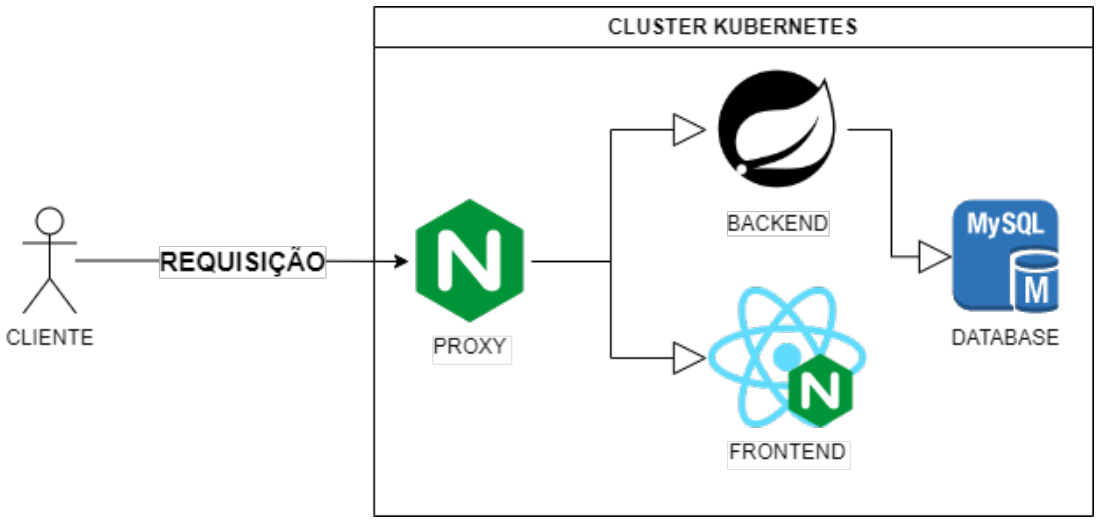
A máquina *host*, ilustrada através do monitor de recursos Ubuntu na **Figura 18**, é equipada com um processador *Xeon E5 2620V3* que possui 6 núcleos físicos, sendo que a tecnologia *Hyper-Threading Intel* oferece 2 segmentos de processamento por núcleo físico, totalizando 12 *threads* disponíveis. Além disso, a máquina *host* conta com 16 gigabytes de memória RAM e executa o sistema operacional Ubuntu na versão 22.04.3 LTS. A configuração do *cluster* é projetada considerando a presença de outros serviços em execução durante os testes, como métricas de consumo de recursos do *cluster*, o próprio *Kubernetes* e a ferramenta *JMeter*. Além disso, ressalta-se que todo o código desenvolvido para as aplicações, infraestruturas e testes está acessível por meio de um repositório[8] no GitHub.

A alocação de recursos no *cluster* ocorre durante a definição da estrutura de toda a infraestrutura da aplicação, e o *Kubernetes* fornece esses recursos conforme as solicitações de cada componente da aplicação. Portanto, a estratégia de gerenciamento de recursos será detalhada nas subseções a seguir.

3.6.1 Infraestrutura da aplicação monolítica

A infraestrutura da aplicação monolítica é notavelmente simples em comparação com a abordagem de microsserviços, pois envolve a configuração de poucos componentes no ambiente. Uma característica distintiva dessa abordagem é a alocação exclusiva de recursos de *hardware* para a única aplicação monolítica, em contraste com os microsserviços, que podem ser escalados horizontalmente para lidar com demandas variáveis.

Figura 19 – *Cluster Kubernetes* monolítico



Fonte: Elaboração própria (2023).

Conforme ilustrado na **Figura 19**, o ambiente de execução é composto por quatro contêineres: a aplicação monolítica do *back-end*, o banco de dados *MySQL*, o servidor *proxy NGINX*[9]

[8] O nome do repositório de código deste trabalho é RealtimeCHAT, disponível no GitHub.

[9] *NGINX* é uma ferramenta de código aberto empregada para várias funções, como servir conteúdo na web, agir como intermediário reverso, armazenar em cache, distribuir a carga de trabalho entre servidores e até mesmo para a transmissão de mídias (NGINX, 2023).

e uma *build ReactJS* fornecida também por um contêiner *NGINX* da aplicação. Vale ressaltar que, em comparação com a aplicação monolítica, os contêineres correspondentes ao serviço de *proxy* (*PROXY*) e a camada de *front-end* (*FRONTEND*) apresentam um consumo de recursos significativamente reduzido, uma vez que desempenham funções de baixo processamento. O *PROXY* atua primariamente como redirecionador de tráfego, enquanto o *FRONTEND* disponibiliza a página principal para que os clientes possam renderizar os componentes da aplicação em seus navegadores.

Tabela 1 – Distribuição de recursos do cluster na aplicação monolítica

SERVIÇO	SOLICITADO		LIMITE	
	CPU (m)	MEM. (M)	CPU MAX (m)	MEM. MAX (M)
BACKEND	1000	1024	5000	8192
DATABASE	500	500	3000	1024
PROXY	50	100	200	250
FRONTEND	50	250	150	500
TOTAL	**1600**	**1874**	**8350**	**9966**

Fonte: Elaboração própria (2023).

A **Tabela 1** apresenta a alocação de recursos realizada no *cluster* monolítico, indicando tanto os limites quanto os recursos solicitados. Vale ressaltar que a métrica principal considerada nesta configuração é a quantidade de recursos solicitados, uma vez que esses recursos são alocados de forma garantida para os serviços que os solicitam. Os limites, por outro lado, são fornecidos aos serviços somente se houver recursos disponíveis para tal. Portanto, mesmo que o processador disponha de 6000 *milicores*, não significa que todos serão alocados exclusivamente para um único serviço solicitante.

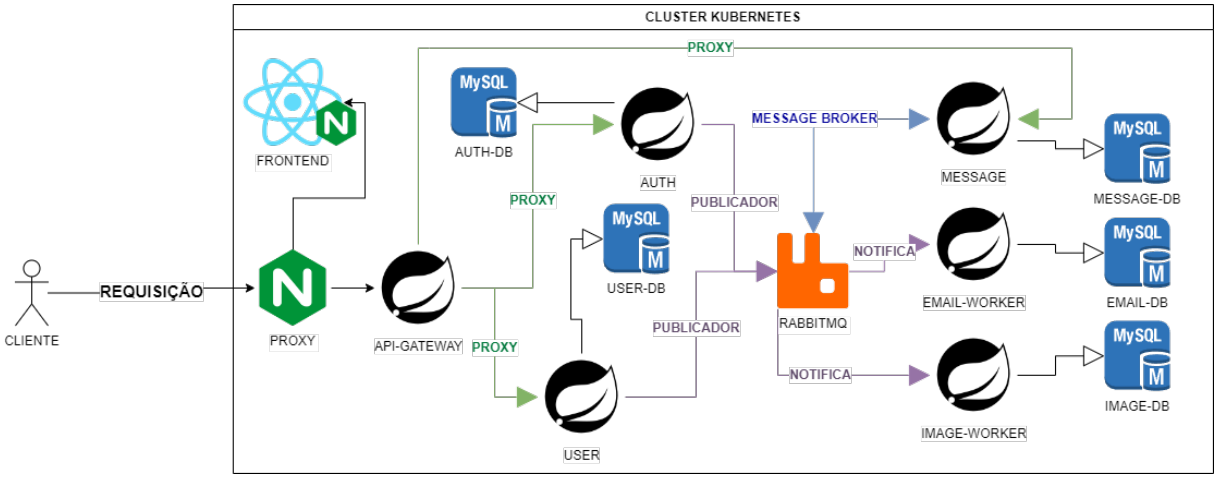
Levando em consideração essa alocação de recursos, a infraestrutura foi planejada de maneira a direcionar o estresse do teste principalmente para os componentes do *back-end*, considerados os principais alvos de carga.

3.6.2 Infraestrutura da aplicação baseada em microsserviços

A abordagem de microsserviços é substancialmente diferente em termos de infraestrutura se comparada à aplicação monolítica. No contexto dos microsserviços, a infraestrutura é mais complexa, uma vez que envolve a implantação e configuração de vários componentes distintos. O principal diferencial dos microsserviços é a flexibilidade que oferecem na alocação de recursos de *hardware*. Diferentemente da abordagem monolítica, onde todos os recursos são dedicados a uma única aplicação, nos microsserviços, os recursos podem ser alocados e escalados horizontalmente para atender a demandas variáveis, o que permite um gerenciamento mais eficaz das cargas de trabalho.

A **Figura 20** apresenta a infraestrutura da aplicação de chat baseada em microsserviços, a qual é composta por diversos componentes, cada um desempenhando funções específicas e

Figura 20 – *Cluster Kubernetes* baseado em microsserviços



Fonte: Elaboração própria (2023).

cruciais. Abordaremos de forma concisa cada um desses componentes a seguir:

- **Auth**: o serviço de autenticação desempenha um papel fundamental, autenticando usuários e gerando *tokens* de acesso para a aplicação. Ele serve como o ponto de entrada para os usuários validarem suas identidades.
- **User**: o serviço de usuário concentra-se no gerenciamento das informações dos perfis dos usuários, abrangendo tarefas como registro, atualização e busca de dados pessoais.
- **Email-Worker**: o serviço de email é um trabalhador assíncrono encarregado do processamento das operações relacionadas ao envio de e-mails, incluindo notificações e comunicações com os usuários.
- **Front-end**: o *Front-end* é a parte da aplicação acessada pelos clientes por meio de navegadores. Sua função primordial é fornecer a interface do usuário e interagir com os demais serviços.
- **Proxy**: o *Proxy* age como um servidor intermediário, redirecionando o tráfego entre o cliente e a aplicação, funcionando como um intermediário nas solicitações dos usuários.
- **API Gateway**: o *API Gateway* atua como um *gateway* de *API* que direciona as solicitações dos clientes para os serviços apropriados e oferece recursos de segurança, monitoramento e resiliência.
- **Image-Worker**: o serviço de imagens é um trabalhador assíncrono responsável pelo processamento de operações relacionadas a imagens, especificamente voltado para o armazenamento dessas imagens, transformação e recuperação.
- **Message**: o serviço de mensagens é essencial para a comunicação em tempo real na aplicação, fornecendo conexões *WebSocket* e utilizando um *message broker* integrado

com o *RabbitMQ*. Essa integração permite a distribuição de mensagens entre múltiplas instâncias do serviço de mensagens, facilitando a comunicação entre os usuários.

- **RabbitMQ**: o *RabbitMQ* é um *message broker* que atua como intermediário na troca de mensagens entre os diversos componentes da aplicação, permitindo uma comunicação assíncrona eficaz.
- **Bancos de Dados**: os serviços com a nomenclatura "DB" são responsáveis pelo armazenamento de dados relacionados a cada serviço.

É importante notar que os *Image-Worker* e *Email-Worker* operam de forma assíncrona, processando tarefas em filas do *RabbitMQ*. Além disso, o serviço de *Message* desempenha um papel crucial na habilitação da comunicação em tempo real, utilizando o *RabbitMQ* para distribuir mensagens entre múltiplas instâncias desse serviço. Os serviços de *User* e *Auth* publicam informações no *RabbitMQ* para serem consumidas pelos trabalhadores assíncronos.

Dada a complexidade da infraestrutura de microsserviços, tornou-se imperativo estabelecer uma clara segregação dos recursos de *hardware* da aplicação. Uma série de testes de execução da aplicação foi conduzida, visando a identificação de valores que estejam alinhados com as limitações da máquina que hospedará a aplicação. Essa abordagem visa otimizar o desempenho e garantir a operação eficiente da aplicação dentro dos recursos disponíveis.

Tabela 2 – Distribuição de recursos do cluster em microsserviços

SERVIÇO	SOLICITADO		LIMITE				
	CPU (m)	MEM. (M)	MÁX. DE RÉPLICAS	CPU MIN (m)	CPU MAX (m)	MEM. MIN (M)	MEM. MAX (M)
frontend	50	250	1	150	150	700	700
proxy	50	100	1	200	250	700	700
auth	200	250	3	1000	3000	500	1500
user	200	250	3	1000	3000	500	1500
email-worker	200	250	3	1000	3000	500	1500
api-gateway	200	250	1	500	500	500	500
image-worker	200	250	3	1000	3000	1000	3000
message	200	250	3	1000	3000	500	1500
rabbitmq	200	250	1	1000	1000	1000	1000
auth-db	100	250	1	250	250	600	600
user-db	100	250	1	250	250	600	600
email-db	100	250	1	250	250	600	600
image-db	100	250	1	250	250	600	600
message-db	100	250	1	250	250	600	600
TOTAL	**2000**	**3350**	**24**	**8100**	**18150**	**8900**	**14900**

Fonte: Elaboração própria (2023).

A **Tabela 2** apresenta a alocação de recursos no *cluster Kubernetes*. Os serviços de autenticação, usuários, e-mail, imagens e mensagens são projetados para escalar horizontalmente, com uma solicitação de apenas 200 *milicores* de processamento e um máximo de 1000 *milicores* por réplica, permitindo que o sistema ajuste automaticamente a capacidade conforme o tráfego de usuários.

Apesar do somatório de poder de processamento atingir 18.150 *milicores*, esse valor só é alcançado na prática quando há uma alta carga de usuários em todos os serviços do sistema e

recursos disponíveis para alocar novas instâncias. Vale ressaltar que cada serviço escalável, na configuração máxima, implica na existência de três instâncias ativas.

Os demais serviços, relacionados aos bancos de dados, *proxy* e *frontend*, consomem recursos mínimos e são instanciados apenas uma vez. Mesmo em cargas de trabalho intensas, o volume de processamento é relativamente baixo para a aplicação desenvolvida, resultando em alocações de recursos otimizadas.

3.7 Fluxo de realização dos testes de carga

Para realizar a comparação do desempenho de execução da aplicação no contexto monolítico e de microsserviços, é preciso utilizar de uma ferramenta que simule um grande tráfego de requisições aos recursos da aplicação, e para isso foi selecionado o *Apache JMeter*.

O *Apache JMeter* é uma aplicação de código aberto em *Java* para testar funcionalidades e medir o desempenho. Pode simular cargas intensas em servidores, redes ou objetos para avaliar a robustez e o desempenho sob diferentes tipos de carga (Apache, 2023).

Por não oferecer suporte nativo para o protocolo *STOMP* via *Websocket*, o qual será empregado para registrar sessões bidirecionais entre os clientes e o servidor, se fez necessário a criação de uma extensão do *Apache JMeter* para simular o fluxo de uso com relação ao serviço *Websocket*.

Os demais serviços operam através de uma *API Rest*, funcionando com o protocolo *HTTP*. Isso possibilita a criação de fluxos de uso para os recursos diretamente com o *JMeter*, sem a necessidade de desenvolver extensões adicionais.

A coleta de resultados foi conduzida por meio dos ouvintes (*listeners*) do *JMeter*. Estes ouvintes têm a responsabilidade de registrar informações sobre a execução do fluxo de testes definido no *Apache JMeter*. Consequentemente, métricas como o tempo de resposta do servidor, média e taxa de erros serão quantificadas para possibilitar o acompanhamento abrangente de toda a execução.

3.7.1 Fluxo de login

O teste relacionado ao processo de login adota a estratégia de avaliar principalmente a capacidade do sistema em gerar *tokens* de autenticação válidos com base nas informações de login do usuário. Esse teste concentra-se na rota "/auth", acessada por meio do método *HTTP POST*, no qual o nome de usuário e a senha são fornecidos no corpo da requisição.

É importante destacar que essa rota não é protegida, ou seja, não envolve a decodificação de *tokens*, dado que se trata de uma rota pública. No contexto de microsserviços, a requisição passa pelo *API Gateway*, seguindo para o servidor de Autenticação, que faz uma chamada interna à *API* de usuários do servidor de usuários.

A carga de usuários selecionada para o teste envolve 2000 usuários realizando operações de login em um período de 100 segundos. Esses valores permanecem consistentes para ambas as arquiteturas testadas, permitindo a comparação dos resultados entre os cenários.

3.7.2 Fluxo de consulta de mensagens

O teste destinado à operação de consulta de mensagens visa avaliar principalmente a capacidade do sistema de recuperar mensagens do bate-papo público. Esse teste concentra-se na rota "/messages/retrieve_messages/by/public" acessada pelo método *HTTP GET*, na qual os usuários realizam a consulta das mensagens públicas disponíveis.

É importante destacar que esta rota requer acesso restrito, sendo acessível somente a usuários autenticados que possuam um *token* válido no cabeçalho da requisição. Na arquitetura de microsserviços, a decodificação do *token* ocorre no serviço *API Gateway*, onde é realizada a verificação de sua validade antes de encaminhar a requisição para o servidor de mensagens.

A operação de consulta de mensagens consiste em 6000 usuários que realizam buscas nas mensagens públicas do bate-papo ao longo de um período de 100 segundos, com o banco de dados inicialmente contendo 100 mensagens públicas. Essas configurações permanecem consistentes em ambas as arquiteturas testadas, permitindo uma comparação objetiva dos resultados entre os diferentes cenários.

3.7.3 Fluxo de conexão Websocket

Para simular a criação de uma conexão *WebSocket* com o servidor nos testes, foi necessário o desenvolvimento de um *plugin* personalizado para o *JMeter*. Isso se deve à falta de um padrão nativo para lidar com o protocolo *STOMP* em conjunto com o cliente *SockJS* usado no *ReactJS*. Como ponto de partida, foi utilizada a ideia do projeto existente no *GitHub* — plataforma de compartilhamento *on-line* de projetos de *software* — de Mohr e Robertson (2020), denominado *Mi Jmeter SockJS Stomp Sampler*, e adaptado todo o fluxo de teste para atender às necessidades do cenário previamente descrito.

O teste foi conduzido na rota pública "/ws", responsável por processar todas as solicitações de novas conexões *WebSocket*. Essa rota requer a decodificação dos *tokens* de autenticação por meio do serviço de autenticação (*auth*) para validação. Após a conexão, ocorre a inscrição no tópico "/app/message", que gerencia a comunicação no bate-papo público. Em seguida, uma mensagem contendo o texto "MESSAGE" é enviada para o tópico ao qual o cliente se inscreveu.

A carga de usuários selecionada consiste em 3000 usuários solicitando conexões no intervalo de 100 segundos. É importante mencionar que essa configuração se aplica a ambas as arquiteturas testadas.

3.7.4 Fluxo de persistência de imagens de perfil

O teste relacionado ao processamento das solicitações de alteração de imagens de perfil de usuário ocorre por meio da rota "/users/upload_profile_image" acessada pelo método POST do protocolo *HTTP*. É importante destacar que essa rota é protegida, o que implica a necessidade de decodificação de *tokens* antes de ser encaminhada para processamento.

Essa funcionalidade exige um considerável poder de processamento, devido à validação, transformação e persistência das imagens necessárias. Na arquitetura de microsserviços, o

fluxo inicia-se com a validação do *token* pelo *API Gateway*, seguida pela recepção e validação dos metadados relacionados à imagem no serviço de usuários. Em seguida, a solicitação de processamento é enfileirada no *RabbitMQ*, aguardando processamento por uma instância do serviço de imagens.

Em contrapartida, a aplicação monolítica realiza o processamento direto da solicitação. Essa distinção no fluxo de processamento destaca uma das diferenças fundamentais entre as arquiteturas monolítica e de microsserviços.

O teste referente ao processamento das solicitações de alteração de imagens de perfil de usuário foi conduzido com a simulação de 1000 usuários durante um período de 100 segundos em ambas as arquiteturas. Essa configuração permite uma análise comparativa dos resultados entre as duas abordagens arquiteturais, enfatizando o desempenho de processamento das imagens sob condições semelhantes em ambas as arquiteturas.

Este capítulo focou na metodologia da aplicação desenvolvida, abrangendo a criação de ambientes com *Kubernetes* alinhados a contêineres *Docker*, bem como cenários de testes definidos para cada uma das arquiteturas. O próximo capítulo, por sua vez, abordará a análise dos dados gerados pelas aplicações ao serem submetidas a testes de carga.

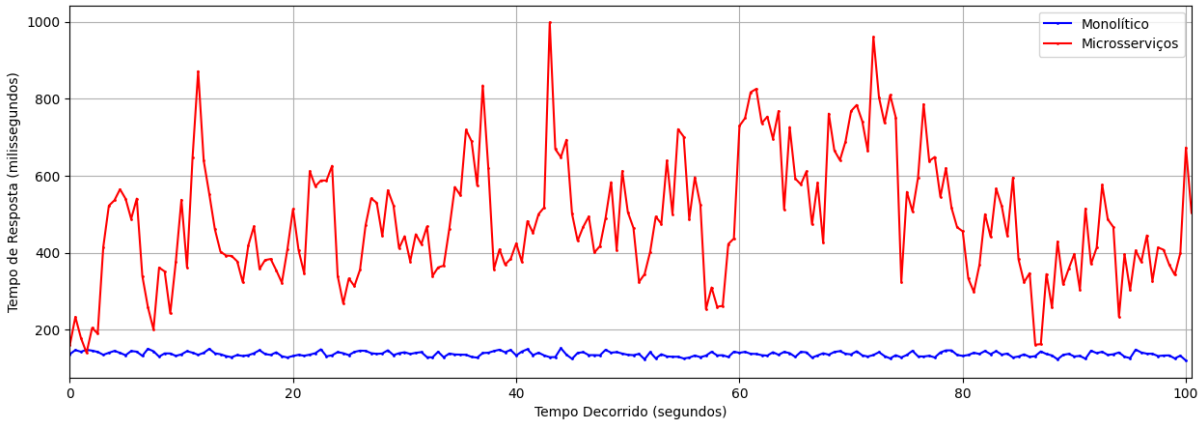
4 RESULTADOS E DISCUSSÕES

Neste capítulo, são apresentados os resultados decorrentes da análise do *software* desenvolvido, que serviu como objeto de estudo nas arquiteturas de microsserviços e monolítica. A principal finalidade deste capítulo é fornecer uma visão abrangente das conclusões obtidas a partir dos testes realizados.

4.1 Teste de carga na funcionalidade de login

O teste iniciou com a configuração do cenário de teste no *JMeter*, uma etapa crucial para simular condições realistas de uso. Nesse cenário, foi simulada a atividade de 2000 usuários realizando operações de login em um período de 100 segundos. A escolha dessa carga específica foi o resultado de testes anteriores, cujo objetivo era determinar a carga ideal que poderia ser executada dentro dos limites de *hardware* da máquina *host*.

Figura 21 – Resultado dos testes de *login*



Fonte: JMeter (Apache, 2023).

O gráfico retratado na **Figura 21** exibe as requisições encaminhadas ao servidor monolítico, representado pela linha de cor azul. Durante o teste, o tempo máximo de resposta registrado pelo servidor foi de 202 milissegundos, apresentando uma média de 136 milissegundos e um desvio padrão de 18,57.

Tabela 3 – Consumo de hardware durante o teste de login na arquitetura monolítica

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-backend	2460	393
webchat-database	76	504
TOTAL	**2536**	**897**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

A **Tabela 3** oferece uma visão do consumo máximo de recursos durante os testes. A aplicação monolítica utilizou um pico de 2460 *milicores* para o poder de processamento e 393

Megabytes de memória RAM. No que diz respeito ao banco de dados, o consumo máximo foi de apenas 76 *milicores* de poder de processamento e 504 *Megabytes* de memória RAM. O total de recursos utilizados por toda a aplicação durante os testes foi de 2536 *milicores* de CPU e 897 *Megabytes* de memória RAM.

Comparando os resultados com a aplicação monolítica, a **Figura 21** apresenta o desempenho da aplicação baseada em microsserviços pela linha de cor vermelha. O volume do teste de carga em questão permaneceu inalterado em relação ao cenário monolítico, resultando em uma média de tempo de resposta de 491 milissegundos, com um tempo máximo de 1796 milissegundos e um desvio padrão de 355,69 milissegundos.

Tabela 4 – Consumo de hardware durante o teste de login na arquitetura de microsserviços

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-api-gateway	67	253
webchat-auth	800	263
webchat-auth	831	266
webchat-auth	891	369
webchat-auth-database	60	460
webchat-image-database	2	390
webchat-image-worker	2	274
webchat-message	7	334
webchat-message-database	2	396
webchat-rabbitmq	9	192
webchat-user	18	229
webchat-user	66	329
webchat-user-database	41	418
TOTAL	**2796**	**4173**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

A **Tabela 4** apresenta a alta utilização dos recursos de *hardware* na aplicação baseada em microsserviços durante o teste de carga. O foco principal do teste recai sobre o serviço de autenticação (*auth*). Para gerenciar a carga, o *Kubernetes* escalou horizontalmente, alcançando o número máximo de instâncias configuradas para o serviço de autenticação. A soma do uso de recursos para todas as instâncias escaladas é a seguinte:

- **Uso de CPU**: 800 *milicores* + 831 *milicores* + 891 *milicores* = 2522 *milicores*.
- **Uso de Memória**: 263 *Megabytes* + 266 *Megabytes* + 369 *Megabytes* = 898 *Megabytes*.

Além disso, o serviço de banco de dados relacionado ao serviço de autenticação (webchat-auth-database) consumiu um máximo de 60 *milicores* de poder de processamento e 390 *Megabytes* de memória RAM.

Outro serviço que respondeu às solicitações internas feitas pelo serviço de autenticação foi o serviço de usuários (*users*). Para lidar com a carga, foram instanciados pelo *Kubernetes*

duas instâncias do serviço para responder às solicitações, resultando em um uso total de recursos de:

- **Uso de CPU**: 66 *milicores* + 18 *milicores* = 84 *milicores* .
- **Uso de Memória**: 229 *Megabytes* + 329 *Megabytes* = 747 *Megabytes*.

O serviço de banco de dados relacionado ao serviço de usuários consumiu apenas 41 *milicores* de poder de processamento e 418 *Megabytes* de memória RAM. Ao total, a infraestrutura de microsserviços consumiu 2796 *milicores* de processamento e 4173 *Megabytes* de memória RAM. Esse maior uso de memória RAM é uma consequência da operação simultânea de outros serviços, os quais requerem um mínimo de recursos de memória para operar de forma eficiente.

O tempo médio de resposta na arquitetura monolítica foi consideravelmente mais baixo, com a vantagem adicional de um menor desvio padrão, tornando-a a opção mais estável nesse cenário de teste. Além disso, é essencial destacar a utilização dos recursos de *hardware*, em que a aplicação baseada em microsserviços demandou uma quantidade de poder de processamento próxima à da aplicação monolítica, porém, requereu uma quantidade substancialmente maior de memória RAM.

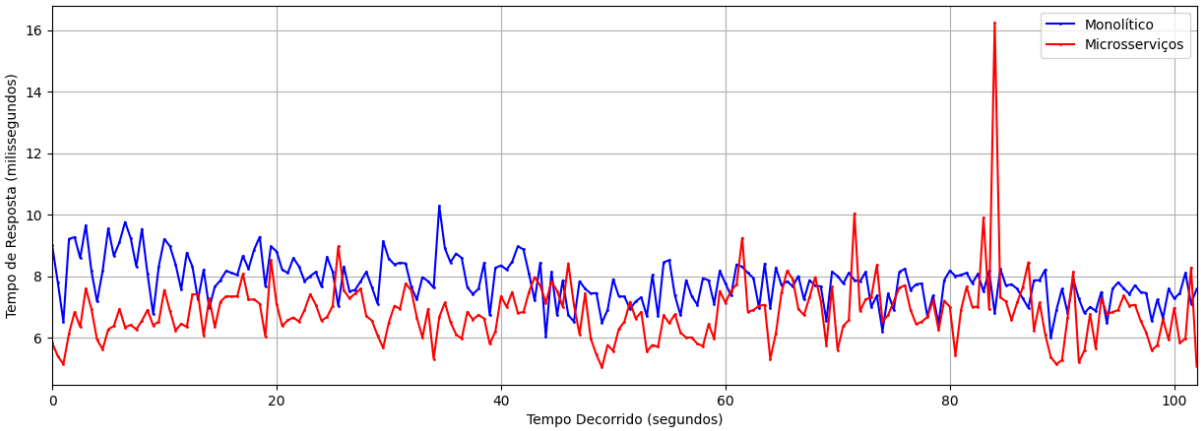
4.2 Teste de carga na funcionalidade de recuperação de mensagens públicas

A funcionalidade relacionada à recuperação de mensagens públicas é relativamente mais simples, uma vez que se limita à listagem das mensagens presentes no banco de dados, juntamente com a validação de autenticação do usuário. Para avaliar essa funcionalidade, foi simulado um cenário com 6000 usuários consultando mensagens em um intervalo de 100 segundos. O banco de dados foi carregado com antecedência com 100 mensagens. Esta escolha foi resultado de testes prévios, que visavam determinar a carga ideal para que o *JMeter* operasse sem enfrentar gargalos na execução do teste.

O gráfico, representado na **Figura 22**, ilustra as respostas do *cluster* monolítico, representados pela linha azul, em milissegundos. Os resultados revelam uma média de 7 milissegundos de tempo de resposta, com um pico máximo de 49 milissegundos e um desvio padrão de 1,67 milissegundos. É importante destacar que a baixa latência se deve, em parte, à natureza relativamente simples dessa operação, como mencionado anteriormente. Além disso, para otimizar o tempo de resposta, o banco de dados pode fazer uso de estratégias de armazenamento em *cache*, garantindo que as mensagens frequentemente acessadas sejam recuperadas de maneira mais eficiente.

No que tange à utilização de recursos pelo *cluster* durante os testes, demonstrado pela **Tabela 5**, foi observado um consumo total de 632 *milicores* de capacidade de processamento e 1002 *Megabytes* de memória, considerando a aplicação e o banco de dados. Esses números

Figura 22 – Resultado dos testes de requisição de mensagens



Fonte: JMeter (Apache, 2023).

Tabela 5 – Consumo de hardware durante o teste de recuperação de mensagens na arquitetura monolítica

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-backend	525	486
webchat-database	107	516
TOTAL	**632**	**1002**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

indicam que, mesmo sob uma carga significativa de usuários na funcionalidade em questão, os recursos utilizados permaneceram relativamente baixos.

Os resultados referentes ao tempo de resposta da aplicação baseada em microsserviços estão apresentados no gráfico ilustrado na **Figura 22** demarcado pela linha de cor vermelha. O teste foi realizado sob as mesmas condições do cenário monolítico, resultando em uma média de tempo de resposta de 6 milissegundos, com um tempo máximo de 55 milissegundos e um desvio padrão de 2,1 milissegundos.

É importante destacar que, embora a funcionalidade seja relativamente simples, o aumento no tempo de resposta em comparação com o contexto monolítico ocorreu devido à necessidade de validar o *token* de autenticação no servidor de recursos (*api-gateway*) antes de encaminhar a requisição para o servidor de mensagens. Esse processo adicional introduziu uma ligeira latência no tempo de resposta, conforme observado nos resultados.

Durante a execução dos testes, o *cluster* fez um uso relativamente baixo dos recursos em relação ao serviço de mensagens (*message*) e ao serviço de recursos (*api-gateway*), conforme evidenciado na **Tabela 6**. A fim de gerenciar eficazmente as solicitações, o *Kubernetes* realizou a escalabilidade do serviço de mensagens para três instâncias, que apresentaram os seguintes valores de uso de recursos

- **Uso de CPU**: 77 *milicores* + 233 *milicores* + 81 *milicores* = 391 *milicores*.
- **Uso de Memória**: 296 *Megabytes* + 286 *Megabytes* + 362 *Megabytes* = 898 *Megabytes*.

Tabela 6 – Consumo de hardware durante o teste de recuperação de mensagens na arquitetura de microsserviços

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-api-gateway	230	267
webchat-auth	3	366
webchat-auth-database	3	456
webchat-image-database	3	387
webchat-image-worker	4	273
webchat-message	77	296
webchat-message	233	286
webchat-message	81	362
webchat-message-database	65	437
webchat-rabbitmq	9	190
webchat-user	3	327
webchat-user-database	3	415
TOTAL	**714**	**4062**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

O serviço de banco de dados, associado às instâncias dos serviços de mensagens, atingiu um consumo de 81 *milicores* de capacidade de processamento e utilizou 362 *Megabytes* de memória RAM. Enquanto isso, o serviço de recursos enfrentou uma demanda considerável de requisições, atingindo o pico máximo de 230 *milicores* de processamento e utilizando 267 *Megabytes* de memória.

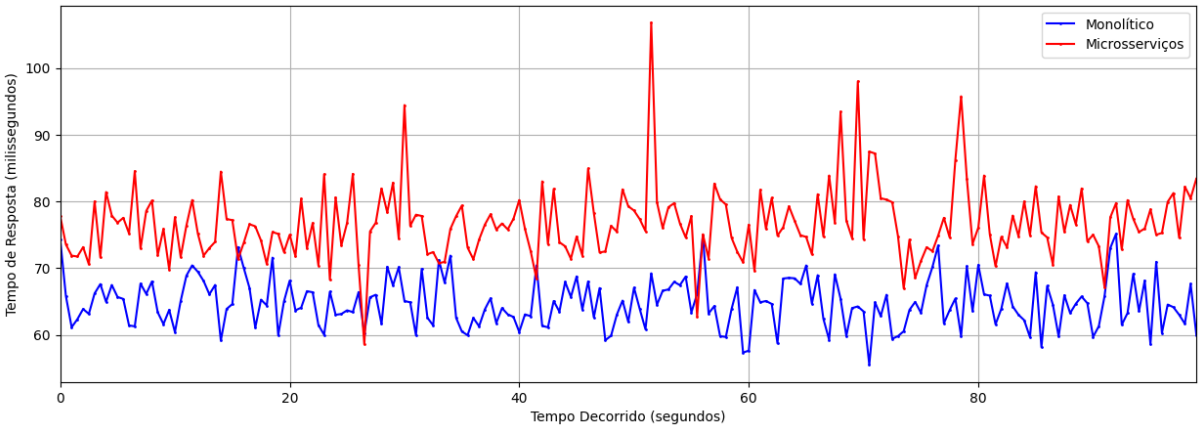
Os testes realizados em ambas as arquiteturas revelaram médias de respostas bastante semelhantes. No entanto, é importante destacar que a arquitetura de microsserviços apresentou um tempo de resposta máximo mais elevado, bem como um desvio padrão superior. Essa discrepância pode ser atribuída ao processo de validação de autenticação conduzido pelo servidor de recursos antes de encaminhar a requisição para o servidor de mensagens na arquitetura de microsserviços. No geral, em relação principalmente às médias de tempo de resposta, ambas as arquiteturas demonstraram um desempenho próximo e satisfatório para essa funcionalidade, e o uso de poder de processamento foi relativamente baixo, mantendo-se abaixo de 800 *milicores*.

4.3 Teste de carga na funcionalidade de conexões em tempo real (Websocket)

A funcionalidade de conexões em tempo real, implementada por meio do protocolo *WebSocket* e seu sub-protocolo *STOMP*, é uma das características fundamentais do sistema em análise. Isso se deve ao fato de que se trata de um aplicativo de bate-papo que exige o gerenciamento de várias sessões simultâneas. Com o propósito de testar essa funcionalidade, uma carga simulada de 3000 usuários foi configurada para estabelecer conexões ao longo de um período de 100 segundos.

A **Figura 23** apresenta um gráfico que ilustra as requisições realizadas durante o teste de carga ao longo de 100 segundos, juntamente com os tempos de resposta correspondentes a cada

Figura 23 – Resultado dos testes de conexões *websocket*



Fonte: JMeter (Apache, 2023).

solicitação. A aplicação monolítica, demarcada pela linha de cor azul, registrou uma média de 64 milissegundos para a execução completa do fluxo de conexão *Websocket*, incluindo a inscrição em um tópico e o envio de mensagens. Além disso, o tempo máximo de resposta foi de 125 milissegundos, com um desvio padrão de 12,18 milissegundos.

Tabela 7 – Consumo de hardware durante o teste de conexão websocket na arquitetura monolítica

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-backend	991	476
webchat-database	82	508
TOTAL	**1073**	**984**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

No que se refere aos recursos consumidos pela arquitetura monolítica, demonstrado na **Tabela 7**, observou-se um consumo relativamente baixo, totalizando 1073 *milicores* de capacidade de processamento e 984 *Megabytes* de memória RAM, abrangendo tanto o uso da aplicação quanto do banco de dados.

No que diz respeito ao teste realizado na arquitetura de microsserviços, observou-se um nível considerável de complexidade na comunicação entre os microsserviços. Para iniciar a análise, podemos examinar o tempo de execução do fluxo de teste *Websocket*, conforme representado no gráfico da **Figura 23** demarcado pela linha de cor vermelha. Nesse cenário, a média do tempo de resposta foi de 76 milissegundos, com um valor máximo de 271 milissegundos e um desvio padrão de 13,76 milissegundos.

Em relação ao uso de recursos pela arquitetura de microsserviços, conforme ilustrado na **Tabela 8**, destaca-se um custo mais elevado em termos de capacidade de processamento em *milicores* para sustentar essa funcionalidade. O serviço de mensagens, em particular, desempenha um papel central no gerenciamento das conexões *Websocket* e foi o foco principal dos testes de carga. Nesse contexto, o serviço de mensagens foi escalado para três réplicas pelo *Kubernetes*,

Tabela 8 – Consumo de hardware durante o teste de conexão websocket na arquitetura de microsserviços

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-api-gateway	166	239
webchat-auth	250	283
webchat-auth	111	343
webchat-auth-database	2	414
webchat-image-database	2	355
webchat-image-worker	2	245
webchat-message	355	383
webchat-message	343	348
webchat-message	188	384
webchat-message-database	50	413
webchat-rabbitmq	150	145
webchat-user	67	292
webchat-user-database	35	378
TOTAL	**1721**	**4222**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

resultando em um pico de uso de recursos, conforme detalhado a seguir:

- **Uso de CPU**: 355 *milicores* + 343 *milicores* + 188 *milicores* = 886 *milicores*.
- **Uso de Memória**: 383 *Megabytes* + 348 *Megabytes* + 384 *Megabytes* = 1115 *Megabytes*.

Para viabilizar a validação da autenticação do usuário, em que o serviço de mensagens (*message*) precisa se conectar ao serviço de autenticação (*auth*), o *Kubernetes* escalou o serviço de autenticação para duas instâncias, levando a uma utilização de recursos conforme detalhado abaixo:

- **Uso de CPU**: 250 *milicores* + 111 *milicores* = 361 *milicores*.
- **Uso de Memória**: 283 *Megabytes* + 343 *Megabytes* = 626 *Megabytes*.

O serviço de autenticação requer informações do serviço de usuário (*user*) para concluir a validação do *token* com sucesso. Isso levou a uma única instância do serviço de usuário, que registrou um pico de uso de recursos de 67 *milicores* de capacidade de processamento e 292 *Megabytes* de memória RAM.

Outro serviço de crucial importância neste teste é o *RabbitMQ* (*rabbitmq*), responsável por fornecer uma fila de mensagens que permite a troca de mensagens entre os usuários, replicando-as entre as instâncias dos serviços de mensagem. Sem ele, se um usuário estivesse conectado a uma instância e outro a outra instância, eles não receberiam as mensagens um do outro. Durante o teste, observou-se um pico de 150 *milicores* de capacidade de processamento e 145 *Megabytes* de uso de memória RAM.

Quanto aos serviços de banco de dados, a instância relacionada aos usuários utilizou 35 *milicores* de capacidade de processamento e 378 *Megabytes* de memória RAM, enquanto a instância associada ao serviço de mensagens consumiu 50 *milicores* de processamento e 413 *Megabytes* de memória RAM. O serviço *API Gateway* desempenhou o papel de encaminhar as requisições dos usuários, utilizando 166 *milicores* de capacidade de processamento e 239 *Megabytes* de memória RAM. No total, o *cluster* utilizou um máximo de 1721 *milicores* de processamento e 4222 *Megabytes* de memória RAM.

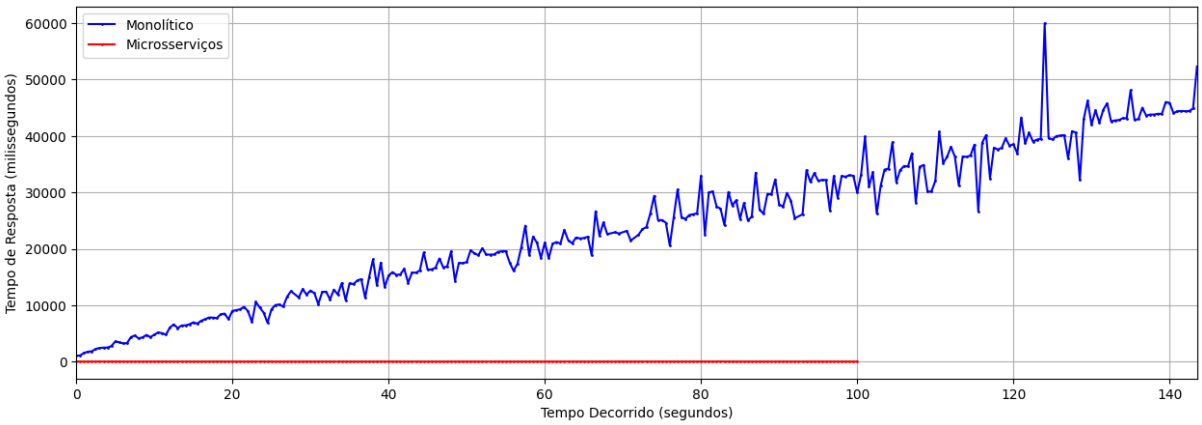
A discrepância na média de tempo de resposta foi de meros 12 milissegundos, com a aplicação monolítica apresentando o menor tempo. Vale notar que o tempo máximo de resposta na aplicação baseada em microsserviços foi cerca de duas vezes maior em comparação com a aplicação monolítica, acompanhado de um desvio padrão de 13,76 milissegundos, que é apenas 1,58 milissegundos superior ao da aplicação monolítica.

Assim, a análise dos dados permite concluir que a funcionalidade em questão operou de forma semelhante nas duas abordagens. No entanto, é importante destacar que a arquitetura de microsserviços demonstrou um consumo de recursos significativamente maior, totalizando 1721 *milicores* de capacidade de processamento. Comparativamente, a arquitetura monolítica utilizou apenas 1073 *milicores*, uma diferença considerável. Além disso, a aplicação baseada em microsserviços demandou mais recursos de memória devido à presença de serviços adicionais em execução.

4.4 Teste de carga na funcionalidade de persistências de imagens de perfil

Para simular o uso da funcionalidade de alteração de imagens de perfil do usuário, foram conduzidos testes preliminares para definir a carga ideal. Dada a sua natureza, essa funcionalidade requer significativos recursos do sistema. Portanto, o cenário de teste apropriado para ambas as arquiteturas consistiu em ter 1000 usuários acessando a aplicação ao longo de 100 segundos.

Figura 24 – Resultado dos testes de persistências de imagens de perfil



Fonte: JMeter (Apache, 2023).

A **Figura 24** apresenta um gráfico que ilustra o tempo de resposta das requisições na

aplicação monolítica, representado pela linha de cor azul. A média desse tempo foi de 24,577 segundos, com um valor máximo de 61,791 segundos e um desvio padrão de 13,123 segundos. Durante os testes, observou-se uma notável latência elevada, que chegou a ultrapassar o limite de 100 segundos estabelecido para o teste, superando 140 segundos.

Tabela 9 – Consumo de hardware durante o teste de persistências de imagens de perfil na arquitetura monolítica

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-backend	5014	2429
webchat-database	15	516
TOTAL	**5029**	**2945**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

A **Tabela 9** revela que a aplicação monolítica foi levada ao seu limite de processamento sob a carga estabelecida, consumindo 5014 *milicores* de capacidade de processamento, juntamente com 2429 *Megabytes* de memória RAM. Em contrapartida, o banco de dados utilizou quantidades mínimas de recursos, com um pico máximo de 15 *milicores* de capacidade de processamento e 516 *Megabytes* de memória RAM.

O ensaio conduzido na estrutura de microsserviços é descrito na **Figura 24**, representado pela linha de cor vermelha, que apresenta os tempos de resposta obtidos sob condições de teste idênticas às empregadas no âmbito da aplicação monolítica. O resultado do teste revelou uma média de 19 milissegundos de tempo de resposta, sendo o valor máximo registrado de 152 milissegundos, com um desvio padrão de 6,04 milissegundos.

Tabela 10 – Consumo de hardware durante o teste de persistências de imagens de perfil na arquitetura microsserviços

SERVIÇO	CPU(milicores)	Memória(Megabytes)
webchat-api-gateway	67	237
webchat-auth	31	333
webchat-auth-database	3	410
webchat-image-database	7	373
webchat-image-worker	997	460
webchat-image-worker	996	447
webchat-image-worker	998	464
webchat-message	2	378
webchat-message-database	3	404
webchat-rabbitmq	50	529
webchat-user	183	313
webchat-user	424	238
webchat-user-database	8	373
TOTAL	**3769**	**4959**

Fonte: Métricas de uso de recursos do Kubernetes (2023).

A realização do teste de carga de imagens impulsionou os serviços associados ao processamento de imagens (*image-worker*) até o ponto de saturação, como ilustrado na **Tabela 10**, o

que demandou a instância de três réplicas pelo *Kubernetes*. Os picos de consumo de recursos durante esse período estão detalhados a seguir:

- **Uso de CPU**: 997 *milicores* + 996 *milicores* + 998 *milicores* = 2991 *milicores*.
- **Uso de Memória**: 460 *Megabytes* + 447 *Megabytes* + 464 *Megabytes* = 1371 *Megabytes*.

Outro serviço que apresentou um consumo significativo de recursos foi o *RabbitMQ* (*rabbitmq*). Nesse caso, a demanda maior estava na utilização da memória, uma vez que era necessário armazenar as imagens em uma fila para posterior consumo pelas instâncias do serviço de processamento de imagens. Em relação ao poder de processamento, observou-se um pico máximo de 50 *milicores* de capacidade de processamento e um consumo de 529 *Megabytes* de memória RAM.

No que tange ao serviço de usuários (*user*), houve a utilização de recursos relacionados ao fluxo de processamento de imagens, uma vez que este serviço é responsável por validar e encaminhar as imagens para serem enfileiradas no *RabbitMQ*, a fim de serem processadas pelas instâncias dos serviços de imagens. Vale destacar que o *Kubernetes* criou duas instâncias do serviço de usuários para a execução dessas tarefas. A seguir, são apresentadas as métricas detalhadas de consumo de recursos:

- **Uso de CPU**: 183 *milicores* + 424 *milicores* = 607 *milicores*.
- **Uso de Memória**: 313 *Megabytes* + 238 *Megabytes* = 551 *Megabytes*.

No contexto deste teste, destaca-se um notável aprimoramento de desempenho da aplicação baseada em microsserviços em relação ao tempo de resposta obtido pela aplicação monolítica. Esse aprimoramento pode ser atribuído, em parte, à alocação específica de recursos para o serviço de imagens, o que impediu a sobrecarga total do *cluster*. Além disso, um fator decisivo para o baixo tempo de resposta é o padrão de implementação relacionado à arquitetura de microsserviços, em particular, a abordagem de enfileiramento no *RabbitMQ* e o consumo assíncrono por parte dos trabalhadores (*workers*).

Esse arranjo permite que as tarefas sejam distribuídas eficientemente, otimizando a utilização de recursos e minimizando o tempo de resposta, demonstrando assim os benefícios práticos da arquitetura de microsserviços em ambientes de alta demanda.

Os testes de carga revelaram o potencial da arquitetura monolítica em otimizar o uso de recursos, proporcionando menor latência na maioria dos casos. Por outro lado, a aplicação baseada em microsserviços demonstrou notável flexibilidade, apresentando latência significativamente inferior e a capacidade de limitação de recursos no último teste. No entanto, observou-se um custo mais elevado na utilização de recursos de hardware nos demais testes. No próximo capítulo, os resultados serão minuciosamente discutidos à luz das questões de pesquisa previamente estabelecidas.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este estudo proporcionou uma análise abrangente das arquiteturas de microsserviços e monolítica no contexto de uma aplicação de chat, com foco em testes de carga nas principais funcionalidades. Com base nos resultados e discussões apresentados nos capítulos anteriores, é possível destacar diversas conclusões que podem auxiliar na tomada de decisões em projetos similares e fornecer *insights* para futuros desenvolvimentos.

5.1 Sumário da pesquisa

No que diz respeito à investigação da **QP1**: A escalabilidade varia significativamente entre a arquitetura monolítica e a arquitetura de microsserviços? Com base nos testes de carga executados, no contexto dos microsserviços, observou-se que o *Kubernetes* escalou de forma horizontal automaticamente múltiplas instâncias do serviço alvo de carga. Em contrapartida, na abordagem monolítica, seria necessário realizar uma escala vertical ou a replicação completa da aplicação a fim de balancear a carga. Essa diferença na estratégia de escalabilidade pode ter implicações significativas no desempenho e na eficiência dos sistemas. Portanto, a escalabilidade varia conforme a arquitetura adotada e as estratégias de dimensionamento utilizadas.

No âmbito da investigação abordada pela **QP2**: as arquiteturas monolíticas e de microsserviços apresentam diferenças notáveis quanto à utilização eficaz de recursos de *hardware* e ao consumo de recursos computacionais em situações de alta demanda? Com base nos testes de carga, fica evidente que, em determinadas funcionalidades, como a recuperação de mensagens públicas, a arquitetura monolítica demonstrou um desempenho superior, resultando em uma utilização mais eficaz dos recursos e tempos de resposta mais rápidos. No entanto, na funcionalidade de persistência de imagens de perfil, a arquitetura de microsserviços destacou-se, mostrando uma utilização mais eficaz de recursos e tempos de resposta significativamente mais baixos, devido à capacidade de alocar recursos de forma específica para funções individuais. Portanto, fica evidente que a eficiência na utilização de recursos e o consumo de recursos computacionais variam consideravelmente com base na natureza das tarefas e nas características da carga de trabalho.

Em relação à investigação da **QP3**: Como a latência e o tempo de resposta de sistemas baseados em arquitetura de microsserviços se comparam aos sistemas monolíticos em situações de carga variável e como essa comparação influencia o desempenho? No contexto dos testes de carga realizados sob situações de carga variável, os resultados destacaram diferenças significativas nos tempos médios de resposta entre a arquitetura monolítica e a arquitetura de microsserviços. Em algumas funcionalidades, como a autenticação no processo de login, a arquitetura monolítica registrou tempos de resposta mais curtos. No entanto, ao investigar a funcionalidade relacionada ao fluxo *Websocket*, notou-se que os tempos de resposta foram aproximados, mas com uma demanda substancial de recursos por parte da aplicação baseada em microsserviços, incluindo uma significativa utilização de poder de processamento. Por outro lado, em funcionalidades como

a persistência de imagens de perfil, a arquitetura de microsserviços sobressaiu-se com tempos de resposta notavelmente mais baixos. Essa distinção marcante é atribuída à adoção de um padrão de enfileiramento de tarefas e ao processamento assíncrono nos microsserviços, juntamente com a capacidade de limitar recursos de forma específica para cada serviço. Consequentemente, a comparação de latência e tempo de resposta é amplamente influenciada pela natureza da funcionalidade analisada e pelas particularidades da carga de trabalho aplicada.

No que diz respeito à investigação da **QP4**: em termos de desempenho, quais são os impactos das comunicações entre componentes em uma arquitetura de microsserviços em comparação com a arquitetura monolítica? Conforme observado nos testes realizados, um dos principais fatores de impacto no desempenho em arquiteturas de microsserviços é a quantidade de chamadas de *APIs* internas. Quanto mais chamadas de *APIs* internas são feitas, maior é a latência gerada na aplicação de microsserviços. Isso ocorre devido ao *overhead* introduzido pela comunicação entre os diversos componentes distribuídos. Esse *overhead* de comunicação pode afetar de maneira significativa o desempenho da aplicação, especialmente em sistemas altamente distribuídos. Portanto, a avaliação dos impactos das comunicações entre componentes deve levar em consideração não apenas a natureza das tarefas, mas também a quantidade de chamadas de *APIs* internas, pois isso pode influenciar diretamente na latência e no desempenho geral.

5.2 Trabalhos futuros

Nesta seção é explorado as direções nas quais este estudo pode ser expandido e aprimorado. Considerando os resultados e análises realizadas, há várias áreas que merecem atenção em pesquisas e desenvolvimentos futuros.

- **Otimização da Arquitetura de Microsserviços**: os resultados destacam o potencial da arquitetura de microsserviços, mas também apontam para a necessidade de otimização na gestão de recursos. Investigar estratégias mais eficazes de escalabilidade e balanceamento de carga pode ajudar a reduzir o consumo de memória e melhorar ainda mais o desempenho.

- **Segurança e Tolerância a Falhas**: novos estudos podem explorar a segurança e a resiliência das duas arquiteturas. Isso inclui a implementação de mecanismos de segurança em ambientes de microsserviços e a avaliação da capacidade de recuperação de falhas em situações de alta carga.

- **Impacto na Experiência do Usuário**: além dos aspectos técnicos, é importante considerar o impacto da escolha arquitetural na experiência do usuário. Avaliar o desempenho sob a perspectiva do usuário final, levando em conta fatores como latência e confiabilidade, pode ser um foco para trabalhos futuros.

REFERÊNCIAS

AL-DEBAGY, O.; MARTINEK, P. A comparative review of microservices and monolithic architectures. *In*: **2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)**. Budapest, Hungary: IEEE, 2018. p. 149–154. Disponível em: https://ieeexplore.ieee.org/abstract/document/8928192. Acesso em: 6 set. 2023. Citado 2 vezes nas páginas 14 e 26.

AMARAL, O.; CARVALHO, M. **Arquitetura de Micro Serviços**: Uma comparação com sistemas monolíticos. Universidade Federal da Paraíba, 2017. Disponível em: https://repositorio.ufpb.br/jspui/handle/123456789/3235. Acesso em: 6 set. 2023. Citado 2 vezes nas páginas 14 e 27.

AMAZON. **O que é o Scrum?** 2023. Disponível em: https://aws.amazon.com/pt/what-is/scrum/. Acesso em: 04 dez. 2023. Citado na página 32.

APACHE. **Apache JMeter**. 2023. Disponível em: https://jmeter.apache.org/. Acesso em: 19 set. 2023. Citado 6 vezes nas páginas 14, 56, 59, 62, 64 e 66.

BASS, L.; CLEMENTS, P.; KAZMAN, R. What is software architecture? *In*: BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 3. ed. Massachusetts, USA: Pearson Education, Inc., 2013. Disponível em: https://edisciplinas.usp.br/pluginfile.php/5922722/mod_resource/content/1/2013%20-%20Book%20-%20Bass%20%20Kazman-Software%20Architecture%20in%20Practice%20%281%29.pdf. Acesso em: 6 set. 2023. Citado na página 21.

BERNERS-LEE, T.; FIELDING, R.; FRYSTYK, H. **Hypertext transfer protocol–HTTP/1.0**. 1996. Disponível em: https://www.rfc-editor.org/rfc/rfc1945. Acesso em: 7 set. 2023. Citado 2 vezes nas páginas 18 e 19.

BOSCH, J. Design and use of industrial software architectures. *In*: **Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)**. [S.l.: s.n.], 1999. p. 404–404. Acesso em: 04 dez. 2023. Citado na página 14.

CANONICAL. **Ubuntu**. 2023. Disponível em: https://ubuntu.com. Acesso em: 1 set. 2023. Citado na página 51.

DOCKER. **Docker overview**. 2023. Disponível em: https://docs.docker.com/get-started/overview/. Acesso em: 19 set. 2023. Citado 2 vezes nas páginas 15 e 51.

DOCKER. **What is a Container?** 2023. Disponível em: https://www.docker.com/resources/what-container/. Acesso em: 10 de set. 2023. Citado na página 25.

DRAGONI, N. *et al.* Microservices: Yesterday, today, and tomorrow. *In*: MAZZARA, M.; MEYER, B. **Present and Ulterior Software Engineering**. Cham: Springer International Publishing, 2017. p. 195–216. Disponível em: https://doi.org/10.1007/978-3-319-67425-4_12. Acesso em: 6 set. 2023. Citado 2 vezes nas páginas 21 e 22.

DUARTE, J. P. L. **Análise comparativa entre arquitetura monolítica e de microsserviços**. Universidade Federal de Santa Catarina, 2017. Disponível em: https://repositorio.ufsc.br/handle/123456789/182309. Acesso em: 6 set. 2023. Citado 2 vezes nas páginas 14 e 27.

FETTE, I.; MELNIKOV, A. **The websocket protocol**. 2011. Disponível em: https://www.rfc-editor.org/rfc/rfc6455. Acesso em: 7 set. 2023. Citado na página 19.

FOWLER, M. **Richardson maturity model**. 2010. Disponível em: http://martinfowler.com/articles/richardsonMaturityModel.html. Acesso em: 9 set. 2023. Citado na página 20.

FOWLER, M. **PolyglotPersistence**. 2011. Disponível em: https://martinfowler.com/bliki/PolyglotPersistence.html. Acesso em: 11 nov. 2023. Citado na página 24.

FOWLER, M.; LEWIS, J. **Microservices**. 2014. Disponível em: http://martinfowler.com/articles/microservices.html. Acesso em: 6 set. 2023. Citado 5 vezes nas páginas 6, 7, 15, 23 e 24.

GOOGLE. **Google Acadêmico**. 2023. Disponível em: https://scholar.google.com.br/?hl=pt. Acesso em: 1 set. 2023. Citado na página 31.

IEEE. **IEEE Xplore**. 2023. Disponível em: https://ieeexplore.ieee.org/Xplore/home.jsp. Acesso em: Acesso em: 1 set. 2023. Citado na página 31.

JACOBSON, D.; BRAIL, G.; WOODS, D. The api opportunity. *In*: JACOBSON, D.; BRAIL, G.; WOODS, D. **APIs**: A strategy guide. O'Reilly Media, Inc., 2012. p. 1–8. Disponível em: https://bit.ly/40Pplkg. Acesso em: 9 set. 2023. Citado na página 20.

KUBERNETES. **Kubernetes**. 2023. Disponível em: https://kubernetes.io/. Acesso em: 19 set. 2023. Citado 9 vezes nas páginas 15, 51, 59, 60, 62, 63, 64, 65 e 67.

KUROSE, J. F.; ROSS, K. W. Redes de computadores e a internet. *In*: KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson Education do Brasil Ltda., 2014. p. 2–5. Citado na página 18.

MOHR, F.; ROBERTSON, M. **Mi Jmeter SockJS Stomp Sampler**. 2020. Disponível em: https://github.com/MovingImage24/JmeterSockJsSampler. Acesso em: 13 set. 2023. Citado na página 57.

NGINX. **What is NGINX?** 2023. Disponível em: https://www.nginx.com/resources/glossary/nginx/. Acesso em: 11 nov. 2023. Citado na página 52.

ORACLE. **MySQL 8.1 Reference Manual**. 2023. Disponível em: https://dev.mysql.com/doc/refman/8.1/en/what-is-mysql.html. Acesso em: 11 out. 2023. Citado na página 35.

ORACLE. **What is a Kubernetes cluster?** 2023. Disponível em: https://www.oracle.com/cloud/cloud-native/container-engine-kubernetes/what-is-kubernetes/cluster/. Acesso em: 11 nov. 2023. Citado na página 51.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. **Communications of the ACM**, v. 17, n. 7, p. 412–421, 1974. Disponível em: https://dl.acm.org/doi/10.1145/361011.361073. Acesso em: 10 set. 2023. Citado na página 24.

PORTNOY, M. Understanding virtualization. *In*: PORTNOY, M. **Virtualization essentials**. John Wiley & Sons, 2012. v. 19, p. 1–18. Disponível em: https://books.google.com.br/books?id=5PSVnKvAVTMC&printsec=frontcover&hl=ptBR#v=onepage&q&f=false. Acesso em: 10 set. 2023. Citado na página 24.

RabbitMQ. **What can RabbitMQ do for you?** 2023. Disponível em: https://www.rabbitmq.com/features.html. Acesso em: 11 out. 2023. Citado na página 45.

REACT. **React – A JavaScript library for building user interfaces**. 2023. Disponível em: https://legacy.reactjs.org/. Acesso em: 19 set. 2023. Citado na página 46.

RICHARDS, M.; FORD, N. Introduction. *In*: RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture**: An engineering approach. USA: O'Reilly Media, 2020. Disponível em: https://amzn.to/47LoRxK. Acesso em: 6 set. 2023. Citado 3 vezes nas páginas 14, 21 e 22.

RICHARDSON, L. **Justice Will Take Us Millions of Intricate Moves**: Act three: The maturity heuristic. 2008. Disponível em: https://www.crummy.com/writing/speaking/2008-QCon/act3.html. Acesso em: 11 nov. 2023. Citado na página 20.

ROSEN, L.; SHKLAR, L. 1. introduction. *In*: ROSEN, L.; SHKLAR, L. **Web application architecture**: Principles, protocol and practices. 1. ed. Chichester, West Sussex, Inglaterra: John Wiley & Sons Ltd., 2009. p. 1–10. Citado na página 14.

ROSEN, L.; SHKLAR, L. 2. before the web: Tcp/ip. *In*: ROSEN, L.; SHKLAR, L. **Web application architecture**: Principles, protocol and practices. 1. ed. Chichester, West Sussex, Inglaterra: John Wiley & Sons Ltd., 2009. p. 11–28. Citado na página 18.

ROSEN, L.; SHKLAR, L. 3. birth of the world wide web: Http. *In*: ROSEN, L.; SHKLAR, L. **Web application architecture**: Principles, protocol and practices. 1. ed. Chichester, West Sussex, Inglaterra: John Wiley & Sons Ltd., 2009. p. 29–42. Citado na página 18.

SPRING. **Spring Boot**. 2023. Disponível em: https://spring.io/projects/spring-boot. Acesso em: 19 set. 2023. Citado na página 37.

SPRING. **Spring Boot Reference Documentation**. 2023. Disponível em: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/. Acesso em: 11 out. 2023. Citado na página 38.

SPRING. **Spring Cloud Gateway**. 2023. Disponível em: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/. Acesso em: 11 out. 2023. Citado na página 46.

STATISTA. **Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025**. 2021. Disponível em: https://www.statista.com/statistics/871513/worldwide-data-created/. Acesso em: 19 set. 2023. Citado 2 vezes nas páginas 14 e 16.

STOMP. **STOMP Protocol Specification version 1.2**. 2012. Disponível em: http://stomp.github.io/stomp-specification-1.2.html. Acesso em: 7 set. 2023. Citado na página 19.

WAINER, J. Métodos de pesquisa quantitativa e qualitativa para a ciência computação. *In*: KOWALTOWSKI, T.; BREITMAN, K. **Atualização em informática 2007**. Sociedade Brasileira de Computação e Editora PUC-Rio, 2007. p. 221–262. Disponível em: https://www.ic.unicamp.br/~wainer/cursos/1s2018/metodologia/Metodos_de_pesquisa_quantitativa_e_qualitativa_par.pdf. Acesso em: 22 out. 2023. Citado na página 29.

WAZLAWICK, R. S. Preparação de um trabalho de pesquisa. *In*: WAZLAWICK, R. S. **Metodologia de pesquisa para ciência da computação**. Rio de Janeiro: Elsevier, 2009. v. 2. Citado 2 vezes nas páginas 29 e 30.