A migração de um sistema monolítico para uma arquitetura de microsserviços: o caso do Cooperative Editor

Trabalho de Conclusão do Curso de
Tecnologia em Sistemas para Internet

Ricardo Elias de Albuquerque Waldow
Orientador(a): Rodrigo Prestes Machado

[1]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Porto Alegre
Av Cel Vicente, 281, Porto Alegre – RS – Brasil

tatow2001@gmail.com, rodrigo.prestes@poa.ifrs.edu.br

Resumo. *Este trabalho tem como objetivo explorar a refatoração do sistema Cooperative Editor, uma aplicação de edição colaborativa que, atualmente, utiliza uma arquitetura monolítica. O estudo foi conduzido em três fases: a investigação do funcionamento do Cooperative Editor, a refatoração do sistema para uma arquitetura de microsserviços com compilação nativa, e a verificação do sistema. Este artigo discute as consequências dessa transição, analisando as melhorias e os impactos na manutenção e escalabilidade do sistema. Os resultados demonstraram que a refatoração para uma arquitetura de microsserviços, aliada à migração para o Quarkus, trouxe melhorias significativas tanto na manutenção quanto na escalabilidade do sistema.*

1. Introdução

A Engenharia de Software desempenha um papel crucial no desenvolvimento e manutenção de sistemas de software. Ela envolve a aplicação de princípios, métodos e práticas para garantir que o software seja desenvolvido com eficiência, qualidade e manutenibilidade. A refatoração de sistemas antigos é um aspecto essencial dessa disciplina, permitindo aprimorar sistemas legados, torná-los mais eficientes e fáceis de manter. A aplicação adequada da engenharia de software nesse contexto pode resultar em melhorias significativas na longevidade e na capacidade de adaptação de sistemas antigos [Fowler 2018].

O sistema Cooperative Editor, por exemplo, é um exemplo de software que enfrenta desafios de manutenção devido à sua arquitetura e tecnologias subjacentes. A arquitetura legada desse sistema tornou-se um obstáculo à sua efetiva manutenção e evolução. Além disso, a utilização do *framework Wildfly*, que já não está alinhado com as práticas e tecnologias modernas, contribui para a dificudalde de escalabilidade do sistema.

A refatoração do sistema para uma arquitetura de microsserviços é crucial para manter sua relevância na era da computação em nuvem. Essa modernização não só alinhará o sistema com as práticas contemporâneas de desenvolvimento de software, mas também possibilitará a sua utilização pelos interessados no sistema no meio acadêmico. Em um ambiente cada vez mais voltado para a colaboração remota, a modularidade dos microsserviços permitirá uma manutenção mais fácil e uma integração mais eficiente em

infraestruturas de nuvem, melhorando significativamente a experiência do usuário e a eficácia do sistema.

A justificativa para este trabalho é a percepção de que o sistema Cooperative Editor poderia continuar sendo utilizado em pesquisas, mas está estagnado devido à sua arquitetura antiga. A refatoração do sistema, voltada para a arquitetura de microsserviços, pode torná-lo mais escalável, flexível e facilmente implementável na infraestrutura de nuvem, atendendo às demandas atuais.

Os objetivos deste trabalho incluem a refatoração do sistema Cooperative Editor, transformando-o em uma arquitetura de microsserviços, tornando-o mais adaptável a ambientes de nuvem e substituindo o *framework* Wildfly pelo *framework* Quarkus, que é mais moderno e eficiente. Além disso, busca-se observar as consequências da migração, avaliando as melhorias e os impactos na manutenção e escalabilidade do sistema.

A metodologia adotada para esta pesquisa envolverá a análise da arquitetura atual do sistema Cooperative Editor, a investigação do funcionamento do sistema, a migração para o *framework Quarkus*, o desenvolvimento de uma arquitetura baseada em microsserviços e a verificação do sistema, visando a avaliação da manutenção e da escalabilidade resultante.

2. Referencial Teórico

2.1. Arquitetura Monolítica

A arquitetura monolítica tem sido um ponto de discussão significativo no campo da engenharia de software. Como afirmado por Martin Fowler, um proeminente autor na área, a arquitetura monolítica é um estilo arquitetural no qual ”todos os componentes do sistema estão combinados em um único programa, geralmente um único processo”[Fowler 2002]. A natureza consolidada da arquitetura monolítica pode resultar em desafios relacionados à escalabilidade, manutenção e flexibilidade do sistema. Entender as implicações dessa arquitetura é crucial para a tomada de decisões informadas no desenvolvimento de software.

A arquitetura monolítica apresenta desafios específicos em relação à escalabilidade e manutenção de sistemas. Em seu livro *”Patterns of Enterprise Application Architecture”*, Martin Fowler aborda a falta de flexibilidade da arquitetura monolítica ao afirmar que ”mudanças em uma parte do sistema podem afetar adversamente outras partes”[Fowler 2002]. Isso pode dificultar a adaptação de sistemas monolíticos a novos requisitos ou à adição de funcionalidades.

2.2. Arquitetura de Microsserviços

Segundo Martin Fowler a arquitetura de microsserviços é um estilo arquitetural em que um aplicativo é dividido em pequenos serviços independentes, cada um executando uma função específica [Fowler 2014]. Esses microsserviços são implantados e escalados separadamente, permitindo uma maior flexibilidade no desenvolvimento e na manutenção de sistemas de software.

A arquitetura de microsserviços surge como uma estratégia eficaz para superar as limitações de sistemas monolíticos, permitindo a construção de sistemas mais flexíveis e escaláveis. A migração de uma arquitetura monolítica para microsserviços é um processo

desafiador, que requer uma reestruturação cuidadosa e a definição clara de serviços independentes, com interfaces bem estabelecidas. A principal vantagem dessa abordagem está na possibilidade de isolar falhas, escalabilidade independente e a agilidade no desenvolvimento, permitindo que equipes tratem de funcionalidades específicas sem afetar o sistema como um todo [Dehghani 2014].

Embora a arquitetura de microsserviços ofereça muitos benefícios, também apresenta desafios específicos. Como descrito por Fowler, "a complexidade de gerenciar várias partes independentes pode ser um desafio significativo"[Fowler 2014]. Além disso, garantir a comunicação eficiente entre os microsserviços e a manutenção de uma visão global do sistema pode requerer esforço adicional.

2.3. Cooperative Editor

O Cooperative Editor, concebido como parte do terceiro estudo da tese de doutorado de Machado (2019), fundamenta-se na prática de Aprendizagem Cooperativa denominada *Circle of Writes – Take Turns* (Círculo de Escritores). Esta prática consiste em posicionar os estudantes ao redor de uma mesa, permitindo que uma folha de papel circule para receber as contribuições individuais para a realização de uma tarefa. O Cooperative Editor, desenvolvido como uma aplicação Web, se alinha a essa prática versátil, expandindo suas possibilidades de aplicação em diversas áreas do conhecimento, como matemática, ciências e línguas. Sua concepção atende ao segundo objetivo específico da pesquisa, buscando implementar uma plataforma que viabilize ações síncronas e cooperativas entre indivíduos com deficiência visual [Machado 2019].

No âmbito do Cooperative Editor, diferentemente do Círculo de Escritores, não há uma ordem circular predefinida para as contribuições dos membros do grupo. Essa abordagem, conforme destacado por Machado e colaboradores (2019), requer a interação constante dos usuários para coordenar suas ações. No entanto, o sistema assegura a participação equitativa, possibilitando que um usuário faça uma nova contribuição somente após todos os outros membros do grupo terem participado. Assim, o Cooperative Editor orienta a prática para atividades síncronas de escrita coletiva, alinhando-se às teorias de Aprendizagem Cooperativa ao fomentar a interação e a interdependência entre os participantes na realização de tarefas educacionais.

A interface do Cooperative Editor é projetada subdividindo-se em seis áreas distintas. A primeira área exibe os usuários conectados, o número de participações restantes e o status de cada membro do grupo. A segunda área consiste no bate-papo textual, funcionando como uma ferramenta de comunicação entre os participantes. As opções de configuração, presentes na terceira área, possibilitam ajustes nos recursos sonoros, adaptando-se às necessidades específicas dos usuários. O objetivo da tarefa, destacado na quarta área, direciona a atividade dos estudantes. A área de rubricas, quinta na ordem, atua como um instrumento de autorregulação da ação dos estudantes. Por fim, a sexta área, o editor, armazena as contribuições dos usuários e serve como recurso para a navegação em um histórico, proporcionando uma experiência rica e orientada para a aprendizagem cooperativa [Machado 2019].

2.4. Quarkus

O Quarkus é projetado para reduzir o tamanho da imagem do Docker, melhorar o tempo de inicialização e otimizar o uso de memória em aplicativos Java. GraalVM de-

sempenha um papel crucial aqui, permitindo a compilação de código Java em código nativo. O *framework* facilita a transição de desenvolvedores Spring para Quarkus, realçando a eficiência da sua arquitetura em comparação com *frameworks* tradicionais [Escoffier and Andrianakis 2020].

A flexibilidade do Quarkus o torna especialmente adequado para o desenvolvimento de microserviços em ambientes de nuvem e Kubernetes, simplificando o processo com uma pilha de tecnologia que abrange desde RESTEasy para serviços REST até Hibernate ORM para acesso a banco de dados, além de um modelo reativo para aplicações escaláveis [Clingan and Finnigan 2020].

Esse framework é pioneiro no desenvolvimento nativo para a nuvem, permitindo que aplicações Java sejam executadas de forma mais eficiente em ambientes de nuvem. A combinação de Quarkus com containers e Kubernetes oferece uma solução poderosa para a implementação de microserviços e que se encaixa no ecossitema Java moderno, oferecendo uma opção robusta para desenvolvedores que buscam otimizar aplicações para a nuvem [Porter 2020].

A compilação nativa é uma das principais funcionalidades do Quarkus, permitindo aos desenvolvedores criar aplicativos que se destacam por iniciarem rapidamente e operarem com uma utilização significativamente menor de memória. Diferentemente dos aplicativos Java tradicionais, que dependem da Máquina Virtual Java (JVM) para execução, a compilação nativa gera executáveis autônomos, eliminando a necessidade de um ambiente de tempo de execução intermediário. Essa abordagem está em total consonância com o foco do Quarkus em ser leve e otimizado para ambientes nativos de nuvem[Kariyakarana 2023].

2.5. K6

A ferramenta K6 é uma plataforma open-source para testes de carga e desempenho que permite a avaliação de sistemas distribuídos sob diferentes condições de uso. Desenvolvida inicialmente pela Load Impact e atualmente mantida pela Grafana Labs, a ferramenta se destaca por sua integração com ecossistemas modernos de observabilidade, como o Grafana, e pelo suporte a scripts baseados em JavaScript, promovendo um ambiente acessível e extensível para desenvolvedores e engenheiros de desempenho [Labs 2024].

Um dos principais diferenciais do K6 é a sua capacidade de simular cenários realistas por meio de uma configuração precisa de estágios de carga. Segundo a documentação da Grafana Labs, a ferramenta suporta tanto cenários estáticos, onde o número de usuários virtuais permanece constante, quanto dinâmicos, como *ramp-up* e *ramp-down*, essenciais para analisar o comportamento de sistemas frente a cargas variáveis.

3. Trabalhos Relacionados

Pesquisas anteriores têm demonstrado a eficácia da migração de sistemas monolíticos para arquiteturas de microsserviços na melhoria da escalabilidade e manutenção de sistemas de *software* [Tostes and Sirqueira 2023]. Um estudo relevante neste contexto é o trabalho que explora a transição para microsserviços utilizando abordagens *serverless*, como exemplificado por ”O processo de migração de um monolito para uma arquitetura

de microsserviços utilizando serverless"[Tostes and Sirqueira 2023]. Este trabalho destaca as limitações dos sistemas monolíticos tradicionais e argumenta em favor da maior flexibilidade, escalabilidade e resiliência das arquiteturas de microsserviços.

O artigo em questão detalha o processo de migração, enfatizando a independência dos componentes e a adoção de serviços serverless, como *AWS Lambda*. Esta abordagem, em contraste com métodos mais tradicionais, sugere reduções significativas em complexidade de gerenciamento e custos de manutenção, conforme demonstrado pelos resultados obtidos. Estes achados corroboram com a literatura existente que aponta para as vantagens da migração para microsserviços em termos de agilidade de desenvolvimento e implantação, alinhando-se com as tendências atuais na engenharia de software [Tostes and Sirqueira 2023].

Em complemento a esta perspectiva, o artigo "Migração de sistemas monolíticos para microsserviços", de Anderson Almeida de Souza Clemente e Evaldo de Oliveira da Silva (2022), fornece uma análise detalhada do processo de transição para arquiteturas baseadas em microsserviços. O estudo destaca as dificuldades associadas com sistemas monolíticos, principalmente no que tange a complexidade e manutenção, e contrasta com os benefícios proporcionados pelos microsserviços, como maior facilidade de atualização, manutenção e escalabilidade[de Souza Clemente and da Silva 2022].

A pesquisa enfatiza que, embora a transição para arquiteturas de microsserviços ofereça numerosas vantagens, ela também pode apresentar desafios significativos. O artigo sublinha a importância do planejamento cuidadoso e consideração de vários aspectos, como a fragmentação da arquitetura, interoperabilidade dos serviços e manutenção da funcionalidade durante a migração. Técnicas específicas como o padrão do estrangulador, a separação entre *frontend* e *backend*, e a extração de serviços são discutidas, cada uma com seus desafios e peculiaridades [de Souza Clemente and da Silva 2022]. Este estudo oferece pontos valiosos para a implementação eficaz de microsserviços, complementando os achados de Tostes (2023) sobre a adoção de abordagens serverless na migração de sistemas.

Por último, o trabalho de Stradolini (2021) também oferece uma visão abrangente sobre a migração de sistemas monolíticos para microserviços, destacando os desafios e benefícios inerentes a este processo [Stradolini 2021]. Este estudo é particularmente relevante por sua abordagem prática e análise detalhada das etapas envolvidas na migração. As discussões focam em aspectos críticos, como a decomposição de um monolito em serviços menores e mais gerenciáveis, a importância de uma comunicação eficiente entre serviços e a necessidade de considerar questões de segurança e consistência de dados. Estas considerações são fundamentais para compreender as nuances envolvidas na migração para uma arquitetura de microserviços, e ressoam com as conclusões de Tostes (2023), em termos da importância da escolha de tecnologias e estratégias adequadas para a transição.

Além disso, o estudo ressalta a importância da escolha cuidadosa de padrões de design e ferramentas de desenvolvimento para facilitar a migração e manutenção dos microserviços. Esta perspectiva é complementar ao trabalho de Tostes (2023), que se concentra na eficiência operacional trazida pelas arquiteturas *serverless*. Ambos os estudos destacam a complexidade e os desafios da migração, mas também reforçam o potencial de melhoria em termos de escalabilidade, flexibilidade e agilidade na entrega de software.

4. Metodologia

O estudo compreendeu três fases: investigação do funcionamento do Cooperative Editor, refatoração do Cooperative Editor para uma arquitetura de microsserviços e compilação nativa, verificação do sistema. Durante a primeira fase foi realizada a investigação do sistema existente, com o objetivo de compreender sua arquitetura, fluxos de dados, e componentes funcionais, bem como identificar limitações e dependências no código. Esta investigação inicial foi crucial para entender as dependências e os desafios técnicos, proporcionando a base para as decisões subsequentes.

Na segunda fase ocorreu a migração do sistema para o *framework Quarkus*, escolhido visando otimizar o desempenho e a escalabilidade do sistema, a versão do framework utilizada foi a ”3.0.3.Final”. Durante esse processo, o sistema foi refatorado para uma arquitetura baseada em, o que envolveu a decomposição do sistema monolítico em componentes e as correções necessárias para o funcionamento das APIs, esforçando-se para evitar ao máximo a reescrita do código, a fim de preservar a integridade da comparação dos resultados. Após isso foi integrado ao *front-end*.

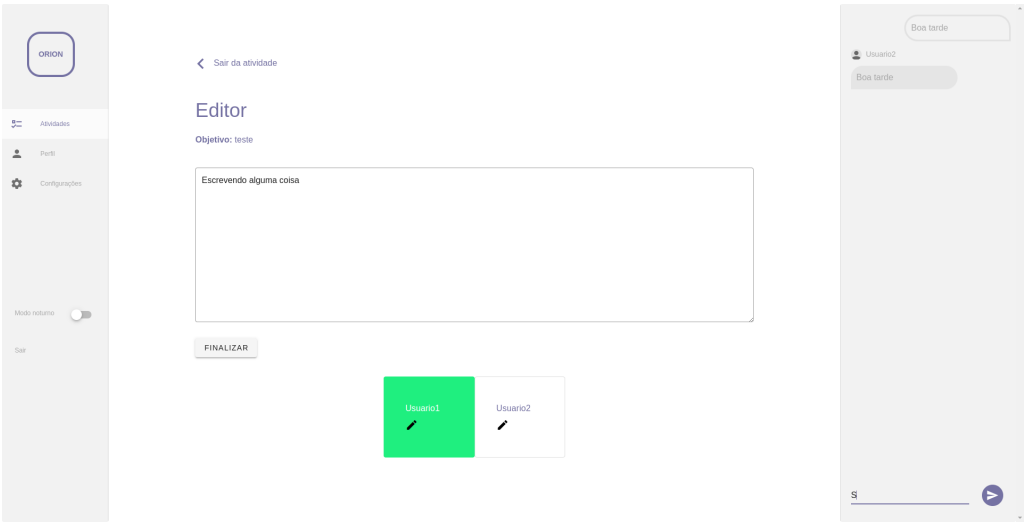

Figura 1. Formato do Front-end

Na terceira fase ocorreu a verificação do sistema por meio de testes. Foram feitos três testes: (1) contagem do número de linhas, (2) Tamanho da imagem (*Docker*) após compilação nativa, (3) tempo de resposta em diferentes cenários (Com e sem balanceador de carga). Os experimentos foram conduzidos em um ambiente computacional configurado com o sistema operacional Ubuntu 22.04.4 LTS, arquitetura x86_64, equipado com um processador AMD Ryzen 7 5700U, uma GPU integrada AMD ATI Lucienne, 20 GB de memória RAM e armazenamento em unidade de estado sólido (SSD).

5. Resultados e Discussões

Após a investigação do sistema original e a migração do *framework* para o *Quarkus*, seguida pela refatoração do sistema para uma arquitetura baseada em microsserviços, na qual o sistema foi segmentado em três componentes principais: autenticação, edição colaborativa e chat. Esta seção abordará os resultados e discussões sobre dois pontos de vista: manutenção e escalibilidade.

5.1. Manutenção

Nesta subseção, serão apresentados e analisados os resultados referentes ao número de linhas de código, considerando sua relação com a complexidade e a manutenibilidade dos microsserviços.

Em relação ao número de linhas de código, observou-se uma diferença marginal após a migração entre os monolitos (Tabela 1), com uma variação de pouco mais de 200 linhas entre as versões comparadas. Contudo, ao se analisar essa diferença no contexto das alterações implementadas, é possível identificar uma otimização significativa na gestão da persistência dos objetos. Anteriormente, essa gestão era realizada por meio de um único objeto DAO, sendo substituída por múltiplos objetos de repositórios especializados para cada entidade, o que resultou em uma estrutura mais organizada e modularizada para as funções associadas a cada entidade. A adoção do Panache nos repositórios possibilitou uma redução no número de linhas de código, que passou de 592 para 496 nas classes afetadas pela mudança. Esse ajuste representou uma diminuição de 16,21% no número total de linhas, o que equivale a 83,78% do código original.

Tabela 1. Número de linhas

Projeto	Linhas
Monolito (Legado)	6364
Monolito (Quarkus)	6131

Ao analisar os números pós-refatoração, observa-se uma redução substancial nas linhas de código do serviço de autenticação, o que pode ser atribuído ao fato de ser um dos serviços mais simples, com menos dependências de repositórios e entidades. Além disso, esse foi o serviço que passou por maior reescrita, uma vez que, no modelo monolítico, o login era validado por atributos de sessão de servlet, abordagem que não seria compatível com a arquitetura REST API adotada pelos microsserviços.

Por outro lado, o serviço de edição colaborativa não apresentou uma diminuição significativa no número de linhas, uma vez que é o serviço de maior porte e com o maior número de dependências. Embora fosse possível reescrever algumas partes do código para torná-lo mais conciso e menos complexo, tais alterações não eram essenciais para o funcionamento imediato do serviço e, portanto, foram adiadas para futuras melhorias. O serviço de chat apresenta uma situação semelhante: embora tenha ocorrido uma redução considerável no número de linhas devido à diminuição das dependências, ainda seria viável reescrevê-lo para reduzir sua complexidade.

O microsserviço de autenticação, por ser relativamente simples e independente de outras partes do sistema, apresenta uma série de vantagens em termos de manutenção e reutilização. Sua estrutura desacoplada facilita a implementação de alterações, uma vez que não há dependências complexas que possam impactar outras funcionalidades. Além disso, devido à sua simplicidade, esse microsserviço pode ser facilmente reutilizado em outros projetos. Sua implementação modular permite que seja integrado rapidamente a novas soluções, sem a necessidade de ajustes significativos, o que aumenta sua flexibilidade e valor em cenários de projetos futuros. Em resumo, a natureza independente do microsserviço de autenticação contribui não apenas para sua manutenção simplificada,

mas também para sua alta capacidade de reutilização.

Por outro lado, os microsserviços de edição colaborativa e chat apresentam uma maior complexidade devido às suas várias dependências e interações com outros componentes do sistema. Essa complexidade aumenta a dificuldade de manutenção, já que mudanças em uma parte do microsserviço podem gerar impactos inesperados em outras áreas, exigindo um esforço contínuo de monitoramento e atualização. Para simplificar sua manutenção e garantir uma maior flexibilidade, seria necessário um esforço considerável de reescrita, visando reduzir a complexidade e modularizar melhor suas funcionalidades. Além disso, a capacidade de reutilização desses microsserviços é limitada, principalmente devido à sua dependência direta do microsserviço de autenticação. Para que fossem reutilizados em outro projeto, seria necessário que o novo sistema adotasse a mesma estratégia de autenticação, ou então seria imprescindível uma reescrita significativa para adaptar o microsserviço a uma nova forma de autenticação, o que implica em um esforço adicional considerável. Assim, a reutilização dos microsserviços de edição colaborativa e chat depende fortemente do contexto do projeto e das dependências relacionadas, o que limita sua flexibilidade em cenários de integração com outros sistemas.

5.2. Escalabilidade

Nesta subseção, serão analisados os resultados relacionados ao tamanho das imagens Docker, o desempenho em termos de tempo de resposta dos microsserviços, tanto com quanto sem o uso de balanceadores de carga, e as implicações desses fatores para a escalabilidade do sistema.

5.2.1. Imagem Docker

Após a migração para o *Quarkus*, mantendo a estrutura monolítica, foi possível observar uma melhoria significativa no tamanho da imagem *Docker* do sistema (Tabela 2). Antes da migração, a imagem ocupava 766 *megabytes*, enquanto, após a transição para o *Quarkus*, o tamanho foi reduzido para 206 *megabytes*, representando uma diminuição de 73,10%. Essa redução equivale a 26,89% do tamanho original da imagem *Docker*, evidenciando os benefícios da mudança para a eficiência geral do sistema. Esse resultado destaca uma melhoria na gestão de recursos, refletida na redução do peso da imagem, o que contribui para uma maior eficiência operacional.

Tabela 2. Tamanho das Imagens Docker

Imagem Docker	Tamanho (mb)
Monolito (Legado)	766
Monolito (Quarkus)	206
ce-authenticate	189
ce-editor	193
ce-chat	188

Na refatoração para microsserviços, observa-se que o tamanho da imagem de cada microsserviço é semelhante ao tamanho da imagem do monolito após sua transição para o

Quarkus. Quando somados, os tamanhos das imagens dos microsserviços totalizam 570 *megabytes*, o que representa 74,41% do tamanho original da imagem de 766 *megabytes*. Esse resultado demonstra que, embora o tamanho combinado dos microsserviços seja superior ao da imagem do monolito migrado para o *Quarkus*, ainda houve uma melhoria na utilização de memória após a divisão do sistema em microsserviços.

À primeira vista, em um cenário sem a reescrita do código para reduzir a complexidade de alguns microsserviços, pode parecer mais vantajoso optar pelo uso do monolito, visando otimizar a memória consumida pelo sistema e obter maior densidade na escalabilidade. No entanto, em um cenário de alta escalabilidade, o uso do monolito pode levar a uma perda na eficiência do uso de memória, já que os microsserviços, por serem menores, permitem o reaproveitamento de componentes isolados. Em contraste, no monolito, é necessário carregar todos os três componentes, resultando em um consumo desnecessário de memória que poderia ser evitado com a utilização dos microsserviços.

Utilizando a arquitetura de microsserviços, em um cenário com seis mil usuários virtuais enviando 1.500 requisições para autenticação, 3.000 para edição e 1.500 para o chat, é possível equilibrar a carga ao criar uma instância adicional do microsserviço de edição. Isso resultaria na distribuição de 1.500 requisições entre as instâncias, com um consumo total de memória das imagens equivalente a 763 *megabytes*. Por outro lado, ao utilizar o monolito migrado para o *Quarkus*, todas as 6.000 requisições seriam direcionadas para a mesma instância, exigindo a criação de mais três instâncias para equilibrar a carga, o que aumentaria o uso de memória para 824 *megabytes*. Com os microsserviços, ocorre uma redução de 7,40% na memória utilizada, evidenciando que, no caso do monolito, esse consumo adicional seria desnecessário, corroborando os argumentos apresentados no último parágrafo.

5.2.2. Tempo de resposta sem balanceamento de carga

Utilizando a ferramenta de teste de carga K6, foram realizados testes para analisar o tempo de resposta das requisições do monolito com *Wildfly* e dos microsserviços com *Quarkus*. No total, três testes foram conduzidos, simulando respectivamente cem, mil e dez mil usuários virtuais. O teste consistia em enviar uma requisição por usuário a cada segundo para as APIs durante um período de trinta segundos. A partir das respostas obtidas, foi calculada a média do tempo de resposta para cada número de usuários virtuais.

Tabela 3. Comparação do Tempo de Resposta Wildfly x Quarkus

Usuários Virtuais	WildFly	Quarkus
100	6.61ms	10.65ms
1000	15.76ms	23.8ms
10000	1.65s	3.03s

É notável que tanto o monolito quanto os microsserviços apresentaram bons tempos de resposta até mil usuários simultâneos, com os tempos do monolito sendo ligeiramente melhores (Tabela 3). No entanto, a partir dos dez mil usuários, o tempo de resposta tornou-se indesejável, com um período significativamente mais longo, alcançando 1,65

segundos no monolito, enquanto no microsserviço esse tempo praticamente dobrou, chegando a 3,03 segundos. Esse comportamento coloca o monolito como a melhor opção em termos de performance em um cenário onde não há escalabilidade.

5.2.3. Tempo de resposta com balanceamento de carga

Com o objetivo de testar a escalabilidade horizontal dos microsserviços, o teste de tempos de resposta descrito na seção anterior foi reutilizado. No entanto, desta vez, o teste foi adaptado para simular o comportamento das instâncias atrás de um balanceador de carga. A simulação do balanceador de carga foi realizada por meio da divisão dos usuários virtuais pelo número de instâncias. Ou seja, o balanceador simulado no teste redirecionaria as requisições de maneira uniforme, distribuindo-as igualmente entre as instâncias disponíveis.

Tabela 4. Comparação Tempo de Resposta e Número de Instâncias

Usuários Virtuais	2 Instâncias	5 Instâncias	10 Instâncias
100	5.15ms	4.91ms	4.49ms
1000	12.45ms	8.5ms	7.13ms
10000	685.16ms	36.66ms	18.46ms

Analisando os dados gerados pela simulação com duas instâncias (Tabela 4), é possível observar que os tempos de resposta foram superados em comparação aos valores previamente apresentados pelo monolito (Tabela 3). Esse resultado demonstra que, com um mínimo de escalabilidade horizontal nos microsserviços, é possível alcançar tempos de resposta desejáveis, nesse caso em específico, em cenários com até mil usuários simultâneos. No entanto, com dez mil usuários simultâneos, o tempo de resposta ainda se mantém dentro de um valor razoável.

No entanto, ao adicionar mais três instâncias (Tabela 4), foi possível observar uma melhoria significativa nos tempos de resposta, até mesmo no cenário com dez mil usuários simultâneos. Nesse caso, os tempos de resposta passaram a estar dentro de valores desejáveis, indicando que a escalabilidade horizontal foi bem-sucedida e capaz de atender a uma carga mais elevada. Esse resultado demonstra que, com uma modificação relativamente simples na infraestrutura, é possível garantir que o sistema suporte eficientemente um número maior de usuários, tornando a escalabilidade para cenários com dez mil usuários totalmente viável e com um desempenho satisfatório.

Por fim, ao analisarmos os últimos resultados apresentados na Tabela 4, com a utilização de 10 instâncias, os tempos de resposta foram excepcionalmente bons, indicando uma performance muito sólida mesmo sob uma carga consideravelmente maior. Esses resultados evidenciam que o sistema é capaz de escalar de maneira eficaz, e o número de instâncias utilizado mostrou-se adequado para suportar uma quantidade ainda maior de usuários simultâneos, mantendo os tempos de resposta dentro dos parâmetros desejáveis. Esse comportamento reforça a capacidade de escalabilidade do sistema, garantindo um desempenho robusto mesmo em cenários de alta demanda.

6. Conclusão e Trabalhos Futuros

Os objetivos deste trabalho foram a refatoração do sistema Cooperative Editor, transformando-o em uma arquitetura de microsserviços, e substituindo o *framework Wildfly* pelo *framework Quarkus* visando observar as consequências da migração, avaliando as melhorias e os impactos na manutenção e escalabilidade do sistema.

A refatoração do monolito para uma arquitetura de microsserviços, juntamente com a migração para o *framework Quarkus*, demonstrou que há melhorias significativas associadas a essa transição. Os resultados indicaram uma melhor manutenção do sistema, devido à utilização do Panache, que proporcionou uma organização mais eficiente das classes e diminuição do número de linhas, reduzindo a complexidade. Além disso, embora o sistema original apresente uma leve vantagem no tempo de resposta, a migração trouxe benefícios notáveis em termos de escalabilidade. A redução no tamanho das imagens *Docker* permitiu aumentar a densidade de instâncias com mais facilidade, possibilitando um maior número de instâncias, o que não apenas supera o tempo de resposta do sistema original, mas também resulta em economia de memória.

É importante salientar que os resultados obtidos estão condicionados às características específicas do modelo do Cooperative Editor, e que outros sistemas, com particularidades diferentes, podem apresentar resultados distintos. Para trabalhos futuros, seria recomendada a reescrita do código dos microsserviços com foco na maximização da eficiência em manutenção, uma vez que essa reescrita foi evitada no presente estudo para não comprometer os resultados dos testes. Além disso, seria interessante explorar até que ponto a utilização de um monolito migrado para o *Quarkus* pode ser mais vantajosa em comparação com uma arquitetura de microsserviços, contribuindo para uma análise mais abrangente das condições que favorecem cada abordagem.

Referências

Clingan, J. and Finnigan, K. (2020). *Quarkus Cookbook: Kubernetes-Optimized Java Solutions*. O'Reilly Media.

de Souza Clemente, A. A. and da Silva, E. d. O. (2022). Migração de sistemas monolíticos para microsserviços. *Caderno de Estudos em Sistemas de Informação*, 7(2).

Dehghani, Z. (2014). Breaking the monolith into microservices. Accessed: 2024-12-05.

Escoffier, C. and Andrianakis, G. (2020). *Quarkus for Spring Developers*. Packt Publishing.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Fowler, M. (2014). Microservices: a definition of this new architectural term. Accessed: 2023-09-22.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Kariyakarana, S. (2023). Quarkus : Turbocharging microservices with native compilation. Accessed: 2024-12-05.

Labs, G. (2024). *Grafana k6 Documentation*. Accessed: 2024-12-05.

Machado, R. P. (2019). Percepção sonora: discutindo os limites e as possibilidades de interação e de interdependência positiva de pessoas com deficiência visual em sistemas web síncronos. *LUME*.

Porter, J. (2020). *Quarkus: Supersonic Subatomic Java*. Packt Publishing.

Stradolini, C. J. (2021). Migração de sistemas monolíticos para microsserviços: estudo de caso de migração de um módulo de pagamentos de e-commerce. *LUME*.

Tostes, R. N. and Sirqueira, T. F. M. (2023). O processo de migração de um monolito para uma arquitetura de microsserviços utilizando serverless. *Caderno de Estudos em Engenharia de Software*, 4(2).