UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

LUCAS MOTA FERREIRA

DESENVOLVIMENTO DE UM MICROSSERVIÇO DE CORRETOR AUTOMÁTICO DE CÓDIGO-FONTE PARA A FERRAMENTA *OPCODERS JUDGE*

Ouro Preto, MG
2025

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

LUCAS MOTA FERREIRA

DESENVOLVIMENTO DE UM MICROSSERVIÇO DE CORRETOR AUTOMÁTICO DE CÓDIGO-FONTE PARA A FERRAMENTA *OPCODERS JUDGE*

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Pedro Henrique Lopes Silva
Coorientador: Prof. Dr. Reinaldo Silva Fortes

Ouro Preto, MG
2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Lucas Mota Ferreira

Desenvolvimento de um Microsserviço de Corretor Automático de Código-fonte para a ferramenta Opcoders Judge

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 2 de Abril de 2025.

Membros da banca

Pedro Henrique Lopes Silva (Orientador) - Doutor - Universidade Federal de Ouro Preto
Reinaldo Silva Fortes (Coorientador) - Doutor - Universidade Federal de Ouro Preto
Adriano Figueiredo de Andrade (Examinador) - Mestre - Núcleo de Tecnologia da Informação - UFOP
Fernando Euzêbio Zimerman (Examinador) - Bacharel - Programa de Pós-Graduação em Ciência da Computação - UFOP

Pedro Henrique Lopes Silva, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 2/04/2025.

Documento assinado eletronicamente por **Pedro Henrique Lopes Silva**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 02/04/2025, às 14:11, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0886283** e o código CRC **B15C36E3**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho aos meus queridos pais, Pelos sacrifícios feitos, para que eu fosse aprovado na UFOP, Pelos sonhos que compartilhamos e pelos desafios que superamos juntos. A cada esforço, a cada lágrima, vocês estiveram lá, Guiando-me com amor, paciência e apoio inabalável. Hoje, celebramos mais um etapa, em busca da jornada que trilhamos, lado a lado, em direção aos meus sonhos. Vocês são a razão pela qual estou aqui, E esta conquista é um reflexo do amor e dedicação que sempre me proporcionaram. Com gratidão eterna,
Lucas Mota Ferreira

Agradecimentos

Primeiramente, expresso minha profunda gratidão a Deus, fonte de toda sabedoria e força, por guiar meus passos e iluminar meu caminho ao longo desta significativa jornada acadêmica.

Agradeço à minha família, que esteve ao meu lado em todos os momentos, mesmo nos períodos de fraqueza, proporcionando apoio e encorajamento.

Aos meus orientadores, mentores e guias neste percurso acadêmico, dedico um sincero agradecimento. Suas orientações, paciência e *insights* foram elementos fundamentais para o meu crescimento e sucesso acadêmico. Agradeço pela sua dedicação em compartilhar seu conhecimento e experiência, contribuindo de maneira significativa para a minha formação.

Que a gratidão que expresso hoje reflita o reconhecimento profundo por todas as influências positivas que moldaram minha jornada acadêmica. Estou verdadeiramente agradecido por ter tido a oportunidade de aprender, crescer e alcançar este marco significativo, e espero continuar a honrar essas influências em meu futuro profissional e pessoal.

"O sucesso é a soma de pequenos esforços repetidos dia após dia."
Robert Collier

Resumo

O *opCoders Judge* é uma plataforma de correção automática de código utilizada em ambiente educacional, voltada para cursos de programação. Inicialmente, a arquitetura da aplicação apresentava limitações quanto à escalabilidade e ao processamento paralelo de submissões. Este trabalho teve como objetivo o desenvolvimento e a integração de um microsserviço dedicado à correção de código. Foram implementadas melhorias ao utilizar uma arquitetura distribuída com a estratégia de Produtor-Consumidor com *Apache Kafka* em *Python*, suporte a paralelismo, isolamento por contêiner *Docker* e capacidade de escalonamento horizontal. Testes de desempenho demonstraram ganhos expressivos em *throughput* com o uso de múltiplas partições no Kafka. Além disso, foram realizados testes com diferentes algoritmos de ordenação para validar a robustez do sistema sob diferentes cargas computacionais. O sistema implementado em *Python* foi capaz de suportar um total de 10.000 requisições simultâneas de correção para um algoritmo de ordenação, o que confirma a viabilidade da proposta e indica que a solução oferece maior estabilidade, desempenho e adaptabilidade para contextos com alta concorrência. A arquitetura implementada também permite futura expansão com novas funcionalidades e tecnologias.

Palavras-chave: Microsserviços. *Threads*. Kafka. Correção automática. Docker. Processamento paralelo.

Abstract

The opCoders Judge is an automatic code evaluation platform used in educational environments, specifically designed for programming courses. Initially, the system's architecture had limitations regarding scalability and parallel processing of submissions. This work aimed to develop and integrate a microservice dedicated to code evaluation. Improvements were implemented through a distributed architecture using the Producer-Consumer strategy with Apache Kafka in Python, supporting parallelism, Docker container isolation, and horizontal scalability. Performance tests demonstrated significant gains in throughput with the use of multiple partitions in Kafka. Additionally, tests were conducted using different sorting algorithms to validate the system's robustness under varying computational loads. The system implemented in Python was capable of handling up to 10,000 simultaneous code correction requests for a sorting algorithm, confirming the viability of the proposed solution and indicating that it offers greater stability, performance, and adaptability in high-concurrency scenarios. The implemented architecture also enables future expansion with new features and technologies.

Keywords: *Flask, Microservices, Threads. Kafka. Automatic code correction. Docker. Parallel processing*.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	*Application Programming Interface*
API RESTful	Interface de Programação de Aplicações baseada em REST
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
DECOM	Departamento de Computação
JSON	*JavaScript Object Notation*
KSQL	*Kafka Structured Query Language*
PBL	*Problem Based Learning*
REST	*Representational State Transfer*
SMT	*Surface Mount Technology*
UFOP	Universidade Federal de Ouro Preto
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
TI	Tecnologia da Informação
TLP	*Thread Level Parallelism*
Zab	*ZooKeeper Atomic Broadcast*

Sumário

1 Introdução

A programação desempenha um papel central no mundo moderno, sendo essencial para o desenvolvimento de softwares, aplicativos móveis, sistemas embarcados e uma ampla gama de soluções tecnológicas. Várias disciplinas presentes nos cursos da área de TI – Tecnologia da Informação (Computação, Informática e áreas afins) apresentam a programação de computadores como foco central, assunto que permanece, ao longo do curso, de forma recorrente. O aprendizado desses conceitos já nas disciplinas do primeiro semestre visa tornar o indivíduo apto a utilizar a lógica de programação como ferramenta para a resolução de diversos problemas computacionais, fator importante e necessário para disciplinas mais avançadas (SANTOS et al., 2015). Segundo Gomes, Henriques e Mendes (2008), numerosos estudos destacam os desafios enfrentados durante o processo de assimilação, sendo frequente a observação de altas taxas de dificuldade nas disciplinas introdutórias de programação. Nestas, conceitos fundamentais de algoritmos e lógica de programação são abordados, muitas vezes resultando em índices significativos de não aproveitamento.

Para superar essa barreira, a metodologia de Aprendizado Baseado em Problemas (PBL) se apresenta como uma solução eficaz. O PBL utiliza problemas da vida real como estímulos para o desenvolvimento do pensamento crítico, habilidades de resolução de problemas e aprendizagem dos conceitos do conteúdo programático (BENEDETTI, 2023). Sua aplicação é particularmente benéfica para alunos iniciantes em programação, pois ajuda a desenvolver um raciocínio diferente do convencional, tornando-se uma ferramenta distinta e eficaz no contexto educacional.

Para que a PBL funcione conforme o esperado no âmbito da programação, as soluções desenvolvidas pelos alunos devem ser analisadas o mais rápido possível, proporcionando um *feedback* rápido, seja ele positivo ou negativo. Contudo, a tarefa de verificar manualmente se um código está correto revela-se extremamente demorada e ineficiente. Isso se deve ao fato de os professores lidarem com diversas turmas e, consequentemente, muitos alunos. Assim, verificar se o código é capaz de executar todos os testes de maneira correta demanda um esforço manual considerável (MENDONÇA, 2023).

Em função disso, muitas pesquisas foram conduzidas para instrumentalizar o professor e dar suporte à sua tomada de decisão. Para exemplificar, alguns pesquisadores propuseram o uso de ambientes de correção automática de código (também conhecidos como juízes online). Além de disponibilizar as atividades criadas pelos instrutores, esses ambientes podem também possuir um ambiente de desenvolvimento integrado, onde o aluno é capaz de desenvolver e submeter as soluções dos problemas e receber um *feedback* imediato, isto é, se a solução desenvolvida para um dado exercício está correta ou errada (PEREIRA et al., 2018).

A Universidade Federal de Ouro Preto (UFOP) atualmente possui um sistema de correção

de código denominado *opCoders Judge* desenvolvido durante os trabalhos de monografia de Brito (2019), Patrocínio (2023), Cedraz (2023) e Mendonça (2023). No entanto, a implementação desse sistema segue uma abordagem monolítica, onde todos os componentes estão altamente interconectados. Diante desse cenário, surge a necessidade de escalabilidade do programa, de diminuir o acoplamento, aumentar a performance e permitir correções simultâneas e sob demanda.

Atualmente, o *opCoders Judge* realiza seu processo de correção por meio do processamento em lote acionado no servidor. De acordo com Faria et al. (2017), o processamento em lote é o método utilizado pelos computadores para concluir periodicamente trabalhos de dados repetitivos de alto volume. Esse método implica na execução de um conjunto de tarefas de maneira sequencial. No entanto, o tempo decorrido entre a entrega da tarefa, a correção realizada e a visualização dos resultados é mais longo do que o desejado, uma vez que o servidor é acionado em um determinado intervalo de tempo, resultando na inconveniência de ter que esperar um tempo considerável, atualizar a página e aguardar para visualizar o *feedback* da correção. Este atraso pode ser frustrante para os usuários, especialmente quando desejam receber um retorno imediato sobre seu trabalho.

A solução proposta neste trabalho consiste na adoção da divisão parcial do corretor em um microsserviço. Em linhas gerais, essa proposta possibilita corrigir diversas tentativas de resolução dos alunos simultaneamente, por meio da criação de um serviço especializado na correção nos moldes de um Produtor-Consumidor utilizando a tecnologia *Flask* e o *Apache Kafka*. Assim, espera-se um tempo de espera menor, devido ao início imediato da correção logo após a entrega da solução, sem a necessidade de aguardar o processamento em lote. Outra vantagem estimada é a possibilidade de acionamento da atualização automática do trecho da página assim que a correção é finalizada. Vale ressaltar que o foco deste trabalho é somente o desenvolvimento do microsserviço do contexto de correção de uma tarefa dentro do *opCoders Judge* e não a correção em si da mesma. O desenvolvimento do *back-end* foi realizado utilizando a linguagem *Python*, com suporte do *framework Flask* e do *Apache Kafka*. A comunicação assíncrona entre os serviços foi viabilizada pelo uso do sistema de mensageria *Apache Kafka*. O *front-end*, por sua vez, foi previamente estruturado em versões anteriores da plataforma *opCoders Judge*, não sendo foco de alteração neste estudo.

1.1 Justificativa

A transição da arquitetura monolítica para microsserviços no *opCoders Judge* é fundamentada em uma série de razões, todas voltadas para o aprimoramento da eficiência, escalabilidade e robustez do sistema. Primeiramente, a arquitetura monolítica pode enfrentar desafios significativos em termos de escalabilidade quando o número de usuários ou submissões aumenta substancialmente, além da dificuldade inerente em correções de *bugs* e a adição de novas funcionalidades. A abordagem de microsserviços oferece uma facilidade maior no desenvolvimento de

um sistema escalável, possibilitando que cada serviço seja dimensionado independentemente, garantindo um desempenho consistente, mesmo durante picos de submissões.

Além disso, a manutenção de uma arquitetura monolítica pode se tornar complexa à medida que o sistema cresce. A mudança para microsserviços facilita a manutenção, permitindo que cada serviço seja atualizado, corrigido ou expandido de forma independente, sem afetar globalmente o funcionamento do sistema. Essa flexibilidade agiliza a introdução de novos recursos e a aplicação de correções de maneira mais eficiente.

Quando se trata do tempo necessário para a correção de códigos, a mudança para microsserviços facilita a execução simultânea de várias correções, garantindo uma resposta mais rápida e eficaz. O microsserviço pode ser instanciado conforme a demanda de correções, garantindo autonomia e agilidade na análise do código avaliado.

1.2 Objetivos

O foco principal deste trabalho reside no desenvolvimento de uma arquitetura de microsserviços dedicada à correção automática de códigos, com integração direta à ferramenta *opCoders Judge*. Além de abordar e solucionar os desafios inerentes à transição entre as arquiteturas monolítica e de microsserviços.

Os objetivos secundários deste projeto visam alicerçar o desenvolvimento do microsserviço de correção automática de código para a ferramenta *opCoders Judge*. Eles podem ser descritos por tarefas a serem cumpridas para alcançar o objetivo principal.

- Avaliar e compreender a estrutura de processamento em lote usada atualmente no *opCoders Judge*.
- Avaliar o desenvolvimento de uma estrutura baseada em *Apache Kafka* para o microsserviços de correção.
- Realizar testes de desempenho para avaliar o desempenho do microsserviço, considerando tempos de resposta e escalabilidade em diferentes cargas de trabalho.
- Submeter o microsserviço a condições extremas com testes de estresse para avaliar sua robustez e identificar possíveis pontos de falha em situações de alta demanda.

1.3 Organização do Trabalho

O Capítulo 2 oferece uma análise detalhada de trabalhos relacionados, destacando as melhores práticas e abordagens existentes em relação a ferramentas de correção de códigos. No Capítulo 3, são apresentados os métodos, tecnologias e processos utilizados para a implementação do microsserviço, incluindo detalhes sobre a análise estática e dinâmica, avaliação de critérios

e integração com a ferramenta *opCoders Judge*. O Capítulo 4 apresenta os experimentos e os resultados obtidos com a execução do microsserviço proposto. Por fim, no Capítulo 5, as descobertas são sumarizadas e discutem-se as implicações práticas e educacionais do projeto, além da sugestão de direções futuras para aprimoramentos.

2 Revisão Bibliográfica

Este capítulo é dividido em duas seções principais: Fundamentação Teórica e Trabalhos Relacionados. Na Seção 2.1 (Fundamentação Teórica), o foco será na apresentação dos conceitos teóricos e fundamentos necessários para compreender a base metodológica do desenvolvimento do microsserviço. Isso incluirá conceitos de análise estática e dinâmica de código, critérios de correção, avaliação de desempenho e integração de sistemas. Já na Seção 2.2 (Trabalhos Relacionados), serão explorados estudos e projetos anteriores que abordam temas semelhantes, fornecendo um panorama das abordagens existentes em correção automática de códigos. A análise desses trabalhos destacará tanto os sucessos quanto as limitações, proporcionando pontos valiosos para a estruturação e implementação do microsserviço no contexto do *opCoders Judge*.

2.1 Fundamentação Teórica

A fundamentação teórica é essencial para fornecer a base conceitual necessária que sustentará as análises e argumentações apresentadas. Os tópicos que serão abordados compreendem uma exploração aprofundada em Arquitetura de Software com ênfase em arquitetura monolítica e microsserviços, uma comparação crítica entre ambas as abordagens, a aplicação dos princípios da API, API *RESTful* e uma análise sobre *Threads*.

2.1.1 Arquitetura de Software

Arquitetura de Software é um excelente modelo para administrar a complexidade de um projeto e pode atuar como um plano de negociação de requisitos de sistema, como também um método de estruturar discussões com clientes, desenvolvedores e gerentes, uma vez que abstrai os detalhes de implementação (SOMMERVILLE, 2011).

Arquitetura de Software, de acordo com Gorton (2006), descreve a estrutura em que um sistema é organizado que pode ser observada na Figura 2.1. Esta estrutura permite a atribuição de tarefas e responsabilidades distintas a cada componente de um sistema, sendo formada por duas camadas principais de uma aplicação: o *backend* e *frontend*.

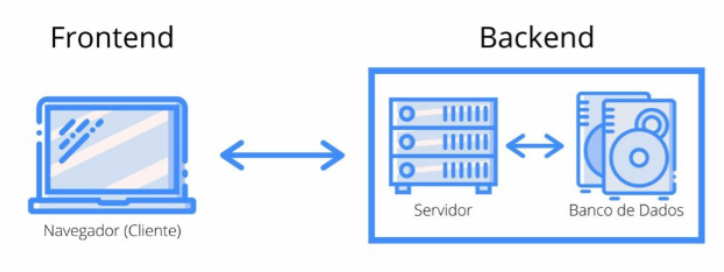
O *backend* é responsável por todo o tratamento dos dados acessados pelo usuário e as ações realizadas, tais como a interação com o banco de dados, validação de segurança das informações, entre outros (ABDULLAH; ZEKI, 2014).

A maioria dos dados e da sintaxe operacional é armazenada e acessada no *backend* de um sistema. Normalmente, o código é composto por uma ou mais linguagens de programação. O *backend* também é chamado de camada de acesso a dados de *software* ou *hardware* e inclui

qualquer funcionalidade que precise ser acessada e navegada por meios digitais (SOUZA; LIMA; CARIDADE, 2022).

No desenvolvimento de sistemas *Web* o termo *frontend* é designado para representar a camada responsável pela interface direta com o usuário. Nessa camada está a aplicação de gerenciamento (ABDULLAH; ZEKI, 2014). A camada acima do *backend* é o *frontend* e inclui todo o *software* ou *hardware* que faz parte de uma interface de usuário. Os usuários humanos ou digitais interagem diretamente com vários aspectos do *frontend* de um programa, incluindo dados inseridos pelo usuário, botões, programas, sites e outros recursos (SOUZA; LIMA; CARIDADE, 2022).

Figura 2.1 – Estrutura de uma Aplicação *Web* baseada sobre cliente/servidor.



Fonte:
<https://marquesfernandes.com/tecnologia/o-que-e-um-desenvolvedor-frontend-e-o-que-ele-faz/>.
Acessado em 14 de Janeiro de 2024.

No contexto deste trabalho, a compreensão da arquitetura de *software* em camadas, especialmente a distinção entre *frontend* e *backend*, orienta diretamente a organização da solução proposta. O *opCoders Judge* é uma aplicação que se beneficia de uma separação clara de responsabilidades: o *frontend* lida com a interface de envio de códigos pelos usuários, enquanto o *backend* é responsável por processar essas submissões. A proposta deste trabalho insere-se especificamente na camada de *backend*, ao desenvolver e integrar um microsserviço de correção automática de código já existente.

2.1.2 Tipos de arquitetura de software

No processo de desenvolvimento de uma aplicação, é essencial estabelecer uma arquitetura robusta que direcione o curso do projeto. Entre as escolhas arquiteturais mais comuns, destacam-se as abordagens monolítica e de microsserviços. A seleção da arquitetura adequada desempenha um papel fundamental, influenciando a escalabilidade, a manutenção e a flexibilidade do sistema.

2.1.2.1 Arquitetura Monolítica

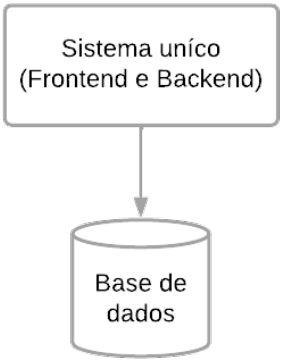
Um sistema baseado em arquitetura monolítica consiste em uma entidade unificada, composta por uma única peça coesa. Nesse modelo, os componentes do sistema são interligados e

interdependentes. Consequentemente, qualquer atualização em um dos componentes requer uma revisão e reescrita de todo o sistema, o que pode ser uma tarefa prejudicial, conforme destacado por Conceição e Pinto (2021).

Apesar das limitações impostas por arquiteturas monolíticas em termos de escalabilidade e desacoplamento, é importante reconhecer que tais modelos ainda oferecem vantagens relevantes. Além disso, eles são considerados mais acessíveis para testes e depuração, tornando o processo de desenvolvimento mais eficiente (CONCEIÇÃO; PINTO, 2021). A simplicidade e a facilidade de desenvolvimento e implantação são iminentes nas aplicações monolíticas, pois a atualização do banco de dados afeta simultaneamente todas as funcionalidades, como indicado por Conceição e Pinto (2021).

O mais comum dessa aplicação se dá pelo código principal que está contido em um único ecossistema, composto pelos elementos essenciais de *backend*, *frontend* e *database* (CLEMENTE; SILVA, 2022) como demonstra a Figura 2.2. Entretanto, é necessário considerar que, conforme observado por Machado (2017), essas aplicações têm a tendência de se tornarem mais complexas com o tempo, tornando a manutenção do código um processo desafiador e dispendioso.

Figura 2.2 – Representação Arquitetura Monolítica.



Fonte: Criado pelo autor.

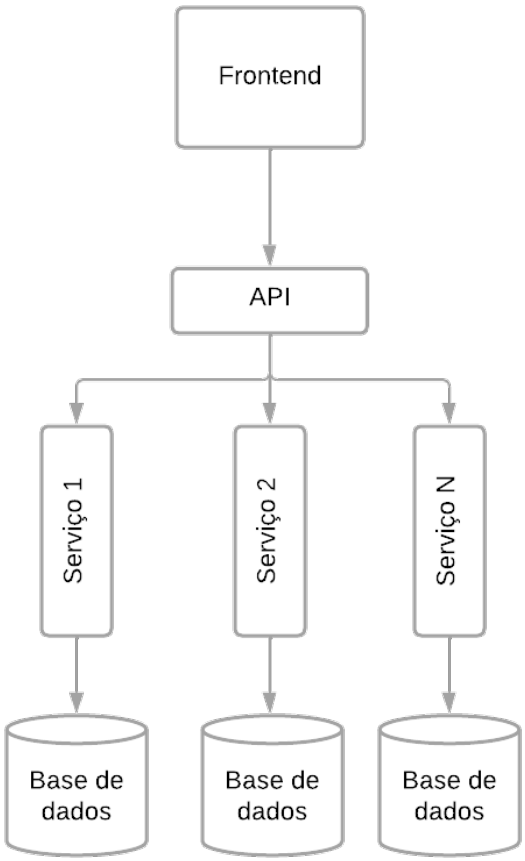
2.1.2.2 Arquitetura de Microsserviços

A arquitetura de microsserviços representa uma abordagem inovadora na concepção de sistemas *backend*. Ela se destaca pela fragmentação do *backend* em diversos serviços independentes, o que proporciona uma flexibilidade notável devido ao baixo acoplamento entre eles. Essa fragmentação permite que cada serviço seja desenvolvido, testado e escalado de maneira individual, possibilitando o aproveitamento de tecnologias específicas para cada um. Como destacado por Neto (2020) em seu estudo sobre desenvolvimento de microsserviços, essa abordagem resulta em serviços mais leves e com propósitos mais específicos, o que contribui para uma arquitetura mais eficiente e escalável.

Segundo o autor, essa abordagem arquitetônica baseia-se em princípios fundamentais, que são essenciais para o sucesso da implementação de microsserviços. Primeiramente, cada serviço deve ter um propósito único e executá-lo de maneira eficaz. Isso significa que cada serviço deve ser concebido para desempenhar uma função específica do sistema, evitando sobrecarga de funcionalidades que poderiam comprometer sua eficiência. Além disso, é crucial manter um baixo acoplamento entre os serviços. Isso significa que as interações entre os serviços devem ser mínimas e bem definidas. Essa independência entre os serviços é fundamental para a manutenção e evolução do sistema de forma ágil e eficaz. Por fim, a alta coesão é essencial para garantir a integridade e a consistência do sistema como um todo.

A arquitetura de microsserviços propõe descentralizar a responsabilidade dos dados, implicando no gerenciamento de atualizações individualmente, sem transações distribuídas atomicamente, de acordo com a Figura 2.3. Cada escopo de transação é tratado pelo microsserviço responsável dentro de seu contexto e limites transacionais adequados, devendo trabalhar com etapas de compensação das operações em mente do início ao fim do processo (GONçALVES, 2020).

Figura 2.3 – Representação Arquitetura Microsserviço.



Fonte: Criado pelo autor.

2.1.2.3 Arquitetura Monolítica vs. Arquitetura de Microsserviços

A evolução da arquitetura de *software* tem sido definida pela escolha entre duas abordagens principais: a arquitetura monolítica e a arquitetura de microsserviços. Estas duas estratégias representam paradigmas distintos para o design e desenvolvimento de sistemas de software, cada um com suas próprias vantagens e considerações únicas.

A escolha entre essas arquiteturas depende das necessidades e requisitos específicos de cada projeto, considerando aspectos como escalabilidade, flexibilidade, complexidade e custo de manutenção.

No contexto da arquitetura de software, é fundamental realizar uma comparação entre diferentes abordagens para entender suas nuances e determinar qual delas é mais adequada para um determinado projeto. A Tabela 2.1 apresenta um comparativo entre as duas arquiteturas proeminentes: a arquitetura monolítica e a arquitetura de microsserviços. Destacam-se suas principais características e diferenças para orientar os desenvolvedores na escolha da melhor opção para suas necessidades específicas.

Tabela 2.1 – Comparativo entre Arquiteturas Monolítica e de Microsserviços

Características	**Arquitetura Monolítica**	**Arquitetura de Microsserviços**
Escalabilidade	Dificuldade em escalar partes específicas.	Facilidade em escalar serviços independentemente.
Manutenção	Alterações requerem atualização de todo o sistema.	Alterações podem ser feitas em serviços individuais.
Acoplamento	Alto acoplamento entre componentes.	Baixo acoplamento, serviços independentes.
Tecnologias	Utilização de uma única tecnologia.	Cada serviço pode usar tecnologias diferentes.
Desempenho	Geralmente eficiente em termos de desempenho.	Desempenho pode variar entre serviços.
Desenvolvimento	Desenvolvimento e implantação centralizados.	Desenvolvimento e implantação descentralizados.
Exemplo de Ferramentas	*Django*, *Ruby on Rails*	*Spring Boot*, *Flask*

Fonte: Criado pelo autor.

A adoção da arquitetura de microsserviços é fundamental para superar as limitações encontradas na abordagem monolítica anteriormente utilizada no *opCoders Judge*. Ao optar por desacoplar a funcionalidade de correção automática em um microsserviço independente, tornou-se possível escalar esse componente de maneira isolada, reutilizá-lo com diferentes tecnologias e tratá-lo como uma unidade autônoma de processamento.

Durante a transição do modelo monolítico para a arquitetura de microsserviços no sistema *opCoders Judge*, foram enfrentados alguns desafios práticos. O primeiro deles foi a necessidade de reorganizar os componentes do sistema, que anteriormente estavam fortemente acoplados, exigindo a definição clara das responsabilidades de cada serviço. Além disso, foi necessário adaptar o fluxo de correção automática, que anteriormente ocorria de forma sequencial e centralizada, para um modelo distribuído e assíncrono.

2.1.3 Avaliação do microsserviço

Neste estudo, será adotada uma abordagem que envolve a aplicação de métricas específicas e metodologias adequadas para avaliar o microsserviço proposto. Segundo Bhuyan (2023), a escalabilidade e o tempo de resposta são métricas interessantes para avaliar a utilização de um microsserviço. A escalabilidade foi analisada com base na capacidade do sistema de manter seu funcionamento e desempenho à medida que o número de requisições simultâneas aumentava, permitindo verificar até que ponto o sistema suportava cenários de alta concorrência sem comprometer sua estabilidade. Já o tempo de resposta foi definido como o intervalo médio entre o envio de uma requisição (submissão de código) e o retorno do resultado da correção. Portanto, para validar o microsserviço de correção, serão conduzidos testes referentes à utilização de microsserviços, com base em estratégias específicas, tais como:

- **Teste de Integração:** A arquitetura de microsserviços exige testes de integração extensivos para garantir a comunicação sem falhas e o comportamento adequado entre os serviços ou com o sistema. O teste de ponta a ponta avalia a funcionalidade e o comportamento de todo o sistema, incluindo o microsserviço implementado, bancos de dados e dependências externas. Ele testa o fluxo completo de uma solicitação do usuário por meio de microsserviços e valida os resultados esperados. Os testes ajudam a identificar problemas relacionados à consistência dos dados, comunicação, tratamento de erros e comportamento geral do sistema.

- **Escalabilidade e Desempenho:** O teste de desempenho avalia o desempenho e a escalabilidade dos microsserviços. Envolve testar o sistema sob diferentes cargas, condições de estresse ou cenários de uso máximo. Os testes de desempenho medem os tempos de resposta, a taxa de transferência, a utilização de recursos e outras métricas de desempenho para identificar gargalos, otimizar o desempenho e garantir que os microsserviços possam lidar com as cargas esperadas sem degradação.

- **Testes de Unidade:** Concentram-se em testar microsserviços individuais de forma isolada. Verifica a funcionalidade de cada microsserviço em um nível molecular, geralmente no nível do código. Os testes de unidade garantem que componentes ou módulos individuais dos microsserviços se comportem como esperado e atendam aos requisitos definidos.

2.1.4 API

API é a sigla em inglês para *Application Programming Interface*, ou interface de programação de aplicações. Segundo TIBC (2022), as APIs são conjuntos de ferramentas, definições e protocolos para a criação de aplicações de software. Elas conectam soluções e serviços, sem a necessidade de saber como esses elementos foram implementados, somente o que espera-se de entrada e saída.

Na Figura 2.4, é demonstrada a arquitetura de uma API, onde atua como um intermediário no processo de comunicação entre o cliente e o servidor. A API é geralmente explicada em termos de cliente e servidor. O cliente pode ser uma aplicação web, móvel ou qualquer outra aplicação que envie solicitações para o servidor. O servidor é responsável por todo o processamento da aplicação e responde às solicitações enviadas pelo cliente.

Figura 2.4 – Atuação da API recebendo requisições do sistema web (frontend) e coordenando as ações com o sistema oculto (*backend*).

Fonte: <https://www.tibco.com/pt-br/reference-center/what-is-an-api-gateway>. Acessado em 14 de Janeiro de 2024.

Uma API é basicamente um conjunto de regras criadas por desenvolvedores do lado do servidor para permitir que programas se comuniquem. A forma e o como isso é feito, depende de aplicação para aplicação. Uma das formas de ser feito é por meio das APIs *RESTful* (interfaces de comunicação entre sistemas), as quais determinam como a API irá se parecer e trabalhar.

O conceito de utilização de APIs representa um elemento central na comunicação entre a solicitação e a correção da aplicação . Através de uma API estruturada, é possível expor os serviços de *backend* de maneira padronizada e acessível para outros sistemas ou interfaces, sem a necessidade de expor a lógica interna.

2.1.5 *Thread*

Uma *thread* é uma sequência de instruções que faz parte de um processo principal. Um software é organizado em processos. Cada processo é dividido em *threads*, que formam tarefas independentes, mas relacionadas entre si. Processadores podem realizar *Simultaneous Multithreading* (SMT) para ter mais desempenho (ALECRIM, 2023).

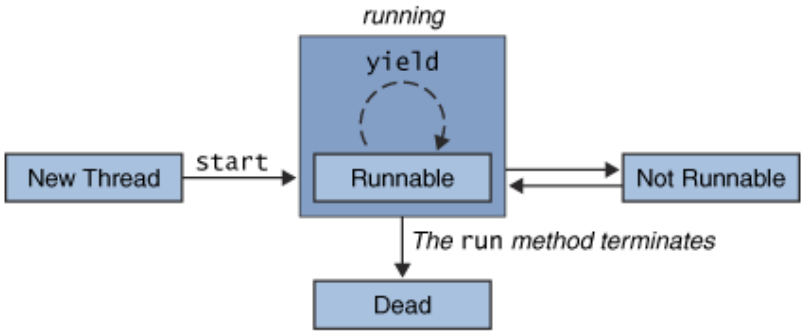
Conforme Júnior (2007), a utilização de *threads* proporciona a execução simultânea e independente de múltiplas tarefas dentro de um mesmo programa, podendo resultar em um aumento significativo de desempenho. Em situações específicas, as *threads* podem compartilhar recursos, como memória, eliminando a necessidade de duplicar dados e otimizando a utilização de recursos computacionais.

Uma das arquiteturas que implementam o paralelismo é o Paralelismo em Nível de *Thread* (TLP, do inglês *Thread Level Parallelism*), que envolve o controle de múltiplas linhas de execução (comumente chamadas de *threads*) do processador, permitindo que partes específicas do

programa sejam distribuídas entre diferentes núcleos e possam ser executadas simultaneamente (MOREIRA; CARRIEL, 2015). Segundo o autor, ao dividir uma grande tarefa em partes menores e executá-las simultaneamente, o uso das *threads* tem o potencial de acelerar a conclusão de uma determinada tarefa como um todo. Esse paralelismo oferece ganhos significativos de desempenho, aumentando a capacidade de processamento e a eficiência na manipulação de requisições em sistemas computacionais. Vale ressaltar que o uso inadequado de *threads* pode resultar em problemas como condições de corrida e concorrência indevida, o que pode comprometer a estabilidade e o funcionamento do sistema.

A Figura 2.5 apresenta uma simplificação do ciclo de vida de uma *thread*. Inicialmente, ocorre a fase de criação, onde a *thread* é instanciada e configurada. Durante a execução, ela realiza suas tarefas designadas, operando de forma independente ou em paralelo com outras *threads*. Ao atingir a obsolescência ou completar seu propósito, a *thread* é encerrada e descartada, evitando assim consumos desnecessários de recursos.

Figura 2.5 – Ciclo de vida de uma *thread*.



Fonte: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/threads/ciclo.html>. Acessado em 16 de Janeiro de 2024.

A utilização de *threads* é essencial na construção da proposta apresentada neste trabalho, principalmente no que diz respeito à natureza do problema que exige um processamento concorrente de requisições de correção de código. A aplicação de paralelismo em nível de *threads* permitiu que múltiplas tarefas fossem executadas simultaneamente dentro do microsserviço, otimizando o tempo de resposta e aumentando a capacidade de processamento.

2.1.6 *Framework* para microsserviços

Existem dois principais *frameworks* para o desenvolvimento de microsserviços: *Flask* e *Nameko* (SEQUEIRA et al., 2020). O *Flask* é um *framework* leve e flexível, voltado para o desenvolvimento de aplicações *web* em Python a partir de microsserviços. Ele oferece aos desenvolvedores controle total sobre a estrutura da aplicação, permitindo que organizem o código conforme preferirem e implementem arquiteturas específicas ou padrões de design personalizados. Por outro lado, o *Nameko* é um *framework* voltado para microsserviços, ideal para aplicações de maior porte, onde o sistema completo já deve estar preparado para essa adaptação.

Inicialmente, o microsserviço de correção foi desenvolvido utilizando o *framework Flask* devido à sua leveza e simplicidade, o que foi útil para uma primeira versão do sistema. No entanto, com o aumento da demanda por escalabilidade e resiliência, especialmente em cenários com múltiplas submissões simultâneas, o *Flask* demonstrou limitações em especial, na incapacidade de lidar com muitas requisições simultâneas e no aumento do tempo de resposta. Isso motivou a adoção de uma arquitetura mais robusta, baseada no *Apache Kafka*, que permitiu comunicação assíncrona, maior paralelismo e melhor separação entre as responsabilidades de produção e consumo de mensagens.

Dessa forma, embora o *Flask* tenha sido utilizado como ponto de partida, a versão atual do microsserviço já está desacoplada do *framework*, operando de forma assíncrona por meio da mensageria (trocas de mensagens) com Kafka, o que representa um passo importante na direção de uma arquitetura de microsserviços mais madura, escalável e tolerante a falhas.

2.1.7 *Confluent Kafka*

Quando se está dentro do ambiente de microsserviços, um dos aspectos mais comuns é a troca de informações/mensagens entre os participantes. Visto isto, estratégias de troca de mensagens distribuídas tornam-se necessárias, e o *Apache Kafka* é um dos exemplos que podem ser usados.

O *Apache Kafka* é uma plataforma distribuída de mensagens que permite o envio, armazenamento e processamento de fluxos de dados em tempo real. Ele funciona como um sistema de *log* distribuído, altamente escalável e tolerante a falhas, em que produtores, aplicações que geram dados e enviam mensagens para tópicos, e consumidores, aplicações que consomem dados e leem essas mensagens de forma assíncrona (RAD; BHATTI; AHMADI, 2017).

Confluent Kafka é uma distribuição comercial do *Apache Kafka*, desenvolvida pelos criadores originais do Kafka (NARKHEDE; SHAPIRA; PALINO, 2017). Ele oferece uma plataforma completa para o *streaming* de dados, fornecendo recursos adicionais e suporte empresarial para facilitar a construção e operação de sistemas de *streaming* de dados em larga escala (SCOTT; GAMOV; KLEIN, 2022). *Confluent Kafka* expande as funcionalidades básicas do Apache Kafka, incorporando ferramentas como o *Confluent Control Center*, que é utilizado para o monitoramento e gerenciamento dos *clusters* Kafka, permitindo que os operadores visualizem e administrem tópicos, consumidores e produtores em tempo real (RAD; BHATTI; AHMADI, 2017).

Outro componente essencial é o *Schema Registry*, que gerencia *schemas* de dados Avro (controle e validação da estrutura dos dados transmitidos). Um *schema* de dados pode ser entendido como uma estrutura que define o formato, os tipos e as regras de validação dos dados trocados entre aplicações e assegura a compatibilidade entre produtores e consumidores (SCOTT; GAMOV; KLEIN, 2022). Este recurso é vital para manter a integridade dos dados e permitir a evolução dos *schemas* sem interrupção dos serviços. Além disso, o *Kafka Connect* é um *framework* que

facilita a integração do *Kafka* com outras fontes e *sinks* de dados, um sistema ou serviço que consome os dados provenientes do Kafka de maneira eficiente e escalável, suportando uma vasta gama de conectores pré-construídos para diferentes sistemas de dados (NARKHEDE; SHAPIRA; PALINO, 2017).

O *KSQL*, sigla para *Kafka Structured Query Language*, é outro destaque da plataforma *Confluent*. Ele permite a realização de consultas e transformações de dados em tempo real utilizando uma linguagem semelhante ao *SQL* (linguagem de consulta utilizada para gerenciar e manipular bancos de dados relacionais) tradicional, mas adaptada para lidar com fluxos contínuos de dados no *Apache Kafka*. Com o *KSQL*, é possível filtrar, agregar, juntar e transformar dados em movimento, sem a necessidade de escrever código Java ou criar aplicações complexas.(RAD; BHATTI; AHMADI, 2017).

Confluent Kafka também oferece suporte a múltiplas linguagens de programação, incluindo Java, Python, *Go* e *C/C++*, tornando-o acessível para uma ampla gama de desenvolvedores (SCOTT; GAMOV; KLEIN, 2022). A plataforma é projetada para ser altamente escalável, suportando a adição de novos *brokers* ao *cluster* sem interrupções significativas (RAD; BHATTI; AHMADI, 2017). Com ferramentas de segurança avançadas, como autenticação baseada em *Kerberos*, *SSL* e controle de acesso granular, *Confluent Kafka* garante que os dados em trânsito estejam protegidos contra acessos não autorizados (NARKHEDE; SHAPIRA; PALINO, 2017). Além disso, a *Confluent Platform* inclui suporte à replicação de dados multi-região, permitindo que as empresas implementem soluções de *disaster recovery* (restaurar os sistemas e dados após falhas críticas) e alta disponibilidade de forma eficiente (SCOTT; GAMOV; KLEIN, 2022).

A arquitetura de *Confluent Kafka* é baseada em *logs* distribuídos, onde os produtores escrevem dados em tópicos e os consumidores leem esses dados de forma assíncrona (RAD; BHATTI; AHMADI, 2017). Cada mensagem no *Kafka* é composta de uma chave, um valor e um *timestamp* (hora e data em que um evento ocorreu), permitindo o processamento e a análise de dados em tempo real (NARKHEDE; SHAPIRA; PALINO, 2017). A plataforma suporta também a compactação de *logs*, o que permite economizar espaço de armazenamento, mantendo apenas as mensagens mais recentes associadas a cada chave (SCOTT; GAMOV; KLEIN, 2022).

A *Confluent Platform* é amplamente utilizada em diversas indústrias, incluindo *fintechs*, *e-commerce*, telecomunicações e saúde, para casos de uso como monitoramento de transações financeiras, personalização de conteúdo e monitoramento de redes (RAD; BHATTI; AHMADI, 2017).

Utilizando os recursos fornecidos pela *Confluent*, como o *Apache Kafka*, o sistema proposto neste trabalho foi estruturado no modelo Produtor-Consumidor, em que a submissão do código por parte do aluno atua como produtor de uma mensagem, e o serviço de correção (o consumidor) processa essa mensagem de forma desacoplada e escalável. Esse modelo permite reduzir o tempo de resposta e melhorar a resiliência do sistema, visto que as mensagens ficam armazenadas até serem processadas, mesmo em situações de alta carga ou falhas temporárias.

Para realizar a orquestração/coordenação de todo o sistema distribuído, o *Confluent Kafka* utiliza o *Apache Zookeeper*, o qual é explicado adiante.

2.1.8 *Apache ZooKeeper*

O *Apache ZooKeeper* é um serviço de coordenação centralizado dos componentes utilizado por sistemas distribuídos para gerenciar configurações, fornecer serviços de nomeação, sincronização e grupos de serviços (HUNT et al., 2010). Originalmente, o *Apache Kafka* era dependente do *ZooKeeper* para gerenciar o estado do *cluster Kafka*. O *ZooKeeper* facilita a coordenação entre os diferentes componentes de um sistema distribuído, mantendo informações sobre a configuração do sistema e ajudando a gerenciar a liderança e o estado dos *brokers* Kafka (JUNQUEIRA; REED, 2013).

No contexto do Kafka, um *broker* é um servidor responsável por armazenar e distribuir mensagens recebidas dos produtores para os consumidores. Em um *cluster Kafka*, múltiplos *brokers* trabalham juntos para garantir disponibilidade, tolerância a falhas e escalabilidade, permitindo que os dados sejam distribuídos eficientemente.

O *ZooKeeper* é projetado para ser altamente disponível e tolerante a falhas, utilizando um protocolo de consenso chamado *Zab* (*ZooKeeper Atomic Broadcast*) para garantir que todas as mudanças no estado do sistema sejam replicadas de forma consistente entre todos os nós do *cluster* (HALOI, 2015). Esse protocolo permite que o *ZooKeeper* ofereça garantias fortes de consistência e durabilidade, essenciais para a coordenação de serviços críticos em sistemas distribuídos (HUNT et al., 2010). A API de *ZooKeeper* é baseada em um modelo de dados hierárquico, onde os dados são armazenados em nós chamados *znodes*, que podem ser persistentes ou efêmeros, permitindo a construção de complexas estruturas de dados distribuídas (JUNQUEIRA; REED, 2013).

Apesar de seu papel essencial, o *Apache Kafka* está em processo de remover a dependência do *ZooKeeper* através do projeto *KIP-500*. Esta mudança visa simplificar a arquitetura do *Kafka*, melhorando a escalabilidade e a confiabilidade (HALOI, 2015). No entanto, até que essa transição esteja completa, o *ZooKeeper* continua sendo uma parte crucial do ecossistema *Kafka*, garantindo a coordenação e a consistência entre os vários nós do *cluster* (HUNT et al., 2010).

Além do *Kafka*, *ZooKeeper* é amplamente utilizado por outros sistemas distribuídos, como *Hadoop*, *HBase* e *Solr*, para gerenciar a configuração e a coordenação entre nós (JUNQUEIRA; REED, 2013). Ele oferece primitivas de sincronização, como *locks* e barreiras, que simplificam a implementação de algoritmos distribuídos complexos (HALOI, 2015). O *ZooKeeper* também suporta notificações de eventos, permitindo que clientes sejam informados sobre mudanças no estado do sistema em tempo real, o que é fundamental para a construção de sistemas reativos e altamente responsivos (HUNT et al., 2010).

O *ZooKeeper* é utilizado como parte da infraestrutura necessária para o funcionamento

do *Apache Kafka*, mais especificamente durante a configuração do ambiente de testes com *Docker Compose*.

2.1.9 *Docker*

Docker é uma plataforma de conteinerização que permite a criação, implementação e execução de aplicativos em contêineres (RAD; BHATTI; AHMADI, 2017). Os contêineres são unidades padronizadas de *software* que empacotam o código e todas as suas dependências para que a aplicação possa ser executada de maneira rápida e confiável em diferentes ambientes de computação (MERKEL et al., 2014). A adoção do *Docker* tem sido impulsionada por sua capacidade de melhorar a eficiência no desenvolvimento e a escalabilidade de aplicações, permitindo o isolamento e a independência de serviços (KRATZKE, 2017).

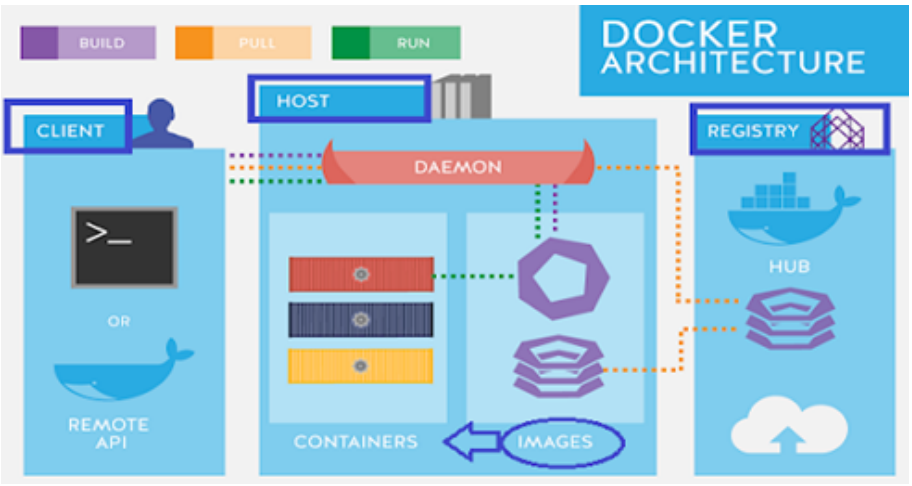
Na implementação de microsserviços, *Docker* possibilita que cada serviço seja isolado em seu próprio contêiner, garantindo modularidade e escalabilidade (KRATZKE, 2017). Grandes empresas utilizam essa abordagem para construir aplicações mais resilientes e distribuídas (RAD; BHATTI; AHMADI, 2017). A utilização de *Docker* para Kafka é um exemplo prático desse benefício. Com *Docker Compose*, é possível configurar e gerenciar um *cluster* (conjunto de servidores) *Kafka* completo, incluindo *brokers*(instâncias do servidor) *Kafka*, *ZooKeeper*, *Schema Registry* e *Kafka Connect*, reduzindo significativamente o tempo de provisionamento e garantindo a consistência entre ambientes de desenvolvimento e produção (MERKEL et al., 2014).

A arquitetura do *Docker* segue o modelo cliente-servidor. O *Docker Daemon* (ou servidor *Docker*) é o processo em segundo plano responsável por gerenciar os contêineres, imagens, volumes e redes. Já o *Docker CLI* (*Command Line Interface*) é a ferramenta que permite aos desenvolvedores interagir com o *Daemon* por meio de comandos. Outro componente essencial é o *Docker Hub*, um repositório online para armazenar e compartilhar imagens, o que facilita o reuso de aplicações pré-configuradas (UPADHYA et al., 2017). Além disso, o *Docker Hub* atua como um repositório público de imagens, permitindo o compartilhamento e a reutilização de componentes, o que acelera o desenvolvimento de aplicações (MERKEL et al., 2014). Essa abordagem elimina a necessidade de reconstrução de ambientes para cada novo desenvolvimento, facilitando a colaboração entre equipes e garantindo que os mesmos contêineres sejam executados de maneira idêntica em diferentes infraestruturas (RAD; BHATTI; AHMADI, 2017).

A Figura 2.6 ilustra o funcionamento do *Docker* com base em uma arquitetura cliente-servidor. O *Docker Client* atua como ponto de entrada para as instruções, podendo ser acessado via terminal ou por meio de APIs remotas. Essas instruções são encaminhadas ao *Docker Daemon*, um serviço que opera em segundo plano e é responsável por gerenciar contêineres, imagens, redes e volumes.

O *Docker Host* representa o ambiente no qual o *Daemon* está em execução, normalmente uma máquina física ou virtual. As instruções processadas pelo cliente resultam na criação ou

Figura 2.6 – Arquitetura do *Docker*: componentes principais e fluxo de funcionamento.



Fonte: <https://www.darede.com.br/arquitetura-docker/>. Acessado em 10 de março de 2025.

execução de contêineres, que são instâncias de imagens *Docker*. Essas imagens, por sua vez, podem ser obtidas de repositórios externos, como o Docker Hub, representado na Figura 2.6 como parte do componente *Registry*.

A eficiência do *Docker* é ampliada pelo uso de imagens em camadas, onde cada camada representa uma alteração incremental no sistema de arquivos (KRATZKE, 2017). Isso permite a reutilização de camadas, reduzindo significativamente o espaço de armazenamento e acelerando o processo de *build* e *deployment* de aplicações (MERKEL et al., 2014). Essa técnica é essencial para garantir que novos desenvolvimentos e atualizações sejam implantados rapidamente sem a necessidade de reconstrução total da aplicação (RAD; BHATTI; AHMADI, 2017). Além disso, *Docker* suporta redes de contêineres isoladas, permitindo comunicação segura entre serviços, o que é crucial para a construção de aplicações distribuídas, escaláveis e resilientes (UPADHYA et al., 2017).

Entre as vantagens do *Docker*, destacam-se:

- **Portabilidade:** Contêineres podem ser executados em diferentes ambientes sem modificações (RAD; BHATTI; AHMADI, 2017).
- **Eficiência de Recursos:** Menos consumo de memória e *CPU* comparado a máquinas virtuais (KRATZKE, 2017).
- **Rapidez na Implantação:** Inicialização quase instantânea de contêineres (MERKEL et al., 2014).
- **Facilidade de Escalabilidade:** Possibilidade de escalar aplicações rapidamente com orquestradores como *Kubernetes* (UPADHYA et al., 2017).

No entanto, algumas desvantagens também devem ser consideradas:

- **Segurança:** Compartilhamento do *kernel* do *host* pode levar a riscos de segurança (UPADHYA et al., 2017).
- **Gerenciamento de Persistência:** Manter estados persistentes pode ser desafiador (RAD; BHATTI; AHMADI, 2017).
- **Compatibilidade com Sistemas *Windows*:** *Docker* é otimizado para *Linux*, o que pode gerar dificuldades ao executar aplicações em ambiente *Windows* sem uma camada de virtualização (MERKEL et al., 2014).

A integração do *Docker* com ferramentas de orquestração, como *Kubernetes*, tem sido amplamente adotada por empresas que buscam alta disponibilidade e eficiência operacional em ambientes de nuvem e *on-premise* (KRATZKE, 2017). Além disso, a segurança dos contêineres é uma preocupação constante, e práticas como a execução de contêineres sem privilégios, a limitação de recursos via *cgroups* e o uso de perfis de segurança como *AppArmor* e *SELinux* são essenciais para mitigar riscos (UPADHYA et al., 2017).

Com base nessas características, *Docker* se estabeleceu como um pilar fundamental no desenvolvimento moderno de *software*, oferecendo soluções eficientes para a criação, distribuição e escalabilidade de aplicações (RAD; BHATTI; AHMADI, 2017). Seu uso contínuo e aprimoramentos constantes indicam que essa tecnologia continuará sendo uma peça central no ecossistema de infraestrutura e desenvolvimento de *software* (KRATZKE, 2017).

O uso do *Docker* foi essencial para viabilizar a modularização, portabilidade e escalabilidade dos componentes do sistema de correção automática. A conteinerização permitiu que o microsserviço de correção fosse executado de forma isolada.

2.1.10 Funcionamento do Modelo Produtor-Consumidor

A crescente demanda por aplicações altamente escaláveis, resilientes e orientadas a eventos levou à ampla adoção do modelo Produtor-Consumidor, uma arquitetura que desacopla a geração e o processamento de mensagens. Esse modelo é particularmente eficiente para sistemas que precisam lidar com grandes volumes de dados, possibilitando o processamento assíncrono, a distribuição equilibrada de carga e a tolerância a falhas (NARKHEDE; SHAPIRA; PALINO, 2017; SCOTT; GAMOV; KLEIN, 2022).

Nesse contexto, uma das tecnologias mais utilizadas é o Apache Kafka, uma plataforma de *event streaming* projetada para lidar com fluxos de dados em tempo real de forma distribuída, escalável e durável. No modelo do *Kafka*, produtores publicam mensagens em tópicos, que são internamente divididos em partições. Essas mensagens são armazenadas temporariamente nos *brokers* do *Kafka* e consumidas posteriormente por consumidores (RAD; BHATTI; AHMADI, 2017; HUNT et al., 2010).

Entre os principais benefícios dessa arquitetura, destacam-se:

- **Escalabilidade horizontal:** novos consumidores podem ser adicionados sem alterar a lógica dos produtores.
- **Tolerância a falhas:** as mensagens permanecem disponíveis até serem processadas, mesmo em caso de falhas temporárias.
- **Distribuição de carga:** o *Kafka* distribui mensagens entre as partições, otimizando o uso dos consumidores.
- **Garantia de ordem:** dentro de cada partição, a ordem das mensagens é preservada.
- **Replicação:** aumenta a resiliência ao replicar dados entre nós do *cluster*.

Além disso, o *Kafka* implementa a noção de grupos de consumidores. Dentro de um grupo, cada partição de um tópico é atribuída a apenas um consumidor, garantindo que cada mensagem seja processada uma única vez. Já entre grupos diferentes, múltiplos consumidores podem ler o mesmo tópico simultaneamente, o que viabiliza diferentes fluxos de processamento com a mesma base de dados (NARKHEDE; SHAPIRA; PALINO, 2017). De forma resumida, tem-se:

- Cada partição é consumida por apenas um consumidor dentro do mesmo grupo.
- Diferentes grupos de consumidores podem processar o mesmo tópico de forma independente.

Esse tipo de estrutura de Produtor-Consumidor dentro do contexto *Kafka* fortalece a robustez dos sistemas distribuídos, permitindo paralelismo com controle, balanceamento dinâmico de carga e maior confiabilidade no processamento dos dados. A arquitetura Produtor-Consumidor foi aplicada com o objetivo de resolver os gargalos identificados na abordagem inicial baseada apenas em *Flask*. O Apache Kafka foi adotado como solução de mensageria por oferecer suporte nativo a paralelismo, escalabilidade horizontal e tolerância a falhas, características essenciais para o processamento de correções de código de forma assíncrona e distribuída.

2.2 Trabalhos Relacionados

Ao longo da evolução da pesquisa em microsserviços, diversos estudos e trabalhos têm contribuído para o entendimento e avanço nesse campo. A revisão de trabalhos relacionados é fundamental para situar o atual estudo no contexto mais amplo da pesquisa acadêmica e identificar lacunas ou áreas que requerem maior pesquisa.

O estudo desenvolvido por Selivon, Bezerra e Tonin (2015), apresenta uma versão aprimorada da ferramenta *Academic*, integrada ao corretor *URI Online Judge*. O objetivo inicial da criação deste portal foi desenvolver funcionalidades que oferecessem uma alternativa ao método

tradicional de ensino de Algoritmos e Programação. No decorrer do estudo, é apresentada uma metodologia que destaca o uso da ferramenta como suporte nas aulas de Algoritmos e Estruturas de Dados. Isso é alcançado por meio da abordagem de problemas que envolvem a prática de conceitos específicos essenciais nessas disciplinas, contribuindo para uma compreensão mais aprofundada por parte dos estudantes. O portal *URI* em questão contém uma variedade de problemas no estilo do *ICPC* (*International Collegiate Programming Contest*) da ACM. Além disso, oferece aos usuários um juiz online que permite testar suas soluções para esses problemas.

O trabalho desenvolvido por Brito (2019) propõe uma API que permite a correção automática de código, abordando duas vertentes. A primeira é a correção dinâmica, que realiza múltiplos testes e compara os resultados obtidos com um valor pré-determinado, analisando a similaridade da solução com o resultado esperado. A segunda abordagem é a correção estática, que verifica a estrutura e a sintaxe do código, incluindo aspectos como laços de repetição e estruturas de dados, entre outros. Após os testes que utilizaram essa ferramenta, os alunos demonstraram melhorias significativas em suas avaliações teóricas, demonstrando que a ferramenta auxilia no aprendizado.

Enquanto o trabalho de Brito (2019) foca nas funcionalidades essenciais de um corretor de código, a proposta do trabalho apresentado por Patrocínio (2023) tem como objetivo principal fornecer uma interface web para o corretor automático denominado *opCoders Judge*. Além disso, uma série de refinamentos e funcionalidades são adicionadas, tornando a ferramenta mais abrangente e completa para todos os usuários. Essas melhorias incluem recursos como o acompanhamento do status das entregas feitas pelos alunos e um sistema de gerenciamento para os professores. Além disso, a implementação de novas tecnologias, como *PHP*, *BootStrap* e *MySQL*, permitiu a correção de diversas linguagens de programação, além de cumprir os objetivos propostos em relação à usabilidade da ferramenta que foram alcançados com sucesso. Os autores também apresentaram possíveis trabalhos futuros, como a geração de listas de exercícios baseadas em tema e nível de dificuldade, bem como a autenticação integrada com o portal da Universidade Federal de Ouro Preto.

No trabalho conduzido por Beltrame (2018), foi desenvolvida uma ferramenta chamada *Online Judge*, que consiste em um sistema de correção de código-fonte baseado em um ciclo de desenvolvimento. Um exemplo amplamente conhecido desse tipo de ferramenta é o *Uniform Resource Identifier Online Judge* (URI). O ciclo de avaliação da ferramenta envolve as etapas de *upload*, compilação, execução, comparação e, por fim, *feedback* ao usuário. Esse ciclo pode ser repetido até que o aluno esteja satisfeito com o resultado obtido. Os resultados do estudo indicaram que a ferramenta contribuiu significativamente para o monitoramento do progresso dos alunos nas disciplinas, permitindo que os professores adotassem estratégias para aprimorar o processo de aprendizado dos estudantes.

Para verificar se o desenvolvimento de corretor é útil na aprendizagem dos alunos, Berssanette (2018) apresentou um estudo que destaca a maneira como um corretor automático de

código pode ser benéfico para o desenvolvimento do raciocínio lógico dos alunos. O autor propôs um modelo de aprendizado que combina uma estratégia educacional conhecida como *Problem Based Learning* (PBL) com o uso de um corretor de códigos amplamente reconhecido, o URI. Além disso, o *feedback* em tempo real fornecido pelo corretor automático foi avaliado de forma positiva pelos alunos, pois lhes permitiu obter uma avaliação imediata de seu desempenho. Essa abordagem que combina PBL com um corretor automático de código demonstrou ser uma estratégia eficaz para promover o desenvolvimento do raciocínio lógico dos alunos no desenvolvimento de algoritmos.

Por fim, para ambientar-se com a ferramenta *Flask*, o trabalho conduzido por Oliveira e Pilan (2018), o objetivo principal do trabalho é demonstrar uma abordagem para o desenvolvimento de uma aplicação *Desktop* capaz de acessar um *website*. Essa tarefa é realizada utilizando *Python*, *Flask*, *PyQt5* e um banco de dados MariaDB. O *Python* e seus módulos são empregados para abrir um *Web Server*, gerenciar uma janela e controlar o banco de dados.

Ademais, adoção do *Apache Kafka* em arquiteturas de microsserviços orientadas a eventos tem-se mostrado eficaz para garantir desempenho e confiabilidade na entrega de mensagens, especialmente em sistemas distribuídos com alta demanda de processamento assíncrono. Os trabalhos de Narkhede, Shapira e Palino (2017) e Scott, Gamov e Klein (2022) já evidenciam esses benefícios, mas estudos mais recentes aprofundam a discussão sobre sua resiliência.

Luu, Janjusic e Yilmaz (2019), no artigo *TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka*, apresenta uma ferramenta que permite avaliar a confiabilidade da entrega de dados no *Kafka* sob diferentes condições de rede. Os autores utilizam contêineres *Docker* e ferramentas de simulação de rede para analisar perdas e duplicações de mensagens com base nas semânticas *at-most-once* e *at-least-once*. Os resultados indicam que a semântica *at-least-once*, embora mais confiável em ambientes com alta perda de pacotes, pode ocasionar duplicações, exigindo mecanismos de controle na aplicação consumidora. Essa análise dialoga diretamente com o contexto deste trabalho, onde o *Kafka* é utilizado como *middleware* para desacoplar a submissão e a correção de códigos, permitindo maior tolerância a falhas e persistência das mensagens até seu processamento.

Complementarmente, Magalhães (2023) propõe uma biblioteca de testes determinísticos para sistemas que utilizam Kafka em comunicação assíncrona. O autor destaca a importância da ordenação de mensagens e da reprodutibilidade dos testes em sistemas distribuídos. Esses conceitos influenciaram diretamente a estratégia experimental adotada neste trabalho, como a simulação de testes com algoritmos de ordenação (*Bubble*, *Selection*, *Insertion*, *Merge*), o uso de tópicos com múltiplas partições e a configuração de grupos de consumidores para garantir paralelismo e escalabilidade.

O trabalho de Fernandes et al. (2024) analisa a correlação entre a complexidade do código e a dificuldade percebida pelos alunos em questões de programação, utilizando dados extraídos do ambiente *CodeBench*. Os autores observaram que, apesar das métricas de complexidade

oferecerem indícios estruturais do código, essas não são suficientes, isoladamente, para prever a dificuldade enfrentada pelos estudantes. Essa limitação evidencia a necessidade de ferramentas de correção mais inteligentes e adaptáveis, constatação que reforça a importância de soluções que visam uma arquitetura distribuída e paralela para correção automática, com escalabilidade e resiliência.

A partir dos trabalhos mencionados acima, fica claro que foram realizados numerosos testes envolvendo a ferramenta de correção de códigos, os quais obtiveram resultados positivos, confirmando a utilidade da ferramenta. Os sistemas desenvolvidos até o momento se baseiam em um banco de dados e uma interface. No entanto, o modelo proposto tem como objetivo introduzir mais tecnologias no sistema e conceitos de engenharia de software.

3 Desenvolvimento do microsserviço

No presente capítulo, será realizada uma abordagem detalhada sobre como foi feita a migração da arquitetura monolítica para o uso do paralelismo de ações simultâneas, com o intuito de aprofundar a sua compreensão e análise em relação ao tema abordado.

As seções a seguir fornecerão uma visão detalhada dos principais aspectos e mudanças dessa implementação. A Seção 3.1 apresenta o funcionamento geral do *opCoders Judge*, detalhando o processo de elaboração e disponibilização de questões para os alunos, além do fluxo de correção das submissões. Posteriormente, a Seção 3.2 descreve os passos realizados para transformar o sistema de correção do *opCoders Judge* em um microsserviço, incluindo mudanças na estrutura, implementação de rotas e gerenciamento de requisições assíncronas. Por fim, aborda a avaliação do microsserviço proposto na Seção 2.1.3.

3.1 O formato de correções do Corretor Online *opCoders Judge*

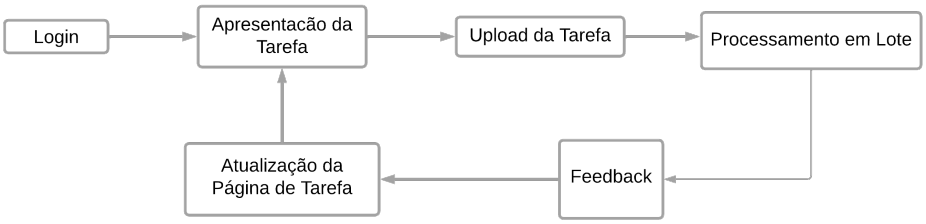
A correção automática de código no sistema*OpCoders Judge* foi implementada utilizando a linguagem *Python*, e é justificada pela sua compatibilidade com os códigos-fonte submetidos, uma vez que a ferramenta é predominantemente compatível com essa linguagem de programação. Além disso, a utilização de funções auxiliares em *Python* tem como objetivo eliminar conflitos de linguagem durante o processo de correção de códigos.

Foi desenvolvido um método para que os professores possam elaborar modelos de questões. A partir desses modelos, o sistema cria diferentes versões de questões, as questões-modelo são caracterizadas por metadados que auxiliam na classificação, como a área de conhecimento, o nível de dificuldade, os tópicos relacionados ao ensino de programação e as *tags* que permitem ao professor estabelecer características específicas para as questões. Após a definição das questões, elas são disponibilizadas para os alunos.

Na Figura 3.1, após o processo de login, os alunos realizam o *upload* de suas resoluções, que são submetidas a um processo em lote. O processamento em lote é uma técnica eficaz para lidar com grandes volumes de dados de uma só vez. Em vez de processar cada entrada individualmente, o sistema coleta um conjunto de entradas e as processa em um único lote. Isso reduz a sobrecarga de processamento, pois o sistema não precisa iniciar uma nova instância de processamento para cada entrada. Em vez disso, ele pode processar várias entradas de uma só vez, economizando tempo e recursos.

Após a correção do código-fonte submetido, o resultado do processamento é avaliado

Figura 3.1 – Fluxo de funcionamento do *opCoders Judge*.



Fonte: Criado pelo autor.

com base na similaridade entre a saída do código e a saída esperada. Essa comparação permite determinar a nota para o código fonte, refletindo a precisão e eficácia do algoritmo implementado.

No contexto do sistema, o processamento em lote é usado para comparar as respostas dos alunos com as respostas esperadas. Isso permite que o sistema avalie várias respostas de uma só vez, em vez de ter que avaliar cada resposta individualmente. Isso é especialmente útil quando o sistema lida com um grande número de respostas, como em um ambiente educacional onde muitos alunos podem estar submetendo suas resoluções ao mesmo tempo.

O processamento apresenta algumas desvantagens e limitações, tais como tempo de resposta mais lento em comparação com o processamento em tempo real, o que pode ser um problema em aplicações que requerem respostas imediatas, latência e escalabilidade. Para superar esses desafios, a adoção da estrutura de microsserviços é fundamental.

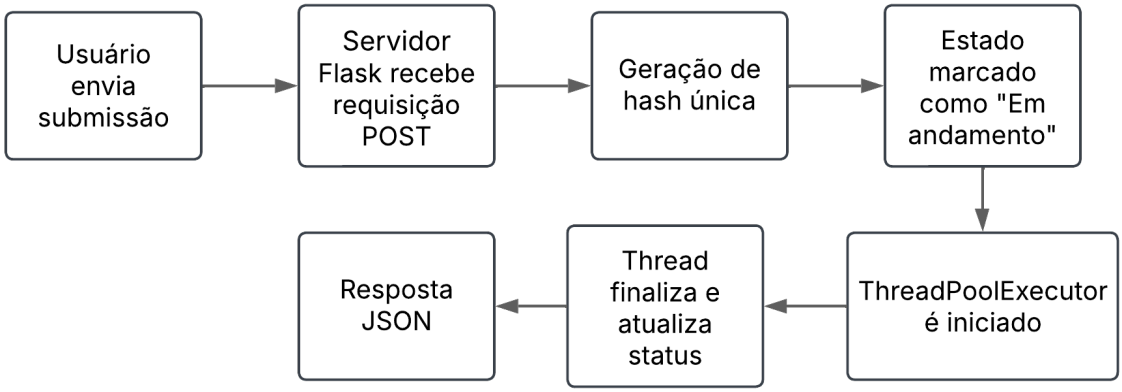
3.2 A correção do *opCoders Judge* como uma API *RESTful*

A migração da arquitetura monolítica para a arquitetura de microsserviços é uma tendência crescente no desenvolvimento de *software* Dragoni et al. (2017). Essa mudança é motivada por uma série de fatores citados anteriormente. Na Figura 3.2, é possível observar o fluxo de execução do microsserviço proposto.

Após o *login* do usuário, existem diversas ações que um usuário pode realizar, como checagem das tarefas concluídas e em aberto, visualização da tarefa aberta e submissão do código-fonte referente a uma tarefa específica. Este último é o objeto de interesse deste trabalho; os demais serão desconsiderados.

Após a submissão do código-fonte, o sistema chama os microsserviços, que processam o código-fonte enviado e realiza-se uma chamada à API do corretor, que é resolvida por uma instância de *thread* para a execução do algoritmo. Posteriormente, o sistema retorna o processamento, fornecendo um *feedback* positivo ou negativo daquela execução. Se o *feedback* for negativo, o aluno pode revisar e submeter o código novamente, caso contrário, se o *feedback* for positivo, o ciclo é concluído. Em comparação com a Figura 3.1, nota-se que a mudança ocorre no processamento, visando um sistema com uma resposta mais eficiente do que o processamento

Figura 3.2 – Fluxo do *opCoders Judge* com Microsserviços.



Fonte: Criado pelo autor.

em lote.

A primeira medida adotada consistiu na redefinição da estrutura de correção, limitando seu uso dos componentes apenas às situações estritamente necessárias, como a leitura dos arquivos e o uso excessivo de variáveis globais. As funções que foram alteradas foram o processamento do arquivo, da correção e a de retornar o resultado, as quais foram submetidas a testes individuais, obtendo o mesmo resultado sem a dependência global e buscando identificar possíveis inconsistências.

Após essa fase de ajustes, cria-se uma rota (*endpoint*) para o microsserviço. Em outras palavras, estabeleceu-se uma chamada para o processo de correção, sendo essa chamada realizada por meio de uma requisição do tipo *POST*. Os atributos da requisição foram:

- *questaoPath*: Caminho da questão no diretório.
- *entregaPath*: Caminho da entrega no diretório.
- *tarefa*: Nome da tarefa.
- *aluno*: Nome do aluno.
- *questao*: Nome da questão.

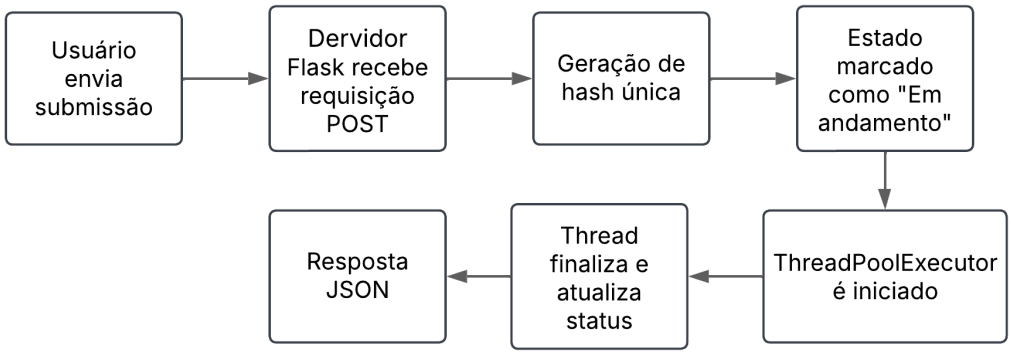
A concepção inicial do *opCoders Judge* considera que todos os arquivos já estão presentes no banco de dados. Dessa forma, torna-se necessária somente a inclusão do ID da tarefa, do aluno e o código resultante.

No funcionamento inicial do sistema, há o *endpoint submeter* responsável por receber todas as requisições do servidor. Cada requisição aciona a instância de uma *thread*, encarregada de realizar a correção.

Para manter o controle sobre a checagem da execução correspondente de cada correção, foi implementada uma rota adicional com um *endpoint checar* responsável por receber o resultado final da execução, indicando se houve algum erro ou se a execução foi bem-sucedida.

A Figura 3.3 demonstra o funcionamento do fluxo de correção baseado em *Flask* com uso de um controlador de *threads* (*ThreadPool*). O processo se inicia com o envio de uma submissão por parte do usuário, que realiza uma requisição *POST* (método HTTP usado para enviar dados do cliente para o servidor) para o servidor *Flask*. Ao receber essa requisição, o servidor gera uma *hash* (código gerado a partir de dados) única, responsável por identificar de forma exclusiva cada submissão, permitindo sua rastreabilidade ao longo de todo o processo.

Figura 3.3 – Funcionamento do microsserviço proposto utilizando o *Flask*.



Fonte: Criado pelo autor.

Após a geração da *hash*, o estado da requisição é registrado como “Em andamento”. Em seguida, inicia-se uma *thread* pelo (*ThreadPool*), que executa a correção de forma paralela em uma *thread* dedicada. Quando a correção é concluída, a *thread* responsável atualiza o estado da requisição para “Concluído”, sinalizando o fim do processamento. Vale ressaltar que este processo torna todo o processo síncrono e lento da correção em si.

Por fim, o servidor retorna uma resposta *JSON* contendo o resultado da correção, bem como a *hash* gerada, que pode ser utilizada para consultar o estado da submissão posteriormente por meio de uma rota específica de status.

No Algoritmo 3.1, é possível observar a chamada do microsserviço em *Flask*. Primeiramente, é criada uma rota para o microsserviço e definido o método *POST*, uma vez que este deve receber dados de uma requisição. Posteriormente, é gerada uma *hash* para controle das requisições de correção que serão instanciadas por uma *thread* que executa o processo de correção.

Algoritmo 3.1 – Desenvolvimento de um Microsserviço de Corretor Automático de Código-fonte para a ferramenta *opCoders Judge*

```
1 @app.route('/calcular', methods=['POST'])
2 def calcular_handler():
3     data = request.get_json()
```

```
    questaoPath = data['questaoPath']
    entregaPath = data['entregaPath']
    tarefa = data['tarefa']
    questao = data['questao']
    aluno = data['aluno']

    input_data = f"{data['questaoPath']}_{data['entregaPath']}
                    _{data['tarefa']}_{data['aluno']}
                    _{data['questao']}_{time.time()}"

    hashed_id = hashlib.sha256(input_data.encode()).hexdigest()

    estado_correcao[hashed_id] = 'Em andamento'

    with ThreadPoolExecutor() as executor:
        resultado_correcao = executor.submit(corrigir,
                                             questaoPath,
                                             entregaPath,
                                             tarefa,
                                             aluno, questao).result()

    estado_correcao[hashed_id] = 'Concluido'
    return jsonify({'result': resultado_correcao,
                    'request_id': hashed_id})
```

Fonte: Próprio autor.

A inicialização de *threads* é realizada pelo *ThreadPoolExecutor*, uma classe em *Python* que fornece uma implementação de um pool de *threads* de execução, ou seja, um conjunto de *threads* que podem ser reutilizadas, permitindo a execução de várias tarefas em paralelo e aproveitando os recursos da *CPU* (em inglês *Central Processing Unit*, ou Unidade Central de Processamento). Por fim, o resultado é retornado caso o processamento tenha sido concluído; caso contrário, o resultado não é retornado, mas é indicado que o processo ainda não foi finalizado.

Há ainda a necessidade de separação total entre o processo de salvar uma submissão no banco de dados e a correção. Os requerimentos que este trabalho tenta resolver são:

> [1] *Como um consumidor desta API, eu desejo submeter o meu código ao corretor e receber um identificador da correção.*

> [2] *Como um consumidor desta API, eu desejo saber se a correção do meu código foi feita e, se sim, qual o feedback, senão, ser avisado que ainda está em processamento.*

Uma vez que os requisitos, processos e a API foram definidos, foi necessário escolher um *framework* para a construção da API *RESTful* A primeira abordagem adotada neste trabalho

utilizou o *framework Flask* para implementar um microsserviço de correção baseado em chamadas síncronas. Nesse modelo inicial, a requisição *HTTP* (mensagem enviada pela aplicação) era recebida, processada imediatamente por uma instância de uma *thread*, e o resultado era retornado na mesma chamada. Embora funcional, essa arquitetura demonstrou limitações consideráveis ao lidar com um número elevado de requisições concorrentes, ocasionando perdas de mensagens, aumento de latência e instabilidade no servidor.

Diante dessas restrições, a arquitetura foi reformulada para um modelo baseado em mensageria de um modelo Produtor-Consumidor, utilizando *Apache Kafka*. Nesse novo cenário, o *Flask* foi descontinuado como ponto de entrada das requisições, e um produtor *Kafka* independente passou a ser responsável por simular o envio das mensagens. Esse produtor atua de forma autônoma, gerando requisições e publicando-as diretamente em um tópico *Kafka*, que são então processadas por consumidores.

3.3 Arquitetura utilizando o Modelo Produtor-Consumidor

A Figura 3.4 ilustra o primeiro modelo de arquitetura adotado no sistema, baseado em um único *broker Kafka*, com apenas um tópico e uma única partição. No ambiente do *Kafka*, um *tópico* é um canal nomeado onde as mensagens são publicadas pelos produtores. Cada tópico pode ser subdividido em múltiplas *partições*, que são unidades lógicas de armazenamento e paralelismo. Nesse cenário, o produtor enviava as mensagens diretamente para esse tópico centralizado, e um único consumidor era responsável por processar todo o fluxo de correções de código.

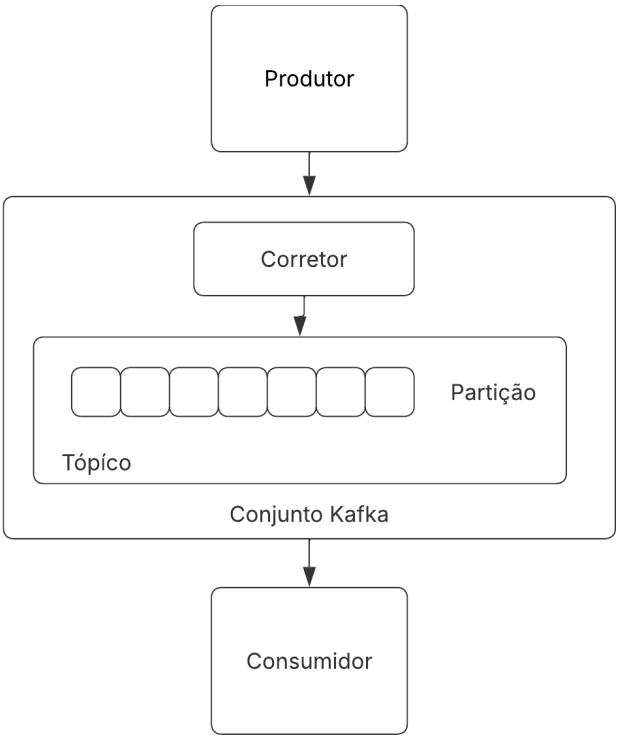
Nesta arquitetura, tem-se o produtor como o responsável por receber as diversas requisições de correção e gerar uma nova requisição de correção armazenada na partição do *Kafka*. Como somente há um único consumidor, este recupera a mensagem da partição, a processa e logo em seguida, como na implementação anterior utilizando exclusivamente o *Flask*, interpreta-se a submissão, corrige e gera-se os resultados.

3.3.1 Melhorias no modelo Produtor-Consumidor

O primeiro problema enfrentado estava relacionado ao tempo de processamento das mensagens. O sistema executa o mesmo código sequencialmente de correção para cada mensagem, tornando a operação ineficiente. Além disso, como apenas um consumidor estava encarregado do processamento, o sistema tende a não conseguir lidar adequadamente com o volume crescente de requisições, resultando em um gargalo significativo. Outro problema crítico é a dependência de um único nó, o que significava que, caso ele falhasse, todas as requisições pendentes seriam perdidas.

Diante das limitações dessa arquitetura inicial baseada em um único tópico com uma partição e apenas um consumidor, tornou-se necessária a implementação de otimizações estratégicas

Figura 3.4 – Modelo do corretor baseado em Produtor-Consumidor com uma partição (*broker*) e um único consumidor. O conjunto *Kafka* é responsável por gerenciar as mensagens e assegurar que o consumidor resolverá (fará a correção) a demanda de um produtor (questão para correção).



Fonte: Criado pelo autor.

para garantir escalabilidade, paralelismo e tolerância a falhas.

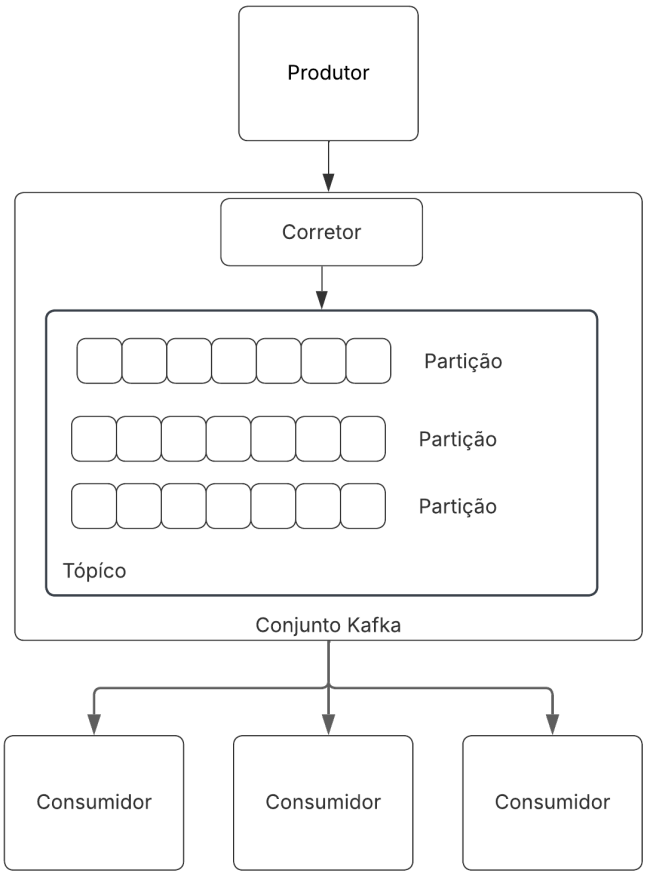
O primeiro passo foi aumentar o número de partições dentro do tópico *Kafka*. Essa mudança possibilitou a distribuição da carga de trabalho entre diferentes canais internos, permitindo que as mensagens fossem divididas de forma balanceada entre múltiplos consumidores. A consequência direta dessa reorganização foi:

- Maior paralelização do processamento das mensagens.
- Redução significativa no tempo total de execução.
- Aumento expressivo no *throughput* (taxa de processamento) do sistema.

A Figura 3.5 ilustra a nova estrutura adotada, com múltiplas partições e consumidores operando em paralelo.

Apesar dos ganhos de desempenho que serão mostrados no Capítulo 4, a nova configuração ainda apresenta vulnerabilidades: caso o único consumidor ativo pare de funcionar, as mensagens

Figura 3.5 – Modelo do corretor baseado em Produtor-Consumidor otimizado com múltiplas partições e grupo de consumidores. O conjunto *Kafka* é responsável por gerenciar as mensagens e assegurar que pelo menos um consumidor resolverá (fará a correção) a demanda de um produtor (questão para correção).



Fonte: Criado pelo autor.

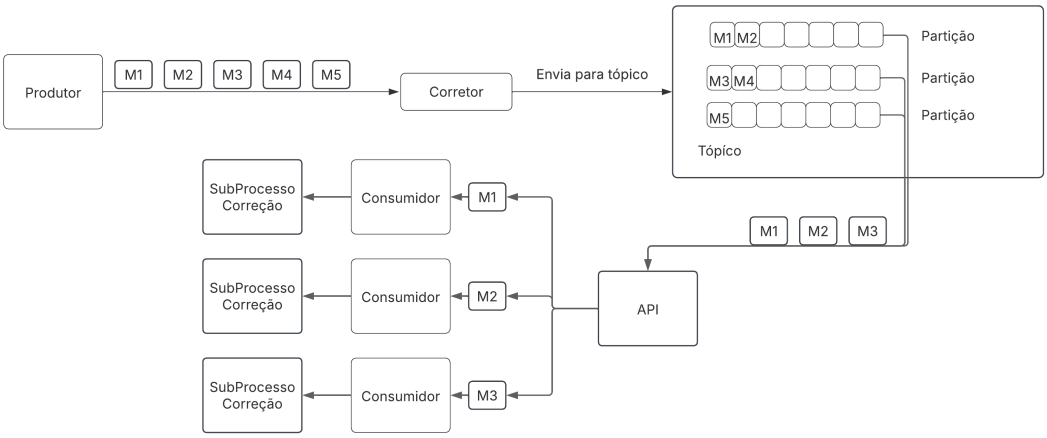
permanecem nas partições, sem serem processadas. Para resolver esse problema, foi implementado um grupo de consumidores.

Nesse modelo, múltiplas instâncias de consumidores pertencem ao mesmo grupo e compartilham a leitura das mensagens de forma coordenada. Se algum consumidor falhar, outro membro do grupo automaticamente assume a partição correspondente, garantindo alta disponibilidade do sistema. Além disso, essa estrutura permite:

- **Escalabilidade horizontal:** novos consumidores podem ser adicionados conforme a demanda cresce.
- **Balanceamento dinâmico de carga:** cada consumidor processa apenas um subconjunto das mensagens.
- **Maior resiliência:** o sistema se adapta a falhas de forma transparente, sem interrupção do serviço.

A Figura 3.6 apresenta o fluxo completo de funcionamento do sistema de correção automática baseado no modelo Produtor-Consumidor utilizando Apache Kafka. Nesse modelo, o produtor é responsável por enviar mensagens que representam submissões de código (identificadas como $M1$, $M2$, ..., $M5$) para o sistema de mensageria *Kafka*.

Figura 3.6 – Funcionamento do microsserviço baseado no modelo Produtor-Consumidor, onde Mi são as mensagens enviadas (requisições de correção), o Produtor / Corretor são os responsáveis pela requisição de correção aos consumidores que fazem a correção no "Subprocesso Correção".



Fonte: Criado pelo autor.

Essas mensagens são primeiro tratadas por um componente chamado corretor do Produtor, que realiza etapas iniciais de empacotamento. Em seguida, o corretor envia as mensagens para um tópico *Kafka*, que por sua vez está dividido em múltiplas partições. Cada partição armazena parte das mensagens recebidas, garantindo a paralelização do consumo. O sistema conta com três consumidores; cada consumidor é responsável por consumir uma mensagem (como $M1$, $M2$ e $M3$) e iniciar um subprocesso de correção que executa a lógica de avaliação dos códigos e opera em paralelo, permitindo maior escalabilidade e redução do tempo de resposta do sistema.

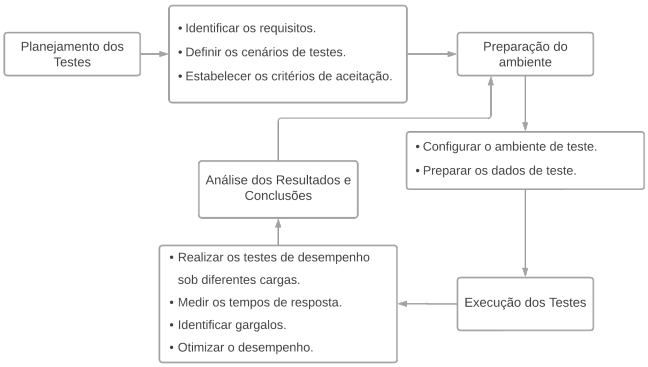
Vale ressaltar que como o processo de correção pode ser demorado, o *endpoint* de checagem é utilizado para checar o estado da requisição, que é atualizado pela *thread* responsável pela execução. Dessa forma, não é necessário que o *frontend* aguarde a conclusão da execução; basta consultar periodicamente o estado da requisição para obter a atualização do progresso.

Considerando um cenário onde sempre há produtores, essa mudança de estratégia não apenas eliminou o ponto único de falha, como também elevou a robustez do desempenho geral do microsserviço de correção, tornando-o capaz de lidar com cenários de alta concorrência com estabilidade e eficiência.

3.4 Planejamento dos testes das arquiteturas

A realização de testes é uma etapa fundamental no ciclo de desenvolvimento de *software*. Ela permite garantir que o produto final atenda aos requisitos de qualidade e funcionalidade estabelecidos. Para isso, é necessário seguir um processo estruturado que envolve diferentes etapas, como demonstra a Figura 3.7.

Figura 3.7 – Fluxo de Planejamento dos Testes.



Fonte: Criado pelo autor.

Esse fluxo de planejamento dos testes foi inicialmente desenvolvido com base em uma arquitetura que utilizava o *framework Flask*, sendo responsável por receber requisições e processá-las diretamente com uso de *threads*. Posteriormente, com a migração para uma arquitetura baseada em *Apache Kafka*, o mesmo fluxo foi adaptado para permitir a preparação de ambientes distribuídos e a execução de testes sob diferentes cenários de consumo e partições. A etapa de preparação do ambiente, portanto, contempla tanto a configuração do servidor *Flask* quanto a inicialização de tópicos, partições e consumidores no *Kafka*, permitindo avaliar a escalabilidade e desempenho do sistema em ambas as abordagens.

O primeiro passo é o planejamento dos testes. Nesta etapa, são identificados os requisitos de teste, definidos os cenários de teste, estabelecidos os critérios de aceitação e selecionadas as ferramentas de teste a serem utilizadas. Em seguida, vem a preparação do ambiente de teste. Nesta etapa, é configurado o ambiente de teste, preparados os dados de teste e instaladas e configuradas as ferramentas de teste. Posteriormente, são realizados os testes de acordo com os cenários definidos; por fim, vem a análise dos resultados dos testes. Nesta etapa, são analisados os resultados dos testes, identificando os problemas encontrados.

Para avaliar o comportamento e desempenho do sistema proposto, foram projetados testes com diferentes tipos de carga computacional e condições extremas de uso. O objetivo foi simular cenários variados de uso real, analisando a estabilidade e a capacidade de resposta do sistema sob diferentes níveis de exigência.

Inicialmente, foram utilizados algoritmos clássicos de ordenação *Bubble Sort*, *Selection*

Sort, *Insertion Sort* e *Merge Sort*, como tarefas processadas pelos consumidores Kafka. A escolha desses algoritmos se deve à sua variação de complexidade computacional, permitindo a simulação de tarefas com diferentes níveis de processamento. O *Bubble Sort* possui complexidade quadrática ($O(n^2)$), o que gera alta carga computacional, enquanto o *Merge Sort* possui complexidade $O(n \log n)$, sendo mais eficiente. Esses algoritmos foram aplicados sobre vetores com 1.000 elementos contendo dados aleatórios com e sem repetição, simulando o comportamento de correções de códigos submetidos ao sistema. Além dos testes com algoritmos, foram realizados dois testes extremos com o objetivo de investigar o comportamento do sistema em condições de estresse:

- O primeiro foi um teste de uso intensivo de CPU, no qual um laço de repetição infinito foi executado para simular um cenário em que uma tarefa nunca é concluída. Esse teste visou observar o comportamento do sistema em casos de saturação do processador.
- O segundo foi um teste de uso progressivo de memória, no qual foi executado um laço com alocação crescente de dados em uma lista. Esse cenário simulou o consumo contínuo e crescente de memória RAM até o ponto de esgotamento dos recursos disponíveis. O teste buscava identificar o momento em que o sistema deixava de responder devido à sobrecarga de memória.

Esses testes foram projetados para cobrir tanto o comportamento sob carga moderada quanto sob condições críticas de uso, utilizando métricas como tempo de resposta e taxa de processamento (*throughput*) para análise. Cada experimento contribuiu para compreender os limites e as capacidades do sistema ao longo de sua execução.

4 Experimentos e Resultados

Este capítulo apresenta os resultados obtidos a partir da implementação da arquitetura baseada em *Apache Kafka* no microsserviço de correção de código do sistema *opCoders Judge*. Os experimentos realizados buscaram validar o desempenho da solução, considerando tanto a escalabilidade quanto a eficiência no processamento de mensagens. Para isso, foram conduzidos testes de carga (execução do sistema sob diferentes volumes de requisições), medições de tempo (tempo médio de resposta entre o envio de uma requisição e o retorno da correção) e avaliações com algoritmos de ordenação aplicados ao processamento paralelo.

4.1 *Setup* de experimentos

Os experimentos realizados neste trabalho foram conduzidos em um ambiente controlado, com o objetivo de avaliar o desempenho do microsserviço de correção automática desenvolvido. A seguir, são descritos os principais componentes do ambiente de desenvolvimento, bem como as ferramentas, bibliotecas e versões utilizadas:

- **Linguagem de Programação:** Python3 versão 3.10.12.
- **Gerenciador de Pacotes:** *pip* versão 24.0.
- **Servidor de Mensageria:** *Apache Kafka* versão 3.4.
- ***ZooKeeper*:** Versão 3.8.1.
- **Produtor e Consumidor *Kafka*:** *kafka-python* versão 2.0.2.
- ***Framework* para construção da *API* da versão inicial dos testes:** *Flask* versão 3.0.3.
- **Execução Paralela:** *ThreadPoolExecutor* do pacote *concurrent*.
- **Geração de *Hash*:** *hashlib* (nativo do *Python*).
- **Serialização *JSON*:** pacote *json*.
- **Ambiente de Conteinerização:** *Docker* versão 26.0.0.
- **Gerenciador de Contêineres:** *Docker Compose* versão v2.26.1-desktop.1.
- **Sistema Operacional:** *Ubuntu 22.04.3 LTS*.

Ademais, para os testes de desempenho com algoritmos de ordenação, foram utilizadas implementações manuais dos algoritmos *Bubble Sort*, *Insertion Sort*, *Selection Sort* e *Merge Sort*, todos desenvolvidos em Python puro, sem bibliotecas externas.

4.1.1 Configuração de *Hardware*

Os testes de desempenho e experimentos descritos neste trabalho foram realizados em uma máquina de uso pessoal com a seguinte configuração de hardware:

- **Processador:** AMD Ryzen 7 5700X, 3.4GHz, 8 núcleos e 16 threads, Cache 32MB.
- **Placa-Mãe:** MSI B550M Pro-VDH WiFi, chipset AMD AM4.
- **Memória RAM:** 16GB DDR4, 3200MHz.
- **Armazenamento:** SSD Kingston NV2, 1TB, M.2 2280 PCIe NVMe (leitura: 3500MB/s, gravação: 2100MB/s).
- **Placa de Vídeo:** MSI NVIDIA GeForce RTX 4060 VENTUS 2X Black OC, 8GB GDDR6, DLSS e Ray Tracing.
- **Fonte:** MSI MAG A650BN, 650W, 80 Plus Bronze, com PFC ativo.

4.2 Análise de carga de mensagens com arquitetura *Flask*

O primeiro teste feito envolvia a realização de múltiplas rodadas de testes com o objetivo de avaliar o comportamento do sistema sob diferentes níveis de carga no contexto do uso exclusivo de uma arquitetura exclusivamente em *Flask*. Inicialmente, foram submetidos lotes de requisições concorrentes em escalas de 5, 10, 15, 20 e 30 requisições simultâneas. A partir de 20 requisições concorrentes, observou-se a ocorrência de gargalos críticos nessa arquitetura.

O desafio inicial enfrentado no sistema estava diretamente relacionado à escalabilidade limitada do *Flask*, que por padrão opera de forma síncrona. Esse modelo restringe o número de requisições que podem ser processadas simultaneamente, resultando em falhas na entrega das respostas, *timeouts* e até mesmo interrupções no funcionamento do servidor.

Durante os testes práticos, os seguintes problemas foram identificados:

- **Perda de requisições:** nem todas as solicitações eram efetivamente processadas, indicando falhas no recebimento ou na geração de resposta por parte do servidor.
- **Queda do servidor local:** o serviço *Flask* apresentou instabilidade durante picos de carga, resultando em falhas críticas e encerramentos inesperados.
- **Aumento expressivo da latência:** os tempos médios de resposta aumentaram de forma não linear com o crescimento do número de requisições simultâneas, comprometendo a experiência do usuário e a eficiência da aplicação.

Esses gargalos foram determinantes para reprovar o uso do *Flask* como solução principal para o microsserviço de correção em ambientes com alta concorrência. Apesar de sua leveza e simplicidade, o *Flask* demonstrou limitações significativas no gerenciamento eficiente de múltiplas requisições paralelas, sobretudo em servidores locais ou sem mecanismos adicionais de balanceamento de carga.

Diante dessas limitações, decidiu-se migrar para uma arquitetura baseada no modelo Produtor-Consumidor com *Apache Kafka*. Essa mudança teve como objetivo contornar os gargalos enfrentados, aumentar a capacidade de resposta do sistema e garantir alta disponibilidade e resiliência no processamento das mensagens. O uso do *Kafka* permitiu o desacoplamento entre a submissão de códigos e o processo de correção, possibilitando que o sistema se tornasse escalável, tolerante a falhas e preparado para demandas maiores sem comprometer sua estabilidade.

4.3 Análise de carga de mensagens com o modelo Produtor-Consumidor

A implementação inicial, utilizando uma única partição e um único consumidor, revelou um crescimento linear no tempo de execução à medida que o número de mensagens aumentava, conforme ilustrado na Tabela 4.1.

Tabela 4.1 – Resultados dos testes com *Kafka* (modelo simples).

Mensagens Processadas	**Tempo Total (s)**	**Tempo Médio (s)**
10	1,0118	0,1012
100	10,0254	0,1003
1.000	100,2462	0,1001
1.230	123,3286	0,1003

Fonte: Criado pelo autor.

Esses dados evidenciam a existência de um gargalo natural próximo de 1.200 mensagens, provavelmente relacionado à limitação da arquitetura inicial, que contava com apenas uma partição e um único consumidor ativo. A partir disso, é possível verificar que essa arquitetura simples também apresentou diversos gargalos durante os testes, tais como:

- **Baixo *throughput*:** foi possível processar entre 1.200 a 1.500 requisições por lote, mas com instabilidade crescente à medida que o volume aumentava.
- **Problemas de conexão e perda de mensagens:** a limitação de uma única partição significava que todas as mensagens formavam uma fila única, aumentando o tempo de resposta.
- **Ponto único de falha:** como havia apenas um consumidor, qualquer falha exigia reinício manual, comprometendo a disponibilidade do serviço.

Com base nessa limitação, foi realizada uma reconfiguração da arquitetura para permitir o uso de múltiplas partições no *Apache Kafka*. Especificamente, a configuração passou a utilizar **50 partições**, o que possibilitou uma divisão mais eficiente das mensagens entre diferentes consumidores. A decisão de utilizar 50 partições foi tomada com base em testes empíricos, que indicaram que, a partir dessa quantidade, o tempo médio de processamento por requisição se estabilizou. Ou seja, o acréscimo de novas partições além desse ponto não resultou em melhorias significativas de desempenho.

Essa alteração teve um impacto expressivo no desempenho, conforme pode ser visto na Tabela 4.2.

Tabela 4.2 – Resultados dos testes com *Kafka* (múltiplas partições).

Mensagens Processadas	**Tempo Total (s)**	**Tempo Médio (s)**
100	9,4770	0,0947
1.000	10,4143	0,0104
10.000	10,6187	0,0011

Fonte: Criado pelo autor.

Conforme ilustrado na Tabela 4.2, o tempo de resposta total e médio do microsserviço foi significativamente reduzido. Enquanto a arquitetura do *Kafka* com uma única partição apresentava um crescimento praticamente linear no tempo de execução, a introdução de múltiplas partições permitiu que o tempo de processamento permanecesse praticamente constante, mesmo com o aumento do número de mensagens.

4.4 Avaliação da Escalabilidade e Resiliência no Processamento de Mensagens com o modelo Produtor-Consumidor

Após a definição da melhor configuração com base nos testes de carga de mensagens, foram conduzidos experimentos para avaliar o desempenho do sistema ao processar tarefas computacionalmente mais exigentes. Para isso, conforme descrito na Capítulo 3, foram utilizados quatro algoritmos clássicos de ordenação: *Bubble Sort*, *Selection Sort* e *Insertion Sort*. Os testes foram aplicados sobre vetores de 1.000 posições, contendo dados aleatórios e repetidos, com o objetivo de analisar como o consumidor lida com diferentes complexidades algorítmicas. A Tabela 4.3 apresenta os tempos de execução obtidos, em segundos, para diferentes quantidades de mensagens processadas.

Os resultados presentes na Tabela 4.3 evidenciam diferenças consideráveis entre os algoritmos, refletindo diretamente suas complexidades computacionais. O *Bubble Sort*, com complexidade $O(n^2)$, apresentou o maior tempo de execução em todos os cenários testados, alcançando mais de 760 segundos para 10.000 mensagens. O *Selection Sort* e o *Insertion Sort*, que também possuem complexidade quadrática, tiveram desempenhos semelhantes, mas ligei-

Tabela 4.3 – Tempo de execução em segundos utilizando os métodos de ordenação como questões a serem corrigidas.

Mensagens Processadas	**Métodos de ordenação**			
	Bubble Sort	*Selection Sort*	*Insertion Sort*	*Merge Sort*
10	13,1171	13,1193	13,1209	13,1089
100	15,4364	13,1423	14,1683	13,1290
1.000	80,4815	70,3831	66,7459	47,0286
10.000	761,8755	604,4136	562,7002	399,0477

Fonte: Criado pelo autor.

ramente melhores. Já o *Merge Sort*, com complexidade $O(n \log n)$, demonstrou desempenho significativamente superior, sendo o algoritmo mais eficiente entre os testados.

Esse comportamento reforça a importância de considerar a complexidade algorítmica das tarefas ao definir a carga de trabalho do consumidor. Em ambientes com alta demanda, o uso de algoritmos mais eficientes como o *Merge Sort* contribui diretamente para a redução do tempo total de processamento, o que impacta positivamente no *throughput* do sistema.

Ademais, os testes demonstram que, mesmo diante de algoritmos com alta complexidade, a arquitetura Produtor-Consumidor foi capaz de suportar a carga, processando até 10.000 mensagens. Outro ponto de atenção é que, embora a quantidade de mensagens tenha aumentado linearmente, o tempo de execução não seguiu essa mesma proporção. Observou-se que, para um aumento de 100 vezes no número de mensagens, o tempo de execução cresceu entre 40 e 60 vezes em códigos com complexidade quadrática e aproximadamente 30 vezes em um código com complexidade linear-logarítmica. Esses resultados indicam um ganho de desempenho em comparação com a estratégia baseada exclusivamente no *Flask*.

4.5 Avaliação da Escalabilidade e Resiliência com o paradigma Produtor-Consumidor em testes extremos

Por fim, foram realizados dois testes extremos para avaliar o comportamento do sistema em condições de sobrecarga intencional. O primeiro consistiu na execução de um *loop* infinito, enquanto o segundo implementou um laço de repetição infinito com alocação de memória que resultava em um consumo progressivo e descontrolado de memória. O objetivo desses testes foi determinar até que ponto o sistema conseguiria manter sua estabilidade antes de colapsar em cada um dos cenários.

No caso do teste extremo por uso de processamento (*loop* infinito), foi observado que o sistema manteve o processamento das mensagens por um tempo maior em relação ao teste de memória, embora apresentasse degradação gradual no desempenho. O alto consumo de CPU provocado pelo *loop* contínuo não resultou na queda do sistema, mas gerou lentidão perceptível na leitura e no consumo das mensagens. Vale ressaltar que o próprio processo de correção do

opCoders Judge inclui estratégias para evitar programas que não têm fim, onde um *timeout* é incluído.

No caso do teste extremo por uso de memória (*loop* infinito com alocação de memória), o processo de consumo de memória foi tão intenso que comprometeu completamente os recursos da máquina, levando ao travamento do sistema operacional e à interrupção forçada do experimento. Observou-se que, antes de atingir esse estado crítico, o sistema conseguiu processar, em média, entre 5.000 e 5.500 mensagens. Isso indica que o consumo excessivo de memória foi o principal limitador da escalabilidade em cenários sem controle adequado de recursos. Uma possível solução para este problema é o uso de contêineres *Docker* para limitar o uso de memória e evitar o esgotamento de recursos do *host* que é responsável pelo microsserviço de correção de questões.

5 Considerações Finais

Este capítulo apresenta as conclusões obtidas neste trabalho (Seção 5.1), bem como os possíveis trabalhos futuros que são vislumbrados após a execução e análise dos testes (Seção 5.2).

5.1 Conclusão

O presente trabalho teve como objetivo principal a reformulação da arquitetura do processo de correção do *opCoders Judge*, por meio da implementação de uma abordagem baseada em microsserviços. Os principais focos eram uma maior escalabilidade, paralelismo e robustez no processo de correção automática de códigos-fonte.

Inicialmente, foi adotada uma solução utilizando o *framework Flask* com execução baseada em *threads*, em que cada requisição era processada paralelamente, porém ainda de forma fortemente acoplada ao servidor *web*. Essa arquitetura mostrou-se funcional em cenários de baixa carga, mas apresentou gargalos significativos a partir de 20 requisições (correções) simultâneas. Entre os problemas observados estavam perdas de requisições, aumento da latência média e interrupções no servidor durante picos de carga, evidenciando limitações severas de escalabilidade.

Diante disso, foi proposta a substituição por uma arquitetura assíncrona baseada no modelo Produtor-Consumidor, utilizando o *Apache Kafka* como sistema de mensageria. Essa mudança permitiu o desacoplamento entre o envio e o processamento das correções, além de possibilitar a utilização de múltiplos consumidores operando em paralelo. A introdução de partições no *Kafka* proporcionou ganhos expressivos de desempenho, com aumento significativo do *throughput* e melhor distribuição de carga entre os consumidores.

Os testes realizados demonstraram que, com ajustes apropriados na arquitetura, como aumento no número de partições e controle de concorrência, o sistema foi capaz de processar uma boa quantidade de mensagens com baixa latência média por requisição. Além disso, testes extremos de estresse, por uso intensivo de memória e CPU, permitiram identificar limites operacionais e reforçaram a importância de mecanismos de controle e isolamento, como a limitação de recursos via contêineres *Docker*.

Conclui-se, portanto, que grande parte dos objetivos propostos foi alcançada com êxito. A arquitetura final demonstrou-se mais robusta, resiliente e adequada para cenários de alta demanda, oferecendo uma solução eficaz para sistemas de correção automática que exigem paralelismo e desempenho escalável. Este trabalho evidenciou, na prática, os benefícios da migração de arquiteturas monolíticas para microsserviços e do uso de mensageria como estratégia de desacoplamento e escalabilidade.

Entretanto, ressalta-se que a integração da nova arquitetura com a interface e o fluxo completo do sistema *opCoders Judge* ainda não foi realizada, permanecendo como uma etapa futura. Os testes e validações foram conduzidos com o módulo de correção de forma isolada, com o intuito de avaliar seu desempenho e estabilidade antes da sua incorporação definitiva ao sistema principal.

5.2 Trabalhos Futuros

Embora os resultados obtidos tenham sido satisfatórios, diversas melhorias podem ser exploradas em versões futuras do sistema:

- Orquestração de contêineres com *Kubernetes*: permitiria maior tolerância a falhas, balanceamento automático de carga e escalabilidade dinâmica dos consumidores.
- Integração com bancos de dados: utilizar o banco atual da disciplina para armazenar os resultados das correções e permitir consulta rápida.
- Limitação de recursos no contêiner: implementar restrições de uso de memória e *CPU* diretamente nos contêineres *Docker*. Essa medida visa prevenir estouros de memória, reduzir o risco de travamentos em ambientes de alta concorrência e garantir uma alocação eficiente de recursos em cenários com múltiplos consumidores ativos.

Referências

ABDULLAH, H. M.; ZEKI, A. M. Frontend and backend web technologies in social networking sites: Facebook as an example. In: IEEE. 2014 3rd international conference on advanced computer science applications and technologies. [S.l.], 2014. p. 85–89.

ALECRIM, A. M. E. O que são threads do processador e quais os benefícios do multithreading? [S.l.], 2023.

BELTRAME, G. A. W. L. F. S. Sistema de correÇÃo automÁtica de cÓdigo- fonte. UFOP, v. 1, p. 2, 2018.

BENEDETTI, T. O que é metodologia baseada em problemas (PBL)? [S.l.], 2023.

BERSSANETTE, A. C. d. F. J. H. Uma proposta de ensino de programação de computadores com base na pbl utilizando o portal uri online judge. UFOP, v. 1, p. 2, 2018.

BHUYAN, A. Comprehensive Guide to Microservices Testing: Ensuring Reliable and Scalable Software. [S.l.], 2023.

BRITO, P. S. S. O uso de ferramentas computacionais para o ensino de programaÇÃo para alunos de engenharia. UFOP, v. 1, p. 38, 2019.

CEDRAZ, V. F. Uma avaliação de usabilidade do corretor de exercícios de introdução à programação opcoders judge. 2023.

CLEMENTE, A. A. de S.; SILVA, E. d. O. da. Migração de sistemas monolíticos para microsserviços. Caderno de Estudos em Sistemas de Informação, v. 7, n. 2, 2022.

CONCEIÇÃO, M. T. da; PINTO, G. S. Arquitetura de microsserviços. Revista Interface Tecnológica, v. 18, n. 2, p. 53–64, 2021.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: Yesterday, today, and tomorrow. Present and Ulterior Software Engineering, Springer, p. 195–216, 2017.

FARIA, G. A. de; ARROYOA, J. E. C.; SANTOSA, A. G. dos; NOGUEIRAB, T. H.; CHAGASA, J. B. C. das. Sequenciamento de tarefas em máquinas paralelas de processamento em lotes com entregas. Simpósio Brasileiro de Pesquisa Operacional (XLIX SBPO), 2017.

FERNANDES, J. C.; CARVALHO, L. S. G. de; OLIVEIRA, D. B. F. de; OLIVEIRA, E. H. T. de; PEREIRA, F. D.; LAUSCHNER, T. Complexidade versus dificuldade: Uma análise da sua correlação em questões de programação em juízes on-line. Revista Brasileira de Informática na Educação, v. 32, p. 22–49, 2024.

GOMES, A.; HENRIQUES, J.; MENDES, A. J. Uma proposta para ajudar alunos com dificuldades na aprendizagem inicial de programação de computadores. Educação, Formação e Tecnologias, Associação Portuguesa de Telemática Educativa, v. 1, n. 01, p. 93–103, 2008.

GONçALVES, M. M. Arquitetura de Microsserviços. [S.l.], 2020.

GORTON, I. Essential software architecture. [S.l.]: Springer Science & Business Media, 2006.

HALOI, S. Apache zookeeper essentials. [S.l.]: Packt Publishing Ltd, 2015.

HUNT, P.; KONAR, M.; JUNQUEIRA, F. P.; REED, B. {ZooKeeper}: Wait-free coordination for internet-scale systems. In: 2010 USENIX Annual Technical Conference (USENIX ATC 10). [S.l.: s.n.], 2010.

JÚNIOR, E. S. Threads. 2007.

JUNQUEIRA, F.; REED, B. ZooKeeper: distributed process coordination. [S.l.]: "O'Reilly Media, Inc.", 2013.

KRATZKE, N. About microservices, containers and their underestimated impact on network performance. arXiv preprint arXiv:1710.04049, 2017.

LUU, T.; JANJUSIC, T.; YILMAZ, C. Trak: A testing tool for studying the reliability of data delivery in apache kafka. In: 2019 IEEE International Symposium on Software Reliability Engineering (ISSRE). [S.l.]: IEEE, 2019. p. 295–305.

MACHADO, M. Micro serviços: Qual a diferença para a arquitetura monolítica. Acessado em, v. 21, 2017.

MAGALHãES, G. M. Uma biblioteca para testes determinísticos em sistemas distribuídos com comunicação assíncrona. Dissertação (Dissertação de Mestrado) — Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, MG, 2023. Disponível em: <https://repositorio.ufmg.br/handle/1843/62872>.

MENDONÇA, T. S. Projeto e desenvolvimento de uma plataforma de gestão de questões dinâmicas para o ensino de programação de computadores. 2023.

MERKEL, D. et al. Docker: lightweight linux containers for consistent development and deployment. Linux j, v. 239, n. 2, p. 2, 2014.

MOREIRA, E. A.; CARRIEL, G. N. Paralelismo em Nível de Thread. [S.l.]: entre, 2015.

NARKHEDE, N.; SHAPIRA, G.; PALINO, T. Kafka: the definitive guide: real-time data and stream processing at scale. [S.l.]: "O'Reilly Media, Inc.", 2017.

NETO, A. S. Desenvolvimento de uma aplicação pwa que comporte outras aplicações usando arquitetura de micro frontend. Pontifícia Universidade Católica de Goiás, 2020.

OLIVEIRA, A. C. de; PILAN, J. R. Desenvolvimento de software desktop responsivo com framework em flask. In: VII JORNACITEC-Jornada Científica e Tecnológica. [S.l.: s.n.], 2018.

PATROCíNIO, J. A. do. Opcoders judge: Uma versÃo online para o corretor automÁtico de exercÍcios de programaÇÃo do projeto opcoders. UFOP, v. 1, p. 26, 2023.

PEREIRA, F. D. et al. Uso de um método preditivo para inferir a zona de aprendizagem de alunos de programação em um ambiente de correção automática de código. Universidade Federal do Amazonas, 2018.

RAD, B. B.; BHATTI, H. J.; AHMADI, M. An introduction to docker and analysis of its performance. International Journal of Computer Science and Network Security (IJCSNS), International Journal of Computer Science and Network Security, v. 17, n. 3, p. 228, 2017.

SANTOS, A.; GORGÔNIO, A.; LUCENA, A.; GORGÔNIO, F. A importância do fator motivacional no processo ensino-aprendizagem de algoritmos e lógica de programação para alunos repetentes. In: SBC. Anais do XXIII Workshop sobre Educação em Computação. [S.l.], 2015. p. 168–177.

SCOTT, D.; GAMOV, V.; KLEIN, D. Kafka in Action. [S.l.]: Simon and Schuster, 2022.

SELIVON, M.; BEZERRA, J.; TONIN, N. Uri online judge academic: Integração e consolidação da ferramenta no processo de ensino/aprendizagem. In: SBC. Anais do XXIII Workshop sobre Educação em Computação. [S.l.], 2015. p. 188–195.

SEQUEIRA, A. M. G. d. C. d. S. et al. Estender a Plataforma de Microserviços Elásticos. Dissertação (Mestrado) — Universidade de Coimbra, 2020.

SOMMERVILLE, I. Software engineering (ed.). America: Pearson Education Inc, 2011.

SOUZA, F. M. C.; LIMA, E. C. S.; CARIDADE, E. R. de S. Criando sistema escalável de agendamentos utilizando typescript com nestjs no backend e nextjs no frontend. Revista Ibero-Americana de Humanidades, Ciências e Educação, v. 8, n. 12, p. 43–57, 2022.

TIBC. O que é um API Gateway? [S.l.], 2022.

UPADHYA, S.; SHETTY, J.; RAJESHWARI, H. R.; SHOBHA, G. A state-of-art review of docker container security issues and solutions. American International Journal of Research in Science, Technology, Engineering and Mathematics, v. 17, p. 33–36, 2017.