Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Uso do Github Actions em Pipeline de Testes para Microsserviços no Contexto de CI/CD e DevOps

Guilherme Ferreira de Souza Sobrinho

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Euler Horta Marinho

Agosto, 2025
João Monlevade–MG

Guilherme Ferreira de Souza Sobrinho

Uso do Github Actions em Pipeline de Testes para Microsserviços no Contexto de CI/CD e DevOps

Orientador: Euler Horta Marinho

Monografia apresentada ao curso de Engenharia de Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Agosto de 2025

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

S677u Sobrinho, Guilherme Ferreira de Souza.
Uso do Github Actions em pipeline de testes para microsserviços no contexto de CI/CD e DevOps.. [manuscrito] / Guilherme Ferreira de Souza Sobrinho. - 2025.
41 f.: il.: color..

Orientador: Prof. Dr. Euler Horta Marinho.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Engenharia de Computação .

1. GitHub (Programa de computador). 2. Engenharia de software. 3. Software - Garantia de qualidade. 4. Software - Testes. I. Marinho, Euler Horta. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.41

Bibliotecário(a) Responsável: Flavia Reis - CRB6-2431

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

FOLHA DE APROVAÇÃO

Guilherme Ferreira de Souza Sobrinho

Uso do Github Actions em pipeline de testes para microsserviços no contexto de CI/CD e DevOps

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

Aprovada em 26 de agosto de 2025

Membros da banca

Dr. Euler Horta Marinho - Orientador - Universidade Federal de Ouro Preto
Dra. Kattiana Fernandes Constantino - Universidade Federal dos Vales do Jequitinhonha e Mucuri
Dra. Gilda Aparecida de Assis - Universidade Federal de Ouro Preto
Dr. Igor Muzetti Pereira - Universidade Federal de Ouro Preto

Euler Horta Marinho, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 28/08/2025

Documento assinado eletronicamente por **Euler Horta Marinho**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 28/08/2025, às 15:48, 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 **0968961** e o código CRC **22CED315**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

A todas as pessoas do mundo, em especial àquelas que vierem a usufruir dele, dedico este trabalho.

Agradecimentos

Agradeço a todos que, de alguma forma, fizeram parte desta jornada. À minha família e amigos, pelo apoio incondicional. Aos meus mentores, por guiarem meus passos e compartilharem sua sabedoria. E a cada desafio, que me ensinou a crescer e a ser mais forte. Este trabalho é o resultado de um esforço coletivo.

"Science is more than a body of knowledge; it is a way of thinking."

— Carl Sagan (1934 – 1996),
in: The Demon-Haunted World: Science as a Candle in the Dark.

Resumo

Este trabalho explorou a aplicação do GitHub Actions como ferramenta para a automação de pipelines de testes em arquiteturas de microsserviços, no contexto de Continuous Integration/Continuous Delivery (CI/CD) e Development and Operations (DevOps). Embora a arquitetura de microsserviços tenha trazido vantagens como flexibilidade e escalabilidade, apresentou também desafios relacionados à garantia da qualidade do software. Para enfrentar esses desafios, foi desenvolvida uma solução de pipeline automatizada, com o objetivo de garantir uma cobertura de testes eficiente para cada microsserviço. O estudo demonstrou como a automação de testes pode melhorar a qualidade do software e a eficiência no processo de desenvolvimento contínuo, evidenciando, nos resultados, a alta eficiência da suíte de testes e a necessidade de aprimoramento da cobertura de branches para cenários de exceção.

Palavras-chaves: GitHub Actions. Testes automatizados. Microsserviços.

Abstract

This work explored the application of GitHub Actions as a tool for automating test pipelines in microservices architectures, within the context of Continuous Integration/-Continuous Delivery (CI/CD) and Development and Operations (DevOps). While the microservices architecture has brought advantages such as flexibility and scalability, it also presented challenges related to ensuring software quality. To address these challenges, an automated pipeline solution was developed, aiming to ensure efficient test coverage for each microservice. The study demonstrated how test automation can improve software quality and efficiency in the continuous development process, highlighting, in the results, the high efficiency of the test suite and the need to improve branch coverage for exception scenarios.

Key-words: GitHub Actions. Automated tests. Microservices.

Lista de ilustrações

Lista de abreviaturas e siglas

AAA Arrange, Act, Assert

CRUD Create, Read, Update, Delete

DTO Data Transfer Object

E2E End-to-End

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

PoC Proof of concept

SGBD Sistema Gerenciador de Banco de Dados

SQL Structured Query Language

Sumário

1 Introdução

O desenvolvimento de software passou por grandes transformações com a adoção de práticas de DevOps e CI/CD *(Continuous Integration/Continuous Delivery)*. Segundo Valente (2022), essas práticas promovem a automação no processo de integração, testes e entrega de novas funcionalidades de forma contínua. Essas práticas permitem que as equipes de engenharia entreguem software de maneira mais rápida, eficiente e escalável, sem comprometer sua qualidade.

Leite et al. (2019) consideram DevOps como sendo [...] um esforço colaborativo e multidisciplinar dentro de uma organização para automatizar a entrega contínua de novas versões de *software*, garantindo sua correção e confiabilidade. Dentro deste contexto, ainda, surgem duas novas definições que são parte integral da cultura de DevOps, que são: CI/CD *(Continuous Integration/Continuous Delivery)*. Ainda segundo Leite et al. (2019) a entrega contínua *(Continuous Delivery)* é um processo em que qualquer versão de software comprometida no repositório deve estar apta a ser implantada em produção, passando por etapas automatizadas como compilação e testes. A implantação ocorre com o pressionar de um botão, garantindo agilidade e segurança no processo. Uma variação deste modelo é a implantação contínua *(Continuous Deployment)*, na qual as versões aprovadas pela pipeline são automaticamente promovidas para produção, sem intervenção manual.

Fowler (2017) define microsserviços como uma pequena aplicação que executa uma única tarefa e o faz com eficiência. Paralelamente, a arquitetura de microsserviços tem sido amplamente adotada devido à sua flexibilidade e escalabilidade, dividindo sistemas monolíticos em pequenos serviços independentes, que podem ser desenvolvidos, testados e implantados de forma isolada. Segundo Valente (2022), essa abordagem permite uma melhor organização do código e facilita a manutenção. No entanto, esse modelo também introduz desafios adicionais em termos de testes e garantia da qualidade do *software*, uma vez que a fragmentação dos sistemas pode dificultar a aplicação eficaz de metodologias de teste tradicionais, como a pirâmide de testes. Chen (2018) reforça que a arquitetura da aplicação pode ser uma barreira importante.

Nesse cenário, o *GitHub Actions* surge como uma poderosa ferramenta para automatizar *pipelines* de CI/CD em cenários de desenvolvimento reais, como discutido por Wessel et al. (2023), especialmente no que se refere à automação de testes em ambientes distribuídos e de microsserviços. Ao automatizar especialmente os testes de unidade, mas eventualmente de integração e *end-to-end*, é possível facilitar a detecção de problemas e assegurar que os serviços atendam a padrões de qualidade ao longo de todo o ciclo de desenvolvimento e implantação.

Portanto, este trabalho tem como objetivo explorar a aplicação do *GitHub Actions* como ferramenta para criação e automação de pipelines de testes em arquiteturas de microsserviços, com foco na melhoria da qualidade de software e na eficiência dos processos de desenvolvimento contínuo. A pesquisa busca demonstrar como a automação de testes pode ser aplicada para diferentes camadas de testes, e como isso impacta positivamente o ciclo de desenvolvimento e a robustez dos sistemas.

1.1 O problema de pesquisa

Com o crescente uso de arquiteturas de microsserviços, novos desafios emergem em relação à garantia da qualidade do software. Como foi lembrado por Valente (2022), a fragmentação do sistema em múltiplos serviços independentes aumenta a complexidade dos testes, dificultando a aplicação tradicional da pirâmide de testes (unidade, integração, *end-to-end*). Além disso, a natureza distribuída dos microsserviços impõe a necessidade de integração contínua, exigindo pipelines de testes que automatizem e garantam a execução eficiente de testes em cada serviço. Nesse contexto, surge a dificuldade de manter uma cobertura de testes ampla e contínua sem comprometer a velocidade de entrega do sistema, característica fundamental do *DevOps*.

1.2 Objetivos

Este trabalho tem como objetivo principal explorar a aplicação do *Github Actions* como ferramenta para a criação de pipelines de testes em ambientes de microsserviços, com foco em garantir a qualidade de software em cada fase do desenvolvimento e integração contínua. Para isso, propõe-se:

- Desenvolver uma pipeline de testes automatizada utilizando o *Github Actions* que seja capaz de validar microsserviços inicialmente de forma individual.
- Explorar diferentes camadas de testes (unidade, de integração e *end-to-end*) e sua aplicabilidade dentro de um *pipeline* CI/CD.
- Avaliar como a automação desses testes pode impactar na entrega contínua, qualidade e manutenção de sistemas baseados em microsserviços.

1.3 Método

O objetivo de pesquisa deste trabalho pretende apresentar um estudo de caso que trata de *pipelines* de testes automatizados no contexto de arquitetura de microsserviços, de modo que os passos para atingir estes objetivos são assim definidos:

- Revisão da literatura disponível sobre o assunto;
- Desenvolvimento de duas Provas de Conceito (do inglês POC - *Proof of concept*) e pipeline automática;
- Validação dos dados obtidos; e
- Análise e discussão dos resultados alcançados.

1.4 Organização do Trabalho

O restante deste trabalho está organizado conforme descrito a seguir.

O Capítulo 2 apresenta a revisão bibliográfica que embasa este estudo, reunindo conceitos fundamentais sobre arquiteturas de microsserviços, práticas de *DevOps*, estratégias de testes automatizados e modelos de *deployment* contínuo. São discutidas abordagens clássicas como a Pirâmide de Testes e abordagens novas, como os Quadrantes Ágeis, além de reflexões recentes sobre a influência do código-fonte na geração de testes por inteligência artificial.

O Capítulo 3 descreve o método empregado no desenvolvimento da pesquisa. Nele, são detalhados os microsserviços utilizados como prova de conceito (Task e Todo)[1], as tecnologias adotadas (NestJS, PostgreSQL, GitHub Actions), e os tipos de testes implementados (unidade, de integração e End-to-End (E2E)), bem como o fluxo de CI/CD aplicado.

O Capítulo 4 apresenta os resultados obtidos a partir da implementação dos *pipelines* de testes automatizados. São discutidos os benefícios observados em termos de confiabilidade e integração contínua, além das limitações enfrentadas durante o desenvolvimento.

O Capítulo 5 encerra o trabalho com as considerações finais, síntese das contribuições, dificuldades encontradas e sugestões de trabalhos futuros relacionados ao tema.

Por fim, o Apêndices A contém *scripts* de um aquivo .yml, de automação completa de um microsserviço.

[1] Os repositórios dos microsserviços podem ser acessados em: <https://github.com/guiinow/task-MS> e <https://github.com/guiinow/todo-ms>.

2 Revisão Bibliográfica

Neste estudo, a fundamentação teórica reúne contribuições que se complementam na compreensão de arquiteturas de microsserviços, práticas de *DevOps* e automação de testes. Valente (2022) estabelece desde cedo a relevância de padrões arquiteturais — como MVC, microsserviços e *publish/subscribe* — para a construção de sistemas escaláveis e de fácil manutenção, e correlaciona esses padrões com práticas de testes (unidade, cobertura de código e TDD) que sustentam a qualidade de software em ambientes *DevOps*, destacando ainda a adoção de Git e *pipelines* de *deployment* contínuo.

Humble e Farley (2010), por sua vez, formalizam o conceito de *pipeline* de entrega contínua, propondo uma sequência automatizada de etapas (*build*, testes e *deploy*) na qual toda versão de código integrável ao repositório está pronta para produção, bastando um único comando para acionar sua liberação. Essa visão é estendida para a implantação contínua, em que cada alteração validada pela pipeline é promovida automaticamente ao ambiente produtivo, reduzindo drasticamente o tempo entre o desenvolvimento e a disponibilidade de novas funcionalidades.

Fowler (2017) complementa essa visão ao detalhar estratégias de qualidade em microsserviços, baseadas em experiências reais de padronização de milhares de serviços. Ela enfatiza a importância de monitoramento, tolerância a falhas e testes distribuídos, orientando como aplicar testes de integração e *end-to-end* que assegurem a robustez em cenários distribuídos.

O estudo de caso apresentado em Chen (2018) ilustra os desafios e benefícios da migração de uma aplicação monolítica para microsserviços com *Continuous Delivery*, ressaltando a necessidade de reestruturação dos processos de desenvolvimento, integração de testes automatizados e adaptação das equipes para suportar a nova cadência de lançamentos.

Huang et al. (2024) trazem um viés inovador ao investigar a influência do código-fonte na geração automática de casos de teste por inteligência artificial. Utilizando técnicas de *machine learning*, eles demonstram que a qualidade dos *prompts* — ou seja, a correção do próprio código-fonte — é determinante para a precisão dos testes gerados, o que reforça a importância de uma base de código rigorosamente testada.

Finalmente, Leite et al. (2019) oferecem uma visão abrangente dos conceitos e desafios do *DevOps*, integrando práticas de CI, CD e implantação contínua. Eles evidenciam que a combinação de automação e colaboração entre desenvolvimento e operações é fundamental para a entrega confiável e eficiente de software, servindo de alicerce para a adoção de pipelines de testes em microsserviços.

Ainda neste contexto, dentre as principais estratégias de *rollout* em ambientes de microsserviços, destacam-se três modelos: *Canary Release, Blue-Green Deployment e Trunk-Based Development*. Segundo Fowler (2014a), no modelo de *Canary Release*, a nova versão do serviço é inicialmente disponibilizada a um subconjunto restrito de usuários ou instâncias, permitindo a avaliação de métricas de desempenho e uso em ambiente real antes do *rollout* completo. Em caso de falhas, basta redirecionar o tráfego de volta à versão anterior, minimizando impactos.

No que se refere ao *Blue-Green Deployment*, a Amazon Web Services (2020) descreve essa estratégia como a manutenção de dois ambientes idênticos — "*blue*" e "*green*" —, dos quais apenas um atende requisições em produção. As alterações são implantadas no ambiente inativo (*green*) e, após validação, o tráfego é instantaneamente alternado para ele. Essa técnica elimina *downtime* e simplifica o *rollback*, pois basta apontar o roteador de volta ao ambiente anterior.

Por fim, de acordo com a Atlassian (2021), o *Trunk-Based Development* é uma prática de versionamento de código em que desenvolvedores integram pequenas alterações diretamente no ramo principal (*trunk*) de forma frequente — geralmente várias vezes ao dia. Esse fluxo reduz conflitos de *merge* e mantém o código sempre em estado integrável, suportando pipelines de CI/CD com execuções contínuas de *builds* e testes automatizados.

No contexto de desenvolvimento ágil, a *Pirâmide de Testes* e os *Quadrantes Ágeis de Testes* são modelos amplamente utilizados para orientar estratégias de testes. A Figura 1 ilustra a *Pirâmide de Testes*, proposta por Fowler (2018), que sugere uma proporção ideal entre diferentes tipos de testes: uma base ampla de testes de unidade, uma camada intermediária de testes de integração e uma camada superior mais estreita de testes de interface gráfica. Essa estrutura visa otimizar a eficiência dos testes, priorizando aqueles que são mais rápidos e menos custosos de manter.

Valente (2022) retoma a Pirâmide de Testes, originalmente proposta por Mike Cohn em Cohn (2009), como uma ferramenta essencial para orientar a estratégia de testes em projetos de software. A pirâmide classifica os testes automatizados em três níveis distintos, organizados de acordo com sua granularidade, custo e velocidade de execução. Na base, encontram-se os testes de unidade, que verificam pequenas partes do código, como métodos ou classes individuais. Esses testes são numerosos, rápidos e de fácil implementação, proporcionando *feedback* imediato aos desenvolvedores e facilitando a detecção precoce de defeitos.

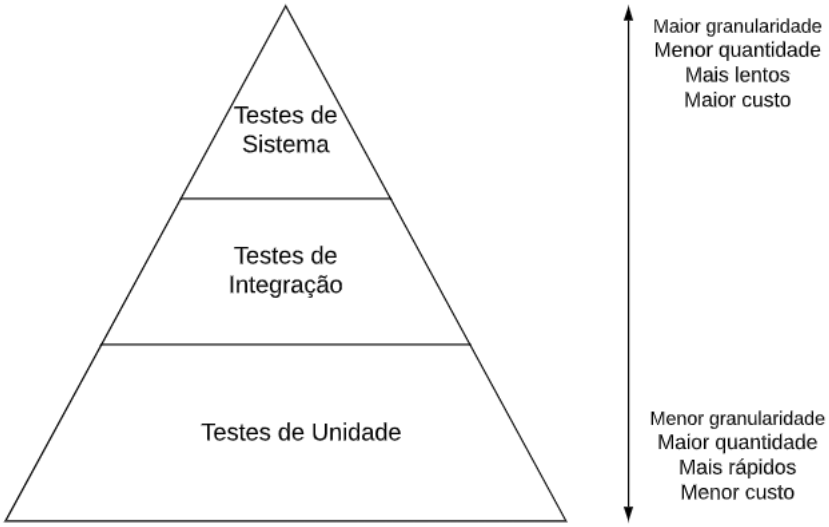
No nível intermediário, situam-se os testes de integração, que avaliam a interação entre diferentes componentes ou serviços do sistema. Esses testes são menos numerosos que os de unidade e exigem maior esforço para implementação e manutenção, uma vez que envolvem múltiplas partes do sistema e, frequentemente, dependências externas, como bancos de dados ou serviços de terceiros. No topo da pirâmide estão os testes de sistema,

também conhecidos como testes de interface ou testes *end-to-end*. Eles simulam o uso do sistema por um usuário real, verificando o comportamento do software como um todo. Devido à sua complexidade e custo elevado, esses testes são mais escassos e suscetíveis a falhas causadas por pequenas alterações na interface ou no fluxo de interação.

Essa estrutura hierárquica sugere uma distribuição proporcional dos testes, como exposto em Whittaker, Arbon e Carollo (2012): aproximadamente 70% de testes de unidade, 20% de testes de integração e 10% de testes de sistema. Tal abordagem visa equilibrar a abrangência e a eficiência dos testes, garantindo uma base sólida de verificação automatizada com custos e esforços controlados.

Graficamente:

Figura 1 – Pirâmide de Testes



Fonte: Valente (2022)

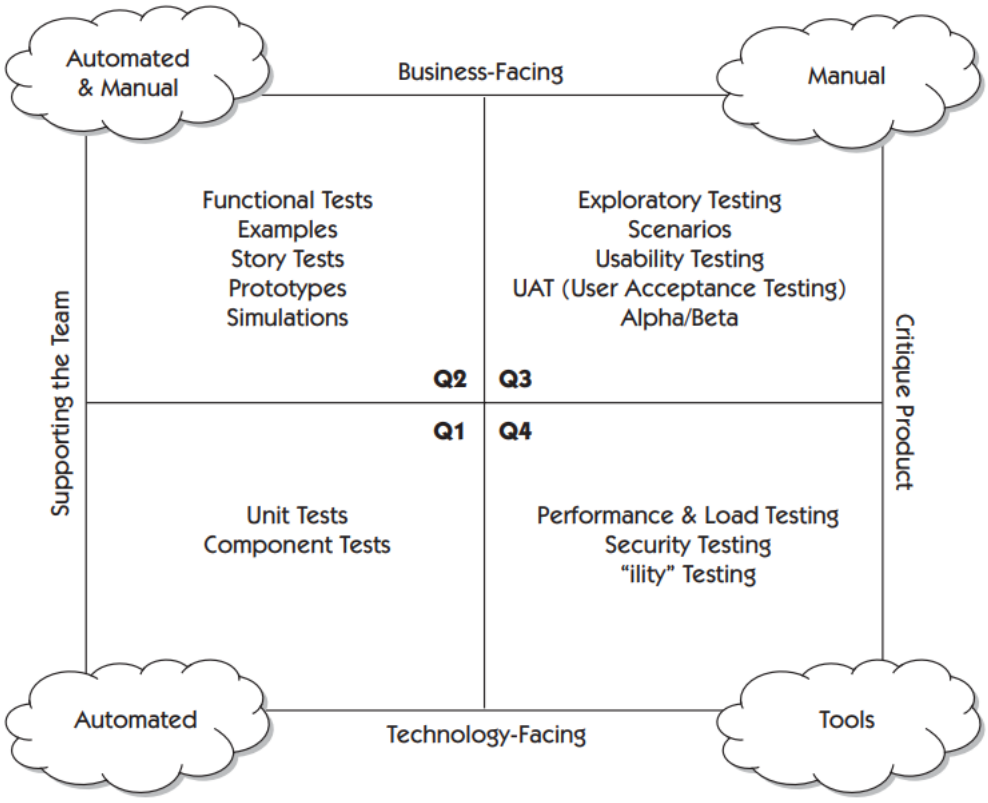
Por outro lado, os *Quadrantes Ágeis de Testes*, desenvolvidos por Brian Marick em Marick (2003) e popularizados por Lisa Crispin e Janet Gregory em Crispin e Gregory (2009), oferecem uma estrutura abrangente para categorizar os diferentes tipos de testes necessários em projetos ágeis. A Figura 2 demonstra os quadrantes ágeis que consistem em dois eixos principais: testes voltados para o negócio *versus* testes voltados para a tecnologia, e testes que suportam o desenvolvimento versus testes que validam o produto. Essa matriz resulta em quatro quadrantes distintos, cada um com foco específico:

- **Quadrante 1**: Testes voltados para a tecnologia que suportam o desenvolvimento, como testes de unidade e de componentes. Esses testes são geralmente automatizados e ajudam a garantir que o código funcione conforme o esperado.

- **Quadrante 2**: Testes voltados para o negócio que suportam o desenvolvimento, incluindo testes de aceitação e de protótipos. Esses testes asseguram que o sistema atenda aos requisitos do cliente e são frequentemente automatizados.
- **Quadrante 3**: Testes voltados para o negócio que validam o produto, como testes exploratórios e de usabilidade. Esses testes são geralmente manuais e ajudam a identificar problemas que afetam a experiência do usuário.
- **Quadrante 4**: Testes voltados para a tecnologia que validam o produto, incluindo testes de desempenho, carga e segurança. Esses testes garantem que o sistema seja robusto e confiável sob diversas condições.

Essa estrutura auxilia as equipes ágeis a planejarem e executarem atividades de teste de forma abrangente, garantindo a entrega de valor ao cliente e a qualidade do produto final.

Figura 2 – Quadrantes ágeis de testes



Fonte: Crispin e Gregory (2009)

3 Desenvolvimento

Este capítulo detalha os principais aspectos relacionados ao desenvolvimento do projeto, incluindo as tecnologias adotadas, a estrutura e funcionamento dos microsserviços implementados, bem como o desenho e a implementação das pipelines de testes automatizados. A seguir, cada um desses elementos é explorado de forma a evidenciar as decisões técnicas tomadas e sua contribuição para os objetivos propostos.

3.1 Principais tecnologias utilizadas

Aqui apresentam-se as principais ferramentas e tecnologias utilizadas durante a construção do objeto de estudo desta monografia. Em grande medida, optou-se por fazer uso de instrumentos já difundidos na comunidade, com sólida aceitação de pares, facilidade de utilização e, claro, um apelo estético alinhado às tendências modernas de desenvolvimento.

3.1.1 Github e Github actions

A escolha do *GitHub* como plataforma de hospedagem de código-fonte e do *GitHub Actions* como ferramenta de integração e entrega contínua (CI/CD) se deu por diversos fatores práticos e contextuais. O *GitHub* é uma das plataformas mais amplamente utilizadas pela comunidade de desenvolvimento de software, oferecendo uma interface moderna, intuitiva e com forte apelo visual, o que facilita a navegação e o acompanhamento das atividades dos projetos. Além disso, sua popularidade permite maior visibilidade e colaboração entre desenvolvedores, incluindo colegas e contatos da própria comunidade acadêmica, o que favorece o compartilhamento de conhecimento e boas práticas. O *GitHub Actions*, por sua vez, integra-se de forma nativa à plataforma, oferecendo uma solução robusta e gratuita para repositórios públicos, com suporte a pipelines automatizadas, testes e *deploys*. Essa integração direta elimina a necessidade de configurar ferramentas externas, simplificando o processo de automação de testes no contexto deste trabalho.

O trecho apresentado na Listagem 3.1 ilustra parte de um arquivo de configuração de um workflow do GitHub Actions, escrito em YAML. Ele demonstra etapas fundamentais da automação adotada neste projeto, como o uso de cache para otimizar a instalação de dependências com *pnpm*, execução de testes de unidade com cobertura, e testes end-to-end utilizando o framework *Jest*.

O uso de `actions/cache@v2` permite armazenar em cache os pacotes previamente baixados, reduzindo o tempo necessário para reinstalar dependências a cada nova execução da pipeline. Em seguida, a etapa `pnpm install` garante a reconstrução do ambiente

de execução da aplicação. As etapas posteriores — `pnpm test`, `pnpm test:cov` e `pnpm test:e2e` — executam, respectivamente, os testes de unidade, a análise de cobertura de código e os testes end-to-end, alinhando-se com as práticas recomendadas de CI/CD.

Esse fluxo automatizado garante que, a cada *push* ou *pull request*, o código seja validado de forma rigorosa, promovendo maior confiança nas alterações realizadas. A integração dos testes ao pipeline reflete um compromisso com a qualidade e a estabilidade da aplicação, conforme proposto na arquitetura de microsserviços explorada neste trabalho.

```
      - name: Cache pnpm store
        uses: actions/cache@v2
        with:
          path: ~/.local/share/pnpm/store
          key: ${{ runner.os }}-pnpm-store-${{ hashFiles('**/pnpm
             -lock.yaml') }}
          restore-keys: |
            ${{ runner.os }}-pnpm-store-

    - name: Install dependencies
      run: pnpm install

    - name: Run Jest Unit tests
      run: pnpm test

    - name: Run Jest test coverage
      run: pnpm test:cov

    - name: Run Jest e2e tests
      run: pnpm test:e2e
```

Listing 3.1 – Exemplo de arquivo de configuração de uma Action

3.1.2 NestJs

A escolha do *NestJS* como *framework* de desenvolvimento neste trabalho fundamenta-se em suas características que favorecem a construção de microsserviços escaláveis e testáveis.

Frameworks são estruturas de software que fornecem uma base padrão para o desenvolvimento de aplicações, promovendo a reutilização de código e facilitando a manutenção. O *NestJS*, um *framework* progressivo para Node.js, utiliza o TypeScript — uma linguagem que adiciona tipagem estática ao *JavaScript* — para oferecer uma arquitetura

modular e orientada a decoradores. Essa abordagem facilita a organização do código e a implementação de testes automatizados, essenciais para pipelines de CI/CD eficientes.

3.1.3 PostgreSQL

Sistema Gerenciador de Banco de Dados (SGBD) são softwares que permitem a criação, manipulação e administração de bancos de dados. O PostgreSQL é um SGBD relacional de código aberto amplamente reconhecido por sua robustez, desempenho e conformidade com padrões Structured Query Language (SQL). Sua compatibilidade com ferramentas de contêineres, como *Test containers*, permite a simulação de ambientes de produção durante os testes de integração, assegurando maior fidelidade nos resultados.

A combinação do *NestJS* e do PostgreSQL proporciona uma base sólida para o desenvolvimento e a automação de testes em arquiteturas de microsserviços, alinhando-se aos objetivos deste trabalho.

3.2 Os microsserviços

No contexto das tecnologias utilizadas, os dois microsserviços implementados neste trabalho — *Task* e *Todo* — não foram projetados para resolver problemas do mundo real, mas sim para atuarem como Proof of concept (PoC). Ambos são comumente adotados em uma empresa específica do ramo de tecnologia como exemplos introdutórios à arquitetura de microsserviços, por sua simplicidade e estrutura didática. A seguir, apresenta-se a estrutura adotada para cada um deles:

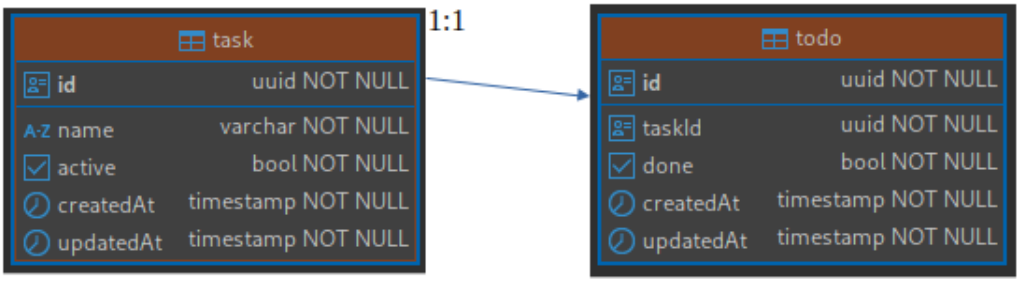
O microsserviço *Task* é responsável por representar uma tarefa genérica e contém funcionalidades básicas de Create, Read, Update, Delete (CRUD). Sua estrutura inclui uma entidade `Task`, com atributos como `id`, `name`, `active`, `createdAt` e `updatedAt`. As operações são expostas por meio de uma *API REST*, construída com NestJS, e testadas principalmente com a biblioteca *Jest*. O foco deste serviço está na validação de testes de unidade e de integração.

Já o microsserviço *Todo* é responsável por associar uma ou mais tarefas (`taskId`) a um estado de conclusão (`done`). Ele complementa o serviço *Task* ao representar a conclusão prática de uma tarefa, e é utilizado para exercitar o consumo de funcionalidades de outro serviço via comunicação. O serviço *Todo* também foi implementado com NestJS, utilizando PostgreSQL como banco de dados e suportando testes de integração com acesso a dados persistidos.

Ambos os microsserviços foram desenvolvidos com o propósito de simular cenários reais de desenvolvimento distribuído, permitindo a construção de pipelines de testes automatizados em conformidade com práticas modernas de CI/CD.

A Figura 3 ilustra o diagrama Entidade-Relacionamento que descreve as estruturas de dados dos dois microsserviços, destacando suas entidades principais e as associações entre elas.

Figura 3 – Diagramas ER de ambos os microsserviços



Fonte: O autor.

3.2.1 Testes Automatizados

Para validar o comportamento dos microsserviços desenvolvidos, foram implementados testes automatizados com foco principal em testes de unidade. Esses testes têm como objetivo verificar, de forma isolada, se as funções, métodos e componentes individuais de cada serviço funcionam corretamente, sem depender de fatores externos como banco de dados ou comunicações com outros serviços.

Em arquiteturas de microsserviços, a complexidade inerente à comunicação entre serviços independentes exige uma adaptação das estratégias de testes tradicionais. A *Pirâmide de Testes* continua relevante, mas é necessário incorporar testes de contrato e testes de componentes para garantir a integridade das interações entre serviços. Esses testes adicionais ajudam a detectar problemas de integração e comunicação que não seriam identificados apenas com testes de unidade ou de interface, como discutido por Fowler (2014b).

Além disso, os *Quadrantes Ágeis de Testes* proporcionam uma visão holística das necessidades de teste em ambientes de microsserviços, permitindo que as equipes equilibrem adequadamente os testes técnicos e de negócio, bem como os testes automatizados e manuais. Como exposto por Crispin e Gregory (2009), essa abordagem é fundamental para lidar com os desafios de qualidade e confiabilidade em sistemas distribuídos.

A biblioteca *Jest* foi utilizada como principal ferramenta para a criação e execução dos testes. Trata-se de um *framework* de testes robusto, amplamente utilizado no ecossistema JavaScript/TypeScript, que oferece funcionalidades como *mocks*, *spies*, *assertions* e cobertura de código.

Nos serviços *Task* e *Todo*, os testes de unidade foram aplicados principalmente às camadas de serviço (`Service`) e controle (`Controller`). Na camada de serviço, foram validadas regras de negócio, como a criação, listagem e atualização de entidades. Já na camada de controle, foram testadas as respostas às requisições HyperText Transfer Protocol (HTTP) simuladas, garantindo que os controladores estivessem corretamente conectados aos serviços e processassem adequadamente os dados recebidos e retornados.

3.2.1.1 Design dos Testes Automatizados

O design dos testes automatizados implementados para o microsserviço *Task* segue uma estrutura que prioriza clareza, isolamento e alinhamento com a arquitetura do NestJS. Foram desenvolvidos testes para as camadas de serviço (*TaskService*) e controlador (*TaskController*), contemplando os principais fluxos de criação, leitura, atualização e remoção de tarefas.

Os testes do *controller* focam na verificação das chamadas feitas aos métodos do serviço, validando se a entrada dos dados (Data Transfer Object (DTO)) resulta em chamadas esperadas aos métodos do *service layer*. Para isso, utiliza-se *spyOn* do Jest para simular o comportamento do serviço, evitando o acoplamento a lógica de negócio ou acesso a dados. Cada operação — como *create*, *getAll*, *getOne*, *update* e *delete* — é testada com entradas representativas e saídas esperadas, garantindo que o controlador está corretamente roteando e respondendo às requisições.

Já os testes da camada de serviço são mais centrados na lógica de persistência e interação com o repositório do TypeORM. Esses testes simulam o comportamento do repositório por meio de objetos *mockados*, permitindo validar o fluxo completo de transformação dos dados (via DTOs), chamada aos métodos como *create*, *find*, *save*, *remove* e validação de existência prévia de entidades. O uso de mocks garante que o teste seja determinístico, rápido e independente de infraestrutura externa (como um banco de dados real).

Uma característica de destaque no design adotado é a separação explícita entre testes de unidade (para o *service*) e testes de integração leve (para o *controller* com *mocks* do *service*), alinhando-se à base da Pirâmide de Testes. Essa estratégia torna os testes rápidos, fáceis de manter e sensíveis a mudanças relevantes no comportamento da aplicação, mas não a detalhes da infraestrutura subjacente. Além disso, a estrutura modularizada e repetível dos testes permite sua expansão futura com testes end-to-end ou testes com banco de dados em memória, como o *SQLite* ou *Testcontainers*.

Dependências como repositórios do TypeORM foram simuladas por meio de *mocks*, permitindo que os testes se mantivessem rápidos e independentes. A execução dos testes foi integrada à pipeline de CI/CD via GitHub Actions, permitindo que, a cada nova alteração no código, os testes fossem executados automaticamente, garantindo maior confiabilidade

e detectando eventuais regressões.

O uso de testes de unidade de *service* e de *controller* com *Jest* contribuiu diretamente para a qualidade do software, proporcionando segurança na refatoração do código e estabelecendo uma base sólida para a evolução dos microsserviços ao longo do desenvolvimento.

A seguir, a Listagem 3.2 apresenta um trecho de teste automatizado da camada de serviço do microsserviço `Task`, escrito com a biblioteca Jest. Este exemplo ilustra o uso explícito da estrutura Arrange, Act, Assert (AAA), um padrão amplamente utilizado para tornar os testes mais legíveis e organizados. Primeiramente, os dados e comportamentos esperados são configurados (*Arrange*), em seguida o método a ser testado é executado (*Act*), e por fim são feitas as verificações das expectativas (*Assert*). Essa abordagem reforça a clareza e facilita a manutenção dos testes à medida que o sistema evolui.

```
1  it('should create a task', async () => {
2    // Arrange
3    const createTaskDto = { name: 'Test Task', active: true };
4    const entity = { id: '1', ...createTaskDto, createdAt: new Date
       (), updatedAt: new Date() };
5    jest.spyOn(repository, 'create').mockReturnValue(entity);
6    jest.spyOn(repository, 'save').mockResolvedValue(entity);
7
8    // Act
9    const createdTask = await service.create(createTaskDto);
10
11   // Assert
12   expect(repository.create).toHaveBeenCalledWith(createTaskDto);
13   expect(repository.save).toHaveBeenCalledWith(entity);
14   expect(createdTask).toEqual(entity);
15 });
```

Listing 3.2 – Exemplo de teste com o padrão AAA

3.2.1.2 Testes de Integração

Os testes de integração têm como objetivo verificar se diferentes módulos ou componentes de um sistema funcionam corretamente quando combinados. Conforme definição de Valente (2022), esse tipo de teste concentra-se nas interações entre as unidades do software — como chamadas entre classes, componentes ou mesmo serviços distintos — a fim de assegurar que o sistema funciona como esperado além do escopo unitário.

Em comparação com os testes de unidade, os testes de integração operam com um maior nível de complexidade e realismo. Eles normalmente acessam bancos de dados,

sistemas de arquivos ou outras dependências externas simuladas ou reais, validando o comportamento do sistema em condições mais próximas das que serão encontradas em produção.

O código a seguir exemplifica um teste de integração para o serviço de tarefas (TasksService), utilizando o banco de dados em memória SQLite, configurado via *TypeORM*. Com esse tipo de abordagem, é possível executar as operações reais de persistência, como inserção, leitura, atualização e remoção de dados, mas sem depender de uma infraestrutura externa permanente. Isso garante maior velocidade e isolamento nos testes, mantendo a fidelidade ao comportamento da aplicação em execução.

```
describe('TasksService (integration)', () => {
  let service: TasksService;

  beforeAll(async () => {
    const module: TestingModule = await Test.createTestingModule
       ({
      imports: [
        TypeOrmModule.forRoot({
          type: 'sqlite',
          database: ':memory:',
          dropSchema: true,
          entities: [TasksEntity],
          synchronize: true,
        }),
        TypeOrmModule.forFeature([TasksEntity]),
      ],
      providers: [TasksService, TasksException],
    }).compile();

    service = module.get<TasksService>(TasksService);
  });

  it('should create and find a task', async () => {
    const dto = { name: 'Integration Task', active: true };
    const created = await service.create(dto as any);
    const found = await service.findOne(created.id);

    expect(created).toHaveProperty('id');
    expect(found.name).toBe('Integration Task');
    expect(found.active).toBe(true);
  });
});
```

Listing 3.3 – Teste de integração com banco em memória

Esse teste segue o padrão AAA (*Arrange, Act, Assert*), organizando a execução em três etapas claras: preparação do cenário com os dados de entrada, execução da funcionalidade a ser testada e verificação do resultado obtido. A adoção desse padrão reforça a legibilidade, a manutenção e a confiabilidade dos testes de integração desenvolvidos.

3.2.1.3 Testes End-to-End (E2E)

Neste trabalho, optou-se por não desenvolver uma interface gráfica (front-end), focando exclusivamente no backend dos microsserviços. Portanto, os testes end-to-end realizados concentram-se no consumo direto da *API REST* por meio de requisições HTTP simuladas. No entanto, em um cenário completo que incluísse uma aplicação front-end, seria possível aplicar testes E2E mais robustos envolvendo a interação com elementos da interface do usuário. Nesse contexto, ferramentas como o *Puppeteer*, uma biblioteca para controle de navegadores *headless* baseada no Chrome, poderiam ser integradas ao *Jest* para a realização de testes automatizados que simulam cliques, preenchimento de formulários e navegação em páginas. Esse tipo de abordagem permite validar a experiência do usuário de ponta a ponta, verificando não apenas a resposta da API, mas também a renderização e o comportamento da interface. A seguir, um exemplo ilustrativo de como seria estruturado um teste E2E utilizando Puppeteer:

```
1  const puppeteer = require('puppeteer');
2
3  describe('E2E Test with Puppeteer', () => {
4    it('should navigate and create a task', async () => {
5      const browser = await puppeteer.launch({ headless: true });
6      const page = await browser.newPage();
7
8      await page.goto('http://localhost:3000');
9      await page.type('#task-name', 'Nova tarefa');
10     await page.click('#submit-task');
11
12     const taskText = await page.$eval('.task-item', el => el.
           textContent);
13     expect(taskText).toContain('Nova tarefa');
14
15     await browser.close();
16   });
17 });
```

Listing 3.4 – Exemplo de teste E2E com Puppeteer

Os testes end-to-end (E2E), também chamados de testes de sistema, têm como propósito verificar o funcionamento completo de uma aplicação, simulando o comportamento de um usuário real ao interagir com o sistema. Segundo Valente (2022), esse tipo de teste é caracterizado por validar a aplicação como um todo, considerando todas as suas camadas e componentes integrados — desde a interface de entrada até o acesso a dados e retorno das respostas.

Diferentemente dos testes de unidade, que exercitam partes isoladas do código, os testes E2E são mais custosos e lentos, uma vez que exigem a aplicação em execução e envolvem múltiplos elementos do sistema. Ainda assim, são essenciais para garantir que os principais fluxos de uso do sistema estão funcionando conforme esperado do ponto de vista do usuário final.

No contexto de uma *API* REST, os testes E2E realizam requisições HTTP reais contra os endpoints da aplicação em execução, verificando se as respostas retornadas condizem com os comportamentos esperados. Esses testes proporcionam maior confiança na estabilidade e coerência do sistema como um todo, especialmente em ambientes com múltiplos microsserviços.

Por fim, os autores ressaltam que apesar da utilização de integração contínua (*Continuous Integration* - CI), por meio do *GitHub Actions* para automação de testes, não foi implementado neste trabalho um fluxo completo de entrega contínua (*Continuous Deployment* - CD). Isso se deve principalmente ao escopo acadêmico do projeto, à ausência de uma aplicação front-end e à não necessidade de promover mudanças automáticas para um ambiente de produção. Em contextos reais de desenvolvimento, a prática de CD é importante para garantir que novas versões aprovadas nos testes sejam disponibilizadas de forma segura e automática. Ainda assim, a estrutura do projeto foi concebida de forma a possibilitar, futuramente, a adoção de pipelines de CD, integrando ambientes de *staging* e produção com validações automáticas antes da liberação de versões.

4 Resultados

Este capítulo apresenta os resultados obtidos com a implementação e execução dos testes automatizados para o microsserviço *task-ms*, que é o foco desta análise. Embora o sistema compreenda dois microsserviços, a atenção aqui se concentra na validação do *task-ms* por meio da configuração de uma pipeline de integração contínua (CI) via *GitHub Actions*. Detalhamos a abrangência dos testes unitários, de integração e *end-to-end* aplicados a ele, a metodologia de organização dos testes (padrão AAA), e uma análise aprofundada da cobertura de código. Serão discutidos os sucessos alcançados, como a eficiência na execução e a aprovação de todos os testes para *task-ms*, e também as lacunas identificadas, especialmente no que tange à cobertura de *branches*, propondo melhorias para a resiliência do sistema.

4.0.1 Resultados da Execução dos Testes Automatizados

Para assegurar a qualidade contínua do microsserviço `task-ms`, implementou-se uma pipeline de integração contínua (CI) utilizando o *GitHub Actions*. Essa automação é acionada em resposta a modificações no repositório, especificamente na *branch* principal (`main`) e em `pull requests`, executando testes unitários, de integração e *end-to-end*.

A listagem 4.1 ilustra um fragmento do arquivo de configuração da *pipeline* [1], destacando as principais etapas da automação.

```
1  name: Run Jest Unit/Integration tests
2  run: pnpm test
3
4  name: Run Jest e2e tests
5  run: pnpm test:e2e
6
7  name: Run Jest test coverage
8  run: pnpm test:cov
9
10 name: Upload test results
11 uses: actions/upload-artifact@v4
12 with:
13 name: task-ms-test-results
14 path: |
15 coverage
16 test-results
```

[1] `.github/workflows/task-microservice.yml`

Listing 4.1 – Trecho do workflow do GitHub Actions para o microsserviço task-ms

4.0.1.1 Classificação dos Testes Executados

Os testes foram organizados segundo os seguintes níveis de granularidade:

- De Unidade: testam funções isoladas no `TasksService`, sem dependências externas.
- De integração: envolvem interações entre controller e service, com simulação de camadas como banco de dados.
- End-to-end (E2E): validam o comportamento completo da aplicação por meio de requisições HTTP, simulando o uso real do microsserviço.

Essa organização permite avaliar a aplicação sob diferentes perspectivas, cobrindo desde a lógica interna até sua interface exposta.

Os resultados da cobertura de testes foram exportados em múltiplos formatos, incluindo `lcov.info`, `coverage-final.json` e um relatório HyperText Markup Language (HTML) completo [2]. Este relatório oferece uma análise detalhada da cobertura por funções, linhas e arquivos, apresentando:

- Relatórios por arquivo-fonte, com destaques visuais para linhas cobertas e não cobertas.
- Divisão por camadas da aplicação (*controllers*, *services*, DTOs, entidades).
- Indicadores quantitativos de cobertura por linha, função e ramo lógico.

Adicionalmente, os resultados dos testes automatizados (no formato JUnit [3] foram persistidos como artefato da execução do *GitHub Action*. Essa abordagem permite integração com ferramentas de análise de código, como *SonarQube* ou *Codecov*. A estrutura dos testes adota o AAA (*Arrange, Act, Assert*), promovendo a organização e facilitando a manutenção do código de testes.

4.0.1.2 Resultados por Suíte de Testes

4.0.1.2.1 `TasksService`.

A suíte de testes `TasksService` apresentou um tempo total de execução de aproximadamente 4,4 segundos e cobriu seis cenários críticos:

[2] `coverage/lcov-report/index.html`
[3] `test-results/junit.json`

- `should be defined`
- `create should create a task`
- `findAll should return an array of tasks`
- `findOne should return a single task`
- `update should update a task`
- `remove should delete a task`

A definição de cenários críticos baseia-se, principalmente, na alta frequência de interação com o usuário. Por serem *endpoints* que serão intensamente utilizados, sua funcionalidade é considerada crítica, justificando a necessidade de testes rigorosos para garantir a confiabilidade do sistema. Todos os testes foram aprovados, validando a integridade da lógica de negócio e do acesso aos dados. O relatório de cobertura para este serviço revelou os seguintes índices: **Statements: 89,65%**, **Functions: 100%**, **Lines: 88,88%**, mas **Branches: 0%**.

A análise do relatório de cobertura identificou lacunas importantes:

- A linha 20 (`if (existentTask)`) do método `validatesName` não foi executada, indicando a ausência de testes para o cenário de nomes de tarefas duplicados.
- Similarmente, a linha 33 (`if (!especificTask)`) do método `validatesTaskById` também não foi testada, o que aponta para a falta de cobertura para IDs de tarefas inválidos.

4.0.1.2.2 `TasksController`.

A suíte `TasksController` executou seis testes focados na validação dos *endpoints* HTTP da API REST, com um tempo total de aproximadamente 0,607 segundos. Todos os testes foram aprovados. Os testes feitos aqui são os mesmos da subseção de testes da TasksService, uma vez que os *controllers* definem *endpoints* que, por sua vez, enviaram requisições e receberão respostas das suas respectivas implementações no *service*.

4.0.1.2.3 `AppController`.

A suíte `AppController` executou um único teste com sucesso em 0,212 segundos:

- `root should return "Hello World!"`

Este teste fundamental confirma a operacionalidade da aplicação e sua capacidade de responder à raiz.

4.0.1.3 Resumo Geral e Discussão

Um total de 13 testes automatizados foram executados com 100% de aprovação e um tempo total inferior a 6 segundos. Essa performance notável demonstra a eficiência da suíte de testes, garantindo execuções rápidas e, consequentemente, elevando a confiança nas entregas contínuas.

Apesar dos índices satisfatórios de cobertura para linhas e funções, a ausência completa de cobertura de *branches* é um ponto crítico que ressalta a necessidade de incorporar testes negativos e tratamento de exceções. Especificamente:

- Recomenda-se o desenvolvimento de testes dedicados para cenários como tentativa de criação de tarefa com nome duplicado e busca por tarefa com ID inexistente.
- É fundamental monitorar a cobertura de *branches* para assegurar a resiliência do sistema frente a entradas inválidas ou situações de erro inesperadas.

Para ilustrar de forma mais concreta os dados obtidos, a seguir são apresentados dois recortes visuais do relatório gerado. As Figuras 4 e 5 mostram o resumo da cobertura de testes, com destaque para os percentuais globais por arquivos e métricas como linhas, funções e ramos lógicos. Já a Figura 6 apresenta a cobertura detalhada de um dos arquivos mais relevantes do microsserviço, evidenciando visualmente as linhas executadas e não executadas durante os testes.

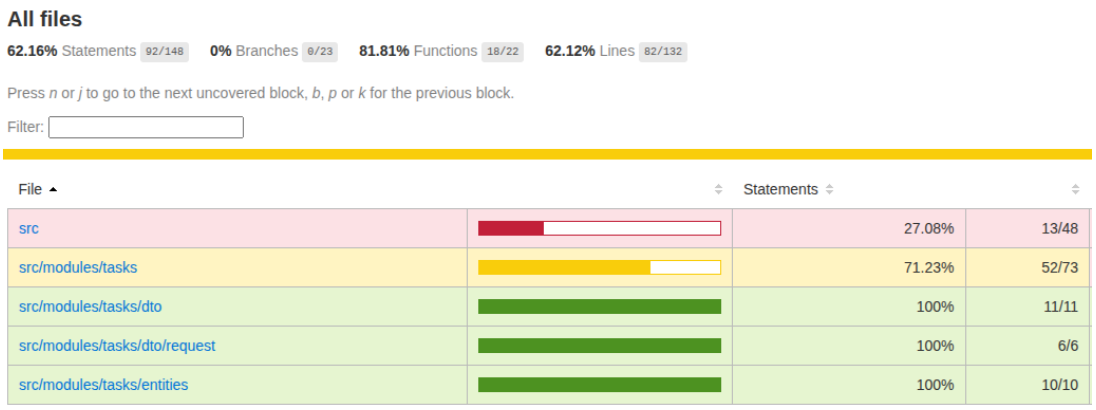
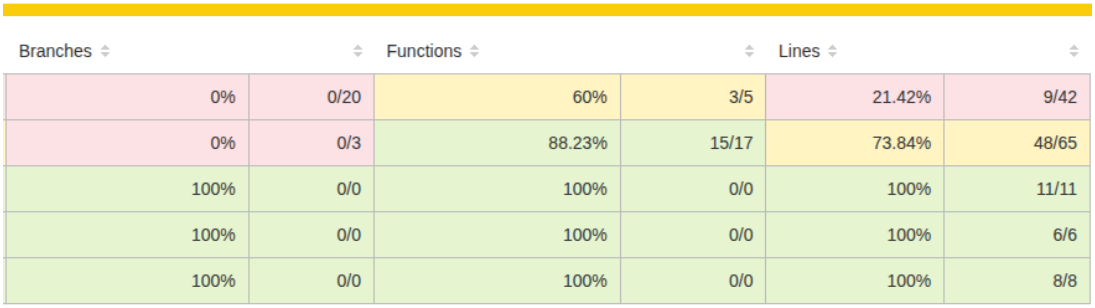


Figura 4 – Resumo da cobertura de testes do microsserviço `task-ms`, com destaque para arquivos, linhas e funções, parte 1.



Branches		Functions		Lines	
0%	0/20	60%	3/5	21.42%	9/42
0%	0/3	88.23%	15/17	73.84%	48/65
100%	0/0	100%	0/0	100%	11/11
100%	0/0	100%	0/0	100%	6/6
100%	0/0	100%	0/0	100%	8/8

Figura 5 – Resumo da cobertura de testes do microsserviço `task-ms`, com destaque para arquivos, linhas e funções, parte 2.

Figura 6 – Cobertura detalhada de um arquivo do microsserviço `task-ms`, o *src/modules/tasks tasks.service.ts*

O alto grau de isolamento entre os testes de serviço e controlador reflete a aplicação de boas práticas no design de testes automatizados. A integração contínua na *pipeline* do *GitHub Actions*, por sua vez, aprimora a rastreabilidade e a integridade do sistema. O registro da execução em `2025-07-06T22:22:18` atesta um estado funcional e estável do código naquele momento.

No contexto atual, a *pipeline* está configurada predominantemente para integração

contínua (CI), sem realizar *deploy* automático (CD). Contudo, o acoplamento dos testes à CI representa um avanço significativo na garantia da qualidade do código e pode ser expandido futuramente para incorporar estratégias de entrega contínua mais robustas.

5 Considerações finais

Este trabalho apresentou a construção de um ambiente de testes automatizados aplicados a microsserviços, com ênfase especialmente em práticas de Integração Contínua (CI) por meio do uso do *GitHub Actions*. Foram exploradas ferramentas consolidadas como o NestJS, Jest e pnpm — todas de código aberto — compondo um cenário realista e alinhado às exigências modernas de desenvolvimento de software distribuído.

Ao longo do projeto, foram desenvolvidos dois microsserviços de prova de conceito, `task-ms` e `todo-ms`, sobre os quais se implementaram testes unitários, de integração e, parcialmente, de testes *end-to-end*. A separação entre os tipos de teste seguiu a pirâmide apresentada e discutida em 2, respeitando as boas práticas em qualidade de software. Os resultados práticos demonstraram que é possível obter uma cobertura significativa com ferramentas simples e acessíveis, favorecendo a rastreabilidade, estabilidade e manutenibilidade do sistema.

Apesar do foco ter recaído majoritariamente sobre a integração contínua, algumas limitações foram assumidas quanto à ausência de um processo completo de entrega contínua (CD), o que impossibilitou a realização de *deploys* automatizados. No entanto, foram descritas abordagens que poderiam ser adotadas futuramente, como a utilização de ambientes de homologação e estratégias de *rollout* do tipo *Blue-Green* ou *Canary*.

Como trabalho futuro, sugere-se a implementação de um *frontend* acoplado aos microsserviços existentes, permitindo a elaboração de testes E2E mais completos, com uso de ferramentas como *Puppeteer*. Além disso, propõe-se a inclusão de ferramentas de análise estática de código na pipeline (como ESLint e SonarQube) e a configuração de um ambiente de CD completo, ampliando o escopo do ciclo de vida DevOps no contexto estudado. Tais aprimoramentos dariam continuidade à proposta deste trabalho, elevando o grau de automação e controle de qualidade na entrega de sistemas baseados em microsserviços.

No fim das contas, não é esse o verdadeiro propósito de testar, automatizar e integrar: continuar perguntando se está tudo funcionando, e por quê?

Referências

Amazon Web Services. *Blue/Green Deployments*. 2020. <https://wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.concept.canary-deployment.en.html>. Acesso em: 18 maio 2025. Citado na página 15.

Atlassian. *Trunk-based development*. 2021. <https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development>. Acesso em: 18 maio 2025. Citado na página 15.

CHEN, L. Microservices: Architecting for continuous delivery and devops. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. [S.l.: s.n.], 2018. p. 39–397. Citado 2 vezes nas páginas 11 e 14.

COHN, M. *Succeeding with Agile: Software Development Using Scrum*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321579364. Citado na página 15.

CRISPIN, L.; GREGORY, J. *Agile Testing: A Practical Guide for Testers and Agile Teams*. 2009. <https://scrummalaysia.com/media/attachments/2020/03/18/agile_testing_-_a_practical_guide_for_testers_and_agile_teams.pdf>. Acesso em: 25 maio 2025. Citado 3 vezes nas páginas 16, 17 e 21.

FOWLER, M. *Canary Release*. 2014. <https://martinfowler.com/bliki/CanaryRelease.html>. Acesso em: 18 maio 2025. Citado na página 15.

FOWLER, M. *Testing Strategies in a Microservice Architecture*. 2014. <https://martinfowler.com/articles/microservice-testing/>. Acesso em: 25 maio 2025. Citado na página 21.

FOWLER, M. *The Practical Test Pyramid*. 2018. <https://martinfowler.com/articles/practical-test-pyramid.html>. Acesso em: 25 maio 2025. Citado na página 15.

FOWLER, S. J. *Microsserviços Prontos Para a Produção: Construindo Sistemas Padronizados em uma Organização de Engenharia de Software*. São Paulo, Brasil: Novatec Editora, 2017. Citado 2 vezes nas páginas 11 e 14.

HUANG, D. et al. *Rethinking the Influence of Source Code on Test Case Generation*. 2024. Disponível em: <https://arxiv.org/abs/2409.09464>. Citado na página 14.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2010. (Addison-Wesley Signature Series (Fowler)). Citado na página 14.

LEITE, L. et al. A survey of devops concepts and challenges. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 52, n. 6, p. 1–35, nov. 2019. ISSN 1557-7341. Disponível em: <http://dx.doi.org/10.1145/3359981>. Citado 2 vezes nas páginas 11 e 14.

MARICK, B. *Agile Testing Project, Entry 2*. 2003. <http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2>. Acesso em: 14 jul. 2025. Citado na página 16.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. 1st. ed. Belo Horizonte, Brasil: Editora Independente, 2022. Citado 7 vezes nas páginas 11, 12, 14, 15, 16, 23 e 26.

WESSEL, M. et al. Github actions: The impact on the pull request process. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 28, n. 6, set. 2023. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-023-10369-w>. Citado na página 11.

WHITTAKER, J. A.; ARBON, J.; CAROLLO, J. *How Google Tests Software*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 0321803027. Citado na página 16.

Apêndices

APÊNDICE A – Arquivo .yml de uma Github Action

O trecho a seguir apresenta o conteúdo completo do arquivo de configuração do GitHub Actions utilizado para automatizar a execução dos testes no microsserviço `task-ms`. Esse *workflow* define a infraestrutura básica de integração contínua (CI) do projeto, englobando:

- configuração do ambiente com `Node.js` e `pnpm`;
- instalação de dependências;
- execução dos testes unitários, de integração e *end-to-end* com `Jest`;
- geração e empacotamento dos resultados de testes e cobertura.

Embora o bloco responsável pela análise estática de código com `ESLint` esteja temporariamente comentado, sua inclusão é recomendada em versões futuras, pois contribui para a detecção precoce de falhas e inconsistências no estilo do código.

A inclusão deste apêndice visa fornecer uma visão transparente e reprodutível da automação empregada no projeto, reforçando o caráter aberto e técnico da solução. Além disso, este *workflow* pode ser expandido futuramente com etapas de **entrega contínua (CD)**, como *builds* e *deploys* automatizados para ambientes de homologação ou produção, alinhando-se às boas práticas de DevOps e evolução constante da infraestrutura.

```yaml
name: task-microservice

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  task-ms:
    name: task-microservice
    runs-on: ubuntu-latest

    steps:
```

```yaml
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install pnpm
        run: npm install -g pnpm

      - name: Cache pnpm store
        uses: actions/cache@v4
        with:
          path: ~/.local/share/pnpm/store
          key: ${{ runner.os }}-pnpm-store-taskms-${{ hashFiles('
            task-ms/pnpm-lock.yaml') }}
          restore-keys: |
            ${{ runner.os }}-pnpm-store-taskms-

      - name: Install dependencies
        run: pnpm install

      # - name: Run ESLint
      #   run: pnpm lint

      - name: Run Jest Unit/Integration tests
        run: pnpm test

      - name: Run Jest e2e tests
        run: pnpm test:e2e

      - name: Run Jest test coverage
        run: pnpm test:cov

      - name: Upload test results
        uses: actions/upload-artifact@v4
        with:
          name: task-ms-test-results
          path: |
            coverage
            test-results
```

Listing A.1 – Workflow completo do GitHub Actions para o microsserviço `task-ms`