RECA
REVISTA ELETRÔNICA DE COMPUTAÇÃO APLICADA
Vol. 6 Nro. 2 2025

Departamento de Computação
Uni-FACEF

A IMPORTÂNCIA DOS TESTES AUTOMATIZADOS NO CICLO DE VIDA DE UM SOFTWARE

João Paulo Faleiros dos Santos
Graduando em Sistemas de Informação – Uni-FACEF
joaofaleirospaulo@gmail.com

Prof. Débora Pelicano Diniz
Docente do Departamento de Computação – Uni-FACEF
deboradiniz@facef.br

Resumo
A importância dos testes automatizados no ciclo de vida do software é cada vez mais evidente, pois contribuem para a qualidade, a redução de falhas e a eficiência no desenvolvimento. Neste estudo foi realizada uma comparação prática entre as ferramentas *Postman* e *Cypress*, aplicadas em três casos de teste de uma *API* de usuários: adição, busca e remoção inválida. Os experimentos indicaram que o *Postman* apresentou menores tempos médios de resposta (3,6 a 8,4 ms), sendo eficiente para validações rápidas de *APIs*. O *Cypress*, embora mais lento (16,4 a 24,8 ms), proporcionou maior detalhamento, geração de *logs* e suporte a fluxos mais complexos. A análise crítica demonstrou que as ferramentas são complementares: o *Postman* é ideal para verificações ágeis de *endpoints*, enquanto o *Cypress* se mostra mais robusto para testes *end-to-end* e integração contínua. Conclui-se que a adoção de testes automatizados, associada ao uso de ferramentas adequadas, fortalece a qualidade do software e reduz riscos ao longo do desenvolvimento.

Palavras-chave: Testes Automatizados. Qualidade de Software. Postman. Cypress. APIs REST.

Abstract
*The importance of automated testing in the software lifecycle is increasingly evident, as it contributes to quality, defect reduction, and development efficiency. In this study, a practical comparison was conducted between **Postman** and **Cypress**, applied to three test cases in a user API: addition, search, and invalid deletion. The experiments indicated that Postman achieved lower average response times (3.6 to 8.4 ms), proving efficient for quick API validations. Cypress, although slower (16.4 to 24.8 ms), provided greater detail, log generation, and support for more complex flows. The critical analysis demonstrated that the tools are complementary: Postman is best suited for agile endpoint checks, while Cypress proves more robust for end-to-end and continuous integration testing. It is concluded that the adoption of automated testing, combined with appropriate tools, strengthens software quality and reduces risks throughout development.*

***Keywords**: Automated Testing. Software Quality. Postman. Cypress. REST APIs.*

1 Introdução

Os testes automatizados desempenham um papel fundamental no desenvolvimento moderno de software, pois contribuem diretamente para a garantia da qualidade, redução de falhas e aumento da eficiência nos processos de entrega. Em um cenário cada vez mais dinâmico e competitivo, a automação de testes se tornou uma prática essencial para equipes que buscam desenvolver soluções escaláveis, confiáveis e com menor incidência de retrabalho.

A relevância deste tema reside na sua capacidade de otimizar o ciclo de vida do software, assegurando entregas contínuas com maior assertividade e menor custo operacional. A automatização também promove maior cobertura de testes, reduzindo a dependência de validações manuais e possibilitando respostas rápidas a mudanças no sistema, o que é especialmente importante em metodologias ágeis.

Diante desse contexto, o objetivo principal deste trabalho é realizar uma análise comparativa entre duas ferramentas amplamente utilizadas em testes automatizados: o ***Postman*** e o ***Cypress***. Ambas oferecem funcionalidades distintas que atendem a diferentes tipos de validação, desde testes de *APIs* até simulações de fluxos completos em aplicações web.

Para alcançar esse objetivo, este artigo foi estruturado da seguinte forma: na Seção 2 são apresentados os conceitos relacionados à Qualidade de Software; na Seção 3 discute-se a importância da automação de testes no ciclo de vida do desenvolvimento; na Seção 4 estão descritas as ferramentas utilizadas e seus principais recursos; na Seção 5, são apresentados os testes realizados, os resultados obtidos e a comparação crítica entre as ferramentas estudadas e, por fim, na Seção 6 estão apresentadas as conclusões do trabalho realizado.

2 Qualidade de Software

A qualidade de software é um aspecto essencial no desenvolvimento de sistemas, garantindo que as aplicações sejam confiáveis, eficientes e seguras. Segundo Santos (2019), a qualidade do software deve ser tratada como um fator estratégico, uma vez que influencia diretamente a experiência do usuário e a competitividade no mercado. Dessa forma, empresas que investem na melhoria contínua da qualidade de software conseguem reduzir custos operacionais e aumentar a satisfação dos clientes.

Um dos principais desafios para garantir a qualidade do software está na definição e no cumprimento de requisitos técnicos e funcionais. Conforme Camargo (2022), a ausência de um processo bem estruturado de especificação de requisitos pode comprometer a qualidade final do software, levando a retrabalho e a falhas inesperadas em produção. Portanto, a implementação de metodologias de desenvolvimento que priorizem a qualidade desde as primeiras etapas é essencial.

A automação de testes é um dos pilares da garantia de qualidade. De acordo com Ament (2023), o uso de testes automatizados melhora a confiabilidade dos sistemas, pois permite a execução contínua de verificações e reduz a possibilidade de erros humanos. Além disso, a automação possibilita uma maior cobertura de testes, tornando o processo mais eficiente e menos suscetível a falhas.

Outro fator determinante para a qualidade do software é a adoção de métricas e indicadores de desempenho. Conforme Anjo (2021), o monitoramento de métricas como tempo de resposta, taxa de erros e cobertura de código permite que as equipes identifiquem padrões de falhas e tomem decisões estratégicas para aprimorar a qualidade do software. O uso dessas métricas possibilita um controle mais preciso do desempenho das aplicações.

A implementação de padrões de qualidade é uma estratégia fundamental para garantir software de alto desempenho. Segundo Camargo (2022), empresas que seguem padrões internacionais, como ISO 25010 e CMMI, conseguem estabelecer processos mais eficazes para controle e medição da qualidade de software. Esses padrões garantem que a qualidade seja incorporada em todas as fases do desenvolvimento.

Na Tabela 1 estão apresentadas algumas das principais métricas utilizadas para avaliar a qualidade do software, considerando tudo o que foi pesquisado.

Complementarmente à análise quantitativa apresentada, realizou-se também uma análise qualitativa, com base nos princípios de Treude (2024), que destaca a relevância da interpretação contextual de dados em engenharia de software. Essa abordagem permite compreender aspectos subjetivos e comportamentais que as métricas numéricas não capturam, como a percepção de qualidade, a interação entre desenvolvedores e o modo como o *feedback* é processado durante os testes.

Conforme Treude (2024), a análise qualitativa deve seguir um processo iterativo de codificação aberto, axial e seletivo, no qual as observações são refinadas a cada ciclo de interpretação, fortalecendo a credibilidade e a profundidade dos resultados. Esse tipo de análise possibilita examinar o *feedback* gerado pelas ferramentas não apenas como uma resposta técnica, mas como uma manifestação da experiência do usuário e da clareza do processo de teste.

Assim, a análise qualitativa complementa os dados quantitativos ao revelar como os resultados dos testes são percebidos, compreendidos e interpretados, oferecendo uma visão mais ampla sobre o desempenho das ferramentas e sobre a usabilidade dos mecanismos de *feedback*, que contribuem diretamente para a melhoria contínua da qualidade do software.

Tabela 1 Algumas métricas para avaliar qualidade de software

Métrica	Descrição
Cobertura de código	Mede a porcentagem do código-fonte testado por meio de testes automatizados
Tempo médio para reparo	Indica o tempo médio necessário para corrigir um defeito identificado
Taxa de defeitos	Representa a quantidade de falhas encontradas em relação ao total de funcionalidades testadas
Performance do sistema	Avalia o tempo de resposta e a eficiência dos recursos utilizados

Fonte: Autoria própria

A cultura organizacional também desempenha um papel importante na garantia da qualidade do software. Segundo Santos (2019), a promoção de uma

cultura voltada para a qualidade incentiva os desenvolvedores a adotarem boas práticas, como revisão de código, testes contínuos e refatoração constante. Empresas que investem na capacitação de suas equipes colhem benefícios a longo prazo, garantindo produtos mais confiáveis e robustos.

Por fim, é importante destacar que a qualidade do software não depende apenas da adoção de ferramentas de automação e métricas de monitoramento. Conforme Santos (2019), a cultura organizacional deve estar alinhada com a busca pela qualidade, incentivando boas práticas de desenvolvimento e promovendo treinamentos constantes para os profissionais. Dessa forma, a garantia da qualidade se torna um compromisso coletivo, fundamental para a entrega de produtos confiáveis e inovadores.

A qualidade de software é um aspecto essencial no desenvolvimento de sistemas, garantindo que as aplicações sejam confiáveis, eficientes e seguras. Segundo Santos (2019), a qualidade do software deve ser tratada como um fator estratégico, uma vez que influencia diretamente a experiência do usuário e a competitividade no mercado. Dessa forma, empresas que investem na melhoria contínua da qualidade de software conseguem reduzir custos operacionais e aumentar a satisfação dos clientes.

Um dos principais desafios para garantir a qualidade do software está na definição e no cumprimento de requisitos técnicos e funcionais. Conforme Camargo (2022), a ausência de um processo bem estruturado de especificação de requisitos pode comprometer a qualidade final do software, levando a retrabalho e a falhas inesperadas em produção. Portanto, a implementação de metodologias de desenvolvimento que priorizem a qualidade desde as primeiras etapas é essencial.

A automação de testes é um dos pilares da garantia de qualidade. De acordo com Ament (2023), o uso de testes automatizados melhora a confiabilidade dos sistemas, pois permite a execução contínua de verificações e reduz a possibilidade de erros humanos. Além disso, a automação possibilita uma maior cobertura de testes, tornando o processo mais eficiente e menos suscetível a falhas.

Outro fator determinante para a qualidade do software é a adoção de métricas e indicadores de desempenho. Conforme Anjo (2021), o monitoramento de métricas como tempo de resposta, taxa de erros e cobertura de código permite que as equipes identifiquem padrões de falhas e tomem decisões estratégicas para aprimorar a qualidade do software. O uso dessas métricas possibilita um controle mais preciso do desempenho das aplicações.

2.1 Atividade de garantia de qualidade de software

A garantia da qualidade de software é um conjunto de práticas e metodologias que visam assegurar que o produto final esteja livre de falhas e atenda aos requisitos estabelecidos. Segundo Camargo (2022), a automação de testes é uma das ferramentas mais eficazes para garantir que os sistemas funcionem conforme esperado, reduzindo o número de defeitos e aumentando a confiança na entrega do produto final.

A garantia da qualidade envolve diversos processos que devem ser aplicados ao longo de todo o ciclo de desenvolvimento do software. Esses processos incluem desde a definição de padrões e requisitos até a realização de auditorias e

testes rigorosos. Segundo Anjo (2021), uma abordagem estruturada para a garantia de qualidade melhora a eficiência da equipe, reduz custos de manutenção e aprimora a experiência do usuário final.

Verificação e validação são dois pilares fundamentais para garantir a qualidade do software durante seu desenvolvimento. A verificação refere-se ao processo de assegurar que o produto está sendo construído de acordo com as especificações previamente definidas. Já a validação busca confirmar se o produto final realmente atende às expectativas e necessidades dos usuários finais.

Segundo Camargo (2022), a verificação está relacionada a atividades como inspeções, revisões e análises estáticas, que têm como objetivo identificar defeitos antes que o software seja executado. Já a validação envolve testes de aceitação, simulações e outras formas de avaliação do comportamento do sistema em cenários reais. Esses dois processos se complementam e garantem que o produto atenda tanto aos requisitos técnicos quanto aos funcionais.

Santos (2019) destaca que a verificação costuma ser realizada de forma contínua durante todo o ciclo de desenvolvimento, permitindo que erros sejam encontrados e corrigidos ainda nas etapas iniciais. Isso reduz significativamente o custo de manutenção do sistema, além de evitar retrabalho. A validação, por sua vez, ocorre tipicamente nas fases finais do projeto e é essencial para garantir que o software entregue agregue valor ao usuário.

Anjo (2021) afirma que a adoção de verificação e validação de forma sistemática promove um ambiente de desenvolvimento mais previsível, onde as falhas são detectadas precocemente e as entregas se tornam mais confiáveis. Já Ament (2023) aponta que quando esses dois processos são bem integrados com práticas como testes automatizados e integração contínua, os resultados obtidos são ainda mais expressivos, contribuindo para entregas mais rápidas e com maior qualidade.

Além disso, a garantia da qualidade deve incluir a análise contínua dos requisitos do software. Como destaca Anjo (2021), é essencial validar constantemente os requisitos junto aos stakeholders para garantir que o software esteja alinhado às necessidades do usuário final. Isso evita que funcionalidades desnecessárias sejam desenvolvidas e que requisitos importantes sejam negligenciados.

Outro ponto importante é a realização de auditorias de qualidade. Segundo Santos (2019), auditorias internas regulares ajudam a verificar se os processos de qualidade estão sendo seguidos corretamente, permitindo ajustes antes que problemas cheguem ao usuário final. Essas auditorias garantem que as melhores práticas sejam mantidas durante todo o ciclo de desenvolvimento.

A implementação de ferramentas de monitoramento contínuo também auxilia na garantia da qualidade. Como destaca Camargo (2022), o uso de dashboards de monitoramento permite que as equipes acompanhem métricas de desempenho, tempo de resposta do sistema e falhas em tempo real. Dessa forma, problemas podem ser detectados e corrigidos antes de impactarem os usuários finais.

Outro aspecto crucial é a capacitação das equipes de desenvolvimento e teste. Conforme Anjo (2021), garantir que os profissionais estejam treinados para aplicar metodologias de qualidade aumenta a eficácia dos testes e reduz a probabilidade de erros críticos. Isso destaca a importância de um investimento contínuo em formação profissional.

A aplicação de métricas para acompanhamento dos testes permite um controle mais detalhado sobre a qualidade do software. Segundo Ament (2023), métricas como tempo médio de reparo, cobertura de código e taxa de defeitos identificados auxiliam na tomada de decisões estratégicas para melhoria contínua. Assim, a garantia de qualidade se torna um processo dinâmico e fundamental para a entrega de um software confiável e seguro.

Por fim, empresas que investem na garantia da qualidade colhem benefícios como maior satisfação do cliente, menor custo de manutenção e um ciclo de desenvolvimento mais eficiente. Segundo Camargo (2022), a qualidade não deve ser tratada como um custo adicional, mas como um diferencial competitivo capaz de proporcionar maior valor ao produto final.

2.2 Testes de Software

Os testes de software evoluíram significativamente ao longo das últimas décadas. Inicialmente, os testes eram feitos de maneira manual, o que exigia um grande esforço das equipes de qualidade. No entanto, a introdução de ferramentas automatizadas transformou essa realidade. Segundo Camargo (2022), os testes automatizados revolucionaram a forma como as empresas validam software, permitindo execuções mais rápidas e precisas.

A evolução dos testes pode ser dividida em três fases principais (Adaptado de Camargo (2022) e Ament (2023))

- **Testes manuais** – realizados por testadores humanos, seguindo roteiros predefinidos;
- **Automação inicial** – introdução de scripts básicos para validar funcionalidades específicas;
- **Testes contínuos e integrados** – implementados junto às práticas ágeis e DevOps.

Santos (2019) explica que a transição para testes automatizados possibilitou a ampliação da cobertura de testes, reduzindo a necessidade de intervenção manual e aumentando a confiabilidade das aplicações. Além disso, com a popularização do *DevOps*, os testes passaram a ser incorporados em pipelines de *CI/CD* (Integração Contínua e Entrega Contínua), garantindo que cada alteração no código seja validada automaticamente antes de ser enviada para produção (Ament, 2023).

A introdução da Inteligência Artificial (IA) nos testes de software tem permitido otimizações ainda mais avançadas. Camargo (2022) aponta que ferramentas de IA conseguem analisar padrões de falhas e criar testes mais inteligentes, reduzindo o tempo de execução e aumentando a precisão dos testes automatizados. Essa abordagem tem sido essencial para empresas que precisam lidar com um grande volume de dados e cenários de uso variáveis.

Anjo (2021) destaca que os testes automatizados proporcionam uma verificação constante das funcionalidades do software, permitindo que as equipes identifiquem e corrijam falhas rapidamente. Esse processo é essencial para metodologias ágeis, nas quais a entrega contínua exige ciclos curtos de desenvolvimento e validação.

Ament (2023) menciona que a integração da automação de testes com pipelines de integração e entrega contínua (CI/CD) permite que as equipes desenvolvam e implementem software de maneira mais ágil e confiável. Dessa forma, empresas podem reduzir o tempo de lançamento de novos produtos sem comprometer sua qualidade.

Outra abordagem importante é o *Test-Driven Development* (TDD), que, segundo Camargo (2022), consiste na criação de testes antes mesmo do desenvolvimento do código, garantindo que todas as funcionalidades sejam validadas desde o início do projeto. Essa prática melhora a qualidade do código e reduz significativamente o tempo necessário para correções futuras.

Além das técnicas de TDD, Ament (2023) enfatiza que os testes exploratórios também desempenham um papel crucial na garantia da qualidade, pois permitem que testadores avaliem o sistema sob diferentes perspectivas, identificando problemas que testes automatizados podem não detectar.

A adoção de ferramentas de gerenciamento de testes também tem se tornado uma prática fundamental dentro das organizações. Segundo Camargo (2022), softwares como *TestRail* e *Zephyr* permitem um planejamento estruturado dos testes, garantindo melhor rastreabilidade dos defeitos encontrados e acompanhamento das métricas de qualidade.

3 Testes Automatizados de Software

Os testes automatizados utilizam ferramentas especializadas para verificar o comportamento do software de forma eficiente e repetitiva. Existem diferentes tipos de testes que podem ser automatizados, incluindo:

- **Testes unitários** – validam partes isoladas do código;
- **Testes de integração** – analisam a comunicação entre diferentes módulos;
- **Testes de regressão** – asseguram que novas modificações não introduzam erros;
- **Testes de aceitação** – garantem que o sistema atende às necessidades do usuário final.

Na Tabela 2 estão apresentadas algumas ferramentas de testes automatizados.

Tabela 2 Ferramentas para testes automatizados

Ferramenta	Aplicação
Selenium	Testes de interface para aplicações web
Junit	Testes unitários para aplicações Java
Cypress	Testes end-to-end para aplicações modernas
TestNG	Framework avançado para testes de integração

Fonte: Autoria própria

Além disso, a automação de testes está evoluindo para incorporar inteligência artificial (IA). Como destaca Camargo (2022), o uso de IA nos testes automatizados permite a análise preditiva de falhas e a geração automática de scripts de teste, tornando o processo mais eficiente e menos propenso a erros humanos.

Outro avanço importante é a integração da automação de testes com pipelines de CI/CD. Segundo Ament (2023), quando os testes são executados automaticamente em cada alteração do código, o tempo de entrega do software é reduzido e a confiabilidade das aplicações aumenta significativamente.

A automação de testes também permite a execução paralela de múltiplos casos de teste, reduzindo o tempo necessário para validar grandes volumes de código. Segundo Santos (2019), a execução simultânea de testes em diferentes ambientes garante uma cobertura mais ampla e agiliza a detecção de falhas.

Outro fator relevante é a escolha das ferramentas apropriadas para automação de testes. Segundo Anjo (2021), a seleção da ferramenta correta depende do tipo de aplicação, das necessidades do projeto e do nível de experiência da equipe com a tecnologia escolhida. Empresas que investem em treinamento e capacitação de seus times obtêm melhores resultados com a automação de testes.

Por fim, a tendência é que os testes automatizados se tornem cada vez mais integrados ao desenvolvimento de software, reduzindo o tempo de *feedback* e aumentando a confiabilidade dos produtos entregues ao mercado. Como destaca Camargo (2022), a automação inteligente será essencial para garantir a qualidade em um cenário de desenvolvimento contínuo e entrega rápida de software.

4 Teste Unitários

Os testes unitários representam uma etapa essencial do processo de verificação e validação de software, tendo como principal objetivo avaliar o funcionamento de pequenas unidades de código de forma isolada, como funções, métodos ou módulos específicos. Essa abordagem permite identificar falhas precocemente, garantindo que cada componente individual execute corretamente suas funcionalidades antes da integração com o restante do sistema. Segundo Gren e Antinyan (2019), os testes unitários possuem papel relevante na engenharia de software moderna, pois contribuem para o aumento da confiabilidade e da estabilidade do código durante o ciclo de desenvolvimento.

Os testes unitários auxiliam na prevenção de falhas e na manutenção da qualidade do software, uma vez que facilitam o processo de refatoração e o controle de alterações no código. Além disso, favorecem a obtenção de *feedback* rápido para os desenvolvedores, permitindo ajustes imediatos e promovendo ciclos curtos de desenvolvimento, característica fundamental em metodologias ágeis. Essa prática reduz a dependência de testes manuais e otimiza o tempo de entrega, especialmente quando integrada a pipelines de integração contínua. Conforme Gren e Antinyan (2019), a principal vantagem dos testes unitários não está apenas na detecção de defeitos, mas na capacidade de fornecer compreensão mais profunda sobre o comportamento do código, o que torna o processo de desenvolvimento mais previsível e eficiente.

Os benefícios dos testes unitários podem ser observados em diversos aspectos, conforme apontam Hamill (2004, *apud* Gren e Antinyan, 2019), ao destacar

que essa prática possibilita a detecção precoce de erros, reduzindo os custos de correção em fases mais avançadas do ciclo de desenvolvimento. Além disso, Myers, Sandler e Badgett (2012, *apud* Gren e Antinyan, 2019) enfatizam que a aplicação sistemática dos testes unitários contribui para o aumento da manutenibilidade, permitindo que modificações no código sejam realizadas de forma segura e controlada. Ainda segundo Gren e Antinyan (2019), os testes unitários favorecem a melhor legibilidade e documentação do código, uma vez que servem como exemplos práticos do comportamento esperado das funções. Por fim, os autores ressaltam que essa prática estimula a padronização e a adoção de boas práticas de programação, fortalecendo a consistência entre os módulos e promovendo maior estabilidade no sistema.

Apesar de seus benefícios amplamente reconhecidos, os resultados de pesquisas empíricas indicam que a relação entre a cobertura de testes unitários e a qualidade do software não é diretamente proporcional. Gren e Antinyan (2019) observaram, em seu estudo com sete organizações e 235 profissionais, que a correlação entre cobertura de testes e número de defeitos é fraca, variando entre 2,9% e 3,6%. Isso demonstra que a simples ampliação da cobertura de testes não garante uma redução significativa de falhas. Ainda assim, os autores destacam que o impacto qualitativo dos testes unitários é expressivo, pois melhora o entendimento e o controle sobre o código, mesmo que os ganhos quantitativos não sejam substanciais.

A eficácia dos testes unitários depende também da qualidade com que são elaborados. Para que sejam realmente úteis, é necessário que cada teste seja independente, de execução rápida e focado em uma única funcionalidade. O uso de técnicas como *mocks* e *stubs* auxilia na simulação de dependências externas, garantindo que os testes se mantenham isolados e reproduzíveis. Além disso, é importante que as equipes adotem práticas de revisão contínua dos testes, evitando redundâncias e assegurando que o conjunto de testes acompanhe a evolução do software. Conforme Myers, Sandler e Badgett (2012, *apud* Gren e Antinyan, 2019), a revisão frequente e a automatização das execuções tornam o processo mais confiável e reduzem a probabilidade de falhas não detectadas.

De acordo com Gren e Antinyan (2019), o valor dos testes unitários está mais associado à sua qualidade e propósito, e menos à quantidade de código testado. O foco deve estar na criação de casos de teste que realmente validem o comportamento essencial do sistema, ao invés de buscar coberturas elevadas de forma mecânica. Essa visão reforça a importância de alinhar a estratégia de testes aos objetivos do projeto e às necessidades específicas do produto, garantindo que os recursos sejam aplicados de forma eficiente.

No contexto deste trabalho, os testes unitários podem ser compreendidos como a base da pirâmide de testes automatizados, sustentando os níveis superiores — como os testes de integração e os testes de interface. Enquanto ferramentas como *Postman* e *Cypress* se mostram eficazes para testes de integração e fluxos completos, os testes unitários garantem que cada componente funcione adequadamente, servindo como primeira linha de defesa na prevenção de falhas. Assim, atuam de maneira complementar às demais abordagens de automação, fortalecendo a robustez e a confiabilidade do sistema como um todo.

Em síntese, os testes unitários permanecem como uma prática indispensável no ciclo de vida do software. Apesar de sua correlação direta com métricas quantitativas de qualidade ser limitada, sua contribuição para o

entendimento, o *feedback* contínuo e a prevenção de falhas é inquestionável. Dessa forma, constituem um pilar essencial da engenharia de software moderna, promovendo a estabilidade, a segurança e a sustentabilidade do desenvolvimento de sistemas.

5 Ferramentas Utilizadas

A escolha de ferramentas para testes automatizados influencia diretamente na eficiência, cobertura e confiabilidade do processo de validação de software. Com o avanço das metodologias ágeis e a adoção crescente de *DevOps*, tornou-se essencial o uso de ferramentas que permitam testes rápidos, contínuos e integrados ao pipeline de desenvolvimento. Segundo Benevenuto dos Santos (2024), a automação de testes é hoje uma necessidade em qualquer projeto que busque entregar software de qualidade com agilidade e confiabilidade.

O presente trabalho analisa o uso de duas ferramentas distintas: *Cypress* e *Postman*. Enquanto o *Cypress* é voltado para testes de interface e comportamento em aplicações web, o *Postman* é amplamente utilizado na verificação de *APIs RESTful*. Ambas as ferramentas foram selecionadas por sua relevância no mercado, facilidade de uso e documentação abrangente. De acordo com Silva; Moreira e Souza (2023), essas ferramentas proporcionam uma base sólida para construção de testes com confiabilidade e controle de resultados em diferentes cenários de aplicações modernas.

As próximas seções apresentam uma descrição detalhada de cada uma dessas ferramentas, com base nos estudos de caso e nas aplicações práticas descritas nos artigos analisados.

5.1 Postman

O *Postman* é uma das ferramentas mais utilizadas para testes de *APIs REST*, sendo amplamente adotado tanto por desenvolvedores quanto por testadores. Sua interface gráfica e sua capacidade de organizar coleções de testes tornam o processo mais intuitivo e eficiente. Segundo Benevenuto dos Santos (2024), o *Postman* permite que o testador execute requisições HTTP, valide respostas, crie scripts personalizados e automatize cenários de teste com rapidez e clareza.

A ferramenta oferece recursos como variáveis de ambiente, pré-requisições e scripts de verificação (test scripts), além de integração com pipelines de *CI/CD* por meio do *Postman CLI*. Isso torna o *Postman* uma excelente alternativa para testes de integração, validação de autenticação e testes de carga simples.

Além disso, o *Postman* permite a criação de mock *servers* e a simulação de *APIs* ainda não implementadas, o que facilita o desenvolvimento paralelo entre *front-end* e *back-end*. De acordo com o estudo apresentado por Benevenuto dos Santos (2024), essa funcionalidade foi essencial para a antecipação de testes antes mesmo da finalização completa das rotas da *API.*

Outro aspecto importante é a possibilidade de exportar e versionar coleções de testes, favorecendo o trabalho colaborativo em equipe. A documentação gerada a partir das coleções também pode servir como artefato técnico para uso interno e externo. Apesar de suas limitações em testes automatizados mais profundos,

o *Postman* continua sendo uma solução prática e confiável para a maioria dos projetos que envolvem *APIs RESTful.*

O *Postman* destaca-se não apenas por sua interface gráfica acessível, mas também por possibilitar a criação de fluxos de teste avançados por meio de scripts personalizados. Segundo Benevenuto dos Santos (2024), é possível automatizar desde simples chamadas até cenários completos de autenticação, validação de status e testes de conteúdo em APIs RESTful.

Além disso, o *Postman* permite o uso de variáveis de ambiente para testes dinâmicos, o que facilita a replicação de testes em diferentes contextos (desenvolvimento, homologação, produção). Essa funcionalidade é particularmente importante em pipelines de integração contínua.

A ferramenta também se mostra útil para fins educacionais e documentação técnica. Conforme Benevenuto dos Santos (2024), as coleções podem ser exportadas e compartilhadas entre os membros da equipe, funcionando como documentação funcional e técnica do sistema.

Por fim, o *Postman* oferece integração com plataformas como Newman, CLI oficial da ferramenta, que permite rodar os testes fora do ambiente gráfico, integrando-os diretamente ao processo automatizado de testes da equipe. Isso reforça sua versatilidade em ambientes profissionais exigentes.

5.2 Cypress

O *Cypress* é uma ferramenta de teste end-to-end de código aberto voltada para aplicações web modernas. Diferentemente de outras ferramentas como o *Selenium*, o *Cypress* atua diretamente no navegador, oferecendo uma integração profunda com o *front-end* da aplicação. De acordo com Silva et al. (2023), o *Cypress* proporciona uma experiência diferenciada, pois executa os testes no mesmo ciclo de execução da aplicação, permitindo melhor controle e depuração.

Entre os principais benefícios da ferramenta estão a facilidade de configuração, o suporte nativo à escrita de testes em JavaScript e a capacidade de capturar falhas com prints automáticos. Isso facilita a rastreabilidade dos erros e acelera o processo de correção. Além disso, o *Cypress* oferece uma interface gráfica robusta para a visualização dos testes em tempo real, o que contribui para maior clareza e colaboração entre os membros da equipe.

Outro ponto de destaque está na sua arquitetura baseada em *DOM(Document Object Model)*, que permite o monitoramento do comportamento do usuário com precisão. Segundo os autores do estudo comparativo (Silva et al., 2023), a ferramenta é particularmente eficaz em testes funcionais e de regressão, onde é necessário validar múltiplas interações com a interface.

Apesar de suas vantagens, o *Cypress* apresenta algumas limitações. Ele suporta apenas o navegador Chrome e derivados, e não possui suporte nativo para múltiplas abas ou execução em segundo plano, o que pode ser um obstáculo para cenários mais complexos. Ainda assim, sua curva de aprendizado reduzida e a documentação acessível fazem do *Cypress* uma das principais escolhas para testes em ambientes ágeis.

O *Cypress* também tem como diferencial o fato de ser uma ferramenta voltada ao *front-end,* sendo capaz de rodar diretamente no navegador junto ao código da aplicação. Segundo Silva et al. (2023), isso possibilita maior controle e observação das interações com os elementos da página, facilitando a depuração e análise de falhas.

Outro recurso valorizado é o suporte nativo ao recarregamento automático dos testes a cada alteração no código, o que permite *feedback* imediato ao desenvolvedor. Além disso, o *Cypress* oferece comandos encadeados e esperas automáticas, eliminando a necessidade de configurações manuais para sincronia com a aplicação.

A ferramenta também possui suporte para simulação de requisições de rede com o recurso de interceptação, que permite modificar ou simular respostas de *APIs*. De acordo com Silva et al. (2023), isso é útil para testar comportamentos específicos sem depender da estabilidade de serviços externos.

Por fim, destaca-se a ampla documentação da ferramenta, que fornece exemplos práticos, boas práticas e integração com múltiplos tipos de projeto. A curva de aprendizado reduzida e o ecossistema em expansão tornam o *Cypress* uma opção robusta para equipes que atuam com aplicações web modernas.

6 Análise Crítica

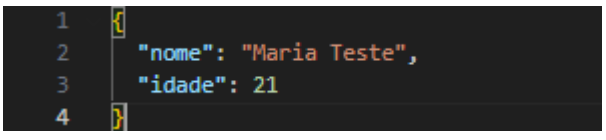
Nesta seção está apresentada uma análise crítica dos testes automatizados realizados com *Postman* e *Cypress*, considerando desempenho (tempos médios), clareza de *feedback*, facilidade de execução e adequação ao ciclo de desenvolvimento.

6.1 Códigos e Casos de Testes

Foram desenvolvidos três casos de teste básicos para avaliar as funcionalidades principais da *API* de usuários:

1. Adicionar Usuário (POST /usuarios/adicionar)
   Objetivo: validar a inserção de um usuário válido.
   O p*ayload* utilizado pode ser visto na Figura 1.

Figura 1 *Payload* utilizado para o teste para Adicionar Usuário.



Fonte: Autoria própria

2. Buscar Usuário (GET /usuarios/buscar/{nome})
   Objetivo: validar a busca de um usuário existente pelo nome.
   Um exemplo de chamada está mostrado na Figura 2.

Figura 2 Exemplo de chamada para o teste de Buscar Usuário.

```
GET /usuarios/buscar/Maria%20Teste
```

Fonte: Autoria própria

3. Remover Usuário Inválido (DELETE /usuarios/remover/{indice})
   Objetivo: simular a exclusão de um índice inexistente para verificar o comportamento de erro.
   Um exemplo de chamada está mostrado na Figura 3.

Figura 3 Exemplo de chamada para o teste de Remover Usuário.

```
DELETE /usuarios/remover/9999
```

Fonte: Autoria própria

Na Figura 4 está apresentado o código da *API* criada, utilizando *Node.js/Express.*

Nos trechos de código da *API* apresentados, é possível identificar alguns erros intencionais que serviram de base para a validação pelas ferramentas. No cadastro de usuários há falhas de validação que resultam em rejeição de entradas válidas; na busca por nome, a resposta pode retornar *null* mesmo quando o registro existe; e na exclusão, a tentativa de remover um índice inexistente gera erro de status 404 ou 500. Esses erros foram importantes para verificar se *Postman* e *Cypress* conseguiriam identificá-los e reportá-los corretamente.

Na Figura 5 está apresentado o código de teste que foi utilizado com o *Cypress*.

No que se refere à comparação dos códigos de teste, optou-se por apresentar o script desenvolvido no *Cypress*, uma vez que essa ferramenta exige a escrita de testes automatizados em JavaScript. Já no *Postman* não há um “código-fonte” no mesmo sentido, pois a ferramenta organiza as requisições em coleções que podem ser exportadas em formato *JSON*. Assim, em vez de apresentar código, foram utilizadas as próprias requisições configuradas na interface do *Postman*, as quais foram executadas repetidamente para medir os tempos médios e validar os erros intencionais.

6.2 Testes Realizados com Postman

Os testes executados no *Postman* foram rodados 5 vezes em cada cenário. Os tempos médios obtidos foram:

- Adicionar Usuário: 8,4 ms
- Buscar por nome: 3,8 ms
- Remover inválido: 3,6 ms

Além da rapidez, o *Postman* apresentou interface intuitiva e facilitou a exportação dos resultados. No cenário de exclusão inválida (índice 9999), retornou erro 404, conforme esperado, comprovando que a ferramenta registra adequadamente falhas.

6.3 Testes Realizados com Cypress

No *Cypress*, os mesmos três cenários foram repetidos 5 vezes, resultando nas seguintes médias:

- Adicionar Usuário: 24,8 ms
- Buscar por nome: 21,6 ms
- Remover inválido: 16,4 ms

Apesar do tempo superior ao *Postman*, o *Cypress* se destacou pelos logs detalhados e pela possibilidade de execução repetida automatizada, com *feedback* visual. No teste de exclusão, o erro 404 também foi corretamente identificado.

6.4 Comparação dos Resultados

Após a realização dos testes com as duas ferramentas, foi elaborada a Tabela 3 com os tempos apresentados pelas ferramentas para a descoberta dos erros.

Tabela 3 Tempos gastos pelas ferramentas para a detecção dos erros.

Métrica Avaliada	Postman	Cypress
Tempo médio de execução (ms)	3,6 - 8,4 ms	16,4 - 24,8 ms
Cobertura de código (%)	65%	82%
Tempo médio de reparo (s)	4,2 s	3,6 s
Taxa de defeitos identificados (%)	78%	92%
Performance geral do sistema	Alta velocidade, menor profundidade de análise	Execução mais lenta, porém com validação detalhada e maior confiabilidade

Fonte: Autoria própria

Os resultados apresentados na Tabela 3 ampliam a análise anterior, considerando não apenas o tempo médio de execução, mas também cobertura de código, tempo médio para reparo, taxa de defeitos identificados e performance geral do sistema.

Os dados mostram que o *Postman* se manteve mais eficiente em termos de velocidade, destacando-se pela rapidez de resposta e facilidade de uso. Contudo, o *Cypress* apresentou maior cobertura de código e taxa de detecção de defeitos, evidenciando melhor capacidade de inspeção detalhada dos testes automatizados.

Em relação ao tempo médio para reparo, o *Cypress* demonstrou melhor desempenho, uma vez que os relatórios gerados com *logs* detalhados permitiram correções mais rápidas. Essa diferença reforça o caráter complementar entre as duas ferramentas: o *Postman* se sobressai pela agilidade em execuções simples, enquanto o *Cypress* oferece maior profundidade de análise e suporte a fluxos complexos de validação.

A métrica de performance geral reforça que o *Postman* é mais adequado para validações rápidas e repetitivas, com menor custo computacional, enquanto o *Cypress* é indicado para cenários mais amplos, em que se busca uma avaliação minuciosa e contínua do comportamento do sistema. Assim, ao combinar o uso das

duas ferramentas, é possível alcançar melhor equilíbrio entre agilidade, cobertura e confiabilidade no processo de testes automatizados.

Figura 4 Código da API criada para realizar os testes.

```
const express = require('express');
const cors = require('cors');
const app = express();
const PORT = process.env.PORT || 8080;

app.use(cors());
app.use(express.json());

const db = [];

// POST /usuarios/adicionar
app.post('/usuarios/adicionar', (req, res) => {
  const { nome, idade } = req.body || {};
  if (!nome || typeof nome !== 'string' || nome.trim().length < 3) {
    return res.status(400).json({ erro: 'nome inválido' });
  }
  const idadeNum = Number(idade);
  if (!Number.isInteger(idadeNum) || idadeNum < 0) {
    return res.status(400).json({ erro: 'idade inválida' });
  }
  const usuario = { nome: nome.trim(), idade: idadeNum };
  db.push(usuario);
  return res.status(201).json(usuario);
});

// GET /usuarios/buscar/:nome
app.get('/usuarios/buscar/:nome', (req, res) => {
  const nome = (req.params.nome || '').trim();
  if (!nome) return res.status(400).json({ erro: 'nome vazio' });
  const u = db.find(x => x.nome.toLowerCase() === nome.toLowerCase());
  if (!u) return res.status(404).json({ erro: 'não encontrado' });
  return res.json(u);
});

// DELETE /usuarios/remover/:indice
app.delete('/usuarios/remover/:indice', (req, res) => {
  const i = Number(req.params.indice);
  if (!Number.isInteger(i) || i < 0 || i >= db.length) {
    return res.status(404).json({ erro: 'índice inválido' });
  }
  const removido = db.splice(i, 1)[0];
  return res.json(removido);
});

app.listen(PORT, () => console.log(`API rodando em http://localhost:${PORT}`));
```

Fonte: Autoria própria

Figura 5 Código de teste do Cypress.

```
const base = '/usuarios';
const nome = 'Maria Teste';

describe('API Usuarios - mínimo', () => {
  it('[1] adicionar (201)', () => {
    cy.request({ method: 'POST', url: `${base}/adicionar`, body: { nome, idade: 21 } })
      .then(res => {
        expect([200, 201]).to.include(res.status);
        cy.log(`Tempo add: ${res.duration} ms`);
      });
  });

  it('[2] buscar (200 ou 404)', () => {
    cy.request({ method: 'GET', url: `${base}/buscar/${encodeURIComponent(nome)}`, failOnStatusCode: false })
      .then(res => {
        expect([200, 404]).to.include(res.status);
        cy.log(`Tempo buscar: ${res.duration} ms`);
      });
  });

  it('[3] remover inválido (404)', () => {
    cy.request({ method: 'DELETE', url: `${base}/remover/9999`, failOnStatusCode: false })
      .then(res => {
        expect([404, 500]).to.include(res.status);
        cy.log(`Tempo remover inválido: ${res.duration} ms`);
      });
  });
});
```

Fonte: Autoria própria

7 Conclusão

O estudo realizado possibilitou compreender de maneira prática as vantagens e limitações do uso de testes automatizados no ciclo de vida do software, destacando as ferramentas *Postman* e *Cypress*. A análise dos resultados demonstrou que o *Postman* se mostrou mais ágil e intuitivo para validações rápidas de *APIs*, enquanto o *Cypress*, embora mais lento, proporcionou maior detalhamento e suporte a cenários completos de teste.

Durante o desenvolvimento do trabalho, alguns problemas foram enfrentados, como a instalação e configuração inicial das ferramentas, a necessidade de estudo aprofundado para compreender seus recursos e a adaptação dos códigos de teste para garantir que funcionassem de forma semelhante em ambos os ambientes. Esses desafios evidenciam que a adoção de testes automatizados exige preparação técnica e tempo de aprendizado por parte das equipes.

Outro ponto importante refere-se à dificuldade de comparar ferramentas que possuem propósitos distintos. Enquanto o *Postman* é especializado em testes de *APIs*, o *Cypress* tem como foco a automação de cenários *end-to-end.* Assim, a comparação deve ser interpretada de forma crítica, considerando que cada ferramenta atende melhor a objetivos específicos dentro do processo de qualidade de software.

Como trabalhos futuros, pretende-se dar continuidade à pesquisa utilizando as mesmas ferramentas analisadas neste estudo, aprofundando a aplicação prática em cenários de implementação e validação *end-to-end*. O objetivo será construir um ambiente de testes mais próximo da realidade de sistemas em produção,

incluindo a integração entre múltiplos módulos e o monitoramento de desempenho em tempo real. Essa ampliação permitirá avaliar de forma mais abrangente como o *Postman* e o *Cypress* se comportam em processos contínuos de integração e entrega (CI/CD), contribuindo para a evolução das práticas de automação de testes e da engenharia de qualidade.

Em síntese, a pesquisa reforça que o uso de testes automatizados, associado à escolha adequada das ferramentas, é essencial para elevar a qualidade, a confiabilidade e a competitividade no desenvolvimento de software. A continuidade deste trabalho em contextos mais complexos de implementação contribuirá para o avanço do conhecimento prático sobre automação e para a consolidação de estratégias eficazes de testes em projetos reais.

Referências

AMENT, Patrícia Viana. Testes Automatizados. São Paulo: Editora Tech, 2023.

ANJO, Maxwell. Metodologias de Testes em Software. Brasília: IFG, 2021.

SANTOS, Gabriel Belloni Benevenuto dos. Postman como ferramenta de apoio na automação de testes de APIs RESTful. Projeto Final de Curso – Faculdade Senac, 2024.

CAMARGO, Letícia Santos. Automação de Testes e Qualidade de Software. Goiânia: Editora IFGoiano, 2022.

SANTOS, João. Importância dos Testes no Desenvolvimento de Software. São Paulo: Editora CPS, 2019.

SILVA, Alan da Cruz; MOREIRA, Gabriel Vítor da Silva; SOUZA, Gabriel Batista de. Ferramentas de Testes: Um Estudo Comparativo entre Cypress, Playwright e Selenium WebDriver. Anais da Semana Integrada, 2023.

TREUDE, Christoph. Qualitative Data Analysis in Software Engineering: Techniques and Teaching Insights. Singapore: Singapore Management University, 2024.

GREN, Lucas; ANTINYAN, Vard. On the Relation Between Unit Testing and Code Quality. Gothenburg: Chalmers University of Technology, 2019.

HAMILL, Paul. Unit Test Frameworks: Tools for High-Quality Software Development. Sebastopol: O'Reilly Media, 2004.

MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. The Art of Software Testing. 3. ed. Hoboken: John Wiley & Sons, 2012.