Ferramentas de Testes: Um Estudo Comparativo entre Cypress, Playwright e Selenium Webdriver.

Testing tools: a comparative study of Cypress, Playwright and Selenium Webdriver.

Arthur Bezerra de Brito[1], **Rebeca Marina dos Santos**[1], **Sheyla Natália de Medeiros**[1]

[1] Análise e Desenvolvimento de Sistemas – Instituto Federal de Pernambuco
Paulista – PE – Brasil

abb@@discente.ifpe.edu.br, rms43@discente.ifpe.edu.br,

sheyla.medeiros@paulista.ifpe.edu.br

Resumo. *Garantir a qualidade do software é essencial, e os testes automatizados estão se destacando cada vez mais por sua capacidade de identificar rapidamente as falhas e melhorar a eficiência do desenvolvimento. Entre as principais ferramentas de automação de testes da Web estão o Cypress, o Playwright e o Selenium WebDriver. O Selenium WebDriver é uma solução tradicional e amplamente adotada, enquanto o Cypress e o Playwright são ferramentas mais recentes que oferecem abordagens mais modernas. A escolha da ferramenta ideal pode ser um desafio, pois cada uma tem suas próprias vantagens e limitações. O objetivo deste estudo é compará-las com base em critérios como suporte a linguagens de programação, compatibilidade com navegadores, facilidade de configuração, recursos de depuração, robustez e resistência a testes com falhas. A pesquisa combina uma revisão teórica com testes práticos, mostrando que, embora o Selenium esteja bem estabelecido e seja versátil, o Cypress e o Playwright se destacam por oferecer recursos mais modernos e otimizados.*
Palavras-chave: qualidade de software; automação de testes; Cypress; Playwright; Selenium WebDriver.

Abstract. *Ensuring software quality is essential, and automated testing is increasingly prominent for its ability to quickly identify faults and improve development efficiency. Among the main web test automation tools are Cypress, Playwright, and Selenium WebDriver. Selenium WebDriver is a traditional and widely adopted solution, while Cypress and Playwright are newer tools that offer more modern approaches. Choosing the ideal tool can be challenging, as each has its own advantages and limitations. This study aims to compare them based on criteria such as programming language support, browser compatibility, ease of configuration, debugging capabilities, robustness, and resistance to flaky tests. The research combines a theoretical review with practical testing, showing that although Selenium is well-established and versatile, Cypress and Playwright stand out for offering more modern and streamlined features.*
Keywords: software quality, test automation, Cypress, Playwright, Selenium Webdriver.

1. Introdução

A tecnologia tem impulsionado a conscientização das empresas sobre a importância de garantir a qualidade e a confiabilidade de seus softwares. Segundo Escobar e Muniz (2021), a área de automação de testes tem apresentado crescimento contínuo devido à qualidade e praticidade que proporciona aos sistemas de software. À medida que a importância do software cresce e as exigências por qualidade se intensificam, surge a necessidade de desenvolver abordagens que aprimorem tanto o processo de desenvolvimento quanto a produção de software (Pressman; Maxim, 2021).

Segundo Myers (2011) testar é o processo de executar um programa com a intenção de encontrar erros, um bom teste é aquele que tem grande probabilidade de encontrar um erro ainda não descoberto. Como destacado por Pandy (2024), a prática de testes de software é fundamental para garantir que os sistemas atendam aos padrões de qualidade necessários, evitando falhas que possam comprometer a operação das empresas.

Testes de software agregam um valor significativo para as empresas, independentemente de seu porte ou segmento. Segundo Rios e Moreira (2006), reduzir os custos de desenvolvimento e aprimorar a qualidade dos produtos finais são objetivos fundamentais das organizações. A realização de testes desde as fases iniciais do desenvolvimento possibilita a rápida identificação de falhas, prevenindo problemas críticos nas etapas posteriores e, consequentemente, reduzindo os custos. Além disso, para ser eficaz, a automação deve se integrar bem ao sistema já utilizado, garantindo um funcionamento adequado, isso torna os processos mais rápidos e melhora a experiência dos clientes, como apontado por uma análise recente sobre o mercado de testes automatizados (Business Research Insights, 2025). De acordo com a IBM – International Business Machines Corporation (2024), a automação é uma estratégia essencial para otimizar esse processo, garantindo execuções mais rápidas, feedback contínuo e validação antecipada do sistema, práticas como rastreamento de defeitos, métricas e relatórios auxiliam no monitoramento da qualidade e na tomada de decisões mais assertivas. Essas abordagens contribuem para uma entrega mais organizada, aprimorando a documentação, a gestão de erros e a confiabilidade do produto final.

Este estudo tem como objetivo comparar as ferramentas de automação de testes Cypress, Playwright e Selenium WebDriver, utilizando as métricas de linguagens de programação suportadas, navegadores compatíveis, facilidade de configuração, recursos de depuração, robustez na execução e resistência a falhas. Com isso, busca-se fornecer uma base para que os leitores possam avaliar qual ferramenta melhor se adequa a diferentes cenários de testes automatizados.

2. Objetivos

Nesta seção, é descrito o objetivo geral e objetivos específicos do estudo.

2.1. Objetivo Geral

O objetivo deste trabalho é comparar as ferramentas de automação de testes Cypress, Playwright e Selenium Webdriver, destacando suas principais características e contribuindo para a escolha da ferramenta mais adequada em projetos de testes de software.

2.2. Objetivos Específicos

Este trabalho visa atingir os seguintes objetivos específicos:

- Desenvolver e executar testes automatizados em uma aplicação web, avaliando o desempenho das ferramentas em um ambiente real;

- Analisar os resultados obtidos, considerando aspectos técnicos e o impacto no processo de automação de testes;
- Comparar as ferramentas nas métricas de linguagens de programação suportadas, navegadores compatíveis, facilidade de configuração, recursos de depuração, robustez na execução e resistência a falhas intermitentes.

3. Fundamentação Teórica

A Seção 3.1 define o que são testes de software, explica seus níveis (unitário, integração, sistema e aceitação) e descreve os principais tipos, como funcionais e não funcionais. A Seção 3.2 apresenta as ferramentas Cypress, Playwright e Selenium WebDriver, destacando suas principais características no uso para automação de testes web.

3.1. Testes de Software

A palavra ”teste”refere-se ao processo de executar um programa para garantir que ele funcione conforme o esperado e atenda aos requisitos definidos (Myers, 2011). Quando ouvimos a palavra ”teste”, frequentemente imaginamos alguém no computador inserindo dados, observando os resultados e reportando problemas (Bernard Homès, 2024). Embora essa parte visível dos testes seja fundamental, os testes de software abrangem uma gama de atividades que não se limitam apenas a executar o sistema, o processo vai muito além. De acordo com a International Software Testing Qualifications Board (ISTQB), o teste de software é um processo mais amplo que inclui planejamento, análise, design e avaliação dos resultados (ISTQB, 2018). Segundo Pressman e Maxim (2021), o teste de software desempenha um papel vital na garantia da qualidade, eles também envolvem a detecção e correção de falhas, além de estratégias para prevenir futuros problemas, garantindo que o software funcione da melhor forma possível. Para assegurar um entendimento eficaz sobre testes de software, é fundamental ter uma compreensão clara dos termos-chave envolvidos, como engano, erro, defeito e falha.

Engano: Ação humana equivocada que resulta na introdução de um defeito no software, como um erro em uma lógica ou fórmula (ISTQB, 2018).

Defeito: Imperfeição no código que pode causar resultados inconsistentes ou falhas no sistema (Delamaro, 2017).

Erro: Resultado inesperado gerado pela execução de um defeito, como um cálculo incorreto (IEEE, 1990).

Falha: Quando o comportamento do software não corresponde ao resultado esperado devido a um erro, podendo ou não ser perceptível (Delamaro, 2017).

O principal objetivo do teste de software é reduzir a probabilidade de falhas durante a operação do sistema em produção (Molinari, 2012). Assim, é crucial entender que um erro cometido por uma pessoa pode resultar na introdução de um defeito no software, que, por sua vez, pode se manifestar como uma falha ou bug. Quando o código defeituoso é executado, ele pode levar à falha do sistema, revelando o impacto negativo do defeito no comportamento do software (ISTQB, 2018).

3.1.1. Níveis de testes

No processo de desenvolvimento de software, é essencial que os testes sejam planejados e executados em diferentes níveis, acompanhando o ciclo de vida do projeto. Essa abordagem permite verificar e validar cada etapa de forma adequada, contribuindo para a qualidade do produto final (Delamaro; Jino; Maldonado, 2017). Os níveis de teste determinam o momento mais apropriado para aplicar diferentes tipos de testes, cada um com foco em aspectos específicos do software, o que ajuda a reduzir custos e evitar retrabalho (Bastos; Rios; Cristalli; Moreira, 2007). Conforme ilustrado na Figura 1, os principais níveis são: Teste de Componentes (ou Teste Unitário), Teste de Integração, Teste de Sistema e Teste de Aceitação (ISTQB, 2018).

Figura 1. Níveis de Testes no Desenvolvimento de Software

Nível de teste	Foco	Onde se encaixa	Responsável
Teste de Unidade	Foca em partes individuais do sistema que podem ser testadas de forma isolada.	Após o desenvolvimento de cada pequena parte do código.	Desenvolvedor
Teste de Integração	Prioriza as interações entre os diversos componentes ou sistemas.	Após o teste de unidade, quando várias unidades ou módulos são combinados.	Equipe de desenvolvimento
Teste de Sistema	Concentra-se no comportamento e nas capacidades de todo o sistema, considerando as tarefas de ponta a ponta e os aspectos não funcionais envolvidos.	Após o teste de integração, quando o sistema completo está pronto.	Equipes de teste ou QA (Quality Assurance)
Teste de Aceitação	Foca no comportamento e na funcionalidade de todo o sistema ou produto.	No final do ciclo de desenvolvimento, antes de entregar o produto ao cliente ou liberar o software para o mercado.	Usuários finais, clientes, ou equipes de desenvolvimento

Fonte: Elaborada pelos autores.

O intuito dos **Testes unitários** ou **Testes de unidade** é garantir que cada unidade de código funcione conforme o esperado. Os **Testes de integração** consistem em examinar como as unidades do software se combinam e como o projeto se integra como um todo. Os **Testes de sistema** envolvem a avaliação do sistema como um todo, verificando se ele atende aos requisitos especificados. Os **Testes de aceitação** são conduzidos do ponto de vista do usuário final e têm como objetivo garantir que o sistema atenda as métricas de aceitação definidos previamente, com a finalidade de verificar se o software está pronto para ser entregue ao usuário final.

Por fim, os objetivos dos testes incluem reduzir o risco, verificar se os comportamentos funcionais e não funcionais estão conforme o projetado e especificado, e construir confiança na qualidade dos componentes, interfaces e do sistema como um todo. Além disso, é essencial identificar e corrigir defeitos e evitar que eles se espalhem para níveis superiores de teste. Os testes também visam validar que o sistema completo, integrado e funcionando em conjunto, atende aos requisitos especificados (ISTQB, 2018).

3.1.2. Tipos de testes

Para garantir a qualidade em todas as camadas do software e evitar problemas, diversos tipos de testes são realizados ao longo do desenvolvimento para avaliar métricas específicas de um sistema ou parte dele, com base em objetivos claros. A classificação dos testes com base nos objetivos são incluir a verificação da qualidade funcional, como integridade e correção; da qualidade não funcional,

como desempenho e segurança; da adequação da estrutura ou arquitetura do sistema; e dos impactos de alterações, como a correção de defeitos e a identificação de possíveis regressões (ISTQB, 2018).

Teste Funcional também conhecido como **Teste caixa-preta**, onde sua estrutura e funcionamento internos não são levados em conta (Neto, 2007). A abordagem consiste em fornecer dados de entrada, executar o teste e comparar o resultado obtido com o resultado esperado. O foco do testador deve ser identificar situações em que o programa não funcione conforme o esperado ou especificado (Myers; Sandler; Badgett, 2011).

O **Teste Não Funcional** examina o desempenho e a qualidade do sistema, concentrando-se em como ele opera sob diferentes condições, incluindo aspectos como segurança, eficiência e usabilidade (ISTQB, 2018). De acordo com Myers (2011) O **Teste Estrutural** ou **Teste caixa-branca** é uma abordagem que possibilita a análise da estrutura interna do software, permitindo uma avaliação detalhada de seu código e funcionamento interno. O teste caixa-branca pode focar na arquitetura do sistema e suas interfaces, exigindo habilidades específicas, como o uso de ferramentas de cobertura de código e análise de consultas de banco de dados (ISTQB, 2018).

Quando alterações são feitas em um sistema, é crucial realizar **Testes de Regressão** para garantir que essas mudanças foram implementadas corretamente, não causaram efeitos colaterais inesperados e que o comportamento do sistema continua conforme o esperado (Sommerville, 2011).

3.2. Ferramentas de Testes Automatizados

São soluções tecnológicas que automatizam a execução de casos de teste, reduzindo a necessidade de intervenção humana. Segundo Sommerville (2011), os testes podem ser manuais ou automatizados, nos testes manuais, um testador executa o software, compara os resultados esperados com os obtidos e reporta possíveis falhas. Já nos testes automatizados, scripts programados realizam essas verificações de forma repetitiva e eficiente.

De acordo com o ISTQB (2018), a automação de testes oferece diversos benefícios, incluindo:

- Redução do trabalho manual repetitivo, eliminando esforços em testes de regressão, configuração de ambientes, reentrada de dados, o que resulta em economia de tempo.
- Maior consistência e repetibilidade, garantindo que os testes sejam executados de forma padronizada, com geração coerente de dados e alinhamento aos requisitos.
- Avaliação objetiva, com base em métricas e cobertura de código.
- Acesso facilitado a informações, como estatísticas, gráficos de progresso, taxas de defeitos e indicadores de desempenho.

Com o crescimento da área de testes de software, ferramentas de automação tornaram-se essenciais para garantir qualidade contínua e confiável. Segundo Rollwagen (2020), essas ferramentas oferecem suporte ao processo de testes, auxiliando na detecção de falhas e na validação do software. Além disso, conforme destaca Pessoa (2020), elas permitem a simulação de interações reais dos usuários, eliminando a necessidade de execução manual e aumentando a eficiência do processo. O mercado oferece diversas opções, cada uma com métricas específicas. Dado o amplo conjunto de ferramentas disponíveis no mercado, cada uma com métricas específicas, é essencial analisar seus benefícios e limitações antes da implementação em um projeto. Neste contexto, este trabalho apresentará as ferramentas Cypress, Playwright e Selenium WebDriver.

3.2.1. Cypress

O Cypress é uma ferramenta de automação de testes voltada para a verificação de aplicações end-to-end ou teste ponta a ponta baseado em JavaScript, conhecido por sua interface intuitiva e velocidade na execução dos testes. Ao contrário de outras ferramentas, o Cypress opera diretamente dentro do navegador, permitindo a execução simultânea dos testes com a aplicação e interação com eventos em tempo real (Khetarpal, 2021) eliminando a necessidade de drivers adicionais. Ele permite testar qualquer aplicação que seja executada em um navegador, oferecendo uma visão detalhada sobre o desempenho dos testes e contribuindo para a melhoria da qualidade do software. Além disso, fornece insights valiosos para aprimorar a integridade do conjunto de testes (cypress.io. Why Cypress?, 2024). A seguir, estão algumas das principais características que fazem do Cypress uma boa ferramenta para automação de testes:

- **Linha do Tempo:** Captura de telas durante os testes, permitindo visualizar cada etapa e facilitando a depuração.

- **Depuração Simplificada:** Integração com *DevTools* do navegador e exibição de erros claros e *stack traces* detalhados para facilitar a identificação de falhas.

- **Espera Automática:** Gerencia automaticamente a espera por elementos, eliminando a necessidade de comandos manuais, o que torna os testes mais confiáveis.

- **Controle de Tráfego de Rede:** Permite interceptar e modificar requisições de rede, simulando diferentes respostas sem depender do servidor.

- **Resultados Confiáveis:** Com uma arquitetura própria, o Cypress garante testes rápidos e estáveis, sem falhas intermitentes.

- **Compatibilidade com Múltiplos Navegadores**: É importante destacar que de acordo com a documentação oficial, oferece suporte para navegadores baseados no Chrome (incluindo Microsoft Edge, Electron e Chromium), além do Firefox e do WebKit, que é o mecanismo de renderização do Safari.
  No entanto, o suporte ao WebKit ainda é experimental. Para habilitá-lo, é necessário ativar a opção experimentalWebKitSupport na configuração do Cypress e instalar o pacote playwright-webkit via npm. Essa funcionalidade foi implementada com base no WebKit do Playwright, com o objetivo de aprimorar a experiência dos usuários em versões futuras da ferramenta. No entanto, por estar em fase experimental, podem ocorrer instabilidades, e eventuais problemas não documentados devem ser reportados no repositório oficial do Cypress no GitHub.

O Cypress também conta com o ”Cypress Cloud”que é uma plataforma que complementa o Cypress, oferecendo recursos avançados para gerenciar, otimizar e analisar testes automatizados na nuvem. Ela facilita a execução de testes em escala, com ferramentas para depuração, monitoramento e integração contínua. Recursos Principais:

- **Repetição de Testes:** Refaça execuções de testes para depuração fácil;

- **Detecção de Flakes:** Identifique e analise testes instáveis;

- **Branch Review:** Avalie o impacto de pull requests no conjunto de testes.

- **Integrações:** Conecte com GitHub, GitLab, Bitbucket, Slack, Jira e Microsoft Teams;

- **Test Analytics:** Acompanhe resultados e tendências de testes ao longo do tempo.

De acordo com Corrêa e Silva (2021) após a instalação, o ambiente de automação está pronto para uso imediato, oferecendo uma instalação e configuração simples. Segundo Khan (2021), o Cypress também leva em conta a visibilidade dos elementos durante a execução do teste. Por exemplo, se o comportamento de um botão estiver sendo testado (seja ele visível ou oculto), e o estado do botão não corresponder à situação esperada, o teste falhará.

Por fim, de acordo com a documentação oficial do Cypress, a ferramenta não é destinada a automação de uso geral, sendo focada principalmente em testes em navegadores. A seguir, estão algumas limitações:

- **Limitação com Múltiplos Navegadores:** O Cypress não permite controlar mais de um navegador simultaneamente. Isso significa que você não pode gerenciar dois navegadores abertos ao mesmo tempo;

- **Superdomínio Único:** No Cypress, cada teste pode interagir apenas com um único superdomínio, ou seja, ele não permite navegação entre domínios diferentes dentro de um único teste. Isso significa que, se o teste envolver múltiplos domínios, é necessário usar recursos específicos, como o comando cy.origin, para habilitar a navegação cruzada.

Embora o Cypress não permita o controle simultâneo de múltiplos navegadores, essa limitação pode ser contornada com o uso do plugin @cypress/puppeteer. Com essa integração, é possível interagir com várias abas dentro de um mesmo teste, expandindo as possibilidades de automação. Isso permite simular cenários mais complexos, como autenticação em uma aba e verificação de dados em outra, tornando os testes mais abrangentes e realistas.

3.2.2. Playwright

O Playwright é um framework robusto para automação de testes de ponta a ponta, projetado para interagir com navegadores de forma eficiente e confiável. Ele se destaca por sua arquitetura moderna, que inclui funcionalidades avançadas, como execução paralela, testes de snapshot e geração automática de relatórios detalhados. Além disso, o Playwright oferece suporte a múltiplos navegadores e permite a automação de interações complexas, garantindo testes mais rápidos e estáveis (Stramer, 2024). De acordo com a documentação do Playwright (2025), a ferramenta é projetada para automação de testes de ponta a ponta, oferecendo recursos avançados que garantem testes rápidos, eficientes e confiáveis. Abaixo estão algumas das principais características que tornam o Playwright uma escolha sólida para automação de testes:

- **Suporte a múltiplos navegadores:** Compatível com Chromium, WebKit e Firefox, permitindo testes em diferentes ambientes;

- **Execução multiplataforma:** Funciona em Windows, Linux e macOS, localmente ou em CI, no modo *headless ou headed*. O modo *headless* permite rodar testes sem exibir a interface gráfica, agilizando a automação, enquanto o modo *headed* exibe a interface, facilitando a

depuração e a análise visual;

- **Compatibilidade com várias linguagens:** Suporte para TypeScript, JavaScript, Python, .NET e Java;
- **Teste de aplicações móveis:** Emulação nativa para Google Chrome (Android) e Mobile Safari (iOS);
- **Sincronização automática:** Aguarda elementos antes de interagir, eliminando a necessidade de timeouts artificiais;
- **Afirmações otimizadas:** Repetição automática de verificações até que as condições esperadas sejam atendidas;
- **Depuração avançada:** Suporte para rastreamento, gravação de vídeos e capturas de tela;
- **Execução eficiente:** Testes rodam fora do processo do navegador, melhorando estabilidade e desempenho;
- **Suporte a múltiplos contextos:** Permite testar múltiplas abas, diferentes origens e múltiplos usuários;
- **Eventos confiáveis:** Simula interações reais com precisão.
- **Isolamento de testes:** Cada teste roda em um contexto de navegador independente;
- **Reutilização de autenticação:** Salva o estado do login para evitar repetição de autenticações;
- **Codegen:** Gere testes automaticamente ao gravar suas ações no navegador, podendo salvar em qualquer linguagem;
- **Inspetor de Seletores:** Inspecione a página, gere seletores e veja os pontos de clique e logs de execução para ajudar na depuração;
- **Trace Viewer:** Capture informações detalhadas para investigar falhas, incluindo vídeos, capturas de tela e o código do teste.

O Playwright é uma ferramenta eficiente para testes de ponta a ponta em aplicações web, oferecendo suporte a múltiplos navegadores, execução paralela e geração de relatórios detalhados. Sua abordagem estruturada garante testes mais rápidos, bem organizados e com análises completas dos resultados, tornando-o uma escolha sólida para a automação de testes.

3.2.3. Selenium Webdriver

O Selenium é uma suíte de ferramentas para automação de testes, composta por Selenium IDE, Selenium RC, Selenium WebDriver e Selenium Grid, essa coleção de ferramentas oferece flexibilidade para diferentes necessidades. O Selenium oferece suporte à automação dos principais navegadores do mercado utilizando o WebDriver, como o Chrome, Firefox, Edge, Internet Explorer e Safari. O

WebDriver utiliza as APIs de automação de navegador fornecidas pelos desenvolvedores dos navegadores para controlar e realizar testes (Selenium, 2024). Cada navegador possui uma implementação específica chamada driver que é responsável por gerenciar a comunicação entre o Selenium e o navegador, delegando as ações a serem executadas (Gonzalez, 2022), ele permite a interação direta com navegadores para testar aplicações web, reduzindo significativamente o esforço de testadores manuais, simulando a navegação do usuário. O selenium Webdriver suporta diversas linguagens como Java, PHP, Python e Csharp, além de múltiplos sistemas operacionais (Contri, 2019).

A ferramenta facilita o processo de depuração de scripts de teste por meio de *breakpoints*, sendo adequado tanto para testes de sistema quanto para testes de regressão (Silva, 2016). É essencial para a automação de testes *end-to-end*, simulando a interação de um usuário final com a aplicação. Sua robustez e flexibilidade o tornam amplamente adotado em grandes projetos, consolidando-se como um padrão na indústria para automação de testes em aplicações web. O Selenium WebDriver, conforme mencionado por Wardhan e Madan (2021), é eficaz em testar páginas dinâmicas, permitindo a interação com elementos que mudam sem a necessidade de recarregar a página. Ele oferece a capacidade de execução em modo *headless*, os navegadores *headless* permitem a execução de testes automatizados de forma mais rápida, pois eliminam a necessidade de renderizar a interface gráfica. Isso melhora o desempenho e reduz o consumo de recursos. No entanto, a ausência da interface visual pode dificultar a depuração de falhas. Segundo Ramya (2017), é o componente mais rápido do Selenium, pois executa comandos diretamente no navegador, sem envolver um servidor intermediário, resultando em maior eficiência e rapidez nos testes.

4. Trabalhos Relacionados

A escolha das ferramentas de automação de testes é crucial para a eficácia dos testes e a qualidade do software. Diversos estudos têm explorado as capacidades dessas ferramentas, com ênfase em seu desempenho e aplicabilidade em diferentes contextos de desenvolvimento. A fim de selecionar os trabalhos mais relevantes para esta pesquisa, foi realizado um levantamento bibliográfico com foco em estudos técnicos, análises comparativas e experiências práticas com as ferramentas Cypress, Playwright e Selenium WebDriver.

A busca foi realizada majoritariamente por meio da plataforma Google Acadêmico, reconhecida por sua ampla cobertura de artigos científicos, dissertações e publicações especializadas. Essa busca foi complementada pela análise de artigos técnicos, blogs especializados e pelas documentações oficiais de cada ferramenta. Foram utilizadas combinações de palavras-chave em português e inglês, como: “ferramentas de automação de testes”, “comparação entre Cypress, Selenium e Playwright”, “automated testing tools comparison” e “Selenium vs Cypress vs Playwright”. Os critérios de seleção incluíram: a presença de comparações entre ferramentas, abordagem técnica ou empírica, relevância prática e a atualidade dos estudos, priorizando publicações no período de 2010 a 2024.

O Selenium é uma ferramenta consolidada e amplamente utilizada para automação de testes em aplicações web. Segundo Veloso (2010), o Selenium é uma das ferramentas mais adequadas para testes funcionais automatizados, sendo base para muitas outras tecnologias devido à sua flexibilidade e à compatibilidade com diversas linguagens de programação. No entanto, conforme apontado por Mane (2016), seu desempenho pode ser impactado em aplicações modernas e dinâmicas, exigindo maior complexidade de configuração e ajustes manuais frequentes.

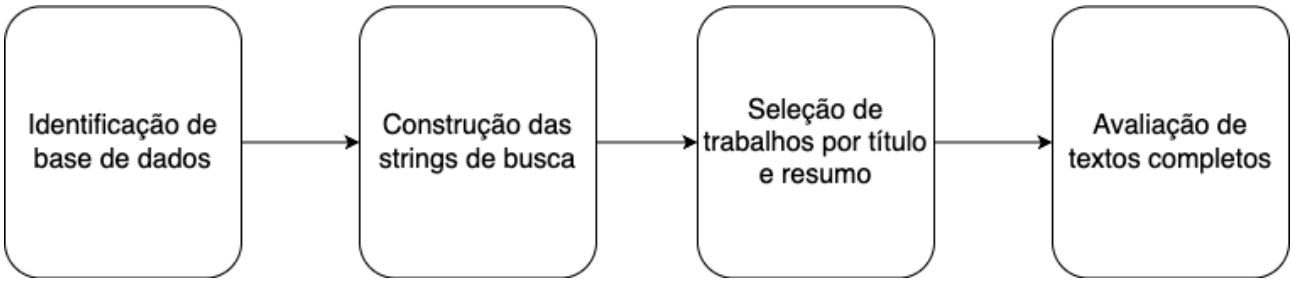
Em contrapartida, o Cypress se destaca como uma solução moderna, projetada para atender às necessidades de aplicações web contemporâneas. Capellini (2021) ressalta que, por adotar uma arquitetura própria e independente do Selenium, o Cypress oferece vantagens significativas em termos

de velocidade, facilidade de uso e integração com frameworks JavaScript modernos. Mobaraya (2019) complementa que, mesmo sendo uma ferramenta mais recente, o Cypress demonstra grande potencial, especialmente por suas funcionalidades que facilitam a depuração e o diagnóstico de falhas.

O Playwright, por sua vez, diferencia-se por sua arquitetura baseada em WebSockets, que permite comunicação contínua entre cliente e servidor durante a execução dos testes. Isso elimina a necessidade de reiniciar conexões a cada requisição, aumentando significativamente a velocidade dos testes. De acordo com Nair (2024), o Playwright introduz o conceito de "contexto do navegador", que possibilita a execução paralela de testes em instâncias isoladas, proporcionando maior controle, isolamento e eficiência nos cenários testados.

O Cypress é amplamente adotado por desenvolvedores JavaScript devido à sua alta performance na execução de testes de interface. No entanto, sua limitação ao suporte exclusivo à linguagem JavaScript pode restringir sua adoção em projetos que utilizam múltiplas tecnologias. Conforme destacado pela BrowserStack, 'o Cypress é uma ferramenta baseada em JavaScript, o que significa que ele só suporta testes escritos em JavaScript ou TypeScript'. Em comparação, tanto o Selenium WebDriver quanto o Playwright oferecem suporte a várias linguagens de programação, como Java, Python, C e Ruby, tornando-se ferramentas mais versáteis e aplicáveis a uma gama mais ampla de tipos de automação, além dos testes tradicionais de interface

Figura 2. String de busca

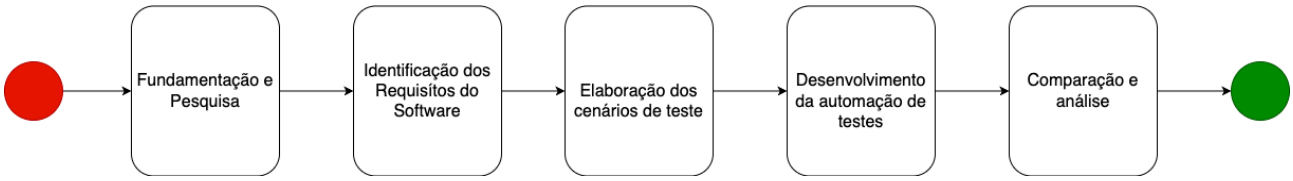


Fonte: Elaborada pelos autores.

5. Metodologia

A metodologia utilizada neste estudo é de abordagem qualitativa e comparativa, baseada na análise documental e em testes práticos. Primeiramente, foram examinadas as documentações oficiais das ferramentas Cypress, Playwright e Selenium WebDriver, considerando os critérios definidos por Sousa (2023) e Silva (2019): linguagens de programação suportadas, navegadores compatíveis, facilidade de configuração e recursos de depuração. Em seguida, foram realizados testes práticos com scripts equivalentes nas três ferramentas, possibilitando uma comparação direta entre elas. As métricas relacionadas à robustez na execução e à capacidade de lidar com falhas intermitentes foram extraídas das documentações oficiais disponíveis nos sites do Cypress (2024), Playwright (2024) e Selenium WebDriver (2024). Essa combinação de análise teórica e prática permitiu uma avaliação das ferramentas em contextos reais de uso.
A Figura 2 apresenta as cinco etapas metodológicas do trabalho, as quais serão discutidas nos próximos tópicos.

Figura 3. Etapas da metodologia.



Fonte: Elaborada pelos autores.

5.1. Fundamentação e pesquisa

A pesquisa foi desenvolvida em três etapas principais. Na primeira etapa, foi conduzido um levantamento bibliográfico com foco nos fundamentos de testes de software. Foram analisadas obras clássicas como Introdução ao Teste de Software de Delamaro, Jino e Maldonado (2017), The Art of Software Testing de Myers (2011) e Engenharia de Software de Pressman e Maxim (2021), que forneceram os conceitos necessários sobre tipos e níveis de testes aplicáveis ao contexto de automação. Na segunda etapa, foi realizada uma análise documental das ferramentas Cypress, Playwright e Selenium WebDriver, com base em suas documentações oficiais atualizadas (Cypress, 2024; Playwright, 2024; Selenium, 2024). Essa análise buscou identificar características técnicas como linguagens suportadas, navegadores compatíveis, facilidade de configuração, recursos de depuração, robustez e capacidade de lidar com falhas intermitentes.

Paralelamente, foram consultados artigos científicos recentes, publicados em periódicos e conferências da área de engenharia de software, para complementar a base teórica com estudos atualizados e específicos sobre automação de testes. Entre eles, destaca-se o artigo Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright (Springer Professional, 2024), que apresentou dados comparativos relevantes sobre desempenho, compatibilidade e facilidade de uso das ferramentas analisadas. Por fim, na terceira etapa, foi selecionada uma aplicação web de grande porte como ambiente de testes, onde foram implementados scripts equivalentes em cada ferramenta. Essa fase prática permitiu observar o comportamento real das soluções, validar informações extraídas dos documentos e identificar limitações ou vantagens no uso em cenários reais. O cruzamento entre os dados teóricos e práticos embasou a análise comparativa final, proporcionando uma avaliação crítica fundamentada das ferramentas.

5.2. Identificação dos requisitos de software

Para a avaliação prática deste estudo, foi escolhida a Queima Diária, uma plataforma de exercícios online, na qual um dos autores trabalha atualmente. A escolha dessa plataforma se justifica pela familiaridade do autor com o ambiente e pelas métricas reais e dinâmicas que ela oferece, proporcionando um contexto adequado para a implementação e teste das ferramentas. Segundo o ISTQB (2018) o System Under Test (SUT), ou ”Sistema em Teste”, refere-se ao software que está sendo avaliado para verificar seu funcionamento correto. A seleção da plataforma a ser testada seguiu condições específicas para assegurar que o sistema estivesse alinhado com os objetivos da pesquisa e permitisse a execução eficiente dos testes automatizados utilizando Cypress, Playwright e Selenium WebDriver.

As condições considerados para a seleção da Queima Diária como SUT foram:

- Ser uma plataforma web com funcionalidades amplamente utilizadas em sistemas reais, como

[1] Disponível em: https://encurtador.com.br/f94Ys

autenticação, navegação e consumo de conteúdo multimídia;

- Representar um sistema com complexidade adequada para permitir a automação de diferentes funcionalidades, assegurando que as ferramentas possam ser testadas de maneira abrangente em um ambiente dinâmico;

- Ser um sistema acessível e configurável para permitir a automação e replicação dos experimentos de maneira simplificada.

A plataforma Queima Diária foi escolhida por sua robustez e ampla utilização, permitindo a criação de cenários de teste variados e realistas para a avaliação prática e comparativa das ferramentas de automação. Trata-se de um serviço de streaming voltado para programas de exercícios em casa, oferecendo funcionalidades como acesso a treinos, gerenciamento de aulas e interação em comunidades.

5.3. Elaboração dos cenários de teste

A criação dos cenários de teste foi fundamentada na análise dos requisitos funcionais do sistema, com o objetivo de validar o comportamento esperado das funcionalidades disponibilizadas ao usuário. Segundo Leffingwell (2010), cenários de teste derivados de casos de uso são essenciais para simular as interações do usuário com o sistema, favorecendo a identificação de falhas e a validação da lógica implementada.

Os requisitos funcionais do sistema abrangem, entre outros aspectos, a autenticação de usuários por meio de login e logout; a gestão de conteúdos em listas personalizadas, como a adição e remoção de programas; a realização de buscas com base em diferentes critérios, como nome de instrutor, tipo de atividade e nome do programa; a aplicação combinada e a limpeza de múltiplos filtros de pesquisa; a criação e exclusão de perfis de usuário; e o redirecionamento correto para o conteúdo de vídeo correspondente. Tais requisitos definem as funcionalidades que o sistema deve oferecer, alinhando-se às necessidades dos usuários e aos objetivos da aplicação.

Com base nesses requisitos, foram elaborados testes de sistema, que simulam o comportamento do sistema como um todo a partir da perspectiva do usuário. Os cenários foram estruturados para representar diferentes fluxos de uso, garantindo que as funcionalidades operassem corretamente sob variadas condições. Cada cenário foi documentado de forma padronizada, contendo identificador, descrição, pré-condições, passos, dados de teste e resultados esperados, assegurando clareza, reprodutibilidade e eficiência no processo de validação. A automação dos testes contribuiu significativamente para ampliar a cobertura e otimizar a execução, permitindo a identificação ágil de falhas e a garantia da conformidade com os requisitos estabelecidos ao longo do desenvolvimento do sistema.

5.4. Desenvolvimento da automação de testes

Nesta etapa, foram desenvolvidos scripts de teste automatizado utilizando Cypress (v14.2.1), Selenium WebDriver (v12.1.9) e Playwright (v1.51.1). Para garantir uma comparação justa entre as ferramentas, todos os testes foram executados em um ambiente de produção controlado, utilizando a mesma aplicação web real e aplicando cenários idênticos nas três ferramentas.

A aplicação testada é uma plataforma de streaming de atividades físicas, na qual os usuários podem criar perfis, buscar programas de treinamento, aplicar filtros personalizados e assistir a vídeos sob demanda. Foram definidos 15 cenários de teste comuns, entre eles: login com sucesso, login com e-mail inválido, login com senha inválida, logout, adição e remoção de programas da lista, busca por

nome do instrutor, tipo de atividade e nome do programa, testes dos campos de busca, aplicação e limpeza de múltiplos filtros, criação e exclusão de perfil e redirecionamento correto para os vídeos.

A implementação dos scripts foi feita em JavaScript, utilizando o Node.js (v18.19.1) como runtime e o Visual Studio Code (v1.85.1) como ambiente de desenvolvimento. Os testes foram executados nos sistemas operacionais Windows, Linux e macOS, assegurando a compatibilidade entre diferentes plataformas. Os scripts foram estruturados com foco em facilidade de manutenção, clareza e padronização, seguindo boas práticas de codificação. Para controle de versão e colaboração, todo o código foi versionado e armazenado em um repositório público no GitHub. Como resultado, foram criadas suítes de teste automatizado para cada ferramenta, devidamente organizadas e documentadas.

5.5. Comparação e análise

Com base nos dados obtidos na revisão bibliográfica e no estudo empírico, foi realizada a análise comparativa entre as ferramentas, utilizando as métricas propostas por Sousa (2023) e Silva (2019), além de critérios técnicos extraídos das documentações oficiais das ferramentas, conforme descrito na seção 5 deste trabalho.

Linguagens de programação suportadas: As linguagens de programação permitem a criação de softwares, com a escrita de instruções interpretadas por sistemas computacionais. Cada linguagem possui uma curva de aprendizado específica, com algumas sendo mais fáceis de aprender que outras. Algumas ferramentas, conscientes da barreira do aprendizado, oferecem suporte a diversas linguagens ou escolhem opções mais fáceis de aprender. Avaliar as linguagens disponíveis para automação de testes é importante, pois programadores com experiência em certas linguagens podem aprender a ferramenta mais rapidamente. Quanto maior a variedade de linguagens compatíveis, mais acessível será a ferramenta.

Navegadores compatíveis: As aplicações web são executadas diretamente em navegadores, e é essencial avaliar seu comportamento em diferentes ambientes para garantir compatibilidade e uma boa experiência do usuário. Para que a automação de testes seja eficaz, as ferramentas devem permitir a execução em múltiplos navegadores, simulando interações reais dos usuários. Em projetos onde a regra de negócio exige compatibilidade com diferentes navegadores, essa métrica torna-se um critério essencial na escolha da ferramenta.

Facilidade de configuração: Algumas ferramentas dependem de outras tecnologias para funcionar corretamente, e configurações complexas ou mal documentadas podem dificultar sua utilização. Em contraste, uma ferramenta bem projetada deve simplificar essa etapa, permitindo que o foco esteja no desenvolvimento de testes automatizados, sem sobrecarregar o desenvolvedor com processos de configuração excessivos. Nesse contexto, foi avaliada a necessidade de dependências adicionais, como drivers ou bibliotecas, juntamente com a avaliação da complexidade de sua configuraçãos, pois impactam diretamente a usabilidade e a adoção das ferramentas. A configuração ideal deve ser intuitiva e eficiente, para que o desenvolvedor possa concentrar-se no que realmente importa: a criação de testes automatizados eficazes.

Recursos de depuração: Depurador de Testes é uma ferramentas que permitem depurar os testes de forma intuitiva, ajudando a identificar e corrigir falhas rapidamente, com uma visão clara do fluxo de execução e das causas dos erros.Durante a análise das ferramentas de automação de testes, foram avaliados diversos recursos que facilitam a escrita, execução e depuração dos testes. Entre os principais, destacam-se:

- **Dashboard:** Uma interface visual que fornece um resumo claro do estado dos testes, mostrando resultados, falhas e progresso da execução. Essencial para o gerenciamento eficiente dos ciclos de testes.

- **Time Travel:** Recurso que permite visualizar o estado da aplicação no momento exato da execução de um comando de teste, facilitando a investigação de falhas ao mostrar os elementos da interface naquele instante.

- **Hot Reload:** Funcionalidade que aplica automaticamente as modificações feitas no código dos testes e reexecuta-os sem precisar reiniciar a aplicação, tornando o ajuste de testes mais ágil.

- **Seletor de Elementos:** Ferramentas que facilitam a captura de identificadores dos elementos da aplicação, como IDs ou classes, garantindo que os testes interajam corretamente com esses elementos.

- **Screenshots e Logs:** Captura automática de screenshots e geração de logs detalhados durante a execução dos testes, oferecendo informações valiosas para análise de falhas e comportamento da aplicação.

- **Execução em Paralelo:** Permite a execução simultânea de múltiplos testes, reduzindo significativamente o tempo de execução dos testes.

- **Integração com CI/CD:** Facilita a integração com pipelines de Continuous Integration/Continuous Delivery, garantindo que os testes sejam executados automaticamente durante o processo de desenvolvimento.

- **Execução Remota:** Permite rodar os testes em diferentes ambientes ou servidores, possibilitando a execução dos testes em diversos contextos sem a necessidade de configuração manual em cada máquina.

- **Múltiplas Instâncias do Navegador:** A capacidade de executar testes em várias instâncias de navegador simultaneamente, garantindo cobertura em diferentes versões e configurações de navegador.

- **Interface de Execução:** A interface usada durante a execução dos testes, que deve ser intuitiva e facilitar o acompanhamento dos testes enquanto são executados.

- **Espera Automática:** Recurso que permite que a ferramenta espere automaticamente por elementos ou eventos específicos antes de executar comandos, evitando falhas por elementos que ainda não estão prontos na página.

- **Relatório de Testes:** Funcionalidade que gera relatórios detalhados dos testes, com informações sobre os resultados, falhas e tempo de execução, essenciais para a análise de performance e identificação de melhorias.

Esses recursos são essenciais para garantir a eficiência, usabilidade e manutenção dos testes automatizados, proporcionando uma forma mais prática de monitorar, depurar e melhorar os testes ao longo do processo de desenvolvimento. A análise desses recursos foi fundamental para avaliar o nível de suporte oferecido por cada ferramenta de automação de testes neste estudo.

Robustez na execução: A robustez de uma ferramenta de automação de testes está relacionada à sua capacidade de executar testes de maneira confiável, sem ser afetada por pequenas variações no comportamento da aplicação. Ferramentas mais robustas oferecem recursos como mecanismos inteligentes de espera para garantir que os elementos estejam disponíveis antes da interação, além de estratégias avançadas para identificar elementos na interface. Além disso, logs detalhados e relatórios completos ajudam a identificar a causa das falhas e a evitar falsos negativos nos testes. Dessa forma, a confiabilidade dos testes automatizados depende diretamente da capacidade da ferramenta de lidar com essas situações, garantindo que os testes falhem apenas quando há um erro real na aplicação, e não por instabilidades do ambiente de execução.

Resistência a Falhas intermitentes: Um dos principais desafios na automação de testes são as falhas intermitentes, conhecidas como *flaky tests*, que ocorrem quando um teste pode passar em uma execução e falhar em outra, mesmo sem nenhuma mudança no código ou na aplicação. Essas falhas geralmente acontecem devido a fatores como carregamento dinâmico de elementos na página, tempos de resposta imprevisíveis da aplicação ou instabilidades no ambiente de execução dos testes. Por exemplo, um teste pode falhar porque um elemento da interface ainda não foi carregado completamente, ou porque há um atraso momentâneo na resposta do servidor. Além disso, se os testes dependerem de serviços externos, como APIs de terceiros, qualquer lentidão ou falha nesses serviços pode afetar os resultados dos testes. Para reduzir esses problemas, algumas ferramentas oferecem recursos como a reexecução automática de testes falhos, garantindo maior estabilidade nos resultados.

6. Resultados

A tabela 1 apresenta um resumo dos resultados da análise comparativa entre as ferramentas Cypress, Playwright e Selenium WebDriver, com base nas métricas aplicadas. Ela indica a presença ou ausência dos componentes avaliados em cada ferramenta, permitindo uma visão geral do desempenho de cada uma nas diferentes categorias analisadas.

Tabela 1. Comparação entre ferramentas de automação de testes

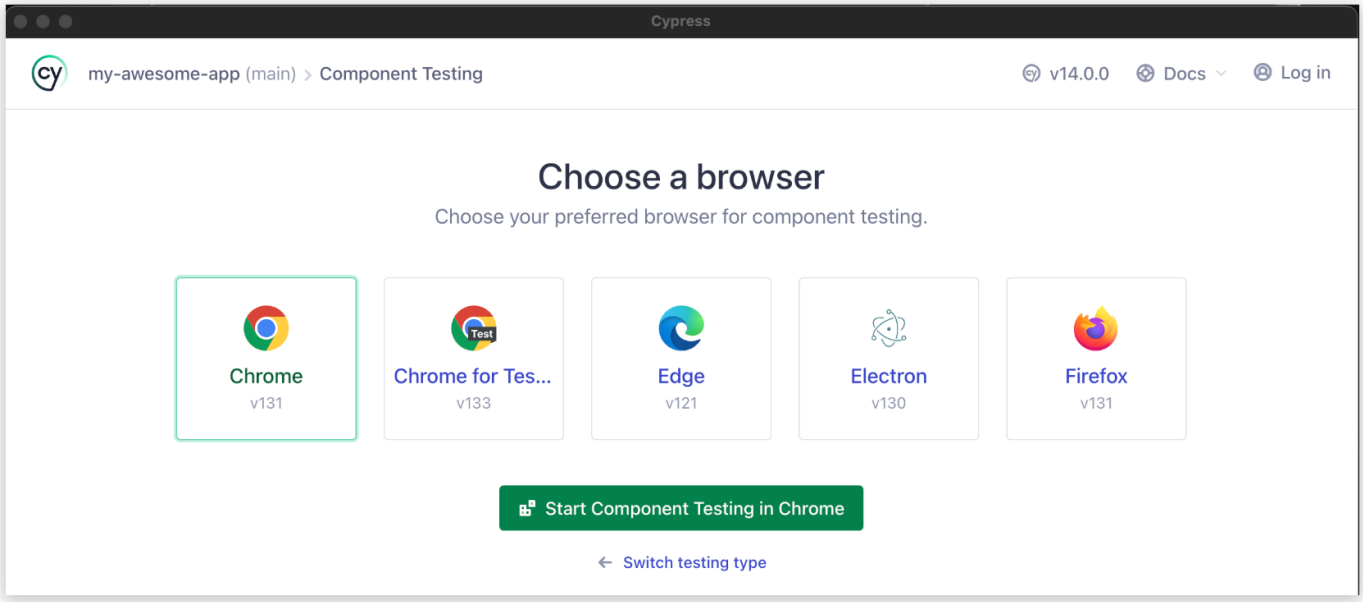
Métrica	Componente avaliado	Cypress	Playwright	Selenium WebDriver
Linguagens suportadas	JavaScript	SIM	SIM	SIM
	TypeScript	SIM	SIM	SIM
	Python	NÃO	SIM	SIM
	Java	NÃO	NÃO	SIM
	PHP	NÃO	NÃO	SIM
Navegadores compatíveis	Chrome	SIM	SIM	SIM
	Edge	SIM	SIM	SIM
	Firefox	SIM	SIM	SIM
	Safari	NÃO	SIM	SIM
	Opera	NÃO	NÃO	SIM
Facilidade de configuração	Instalação direta	SIM	SIM	SIM
	Drivers automáticos	SIM	SIM	NÃO
	Configuração simples	SIM	SIM	SIM
Recursos de depuração	Captura automática de erros	SIM	SIM	SIM
	Relatórios detalhados	SIM	SIM	NÃO
	Debug interativo	SIM	SIM	NÃO
Resistência a falhas intermitentes	Execução repetida de testes instáveis	SIM	SIM	NÃO
	Resistência a variações de tempo	SIM	SIM	NÃO
	Gerenciamento de falhas temporárias	SIM	SIM	NÃO
Robustez na execução	Estabilidade em execuções repetidas	SIM	NÃO	SIM
	Tolerância a pequenas mudanças na interface	NÃO	NÃO	SIM
	Manuseio de carregamento assíncrono	SIM	SIM	NÃO

Fonte: Elaborado pelos autores.

O suporte a diferentes **linguagens de programação** o Cypress e o Playwright oferecem suporte completo para JavaScript e TypeScript, já o Playwright inclui o suporte para Python. Em contrapartida, o Selenium Webdriver se destaca por sua maior flexibilidade, suportando mais linguagens como JavaScript, TypeScript, Python, Java e PHP, o que o torna a opção mais versátil entre as três ferramentas. O Cypress é restrito ao JavaScript, o que é uma limitação da ferramenta, pois não oferece suporte a outras linguagens de programação. O Playwright, por sua vez, oferece uma alternativa ao incluir Python. Já o Selenium WebDriver é a escolha mais abrangente, suportando uma variedade maior de linguagens e sendo ideal para projetos que utilizam diferentes tecnologias. A escolha da ferramenta mais adequada depende das linguagens e das necessidades específicas de cada projeto.

A **compatibilidade entre navegadores** é um aspecto essencial para aplicações web, garantindo que sua exibição e funcionalidade sejam consistentes em diferentes navegadores e versões. O Cypress possui uma compatibilidade mais restrita com navegadores, oferecendo suporte principalmente aos navegadores baseados no Chrome, além de incluir o Electron como uma opção integrada para testes. No entanto, sua principal vantagem é a simplicidade de configuração, já que ele detecta automaticamente os navegadores disponíveis no sistema operacional, como mostrado na figura 3, permitindo que os testes sejam executados nos navegadores instalados localmente sem necessidade de ajustes adicionais.

Figura 4. Tela de seleção do navegador.



Choose your browser

Fonte: Cypress.io.
Disponível em: cypress.io

Por outro lado, o Playwright oferece uma compatibilidade mais ampla, suportando diversos navegadores populares como Chromium, Firefox e WebKit, como mostrado na figura 4, ele vem configurado por padrão para trabalhar com três navegadores, proporcionando uma experiência pronta para uso, com a flexibilidade de ajustes diretamente no arquivo de configuração do projeto. Isso simplifica a configuração inicial e torna o Playwright uma opção atraente para projetos que necessitam de compatibilidade com múltiplos navegadores de forma fácil e rápida.

Figura 5. Configuração de Navegadores no Playwright.

```
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({
  projects: [
    /* Test against desktop browsers */
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },
    {
      name: 'webkit',
      use: { ...devices['Desktop Safari'] },
    },
    /* Test against mobile viewports. */
    {
      name: 'Mobile Chrome',
      use: { ...devices['Pixel 5'] },
    },
    {
      name: 'Mobile Safari',
      use: { ...devices['iPhone 12'] },
    },
    /* Test against branded browsers. */
    {
      name: 'Google Chrome',
      use: { ...devices['Desktop Chrome'], channel: 'chrome' }, // or 'chrome-beta'
    },
    {
      name: 'Microsoft Edge',
      use: { ...devices['Desktop Edge'], channel: 'msedge' }, // or 'msedge-dev'
    },
  ],
});
```

Fonte: Playwright.
Disponível em: playwright.dev

O Selenium WebDriver se destaca pelo amplo suporte a diversos navegadores, conforme ilustrado na Figura 5, graças à sua arquitetura baseada em WebDrivers. Essa estrutura permite que os testes sejam executados em diferentes ambientes, oferecendo maior flexibilidade e personalização. A escolha do navegador pode ser feita diretamente no código, o que facilita a adaptação dos testes

a diferentes cenários. Por exemplo, ao utilizar o Microsoft Edge, é possível inicializar o navegador com configurações específicas, permitindo um controle mais refinado da execução dos testes. Um exemplo básico de inicialização com opções personalizadas pode ser visto a seguir.

Figura 6. Inicialização do Edge no Selenium com JavaScript.

Java Python CSharp Ruby JavaScript Kotlin

```
let options = new edge.Options();
driver = new Builder()
  .forBrowser(Browser.EDGE)
  .setEdgeOptions(options)
  .build();
```

Fonte: Selenium.
Disponível em: selenium.dev

A escolha da ferramenta de automação deve levar em conta a compatibilidade com os navegadores utilizados no projeto, garantindo que os testes sejam executados de forma eficiente no ambiente desejado. Esse fator é essencial para assegurar a confiabilidade dos testes e a cobertura adequada das funcionalidades em diferentes plataforma.

Em relação à **facilidade de configuração**, foram considerados três aspectos principais: instalação direta, uso de drivers automáticos e simplicidade de configuração inicial. O Cypress se destaca por seu processo de instalação extremamente prático, bastando um único comando via npm ou yarn para que a ferramenta esteja pronta para uso. Todas as dependências vêm embutidas, o que elimina a necessidade de configurações adicionais. Essa abordagem torna o Cypress uma das ferramentas mais acessíveis para quem está começando com testes automatizados, oferecendo um ambiente integrado de testes e facilitando a execução imediata dos scripts.

O Playwright também apresenta uma instalação simplificada, exigindo poucos comandos para iniciar. Um de seus diferenciais é a instalação automática dos navegadores necessários, como Chromium, Firefox e WebKit, o que contribui para um ambiente de testes sempre atualizado e pronto para uso. Embora o processo inicial também seja bastante intuitivo, pode haver a necessidade de pequenos ajustes conforme o contexto do projeto, como a escolha da linguagem de programação, definição de diretórios de testes ou configuração de arquivos para integração contínua, especialmente em ambientes CI/CD.

O Selenium WebDriver, por sua vez, apesar de contar com uma instalação relativamente direta, requer a configuração manual de drivers específicos para cada navegador utilizado nos testes, como o ChromeDriver e o GeckoDriver. Esse processo pode representar um desafio inicial, sobretudo para usuários com menos experiência. Mesmo com o auxílio de bibliotecas como o WebDriverManager, que automatizam parte da instalação e atualização dos drivers, a configuração inicial ainda exige mais etapas em comparação com as outras ferramentas analisadas.

No que diz respeito ao uso de drivers automáticos, Cypress e Playwright oferecem soluções que reduzem significativamente a complexidade. O Cypress interage diretamente com o navegador

por meio de sua API, dispensando a configuração manual de drivers. O Playwright vai além, permitindo a instalação e atualização automática de múltiplos navegadores, o que assegura compatibilidade contínua e facilidade na execução dos testes. Em contrapartida, o Selenium exige atenção constante a mudanças nas versões dos navegadores e seus respectivos drivers, aumentando o esforço necessário para manter o ambiente funcional.

As ferramentas oferecem diferentes níveis de **suporte a recursos de depuração**. O Cypress e o Playwright se destacam pela captura automática de erros, geração de relatórios detalhados e suporte à depuração interativa, o que facilita a identificação e correção de problemas durante a execução dos testes. Por outro lado, o Selenium cuida de relatórios detalhados e de depuração interativos, focando principalmente na execução remota e na utilização de múltiplas instâncias de navegador, o que pode limitar a eficiência nas etapas de depuração.

No que diz respeito à **resistência a falhas intermitentes**, foram analisados três aspectos principais: a reexecução de testes instáveis, a tolerância a variações de tempo e o gerenciamento de falhas temporárias. O Playwright oferece suporte nativo à reexecução automática de testes considerados flakey, o que significa que testes que falham de forma ocasional por motivos temporários podem ser repetidos sem intervenção manual, garantindo maior confiabilidade. O Cypress também lida bem com esse tipo de instabilidade por meio de mecanismos de auto-retry para comandos específicos, como a busca por elementos, embora a repetição completa de testes falhos exija configurações adicionais. Já o Selenium WebDriver não possui suporte automático para reexecução, sendo necessário configurar essa funcionalidade por meio de frameworks de testes ou scripts personalizados, o que aumenta a complexidade.

Quanto à resistência a variações de tempo e ao gerenciamento de falhas transitórias, o Playwright novamente se destaca, apresentando boa tolerância a pequenas mudanças na interface ou nos tempos de carregamento, reduzindo o número de falhas causadas por atrasos pontuais. O Cypress também se mostrou estável nesse aspecto, principalmente por contar com esperas automáticas para interações mais simples, embora cenários mais complexos possam exigir ajustes manuais de tempo. Por outro lado, o Selenium WebDriver depende fortemente do uso de explicit waits, o que demanda mais configuração por parte do desenvolvedor. No gerenciamento de falhas temporárias, Playwright e Cypress contam com mecanismos que auxiliam na estabilidade dos testes, mas no caso do Selenium, esse tipo de tratamento precisa ser implementado de forma manual, tornando o processo mais suscetível a interrupções causadas por instabilidades do ambiente.

A **robustez na execução** foi analisada com base em três fatores principais: estabilidade em execuções repetidas, tolerância a pequenas mudanças na interface e manuseio de carregamento assíncrono. Nos testes, o Playwright foi o que mais apresentou problemas de estabilidade, a cada script executado, um ou outro teste que na etapa anterior tinha passado, apresentava erro posteriormente e quando executado novamente, passava. O Cypress demonstrou boa estabilidade, mas em alguns casos onde elementos mudavam dinamicamente de estado foi preciso ajustes adicionais no código dos testes. O Selenium WebDriver, também apresentou ótima estabilidade nas execuções. Quanto à tolerância a pequenas mudanças na interface o Playwright e o Cypress apresentaram erros a mudanças que envolvessem mudanças de seletores, por menor que tivesse sido essa mudança.
Já o Selenium WebDriver apresentou localizadores mais flexíveis e inteligentes para ainda garantir a execução do script. Por fim, no manuseio de carregamento assíncrono, o Playwright e o Cypress apresentaram eficiência em relação ao carregamento assíncrono, um com a sua capacidade de aguardar automaticamente a estabilidade da página antes de interagir e o outro com sua reexecução automática, respectivamente. Já nos testes com o Selenium WebDriver, a ferramenta dependeu fortemente do uso

de esperas explícitas e estratégias manuais para lidar com carregamentos assíncronos, tornando esse processo mais suscetível a erros, caso não seja configurado corretamente.

7. Conclusão e trabalhos futuros

Este estudo teve como objetivo comparar as ferramentas de automação de testes Cypress, Playwright e Selenium WebDriver, destacando suas principais métricas e diferenças. Para isso, foram analisadas métricas como linguagens de programação suportadas, navegadores compatíveis, facilidade de configuração, recursos de depuração, robustez na execução e resistência a falhas. A partir dessa análise, foi possível evidenciar os pontos fortes e limitações de cada ferramenta, permitindo uma visão mais clara sobre suas aplicações em diferentes cenários de testes automatizados.

A automação de testes se destaca como uma ferramenta indispensável, proporcionando maior agilidade, precisão e eficiência na detecção de erros. A utilização de softwares de teste automatizados é fundamental para comparar os resultados reais com os esperados, além de automatizar tarefas repetitivas que demandariam grande esforço manual. Esses sistemas desempenham um papel crucial nas operações comerciais de muitas empresas, acelerando o crescimento do mercado ao oferecer uma melhor experiência para os clientes. Embora o trabalho humano ainda seja importante, a automação reduz a necessidade de intervenção manual em testes repetitivos, eliminando redundâncias e aumentando a eficiência operacional.

A partir da revisão teórica e dos testes práticos realizados, foi possível observar que o Playwright se destacou pela robustez e suporte a múltiplos navegadores, oferecendo esperas automáticas inteligentes, melhor tolerância a falhas intermitentes e uma execução mais confiável em diferentes ambientes e também a sua compatibilidade com Chromium, Firefox e WebKit sem necessidade adicional de configuração o que acaba tornando uma opção versátil para projetos que exigem testes multiplataforma. Outro diferencial foi o suporte nativo e execução paralela o que facilitou o tempo de execução dos testes.

Enquanto isso, o Cypress apresentou vantagens em depuração e execução de forma mais rápida, sendo uma ferramenta de fácil configuração e com uma abordagem mais intuitiva para criação de testes. O suporte ao Time Travel e dashboard interativo simplificou a análise de falhas, tornando a depuração mais eficiente. Por outro lado, as limitações da ferramenta incluem suporte reduzido a múltiplos navegadores e dificuldades na automação de múltiplas janelas e abas, o que pode impactar em projetos mais robustos.

Já o Selenium, apesar de ser uma solução consolidada e amplamente utilizada, demonstrou a necessidade de maior configuração manual para alcançar um desempenho equivalente às outras ferramentas. O Selenium acaba sendo uma escolha segura quando é preciso configurar projetos que precisam de ampla compatibilidade com navegadores e execução remota. Mas a sua dependência de drivers externos, a necessidade de configurar de forma manual e a ausência de funcionalidades avançadas de depuração, tornam a ferramenta um pouco complexa.

Os resultados obtidos contribuem significativamente para a escolha mais estratégica dessas ferramentas no mercado de testes automatizados, auxiliando desenvolvedores e equipes de qualidade a optarem pela solução mais adequada para seus projetos. Além disso, a análise reforça a importância da automação de testes na garantia da qualidade de software, permitindo detecção rápida de falhas, maior eficiência no desenvolvimento e redução de custos operacionais.

Contudo, é importante destacar que esta pesquisa foi realizada dentro de um ambiente específico, podendo apresentar variações conforme o contexto e a infraestrutura utilizada. Como

sugestão para trabalhos futuros, recomenda-se a investigação dessas ferramentas em ambientes distribuídos, a integração com pipelines CI/CD complexos e a avaliação do impacto de novas atualizações nas métricas analisadas. Dessa forma, espera-se que este estudo possa servir como base de referência para profissionais da área, contribuindo para decisões mais embasadas na adoção de ferramentas de automação de testes.

Referências

BARTIE, A. **Garantia da qualidade de software**. [S.l.]: Gulf Professional Publishing, 2002.

BASTOS, Aderson; RIOS, Emerson; CRISTALLI, Ricardo; MOREIRA, Trayahu. **Base de conhecimento em teste de software.** 3. ed. São Paulo: Editora Martins, 2012.

BUSINESS RESEARCH INSIGHTS. **Automated testing software market.** 2025. Disponível em: https://www.businessresearchinsights.com/market-reports/automated-testing-software-market-103370. Acesso em: 8 abr. 2025.

BROWSERSTACK. **Playwright vs Cypress.** Disponível em: https://www.browserstack.com/guide/playwright-vs-cypress. Acesso em: 8 Abr. 2025.

CAPELLINI, L. A. Cypress: **o novo conceito em testes automatizados.** 2021. Disponível em: https://atech.com.br/cypress-o-novo-conceito-em-testes-automatizados/. Acesso em: 15 jul. 2024.

CONTRI, Maria Eduarda. **Selenium como uma ferramenta para testes de software.** 2019. Disponível em: https://medium.com/@dudacontri65/seleniumcomo-uma-ferramenta-para-testes-de-software-22541584b960. Acesso em: 25 abr. 2023.

CORRÊA, Jonas Santos; SILVA, Welington Tierry. **Elaboração de um mínimo processo viável para automação de testes de interface em fábricas de software.** 2021. Monografia (Graduação em Engenharia de Computação) – UniEVANGÉLICA, Anápolis, 2021.

CYPRESS.IO. Why Cypress? 2024. Disponível em: https://www.cypress.io/. Acesso em: 27 jul. 2024.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software.** Rio de Janeiro: Elsevier, 2017.

ESCOBAR, Daniel de Andrade; MUNIZ, Vitor Natariani. **Testes automatizados: teoria e prática. 2021.** Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Federal de Alfenas, Alfenas, 2021. Acesso em: 6 mai. 2025.

Exploring Browser Automation: **A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright (Springer Professional, 2024**). Acesso em: 10 abr. 2025

FOWLER, M. **The Practical Test Pyramid.** Martin Fowler, 2018. Disponível em: https://martinfowler.com/articles/practical-test-pyramid.html. Acesso em: 11 nov. 2019.

GONZALEZ, R. S. S.; URRIBARRI, D. K.; LARREA, M. L. **Automation tools for web testing. Beyond Selenium.** Memorias de las JAIIO, v. 8, n. 3, 2022.

HOMES, Bernard. **Fundamentals of Software Testing**. 2. ed. rev. e atual. Wiley-ISTE, 2024.

INTERNATIONAL SOFTWARE TESTING QUALIFICATIONS BOARD. (2018). ISTQB Glossary of Terms used in Software Testing (Version 3.1).

IBM (International Business Machines Corporation, 2024). **O que é teste de software?.** Disponível em: https://www.ibm.com/br-pt/topics/software-testing. Acesso em: 27 jul. 2024.

KHAN, Shahnawaz. **Advantages and Disadvantages of Cypress (End to End Testing Tool) before choosing it as your Testing Automation Tool.** 2021. Disponível em: https://skakarh.medium.com/advantages-and-disadvantages-ofcypress-end-to-end-testing-toolbefore-choosing-it-as-your-347b6436dec8. Acesso em: 26 abr. 2024.

KHETARPAL, A. **What is Cypress? Cypress Architecture, Features and Introduction.** TOOLSQA, 2021. Disponível em: https://www.toolsqa.com/cypress/what-is-cypress/. Acesso em: 27 jun. 2024.

KARALE, Jaydeep. What Is Playwright: **A Tutorial on How to Use Playwright.** 2024. Disponível em: https://www.lambdatest.com/playwright. Acesso em: 27 out. 2024.

LEFFINGWELL, D. **Agile software requirements: lean requirements practices for teams, programs, and the enterprise.** [S.l.]: Addison-Wesley Professional, 2010.

MALM, Anu. **UI-testiautomaation aloitus Robot Frameworkia hyväksi käyttäen**. Tampere: Tampere University Of Applied Sciences, 2020. Disponível em: https://www.theseus.fi/bitstream/handle/10024/349265/MalmAnu.pdf?sequence=2. Acesso em: 15 ago. 2024.

MANE, D.; BHADEKAR, G.; SALUKHE, S. **Text and keyword driven automation testing using selenium web driver.** International Research Journal of Engineering and Technology, p. 515–519, 2016.

MOLINARI, Leonardo. **Teste de Software – Produzindo sistemas melhores e mais confiáveis.** 4. ed. São Paulo: Editora Érica Ltda, 2009.

MONITORA. **Quais os tipos de testes de software e por que automatizá-los?** 11 de fev. 2019. Blog Monitora. Disponível em: https://www.monitoratec.com.br/blog/tipos-de-testes-de-software/.

MOBARAYA, F.; ALI, S. et al. **Technical analysis of selenium and cypress as functional automation framework for modern web application testing.** In: 9th International Conference on Computer Science. [S.l.: s.n.], 2019.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing.** 3. ed. Wiley, 2011.

NETO, A. **Introdução a teste de software. Engenharia de Software Magazine**, v. 1, 2007.

NAIR, Athira. **Análise da Arquitetura: Selenium, Cypress e Playwright, 2024**. Disponível em: https://www.testingmavens.com/blogs/architecture-breakdown-selenium-cypress-and. Acesso em: 27 set. 2024.

PANDY, Gokul; PUGAZHENTHI, Vigneshwaran Jagadeesan; MURUGAN, Aravindhan. **Advances in Software Testing in 2024: Experimental Insights, Frameworks, and Future Directions.** International Journal of Advanced Research in Computer and Communication Engineering, v. 13, n. 11, p. 56-72, nov. 2024. Acesso em: 6 mai. 2025

PESSOA, Clinton H. M. **Avaliação de ferramentas de automação de teste: uma análise documental e comparativa.** 2020. Trabalho de Conclusão de Curso (Graduação em Engenharia de Software) – Universidade Federal do Amazonas, Manaus, 2020. Disponível em: https://riu.ufam.edu.br/bitstream/prefix/5831/2/TCC$_C lintonPessoa.pdf$. Acesso em: 15 ago.

PLAYWRIGHT. (2024). **Playwright Features.** Disponível em: https://playwright.dev. Acesso em: 14 jul. 2024.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software: Uma Abordagem Profissional.** 9. ed. Porto Alegre: AMGH Editora, 2021.

RAMYA, P.; SINDHURA, V.; SAGAR, P. V. **Testing using selenium web driver**. 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), p. 1–7, 2017.

RIOS, E.; MOREIRA, **T. Teste de software.** [S.l.]: Alta Books Editora, 2006.

ROLLWAGEN, Andre Fernando et al. **Comparativo entre ferramentas de automação de testes de software para sistemas web.** Salão do Conhecimento – Unijuí, 2020. Disponível em: https://publicacoeseventos.unijui.edu.br/index.php/salaoconhecimento/article/view/18489/17223. Acesso em: 20 set. 2024.

SELENIUM (2024). WebDriver Documentation. Disponível em: https://www.selenium.dev/ptbr/documentation/webdriver/. Acesso em: 27 out. 2024.

SOUSA, Raíssa E. **Um estudo comparativo entre ferramentas de automação de testes: Selenium e Cypress. 2023.** Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) – Universidade Federal Do Ceará, Sobral, 2023.

SILVA, Carlos G. G. de M. **Estudo comparativo de ferramentas de testes de ponta a ponta automatizados em sistemas web.** 2019. Trabalho de Conclusão de Curso – Universidade Federal do Rio Grande do Norte, Natal, 2019.

STRAMER, Avi. (2024). **Playwright Test vs. Playwright Library.** Disponível em: https://blog.testable.io/playwright-test-vs-library/. Acesso em: 27 jan. 2025.

VELOSO, J. de S.; NETO, P. d. A. dos S.; SANTOS, I. de S.; BRITTO, R. de S. **Avaliação de ferramentas de apoio ao teste de sistemas de informação.** iSys – Brazilian Journal of Information Systems, 2010.

VIEIRA, Josimar de Souza. **Avaliação e comparação entre frameworks de desenvolvimento de testes.** Monografia (Bacharel em Ciência da Computação) – UNISUL, 2021. Disponível em: https://repositorio-api.animaeducacao.com.br/server/api/core/bitstreams/73748cd3-8728-4ec1-9f0b-0a898c2a90fc/content.

WARDHAN, Harshita; MADAN, Suman. **Study on functioning of selenium testing tool.** International Research Journal of Modernization in Engineering Technology and Science, v. 3, n. 4, p. 1342–1346, abr. 2021. Disponível em: https://www.irjmets.com/uploadedfiles/paper/volume3/issue$_{4a}pril_2021/9272/1628083377.pdf$. Acesso em: 6 maio 2025.