UNIVERSIDADE CATÓLICA DE PERNAMBUCO

UNICAP - ICAM TECH

CURSO DE CIÊNCIA DA COMPUTAÇÃO

FRAMEWORKS DE AUTOMAÇÃO DE TESTES DE SISTEMA: UM ESTUDO COMPARATIVO

Aluno:

Vinícius Lopes da Silva

Orientador:

Prof. Dr. Léuson Mário Pedro da Silva

RECIFE

Junho, 2023

FRAMEWORKS DE AUTOMAÇÃO DE TESTES DE SISTEMA:
UM ESTUDO COMPARATIVO

Trabalho de Conclusão de Curso submetido à Coordenação do Curso Bacharelado em Ciência da Computação da Universidade Católica de Pernambuco como requisito parcial à obtenção do título de Bacharel.

Área de concentração: Computação

Aprovado em: ___/___/_____.

BANCA EXAMINADORA

_____
Profa. Andréa Maria Nogueira Cavalcanti Ribeiro
UNICAP - Universidade Católica de Pernambuco

_____
Prof. Jheymesson Apolinário Cavalcanti
UNICAP - Universidade Católica de Pernambuco

_____
Prof. Marcos José Canejo Estevão de Azevedo
UNICAP - Universidade Católica de Pernambuco

SUMÁRIO

RESUMO

Durante o desenvolvimento de software, a fase de testes é responsável por garantir que o sistema desenvolvido atende às necessidades do usuário (validação) e exerce suas funções de maneira correta (verificação). Considerando a alta demanda por software, é praticamente inviável a realização de constantes validações manualmente pelo time de QA (analista de qualidade). Desta forma, a adoção e uso de testes automatizados visa permitir a redução de custos e de esforço para o time durante as validações. Por consequente, pode-se observar um amplo acervo de diferentes *frameworks*, que são utilizados durante a fase de testes para a utilização/manutenção de testes automatizados. Assim, neste trabalho busca-se avaliar diferentes *frameworks* de testes de sistema para o um ambiente web que é um dos ambiente mais comuns a serem utilizados pelos usuários. Com isso, busca-se realizar comparações entre estes *frameworks* realizando uma avaliação quantitativa de suas funcionalidades e recursos. Para tanto, uma avaliação foi realizada usando um sistema Web a partir de diferentes *frameworks* para JavaScript (Selenium, Cypress, Robot Framework). Para avaliar cada *framework*, um conjunto de métricas foi usado, tais como tempo de execução, linhas de código (LOC) e testes inválidos. Inicialmente, um estudo do sistema em análise foi feito pelo pesquisador do trabalho que possui experiência na área de QA, bem como a criação de testes usando os *frameworks* citados previamente. Por fim, baseado nas limitações reportadas, uma análise de perguntas no StackOverflow foi realizada a fim de entender como os QAs lidam com essas limitações. Como resultado, foi observado que o *framework* do Robot Framework obteve o melhor desempenho de acordo com as métricas utilizadas. Adicionalmente, foi identificado duas limitações no *framework* do Cypress durante o processo de automação que causou o impedimento para automatizar dois casos de teste para esse *framework*. Ao final deste trabalho, espera-se apoiar a comunidade de QAs, durante o processo de escolha de *frameworks* para a criação e manutenção de testes de sistemas automatizados.

Palavras-chave: Testes automatizados; Cypress; Selenium; Robot Framework;

ABSTRACT

During software development, the testing phase is responsible for ensuring that the developed system meets user requirements (validation) and performs its functions correctly (verification). Considering the high demand for software, it is practically unfeasible to perform constant manual validations by the QA (quality analyst) team. Thus, the adoption and use of automated tests aim to reduce costs and effort for the team during validations. Consequently, there is a wide range of different frameworks used during the testing phase for the creation and maintenance of automated tests. This study aims to evaluate different system testing frameworks for a web environment, which is one of the most common environments used by users. Therefore, a quantitative assessment of their functionalities and features is performed, comparing these frameworks. For this purpose, an evaluation was conducted using a web system with different JavaScript frameworks (Selenium, Cypress, Robot Framework). Each framework was evaluated based on a set of metrics, such as execution time, lines of code (LOC), and invalid tests. Initially, the researcher, who has experience in the QA field, studied the system under analysis and created tests using the aforementioned frameworks. Lastly, based on the reported limitations, an analysis of StackOverflow questions was conducted to understand how QAs deal with these limitations. The results showed that the Robot Framework performed the best according to the metrics used. Additionally, two limitations were identified in the Cypress framework during the automation process, which prevented the automation of two test cases for this framework. By the end of this study, it is expected to provide support to the QA community in the process of selecting frameworks for the creation and maintenance of automated system tests.

Keywords: Automated tests; Cypress; Selenium; Robot Framework;

1 INTRODUÇÃO

A qualidade de um *software* é fundamental para impactar de forma positiva a competitividade de uma empresa de desenvolvimento de *software* no mercado atual, bem como a qualidade do software desenvolvido (Noello, 2016). Porém, as constantes mudanças necessárias para a evolução e adequação do software para implementar novas tecnologias, pode levar a ocorrência de erros que podem impactar negativamente o produto final (Izabel, 2014).

Dessa forma, a metodologia utilizada no desenvolvimento de *software* exige um grande esforço dos membros do projeto para se adaptar às constantes mudanças que acontecem. Com isso, torna-se necessário a constante participação do QA para validar as mudanças realizadas e garantir o comportamento do software mediante as suas especificações (Vilas Boas, 2020). Neste sentido, foi criada a necessidade de conhecer e desenvolver *scripts* de automação, que possam realizar a cobertura dos testes end-to-end que são os tipos de testes que simulam a interação de um usuário em um sistema. Assim, o que antes era necessário muito mais esforço e alocação de vários profissionais de qualidade, realizando atividades manuais, agora torna-se uma atividade automatizada, utilizando o scripts automatizados e tornando a atividade mais prática (Dustin, Elfiede e Garrett, Thom, 2009).

Porém, em um mercado tão aquecido e com diversas *frameworks* diferentes de automação, verifica-se a necessidade de adquirir conhecimento (caso não possua) para avaliar os *frameworks* disponíveis e suas limitações. Com isso, o objetivo desse trabalho é realizar uma análise comparativa de três *frameworks* para automação de testes de sistema (Selenium, Cypress e Robot Framework) que são alguns dos mais utilizados no mercado atual. Como resultado, espera-se identificar as limitações e pontos positivos e negativos de cada um dos *frameworks* analisados, bem como verificar quão recorrente as limitações observadas impactam QAs a partir da mineração de fóruns online de perguntas (StackOverflow).

Para tanto, foi realizado um levantamento de casos de testes baseados nas funcionalidades do site de e-commerce da Amazon [1], que posteriormente foram automatizados em cada um dos *frameworks* analisados. Um grupo de métricas (tempo de escrita dos casos de teste, tempo de execução, linhas de código (LOC) e testes inválidos) foi selecionado para auxiliar na análise comparativa entre os *frameworks* de automação baseadas nos resultados obtidos na automação dos casos de testes. Além disso, a fim de mensurar quão frequentemente QAs experienciam as limitações identificadas no estudo, um estudo empírico

[1] https://www.amazon.com

foi realizado a partir da mineração de perguntas usando a API do StackOverflow.[2] Assim, espera-se coletar informações sobre os problemas mais recorrentes, soluções adotadas e suas variações, bem como a assiduidade da comunidade de cada *framework*.

Ao final desse trabalho, 12 casos de testes, previamente selecionados, foram automatizados permitindo uma análise de acordo com as métricas comentadas anteriormente. Diante destes resultados, o Cypress se revelou como um *framework* de natureza minimalista, requerendo uma quantidade significativamente menor de linhas de código em comparação com seus concorrentes. No entanto, uma desvantagem notável do Cypress foi o tempo de execução dos casos de teste, pois exigiu um tempo maior para concluir a execução de todos os scripts. Por outro lado, o Robot Framework se destacou positivamente nessa métrica de tempo de execução, sendo o *framework* que exigiu menos tempo para executar todos os casos de teste. Além disso, o Robot Framework também obteve destaque positivo nas métricas de tempo de escrita dos casos de teste, e também não teve nenhum caso de teste bloqueado.

No que diz respeito a limitações dos *frameworks*, duas limitações foram reportadas no *framework* do Cypress, resultando no impedimento que dois dos casos de testes fossem automatizados para esse *framework*. A fim de entender como QAs lidam com estas limitações, perguntas foram mineradas do Stack Overflow, onde foi possível verificar que estes problemas acontecem e diferentes ações são tomadas. Por exemplo, uma solução para uma das limitações encontradas seria validar a presença da URL, verificando diretamente o elemento href no HTML da página.

O restante deste documento está organizado da seguinte forma. A fundamentação teórica é localizada na Seção 3. Na Seção 4 é abordado sobre o sistema escolhido e os *frameworks* para o estudo. Além de explicar sobre as métricas utilizadas para realizar as comparações e o fluxo a ser seguido para a execução da pesquisa. Na Seção 5 são abordados os resultados para o estudo. Por fim, as ameaças à validação, os trabalhos relacionados e a conclusão foram abordados nas Seções 6, 7 e 8 respectivamente.

[2] https://stackoverflow.com/

2 EXEMPLO MOTIVADOR E FUNDAMENTAÇÃO TEÓRICA

Nesta seção é apresentado um exemplo motivador caracterizando o problema abordado neste trabalho, bem como a descrição dos principais conceitos que foram utilizados neste trabalho.

2.1 Motivação

Atualmente com software cada vez mais complexos, é comum os times de qualidade precisarem planejar diversos casos de testes. Esses casos são constantemente executados a cada novo lançamento que o produto irá realizar para poder verificar se nenhum bug/falha foi gerado pela modificações realizadas. Com a grande quantidade de casos de testes levantados pelo time de qualidade é cada vez mais inviável a execução de forma manual. Primeiramente, dependendo da quantidade de testes manuais criados, pode ser impossível que o time de QA consiga validar todos os casos antes de um lançamento, além de gerar muito esforço e um extremo gasto de tempo.

Dessa forma, é de extrema importância o conhecimento dos testadores sobre os *frameworks* presentes no mercado. A utilização de um *framework* de automação vai poder ajudar e poupar o time de QA de um trabalho de extremo desgaste. Porém, na medida que novas funcionalidades são implementadas no produto é aceitável que o *framework* possua algumas limitações ou problemas que irão impossibilitar de realizar a automação de todos os casos planejados.

Com isso, é importante possuir testadores experientes no time que possam encontrar uma solução para o problema ou um paliativo, desde implementar uma nova funcionalidade no *framework* ou utilizar uma biblioteca que seja a solução. Porém, algumas vezes o problema está localizado no *framework* escolhido e é necessário o conhecimento sobre outro *frameworks* que possam substituir o atual para evitar os problemas atuais. O conhecimento e prática em diversos *frameworks* é extremamente importante para o profissional de QA, dessa forma ele irá prever possíveis problemas de limitações e poderá identificar o melhor *framework* para ser adotado seu time.

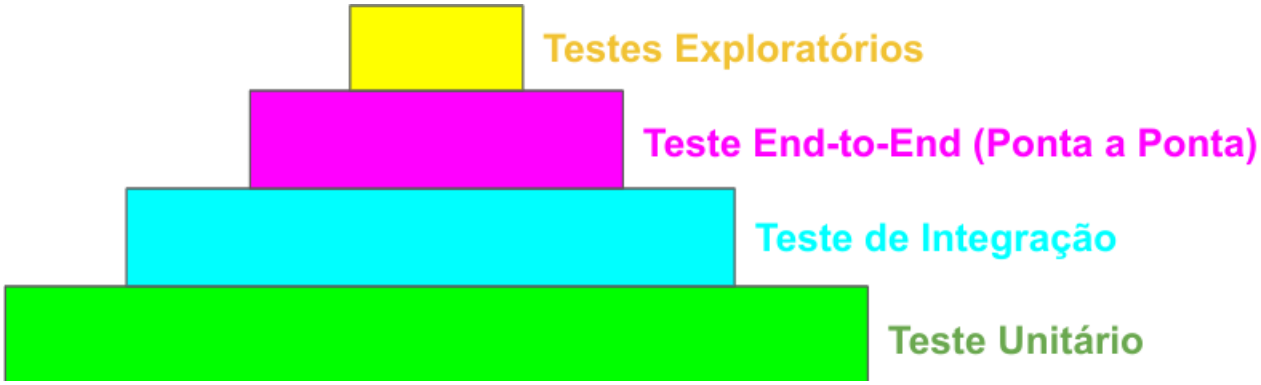
2.2 Teste de Software

O teste de *software* é um processo que é executado antes, durante e depois do desenvolvimento do *software*. Esse processo tem como objetivo principal identificar possíveis falhas/bugs que necessitam ser corrigidos antes que o *software* seja disponibilizado para os

usuários. Além de identificar mal funcionamento de alguma funcionalidade, a etapa da validação do *software* é importante para observar se algum requisito proposto não foi desenvolvido/entregue pelo time de desenvolvimento (Izabel, 2014).

Dessa forma, podemos dizer que esse processo é responsável por medir a qualidade do *software* como um todo e tomar a decisão se o sistema está apto para ser lançado ou não. Para isso o time de qualidade utiliza alguns diferentes tipos de testes que têm diferentes objetivos para poder garantir a qualidade do *software*.

Figura 1: Pirâmide de Testes



Fonte: Elaborado pelo Autor.

A base da pirâmide é onde se deve precisar de menos tempo para executar e será menos custoso para o time, além de ser o setor que possui uma maior quantidade de testes criados. Quanto mais vai se aproximando do topo da pirâmide, então os testes vão precisar de mais tempo para execução (Valente, 2020). As próximas seções deste trabalho vão explicar detalhadamente qual o objetivo de cada parte da pirâmide.

2.2.1 Teste de Software

O teste unitário é a base da pirâmide e são executadas muito rapidamente e com um custo muito baixo. Essa é a forma de testar partes individuais do código (funções, métodos, classes e etc). Essa é a menor parte do sistema que é testada e tem como objetivo verificar se falhas/bugs não foram gerados quando são realizadas mudanças dentro de alguma parte específica do código. Esse tipo de teste é responsabilidade do time de desenvolvimento (Cachoni, Mônica e Vilas Boas, Carol, 2020).

2.2.2 Teste de Integração

O teste de integração é a integração de diversos módulos do sistema para verificar se a junção de diversas partes diferentes do código foram desenvolvidas de forma separada, se quando unidas serão executados corretamente (Noello, 2016). O objetivo desse tipo de teste é verificar a conformidade do sistema com os requisitos funcionais, como uma comunicação com banco de dados e micro serviços. Esse tipo de teste é responsabilidade do time de desenvolvimento.

2.2.3 Teste End-to-End (Ponta a Ponta)

O teste end-to-end verifica o fluxo do funcionamento do sistema do começo ao fim. O seu objetivo é identificar se a integridade dos dados está sendo mantida para os diversos componentes que formam o sistema (Clerissi et al.., 2017). Esse tipo de teste tenta simular um comportamento real do usuário e é um ótimo indicador já que verifica a aplicação como um todo, porém é uma execução que precisa de mais recursos, como um gasto de tempo a mais para ser executado pelo time de qualidade e um maior esforço em comparação com os tipos de testes anteriores. Esse tipo de teste é responsabilidade do time de qualidade do produto.

2.2.4 Testes Exploratórios

Os testes exploratórios são realizados em grande parte para poder aprender sobre uma funcionalidade específica ou até mesmo para se conhecer o sistema como um todo. Esse tipo de teste não possui passos específicos, a pessoa que está executando deve interagir com o sistema baseado em seu conhecimento geral do sistema ou de algum parecido (Vilas Boas e Gallardo et al, 2020). Esse tipo de teste é utilizado também para poder interagir livremente com o sistema e poder encontrar novas falhas/bugs que não são encontrados com tanta frequência quando é seguido passos concretos de alguns casos de testes específicos. Esse tipo de teste é muito importante para medir a qualidade do produto, mas como não tem um escopo definido, ele é muito exaustivo para execuções constantes . Esse tipo de teste pode ser executado tanto pelo time de qualidade quanto pelo time de desenvolvimento ou por qualquer outra pessoa que queira adquirir conhecimento sobre um determinado produto.

2.2.5 Testes Automatizados

Os testes manuais são muito bons para poder realizar as validações dos sistemas. É incomparável que a execução dos casos de testes no formato manual feita por uma pessoa, é muito grande as chances que possíveis erros/bugs sejam encontrados e evitados, já que no formato manual os passos seguidos sempre vão ser executados de forma diferentes de uma pessoa para outra (Vilas Boas e Gallardo et al, 2020). Porém considerando a alta demanda por *software* cada vez mais estáveis e confiáveis, é extremamente importante que a maior quantidade de testes sejam executados para verificar a estabilidade do sistema. Para atingir uma quantidade elevada de execuções é necessário a adoção dos testes automatizados que são scripts que irão simular os passos que o time de QA (analista de qualidade) deveria realizar para fazer a validação dos casos de teste.

A vantagem dessa abordagem é que um número maior de casos de testes vão poder ser executados em um curto espaço de tempo de sem demandar uma grande exigência do time de QA, mas para isso é importante que a manutenção dessas automações sejam feitas de forma constante para garantir que os resultados são confiáveis e positivos. Dessa forma, o time de QA vai poder de forma constante verificar possíveis problemas causados nas funcionalidades e os possíveis causadores (Dustin, Elfiede e Garrett, Thom, 2009). Em resumo, os testes automatizados ajudam o time a evitar o trabalho repetitivo, facilita o retorno sobre os resultados e o funcionamento do sistema e economizar tempo que podem ser investidos na prática de outro tipo de testes como o exploratório para fazer um complemento na verificação do produto.

A desvantagem dos testes automatizados é relacionado aos passos que o script percorre que sempre serão os mesmos independente do momento da execução, ao contrário do manual que os passos devem ser executados de forma diferente de acordo com o QA que está executando, dessa forma falhas/bugs que não estão ocorrendo nos passos que são cobertos pela automação, nunca serão encontrados pelos passos dos testes automatizados (Watkins, 2009).

2.3 Selenium WebDriver

Selenium é um dos *frameworks* mais utilizados e conhecidos no mercado de desenvolvimento de testes automatizados para os testes *end-to-end* em sistemas Web. Ela pode ser utilizada usando diversas linguagens como Java, Python, C# e muitas outras. O WebDriver é uma derivação do selenium que utiliza uma API compacta e orientada a objetos

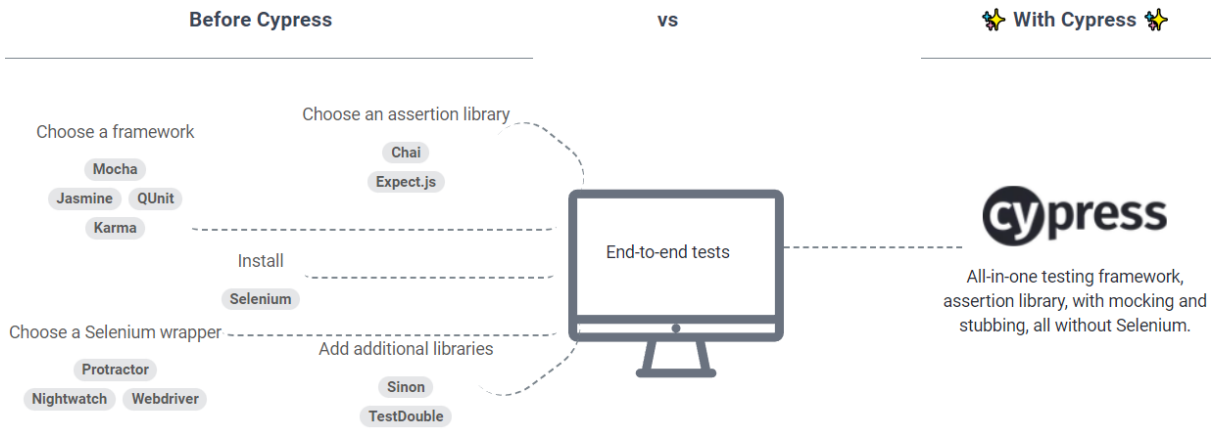
e é um *framework* que necessita da instalação de diversas bibliotecas para poder adquirir as funcionalidades que serão utilizadas (SELENIUM, 2022).

2.4 Cypress

Cypress é um *framework* de automação que tem como objetivo revolucionar o mercado de desenvolvimento de testes automatizados. Cypress utiliza o Node JS como interpretador de sua linguagem JavaScript. Esse *framework* foi totalmente baseada em uma arquitetura completamente diferente do Selenium, como a maior parte dos *frameworks* de automação são baseadas em Selenium, então o Cypress traz uma arquitetura completamente diferente para poder evitar os problemas frequentes que seus concorrentes possuem (CYPRESS 2021).

O Cypress possui um painel que exibe a execução da automação durante o desenvolvimento, então o desenvolvedor de testes automatizados pode durante a escrita da automação já verificar como está ocorrendo a execução e se está causando algum erro na compilação ou no resultado final do teste (CYPRESS 2021). Outra vantagem do Cypress é evitar que seja necessária a instalação de diversas bibliotecas para serem utilizadas na automação, o Cypress possui diversas bibliotecas mais utilizadas executando de forma conjunta, como pode ser observado na seguinte imagem:

Figura 2: Comparação entre os demais *frameworks* e o Cypress



Fonte: https://www.cypress.io/how-it-works

2.5 Robot Framework

O Robot Framework é um *framework* de automação de testes que é baseado em Python e ele utiliza diversas bibliotecas que são implementadas em Python, Java ou em outras linguagens para poder facilitar o trabalho do desenvolvedor de testes como a biblioteca do *SeleniumLibrary* que é um biblioteca do Selenium. Esse *framework* foi desenvolvido pela Nokia e é comumente utilizado até hoje, porém agora ele é apoiado e desenvolvido pela fundação Robot Framework que conta com o apoio de outras empresas para financiar o desenvolvimento e manutenção do *framework* (ROBOT 2017).

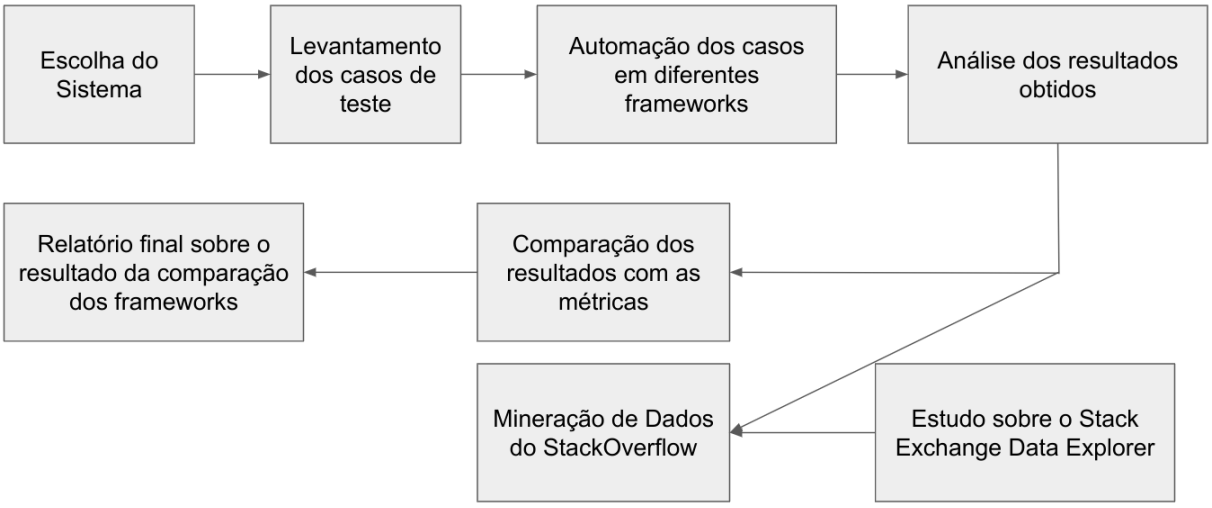
Esse *framework* é fácil de ser utilizada inclusive por usuário que não possuem conhecimento de programação, já que ela utiliza abordagens keyword-driven, data-driven e/ou behaviour-driven (BDD) que diminui a necessidade de precisar implementar comandos de linguagens de programação, já que essas abordagens será necessário somente declarar no arquivo do teste a frase utilizada para implementar as ações que serão executadas e as ações são escritas em um outro arquivo somente com a utilização das funções (click, wait, open, close e etc). Porém, para implementar uma nova biblioteca ou compreender alguma existente será necessário o conhecimento de linguagem de programação (ROBOT 2017).

3 METODOLOGIA

Nesta seção, é apresentada a metodologia adotada para a realização deste trabalho. Neste sentido, uma análise comparativa entre *frameworks* de automação de testes de sistema foi realizada, e, portanto, a identificação da comparação entre os benefícios e limitações entre si. A Figura 3 apresenta o conjunto de atividades identificadas, e o fluxo no qual as mesmas foram realizadas.

Cada etapa da pesquisa encontra-se representada na Figura 3 por meio de um retângulo, onde cada etapa corresponde a uma subseção nesta seção. Assim, para cada subseção foi apresentado o que foi feito naquela etapa, e a motivação e objetivo associados. A ordem de execução das etapas é sequencial, indicado pelo fluxo das setas.

Figura 3 - Fluxo da Metodologia



Fonte: Elaborado pelo Autor.

3.1 Sistema

Nesta etapa, foi realizada a escolha do sistema utilizado como base para o planejamento dos casos de testes automatizados. Para tanto, uma busca foi feita baseada em alguns requisitos, como explicados a seguir. Inicialmente, é necessário um sistema, que seja usado por uma boa base de usuários. Faz-se necessário um sistema popular, considerando a necessidade de automatizar funcionalidades de significância elevada, valorizando a importância da automação e o seu significado para o projeto. Outro requisito importante diz respeito às tecnologias usadas para a implementação do sistema; assim, espera-se um sistema

que recorra a tecnologias atuais, como JavaScript ou React para o desenvolvimento do front-end, viabilizando a elaboração de casos de teste fundamentais.

O sistema escolhido para ser utilizado neste trabalho foi o site de *e-commerce* da Amazon (https://www.amazon.com). Essa empresa teve um grande crescimento nos últimos anos e sua plataforma é um dos sistemas de vendas mais comuns, ela é responsável por 45% das vendas de forma digital nos Estado Unidos.

O conhecimento sobre a plataforma e a constante utilização de sistemas de *e-commerce* irão auxiliar o testador a realizar o planejamento dos casos de teste e irá qualificar a importância da automação como foi dito anteriormente.

3.2 Planejamento e Criação de Testes

Com base no conhecimento adquirido na utilização do sistema e na execução de testes exploratórios, o pesquisador criou alguns casos de teste que cobrem as principais funcionalidades utilizadas por um *framework* de automação. Uma planilha (Test Map) foi criada para adicionar os casos pretendidos com suas classificações de prioridade conforme a importância do teste para o sistema. Um arquivo Docs (Test Plan) foi criado para determinar os passos acerca de cada teste, bem como para padronizar as navegações e validações utilizadas em cada um dos *frameworks*. Desta forma, foi possível garantir que os testes fossem implementados de forma similar.

Os casos de teste foram planejados visando validar diferentes funcionalidades dos *frameworks* importantes para automatização de testes para sistemas web, como interagir com diferentes elementos HTML em uma página web, *mouseover*, *scroll down e up* na página, *upload* de arquivos e outros. Essa abordagem foi escolhida já que o sistema da Amazon deve possuir diversos casos de teste e automações para cobrir as funcionalidades do produto por se tratar de uma plataforma muito grande e popular, então não é necessário criar casos que tenham o objetivo de validar as funcionalidades do sistema escolhido.

Depois da etapa de criação dos casos de testes, o pesquisador iniciou a etapa da automatização, onde foi automatizado todos os casos de testes criados na etapa anterior. Para a realização da automação, foram escolhidos os *frameworks* Selenium, Cypress e Robot, como *frameworks* de estudo para esse trabalho.

3.2.1 Levantamento dos Casos de Teste

Depois da etapa de exploração manual do sistema, foi iniciado o processo de levantamento dos casos de teste. Nesse momento, foi escrita a descrição dos 12 casos de teste

na planilha do Test Map (Apêndice Online) e a área do produto que está sendo coberta pelo mesmo, além de informar sua prioridade para o sistema. Para realizar a classificação da prioridade do teste, o pesquisador (QA do projeto) analisou qual caso a seguir melhor representa a importância do teste em análise:

- P5 - Indica que o teste cobre uma funcionalidade extremamente importante do sistema, ou seja, uma funcionalidade indispensável para o funcionamento do produto.
- P4 - Indica que o teste cobre uma funcionalidade importante do sistema, ou seja, uma funcionalidade significativa para o produto, mas que não impede completamente o seu funcionamento.
- P3 - Indica que o teste cobre uma funcionalidade não fundamental do sistema, ou seja, uma funcionalidade que não vai impactar de grandes formas o cliente que utiliza o produto ou é pouco provável que deva impactar o cliente.
- P2 - Indica que o teste cobre uma funcionalidade trivial do sistema, ou seja, uma funcionalidade que não vai causar nenhuma perda/dano para o cliente. Essa deve ser uma funcionalidade que não seja muito notória na utilização do produto.

Tabela 1 - Casos de testes e seus objetivos

Casos de Teste	Objetivo do Teste
Abrir um produto em uma nova aba	Validar se o *framework* consegue controlar e interagir com mais de uma aba do navegador abertas ao mesmo tempo durante a execução
Acessar o site da amazon via serviço de busca	Verificar se o *framework* consegue utilizar diferentes domínios de página durante a execução do caso de teste
Navegar pelo pelo menu de navegação lateral	Verificar se o *framework* consegue interagir com o menu lateral do site para validar os clique e localização dos elementos do menu
Navegar pelo menu suspenso das opções do usuário	Verificar se o *framework* consegue interagir com o modal do usuário que somente aparece quando faz um *mouseover* no elemento do menu
Navegar pelo widget tipo Slider	Verificar se o *framework* consegue interagir com o menu slider que é um menu que os botões somente são visíveis após um *mouseover*
Interagir com o modal de endereços salvos	Verificar se o *framework* consegue focar e interagir com o modal que aparece na tela
Usar o filtro de busca antes de realizar uma pesquisa	Verificar se o *framework* consegue interagir com o campo *select* do filtro de busca que se consegue selecionar qualquer uma das opções
Autocomplete durante uma pesquisa	Verificar se o *framework* consegue visualizar e interagir com as opções do autocomplete da barra de busca
Scroll down e up na tela de um produto	Verificar se o *framework* consegue realizar *scroll down e up* na página e se consegue fazer a validação pela profundidade da página
Adicionar novos filtros para o resultado dos produtos	Verificar se o *framework* consegue interagir selecionando o checkbox para aplicar um filtro nos resultados dos produtos
Interagir com as funcionalidades do reprodutor de vídeo dos produtos	Verificar se o *framework* consegue interagir com o reprodutor de vídeo que existe na página e se consegue interagir com os ícones de reprodução sem nenhum problema de visibilidade
Escrever uma avaliação para um produto	Verificar se o *framework* consegue interagir com uma espécie de formulário realizando *upload* de imagem, escrevendo no campo de texto e submetendo a avaliação

Fonte: Elaborado pelo Autor.

3.2.2 Automação dos Casos de Testes

Após o levantamento dos casos de teste, e com o preenchimento da planilha de casos de testes, foi realizada a escrita dos casos de testes automatizados em cada um dos *frameworks* analisados neste estudo. Os testes foram automatizados do mais prioritário (P5) para o menos prioritário (P2), sempre intercalando entre a automação do *frameworks* 1 para o *framework* 2 e os passos utilizados para implementar a automação foi obedecendo o que foi determinado no Test Plan (Apêndice Online) pelo pesquisador. Após a automação de cada caso para um determinado *framework*, o pesquisador atualizou o *status* final da execução do caso de teste para a coluna do *framework* que foi utilizado (Figura 4), seguindo esses significados:

- Não Automatizado - Indica que o caso de teste ainda não foi automatizado (Status padrão).
- Automatizado - Indica que o caso de teste foi automatizado completamente.
- Automação Bloqueada - Indica que o caso de teste não pode ser automatizado completamente por uma limitação do *framework*. Deve ser colocado um comentário na última coluna da planilha para descrever o motivo.
- Falha - Indica que o teste não pode ser automatizado devido a um bug/falha do sistema.

3.3 Frameworks de Testes

Os *frameworks* selecionados para essa pesquisa foram Selenium WebDriver, Cypress e Robot Framework. Esses *frameworks* foram selecionados por possuírem mais perguntas criadas na plataforma do StackOverflow. O Selenium WebDriver possuía 52.423 perguntas criadas na plataforma, enquanto que o Cypress possuía 9.554 perguntas e o Robot Framework possuía 6.699 perguntas. Essas consultas foram realizadas no dia 02/06/2023 utilizando a plataforma do Stack Exchange Data Explorer.[3] Esses *frameworks* foram utilizados em diferentes linguagens conforme a utilização mais comum pela comunidade e recomendada na documentação. O Selenium WebDriver foi utilizado com a linguagem Java, Cypress foi utilizado com a linguagem JavaScript e o Roboto Framework foi utilizado com a linguagem Python. Para informações mais detalhadas sobre esses *frameworks*, deve ser consultada a seção de Fundamentação Teórica.

[3] https://data.stackexchange.com

3.3.1 Selenium WebDriver

Para realizar a automação com Selenium WebDriver foi utilizada a linguagem Java para escrever o código da automação. Os testes foram executados utilizando o navegador do Google Chrome (versão 110) sendo o navegador mais utilizado no mercado, segundo a pesquisa do StatCounter,[4] e a IDE escolhida foi IntelliJ IDEA (Community Edition versão 17.0.6), sendo uma IDE muito utilizada no mercado, segundo a pesquisa do Stack Overflow Developer Survey 2022,[5] onde o IntelliJ ficou em 3º lugar na preferência dos programadores. Para a utilização do Selenium WebDriver com o navegador escolhido, foi necessário fazer o download do ChromeDriver (versão 110.0.5481.77 ou mais atual) sendo o driver do Selenium para utilização do Google Chrome.

3.3.2 Cypress

Para realizar a automação com Cypress foi utilizada a linguagem JavaScript para escrever o código da automação, essa é a linguagem utilizada pelo *framework*. Os testes foram executados utilizando o navegador do Google Chrome (versão 110) sendo o navegador mais utilizado no mercado, segundo a pesquisa do StatCounter, e a IDE escolhida foi Visual Studio Code (versão 1.73 ou mais atual), a IDE mais utilizada no mercado, segundo a pesquisa do Stack Overflow Developer Survey 2022, onde o Visual Studio Code foi eleito a IDE preferida dos programadores.

3.3.3 Robot Framework

Para realizar a automação com Robot Framework (versão 6.0.1), foi utilizada a linguagem Python para escrever o código da automação; essa é a linguagem mais utilizada para esse *framework*. Os testes foram executados utilizando o navegador do Google Chrome (versão 110), navegador mais utilizado no mercado, segundo a pesquisa do StatCounter, e a IDE escolhida foi Visual Studio Code (versão 1.73 ou mais atual), a IDE mais utilizada no mercado, segundo a pesquisa do Stack Overflow Developer Survey 2022, onde o Visual Studio Code foi eleito a IDE preferida dos programadores.

3.4 Métricas

Para realizar a comparação entre os *frameworks* escolhidos, foram utilizadas as métricas de Tempo de Execução, Linhas de Código (LOC) e Testes inválidos. Essas métricas

[4] https://gs.statcounter.com/, Acessado em 06/06/2023
[5] https://survey.stackoverflow.co/2022/#integrated-development-environment

foram selecionadas baseando-se na análise das métricas utilizadas pelos trabalhos e relacionados, verificando quais métricas deveriam ter sido incluídas para participarem desse estudo. Essas métricas auxiliaram a determinar qual *frameworks* possuiu o melhor desempenho para cada uma das métricas.

3.4.1 Tempo de Escrita dos Casos de Teste

Essa métrica foi responsável por medir o tempo necessário para automatizar cada caso de teste para cada um dos *frameworks*. Após a execução de cada caso, a aba "Tempo de Escrita dos Casos de Teste" da planilha (Figura 5) foi atualizada com o tempo necessário em minutos (min) para a automação de cada caso de teste. O tempo utilizado para automação de cada caso foi contabilizado do momento que começa a trabalhar na automação até o momento que a automação é finalizada e que esteja sendo executada com sucesso. No final da aba da planilha foi calculado o tempo total em minutos que cada um dos *frameworks* necessitou para a implementação de todos os casos de teste. O objetivo desta métrica é poder observar qual o *frameworks* que demandou mais esforço para ser automatizado por necessitar de mais tempo de dedicação do pesquisador.

3.4.2 Tempo de Execução

Essa métrica foi responsável por medir o tempo necessário para executar cada caso de teste para cada um dos *frameworks*. Após a execução de cada caso, a aba "Tempo de Execução dos Frameworks" da planilha (Figura 6) foi atualizada com o tempo necessário em segundos (s) para a execução completa do caso de teste. O valor do tempo de execução em segundos é informado no terminal da execução do teste. No final da aba da planilha foi calculado o tempo total em segundos que cada um dos *frameworks* necessitou para a execução de todos os casos de teste. O objetivo desta métrica é poder observar qual o *frameworks* que executa os casos de testes de forma mais rápida.

Todos os casos de teste foram executados utilizando o mesmo ambiente para não haver discrepância causada pela execução em diferentes ambientes (Tabela 2)

Tabela 2 - Informações da máquina de execução dos testes

Computador (modelo)	Dell Latitude 5511
Processador	Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz
Memória	32GB 3200MHz DDR4
Sistema Operacional	Fedora Linux 35
Navegador	Chrome 110

Fonte: Elaborado pelo Autor.

Figura 6 - Tempo de Execução dos Frameworks

	A	B	C	D	E	F	G	H
1				**Descrição do Teste**	**Selenium (ms)**	**Cypress (ms)**	**Robot Framework (ms)**	**Prioridade**
2								
3	Acessar a Plataforma							
4		Criar Conta						
5				Teste 1	0	0	0	P5
6				Teste 2	0	0	0	P4
7		Login						
8				Teste 1	0	0	0	P5
9				Teste 2	0	0	0	P3
10		Log Out						
11				Teste 1	0	0	0	P5
12				Teste 2	0	0	0	P2
13	Tempo Total (ms)				0	0	0	

Fonte: Elaborado pelo Autor.

3.4.3 Linhas de Código (LOC)

Essa métrica foi responsável por determinar a quantidade de linhas que cada caso de teste necessitava para ser escrito. Após a automação de todos os casos de teste, a aba “LOC dos Frameworks” da planilha (Figura 7) foi atualizada com a quantidade de linhas que foi preciso para cada um dos casos de teste. Esse valor foi obtido pelo pesquisador contando as linhas de cada um dos testes automatizados. No final da aba da planilha foi calculada a quantidade de linhas que cada um dos *frameworks* necessitou para a automação de todos os casos de teste. O objetivo desta métrica é avaliar qual o *framework* que exige menos linhas para automatizar os testes.

Figura 7 - LOC dos Frameworks

	Descrição do Teste	Selenium	Cypress	Robot Framework	Prioridade
Acessar a Plataforma					
Criar Conta					
	Teste 1	0	0	0	P5
	Teste 2	0	0	0	P4
Login					
	Teste 1	0	0	0	P5
	Teste 2	0	0	0	P3
Log Out					
	Teste 1	0	0	0	P5
	Teste 2	0	0	0	P2
Total de Linhas		0	0	0	

Fonte: Elaborado pelo Autor.

3.4.4 Testes Inválidos

Essa métrica foi responsável por verificar qual o *framework* que obteve a maior taxa de testes inválidos. Os testes inválidos são testes que por algum motivo não foram possíveis de automatizar, os motivos podem ir de uma limitação da plataforma, falta de conhecimento do testador sobre as funcionalidades do *framework* ou uma limitação do *framework*. Por exemplo, se um determinado teste precisa abrir uma segunda aba no navegador, alguns *frameworks* podem não possuir uma funcionalidade que abra uma nova aba. Os testes inválidos foram os testes que estiveram com o status de automação como "Automação Bloqueada" na aba "Testes inválidos dos Frameworks" da planilha (Figura 8). Então o seguinte cálculo foi utilizado através das informações da aba da planilha e foi exibido o resultado no final da mesma:

$$PorcentagemInválidos = \left(\frac{NúmeroTestesInválidos}{NúmeroTotalTestes}\right) \times 100$$

Figura 8 - Testes inválidos dos Frameworks

	Descrição do Teste	Selenium	Cypress	Robot Framework	Prioridade
Acessar a Plataforma					
Criar Conta					
	Teste 1	Não Automatizado	Não Automatizado	Não Automatizado	P5
	Teste 2	Não Automatizado	Não Automatizado	Não Automatizado	P4
Login					
	Teste 1	Não Automatizado	Não Automatizado	Não Automatizado	P5
	Teste 2	Não Automatizado	Não Automatizado	Não Automatizado	P3
Log Out					
	Teste 1	Não Automatizado	Não Automatizado	Não Automatizado	P5
	Teste 2	Não Automatizado	Não Automatizado	Não Automatizado	P2
Porcentagem de Testes Inválidos		0.00%	0.00%	0.00%	

Fonte: Elaborado pelo Autor.

3.5 Relatórios

Após o preenchimento das abas da planilha referente a cada métrica, os resultados das abas individuais de cada métrica foram adicionados na aba final dos resultados (Relatório Final dos Frameworks, Figura 9). Nessa aba, apresenta-se um resumo dos resultados coletados nas métricas para cada um dos testes automatizados. Dessa forma, tivemos uma visão na planilha da comparação de cada um dos testes para os diferentes *frameworks* e tivemos um visão da comparação entre os diferentes *frameworks*.

Figura 9 - Relatório Final dos Frameworks

	A	B	C	D	E	F	G	H	I	J
1	**Framework**	**Descrição do Teste**	**Tempo de execução (ms)**	**LOC**	**Detecção de Bugs**	**Testes Inválidos**	**Indice de Severidade**	**Prioridade**	**Arquivo do Teste**	**Nome do Teste**
2										
3	Selenium	Teste 1	0	0	Não	Não Automatizado	0	P5	TestFile1	TestName
4		Teste 2	0	0	Não	Não Automatizado	0	P4	TestFile1	TestName
5		Teste 3	0	0	Não	Não Automatizado	0	P5	TestFile2	TestName
6		Teste 4	0	0	Não	Não Automatizado	0	P3	TestFile2	TestName
7										
8	Cypress	Teste 1	0	0	Não	Não Automatizado	0	P5	TestFile1	TestName
9		Teste 2	0	0	Não	Não Automatizado	0	P4	TestFile1	TestName
10		Teste 3	0	0	Não	Não Automatizado	0	P5	TestFile2	TestName
11		Teste 4	0	0	Não	Não Automatizado	0	P3	TestFile2	TestName
12										
13	Robot Framework	Teste 1	0	0	Não	Não Automatizado	0	P5	TestFile1	TestName
14		Teste 2	0	0	Não	Não Automatizado	0	P4	TestFile1	TestName
15		Teste 3	0	0	Não	Não Automatizado	0	P5	TestFile2	TestName
16		Teste 4	0	0	Não	Não Automatizado	0	P3	TestFile2	TestName
17										
18	Resumo do Resultado									
19		Selenium	0	0	0	0	0			
20		Cypress	0	0	0	0	0			
21		Robot Framework	0	0	0	0	0			

Fonte: Elaborado pelo Autor.

Com isso, foi possível alcançar o objetivo final desse trabalho que seria identificar qual *framework* tem um desempenho melhor em comparação com cada uma das métricas e sinalizar as limitações de cada um dos *frameworks*.

3.6 Mineração no StackOverflow

Foi realizada uma mineração dos dados presentes de perguntas e ocorrências na plataforma do StackOverflow baseado nas limitações e dificuldades encontradas durante a fase de utilização dos diferentes *frameworks* para poder identificar como a comunidade soluciona as limitações observadas.

3.6.1 Estudo sobre o Stack Exchange Data Explorer

Foi necessário realizar um estudo para compreender o funcionamento do Stack Exchange Data Explorer que é uma ferramenta de consulta de dados públicos, que pode ser utilizado para realizar consultas aos dados do StackOverflow. Dessa forma, foi possível identificar os parâmetros necessários para realizar uma consulta e como operar as buscas.

3.6.2 Mineração de Dados do StackOverflow

Com o intuito de verificar como a comunidade soluciona os problemas encontrados pelo testador durante a utilização dos *frameworks* investigados, consultas foram realizadas usando o Stack Exchange Data Explorer. Desta forma, foi possível quantificar o número de ocorrências que os testadores utilizaram a plataforma em busca de alguma solução para as limitações observadas. Com isso, foi observado a quantidade de perguntas feitas sobre o problema e as soluções para contornar a limitação.

4 RESULTADOS

Nesta seção foram apresentados os resultados deste trabalho. Para tanto, foram planejados a automatização de 12 casos de testes, onde foi possível explorar as funcionalidades principais dos *frameworks* analisados neste estudo. Dos 12 casos de teste planejados no estudo, somente 2 não puderam ser automatizados; nestes casos, devido à limitação do *framework* Cypress. Para cada caso de teste automatizado, métricas foram computadas a fim de permitir a comparação entre os resultados obtidos pelos *frameworks* analisados, dentre elas: tempo de escrita dos casos de teste, tempo de execução, linhas de código (LOC) e testes inválidos.

4.1 Casos de Teste

A Tabela 3 apresenta os 12 casos de teste criados, bem como os status da automação de cada caso em relação aos *frameworks* selecionados para a pesquisa. Os casos de teste selecionados buscam validar diferentes funcionalidades dos *frameworks,* baseados no suporte para automatização de testes para sistemas web. De maneira geral, foi explorado como identificar e interagir com diferentes elementos HTML em uma página web, *mouseover*, *scroll down* e *up* na página, *upload* de arquivos, interação envolvendo mais de uma aba do navegador, para mais detalhes verificar a seção “Planejamento e Criação de Testes”. Os passos planejados para fazer a navegação e as validações de cada caso de teste podem ser analisados no Test Plan da pesquisa (Apêndice Online).

Tabela 3 - Casos de teste e status da automação

Casos de Teste	Selenium	Cypress	Robot
Abrir um produto em uma nova aba	Automatizado	Bloqueado	Automatizado
Acessar o site da amazon via serviço de busca	Automatizado	Bloqueado	Automatizado
Navegar pelo pelo menu de navegação lateral	Automatizado	Automatizado	Automatizado
Navegar pelo menu suspenso das opções do usuário	Automatizado	Automatizado	Automatizado
Navegar pelo widget tipo Slider	Automatizado	Automatizado	Automatizado
Interagir com o modal de endereços salvos	Automatizado	Automatizado	Automatizado
Usar o filtro de busca antes de realizar uma pesquisa	Automatizado	Automatizado	Automatizado
Auto Complete durante uma pesquisa	Automatizado	Automatizado	Automatizado
Scroll down e up na tela de um produto	Automatizado	Automatizado	Automatizado
Adicionar novos filtros para o resultado dos produtos	Automatizado	Automatizado	Automatizado
Interagir com as funcionalidades do reprodutor de vídeo dos produtos	Automatizado	Automatizado	Automatizado
Escrever uma avaliação para um produto	Automatizado	Automatizado	Automatizado

Fonte: Elaborado pelo Autor.

4.2 Tempo de Escrita dos Frameworks

Para cada caso de teste, foi computado o tempo necessário para a sua escrita. A Tabela 4 apresenta os resultados desta métrica para cada caso de teste. É possível observar que Cypress foi o *framework* que precisou de menos tempo para o pesquisador implementar todos os casos de teste, seguido dos *frameworks* Robot Framework e Selenium. Entretanto, considerando que o Cypress teve dois casos de teste bloqueados por uma limitação do

framework, foi necessário calcular o desvio padrão entre todos os casos de teste, e assim, realizar a comparação entre os desvios para cada *framework*.

Dessa forma, foi observado que o Robot Framework teve o menor tempo para automatizar todos os casos de teste, bem como a menor taxa de desvio padrão. Com isso é possível concluir que, apesar dos diferentes casos de teste, este *framework* apresentou a menor disparidade de tempo entre os demais *frameworks*. Analisando alguns casos de teste, observa-se que Robot foi o *framework* que precisou de mais tempo nos 3 primeiros casos de teste (linhas 2 a 4), com praticamente o dobro do tempo se comparado com o Selenium.

O principal influenciador para isso foi a estrutura de organização do Robot. Utiliza-se um arquivo onde se declaram frases que explicam o que vai ocorrer na execução do teste para um determinado passo, e outro arquivo chamado resource, onde são declaradas as funções da biblioteca do SeleniumLibrary e os localizadores da página. Dessa forma, a utilização de dois arquivos para organizar os casos de teste mencionados acima causou um tempo mais estendido. Outro influenciador possível foi a falta de conhecimento do pesquisador que não dominava nenhum dos três *frameworks* selecionados para a pesquisa, dessa forma a falta de domínio sobre os *frameworks* pode ter causado a necessidade de utilização de mais tempo para conseguir automatizar alguns dos casos de teste.

Apesar de o Cypress apresentar dois casos de teste não automatizados, ao computar o desvio padrão para os casos automatizados, observa-se um desvio padrão de 17.7 minutos; desvio maior do que o observado com Selenium ou Robot. Assim, é possível concluir que mesmo mediante tal comparação, Robot ainda foi o *framework* com o melhor resultado para esta métrica.

Tabela 4 - Tempo de Escrita dos Frameworks

Casos de Teste	Selenium (min)	Cypress (min)	Robot (min)
Caso de Teste 1	25	N/A	60
Caso de Teste 2	8	N/A	13
Caso de Teste 3	23	20	40
Caso de Teste 4	15	11	8
Caso de Teste 5	30	30	10
Caso de Teste 6	50	18	37
Caso de Teste 7	25	10	19
Caso de Teste 8	8	9	25
Caso de Teste 9	40	20	23
Caso de Teste 10	38	60	42
Caso de Teste 11	50	40	20
Caso de Teste 12	60	50	45
Tempo Total (minutos):	372	268	342
Desvio Padrão do Tempo Total (minutos):	16.90	17.77	16.12

Fonte: Elaborado pelo Autor.

4.3 Tempo de Execução

Para cada caso de teste automatizado, foi computado o tempo de execução. A Tabela 5 apresenta o tempo requerido para a execução de cada caso de teste. É possível observar que o Robot Framework teve o menor tempo de duração, seguido de um empate entre os *frameworks* Selenium e Cypress. Considerando a não automatização de dois casos de teste pelo *framework* Cypress, foi novamente calculado o desvio padrão entre os tempos de execução de cada caso de teste. Dessa forma, mesmo em tais condições, o Robot Framework ainda apresenta o menor desvio quando comparado com os demais *frameworks*, e portanto, o *framework* com a execução mais rápida.

Esse resultado negativo para o Cypress, utilizando essa métrica, pode ter sido influenciado pela característica do *Framework*. O Cypress espera que todos os elementos e requisições da página sejam carregados para poder avançar para a próxima ação do teste, o que não acontece com o Selenium e o Robot Framework. Esses últimos tendem a executar as ações do teste de forma sequencial, sem aguardar o carregamento, a menos que seja declarado um comando de espera até que um determinado elemento apareça na página. A plataforma da Amazon que foi utilizada para executar os testes é uma plataforma que possui diversos elementos em cada página e acaba afetando a execução do Cypress que tende a esperar o carregamento completo dos elementos. Entretanto, mesmo com essa característica que contribuiu com um desempenho ruim para o Cypress, podemos observar que nos casos 3 e 9 o Cypress teve um tempo de execução similar ao Robot Framework.

Tabela 5 - Tempo de Execução dos Framework

Casos de Teste	Selenium (s)	Cypress (s)	Robot (s)
Caso de Teste 1	4	N/A	4
Caso de Teste 2	7	N/A	8
Caso de Teste 3	12	7	7
Caso de Teste 4	11	16	9
Caso de Teste 5	16	7	4
Caso de Teste 6	16	16	14
Caso de Teste 7	7	10	7
Caso de Teste 8	6	7	5
Caso de Teste 9	4	3	3
Caso de Teste 10	11	12	9
Caso de Teste 11	10	17	14
Caso de Teste 12	21	30	17
Tempo Total (s):	125	125	101
Desvio Padrão do Tempo Total (s):	5.24	7.73	4.48

Fonte: Elaborado pelo Autor.

4.4 Linha de Código (LOC)

Para cada caso de teste foi computado o número de linhas de código necessário para a automação. Assim, considerando as diferentes linguagens usadas para a automação, a Tabela 6 apresenta os valores computados. É possível observar que o Cypress foi o *framework* que precisou da menor quantidade de linhas de código para a automação dos testes, seguido de Selenium. Isso ocorreu por conta da linguagem suportada pelo *framework* (JavaScript), conhecida por ser uma linguagem menos verbosa, quando comparado com Java (Selenium). O *framework* que precisou do maior número de linhas de código para a implementação dos testes foi o Robot Framework (Python).

Considerando que o Cypress teve dois casos de teste bloqueados por uma limitação do *framework*, foi calculado o desvio padrão da quantidade de linhas para cada *framework*. Mesmo em tais condições, não foi observado nenhuma mudança no que diz respeito ao desempenho dos *frameworks* analisados. O Robot Framework permaneceu como o *framework* que precisou da maior quantidade de linhas para a automação dos testes, seguido de Selenium e Cypress.

Tabela 6 - LOC dos Frameworks

Casos de Teste	Selenium	Cypress	Robot
Caso de Teste 1	39	N/A	49
Caso de Teste 2	29	N/A	36
Caso de Teste 3	36	14	38
Caso de Teste 4	39	16	35
Caso de Teste 5	38	19	43
Caso de Teste 6	60	30	65
Caso de Teste 7	48	18	62
Caso de Teste 8	32	11	21
Caso de Teste 9	37	14	36
Caso de Teste 10	53	29	52
Caso de Teste 11	50	27	51
Caso de Teste 12	58	23	62
Total de Linhas:	519	201	550
Desvio Padrão do Total de Linhas:	9.77	6.77	13.35

Fonte: Elaborado pelo Autor.

4.5 Testes Inválidos

Por fim, a última métrica avaliada diz respeito à ocorrência de testes inválidos durante o processo de automação dos testes. A Tabela 8 apresenta os resultados referentes a esta métrica. É possível observar que apenas dois casos de teste não puderam ser automatizados no Cypress, por decorrência de uma limitação do *framework*, enquanto os demais testes foram automatizados nos *frameworks* Selenium e Robot Framework. Dessa forma, o Cypress teve uma taxa de 16,67% de testes inválidos/bloqueados.

Uma das limitações observadas estava relacionado com a possibilidade de controlar mais de uma aba do navegador durante a execução da automação (Linha 1, Coluna 1 na Tabela 1). O teste tinha como objetivo poder abrir um dos produtos do site da Amazon em uma segunda aba do navegador e verificar se seria possível interagir entre as duas abas existentes durante a execução da automação. Neste sentido, o Cypress não fornece suporte a

essa funcionalidade já que o *framework* para ser executado abre uma janela do navegador escolhido, onde acessa o painel de execução dos testes do Cypress. Dessa forma, quando se tenta criar uma automação para abrir uma segunda aba durante a execução de uma automação, ocorre a tentativa de acessar uma aba diferente da qual o Cypress define como sua página de execução e, com isso, as verificações não podem ser realizadas. Essa limitação está presente na documentação do Cypress e, como alternativa, recomenda-se que o QA realize outras validações para contornar o problema. Por exemplo, verificar se o elemento *href* possui a URL correta que redireciona a página quando clicar em um texto ou botão.

A fim de entender como esta limitação afeta QAs em suas atividades, uma consulta foi realizada na base de dados do Stack Overflow.[6] Para tanto, os termos relacionados à limitação previamente reportada foram usados na *query* como *keywords* (Figura 10), verificando a ocorrência dos termos nas tags e corpo da postagem, respectivamente. Como resultado, 04 perguntas foram retornadas. A partir deste conjunto de postagens, 03 perguntas foram analisadas a fim de identificar como a limitação era observada e solucionada pela comunidade.

Por exemplo, em três perguntas em que os usuários reportaram o mesmo problema da limitação anterior, foi possível verificar que duas das sugestões da comunidade para solucionar o problema seriam: verificar se o elemento do href apresentava a URL da página que deveria ser aberta.[7,8] Nessas circunstâncias, caso houvesse um clique no elemento em que o *href* está contido, a URL poderia ser extraída do HTML da página e que é carregada na aba em que a automação está sendo executada, verificando assim se a URL era válida. Essa sugestão está presente na documentação do Cypress como uma possível solução para o problema encontrado.

Outra solução identificada diz respeito ao elemento *href* não existir na página HTML.[9] Neste caso, a solução apontada seria utilizar o comando *cy.intercept,* usado no Cypress para interceptar solicitações de rede durante um teste, simulando o comportamento de uma solicitação de rede, como modificar ou simular uma resposta HTTP e controlar sua resposta. Dessa forma, o comando foi utilizado para modificar a propriedade *target,* alterando-a de "*_blank*" para "*_self*", evitando a abertura de uma nova aba do navegador ao clicar no elemento que contém o link.

---

[6] https://data.stackexchange.com/stackoverflow/query/new, consulta realizada no dia 02/06/2023.

[7] https://stackoverflow.com/questions/62517113/is-there-a-way-to-force-cypress-to-open-in-same-tab-instead

[8] https://stackoverflow.com/questions/48348354/cypress-get-href-attribute

[9] https://stackoverflow.com/questions/73245735/clicking-on-links-that-doesnt-contains-href-in-cypress

Figura 10 - Consulta SQL relacionada com a limitação de uso de múltiplas abas no Cypress

```sql
SELECT *
FROM Posts
WHERE PostTypeId = 1
AND Tags LIKE '%cypress%'
AND Body LIKE '%multiple tabs%'
AND AcceptedAnswerId IS NOT NULL;
```

Fonte: Elaborado pelo Autor.

A outra limitação observada é referente à utilização de diferentes super-domínios de páginas (Linha 2, Coluna 1 na Tabela 1). Neste sentido, o Cypress não possuía tal funcionalidade até a versão 12.0.0, quando adicionou suporte à utilização da nova função *cy.origin*. Esta função deve ser utilizada depois do comando *cy.visit* responsável por carregar a página da URL que foi determinada no parâmetro da função. Logo em seguida, o *cy.origin* é utilizado para indicar para o Cypress o novo super-domínio que será utilizado no restante da automação. Essa abordagem se faz necessária, pois o Cypress considera que o primeiro site que ele acessa, quando inicia a automação, é o site em que ele deve iniciar e finalizar o teste. No entanto, para casos em que é necessário acessar uma página de um super-domínio diferente do original, é preciso utilizar o *cy.origin* para indicar ao *framework* que a mudança será necessária. A Figura 11 e Figura 12, a seguir, apresentam um exemplo de um caso de sucesso e de falha para essa função, respectivamente.

Figura 11 - Teste será executado com erro por não utilizar o cy.origin

```
it('navigates', () => {
    cy.visit('https://www.google.com');
    cy.visit('https://www.amazon.com');
    cy.get('[aria-label="Amazon.com.br"]').should("exist");
});
```

Fonte: Elaborado pelo Autor.

Figura 12 - Teste será executado sem erro por utilizar o cy.origin

```
it('navigates', () => {
    cy.visit('https://www.google.com.br');
    cy.visit('https://www.amazon.com.br');
    cy.origin('https://www.amazon.com.br', () => {
        cy.get('[aria-label="Amazon.com.br"]').should("exist");
    });
});
```

Fonte: Elaborado pelo Autor.

O caso de teste planejado diz respeito a uma busca no Google, pesquisando pelo site da Amazon. Em seguida, ao selecionar um dos resultados da pesquisa, o caso de teste deveria verificar que o site da Amazon foi carregado a partir do site de busca acessado previamente. Entretanto, não faz sentido para o caso proposto utilizar o comando *cy.visit* para carregar o site da Amazon, já que isso seria contra um dos passos e o intuito do caso de teste. Portanto, esse caso foi classificado como bloqueado, já que o Cypress precisa que o site seja carregado pelo seu comando *cy.visit* (Linha 3 da Figura 12), e só então utilizar o *cy.origin* (Linha 4 da Figura 12).

Assim como realizado para a limitação apresentada previamente, a corrente limitação também foi analisada a fim de entender como a comunidade lida com esta limitação. Novamente, uma nova consulta foi realizada na base de dados do Stack Overflow, usando os termos associados com a limitação, como apresentado na Figura 13.[10] Diferente da consulta anterior, onde as keywords foram usadas apenas com o corpo das perguntas, nesta foi utilizado também o título por conta da falta de resultados quando se utilizava somente o corpo das perguntas. Como resultado, 03 perguntas foram retornadas, onde elas exploravam um contexto diferente do que é explorado aqui. Duas perguntas foram relacionadas com a reutilização do mesmo teste para evitar duplicidade de código já que a diferença entre os testes era somente a URL da página. A outra pergunta era relacionada com um problema de Iframe quando se utiliza o Cypress para automatizar a plataforma do Microsoft Teams.

Figura 13 - Consulta SQL relacionada com a limitação de diferentes domínios no Cypress

```
1  SELECT *
2  FROM Posts
3  WHERE PostTypeId = 1
4  AND Tags LIKE '%cypress%'
5  AND (Title LIKE '%multiple domains%' OR Body LIKE '%multiple domains%' OR
6  Title LIKE '%different domains%' OR Body LIKE '%different domains%' OR
7  Title LIKE '%multiple origin%' OR Body LIKE '%multiple origin%')
8  AND AcceptedAnswerId IS NOT NULL;
```

Fonte: Elaborado pelo Autor.

[10] Consulta utilizada no dia 02/06/2023

Tabela 8 - Testes inválidos dos Frameworks

Casos de Teste	Selenium	Cypress	Robot
Caso de Teste 1	Automatizado	Bloqueado	Automatizado
Caso de Teste 2	Automatizado	Bloqueado	Automatizado
Caso de Teste 3	Automatizado	Automatizado	Automatizado
Caso de Teste 4	Automatizado	Automatizado	Automatizado
Caso de Teste 5	Automatizado	Automatizado	Automatizado
Caso de Teste 6	Automatizado	Automatizado	Automatizado
Caso de Teste 7	Automatizado	Automatizado	Automatizado
Caso de Teste 8	Automatizado	Automatizado	Automatizado
Caso de Teste 9	Automatizado	Automatizado	Automatizado
Caso de Teste 10	Automatizado	Automatizado	Automatizado
Caso de Teste 11	Automatizado	Automatizado	Automatizado
Caso de Teste 12	Automatizado	Automatizado	Automatizado
Porcentagem de Testes Inválidos:	0,00%	16,67%	0,00%

Fonte: Elaborado pelo Autor.

5 AMEAÇAS À VALIDADE

Esta seção discute as ameaças que podem ter influenciado os resultados relacionados à execução deste trabalho. A primeira possível ameaça está relacionada ao impacto da experiência do pesquisador, responsável pela automação dos testes, perante os *frameworks* avaliados neste estudo. Neste sentido, apesar do pesquisador ter conhecimento e experiência em *frameworks* similares de automação, ele não tinha experiência prática em nenhum dos *frameworks* avaliados. Desta forma, acredita-se que esta ausência de experiência em comum a todos os *frameworks* não represente uma ameaça aos resultados.

A segunda possível ameaça diz respeito às linguagens de programação suportadas pelos *frameworks*, e por consequente, usadas neste trabalho. Os testes usando Cypress foram automatizados usando a linguagem JavaScript, enquanto Python foi adotado para uso do Robot Framework. Por fim, os testes no Selenium foram automatizados usando a linguagem Java. Considerando que JavaScript e Python são linguagens menos verbosas comparadas com Java, logo, espera-se que a métrica de Linhas de Código (LOC) tenha sido impactada por estas características.

A terceira possível ameaça está relacionada com a experiência prévia do pesquisador do trabalho com a linguagem Java. Por conta desta experiência prévia, acredita-se que isso possa ter facilitado no funcionamento e automação de testes usando o *framework* Selenium. Por consequente, a inexperiência nas linguagens suportadas pelos demais *frameworks* (JavaScript e Python) pode ter impactado o resultado da métrica de Tempo de Escrita dos Frameworks. Considerando que o *framework* que demandou mais tempo durante a automação foi o Cypress, que utiliza o JavaScript como linguagem principal.

Por fim, a utilização dos *frameworks* avaliados nesta pesquisa utilizando outras linguagens pode resultar em diferentes conclusões, principalmente para a métrica de Linha de Código (LOC) e Tempo de Escrita dos Casos de Teste. Uma vez que os *frameworks* avaliados são suportados por diferentes linguagens, logo, a adoção de linguagens diferentes do grupo adotado neste estudo, representa uma ameaça à generalização dos achados do estudo. Considerando que diferentes linguagens possuem verbosidades diferentes, bem como complexidades distintas impactando na automação de testes.

6 TRABALHOS RELACIONADOS

Nesta seção são discutidos trabalhos, que investigam um problema relacionado com a proposta apresentada neste trabalho.

Fatini Mobaraya e Shahid Ali (2019) apresentam um estudo comparativo sobre Selenium e Cypress. Semelhante ao objetivo deste trabalho, os autores utilizam um subconjunto das métricas adotadas do presente estudo (tempo de execução, eficiência do Teste e cobertura de teste baseada em requisitos) para avaliar a eficiência dos *frameworks* selecionados. Como resultado, os autores concluíram que o uso de Selenium para a automação requer mais esforço para a escrita dos testes automatizados, bem como mais tempo para executar os casos de teste, quando comparado com os resultados obtidos pelo *framework* Cypress.

Candido Silva et al. (2021) apresentam um estudo comparativo entre Selenium WebDriver e Cypress. Semelhante ao objetivo deste trabalho, os autores utilizam um conjunto de vasto de características (Simplicidade, Chamadas HTTP, Uso de Wait, Métodos Javascript, Múltiplas abas, Tempo de execução, Linhas de código, Documentação, Artefatos de teste e Suporte a execução dos testes) para avaliar a eficiência dos *frameworks* selecionados. Como resultado, os autores concluíram que utilizando o *framework* Cypress, o profissional da área de qualidade poderia escrever testes automatizados utilizando menos linhas de código e redução no tempo de execução dos testes, quando comparado com o *framework* Selenium. Adicionalmente, os autores observaram que Cypress apresenta uma limitação sobre o suporte à interação com múltiplas abas e janelas.

Aussourd (2022) apresenta um estudo comparativo entre Selenium IDE e Selenium WebDriver. Semelhante ao objetivo deste trabalho, a autora utilizou um subconjunto de métricas (número de falhas, índice de severidade de defeitos, velocidade de execução e testes inválidos) para avaliar a eficiência dos *frameworks* selecionados. Como resultado, o estudo concluiu que a automação de testes utilizando Selenium IDE tem uma cobertura maior, já que o *framework* possui menos limitações se comparado com sua concorrente. Enquanto o Selenium IDE pode executar 5 testes por hora, o Selenium WebDriver executou somente 1,6 testes por hora. Essa diferença ocorre devido ao aumento no número de testes inválidos quando há o uso do Selenium WebDriver. Assim, o estudo conclui que Selenium IDE foi o *framework* mais indicado para uso devido à sua facilidade de escrita de código e eficiência.

Semelhante aos estudos prévios, este trabalho também busca comparar o suporte de diferentes *frameworks* para testes de sistemas automatizados, porém um conjunto de métricas diferentes foram selecionadas para poder utilizar um conjunto de métricas que sejam mais importantes no quesito de comparação da qualidade de um *framework*. Dessa forma, enquanto os trabalhos avaliam Cypress e variações de Selenium, neste trabalho há a avaliação destes *frameworks* mas também do Robot Framework. Adicionalmente, neste trabalho diferentes métricas foram selecionadas como, tempo de escrita dos casos de teste, tempo de execução, quantidade de linhas de código e testes inválidos. Por fim, este estudo foi realizado explorando diferentes linguagens de programação como, Java (Selenium), JavaScript (Cypress) e Python (Robot Framework). Fatini Mobaraya e Shahid Ali (2019) e Candido Silva et al. (2021) utilizaram Java e JavaScript para Selenium e Cypress respectivamente, enquanto que Aussourd (2022) optou pela utilização do Python no Selenium IDE e WebDriver. Por fim, neste trabalho também foi investigado como QAs lidam com as limitações dos *frameworks* por meio da análise de fóruns de perguntas.

7 CONCLUSÃO

Este trabalho teve como objetivo fazer uma análise comparativa entre os *frameworks* de automação de testes web Selenium, Cypress e Robot Framework. Para tanto, um conjunto de casos de testes foi identificado a partir do site da Amazon, e posteriormente, automatizados em cada um dos *frameworks* citados. Durante este processo, uma avaliação foi realizada a partir da seleção de algumas métricas, selecionadas para mensurar diferentes aspectos dos *frameworks* avaliados, como tempo de execução e escrita, linhas de código e testes inválidos. Como resultado geral, foi possível observar que o *framework* que apresentou a melhor eficiência perante às métricas avaliadas foi o Robot Framework que obteve destaque para as métricas de Tempo de Escrita dos Frameworks, Tempo de Execução dos Framework e não teve nenhum caso de teste bloqueado, dessa forma apresentando uma melhor eficiência em 3 das 4 métricas utilizadas por essa pesquisa.

O Cypress demonstrou ser um *framework* minimalista, onde faz-se necessário uma quantidade bem inferior de linhas de código do que se comparado com seus concorrentes. Esta observação é influenciada pelo fato do *framework* ser apoiado pela linguagem JavaScript, uma linguagem menos verbosa. Porém, o Cypress obteve destaque negativo em relação ao tempo de execução dos casos de teste, sendo o *framework* que precisou de mais tempo para executar todos os scripts. Isso ocorreu pela complexidade da plataforma da Amazon, que possui diversos elementos na tela carregados/renderizados durante uma requisição ao sistema, e o Cypress possui a característica de aguardar que todas as requisições sejam finalizadas para seguir com o próximo passado da automação. Por outro lado, essa característica não é observada com o Robot Framework que teve destaque nesta métrica de tempo de execução sendo o *framework* que precisou de menos tempo para executar todos os casos de teste.

O Cypress foi o único *framework* selecionado neste estudo, onde foi possível observar limitações impedindo a automação de alguns casos de teste. Uma das limitações notadas foi a ausência de suporte do Cypress para a interação e possibilidade de controlar mais de uma aba/janela do navegador durante a execução da automação. Outra limitação observada foi a obrigatoriedade de iniciar e finalizar os testes com o mesmo super-domínio utilizado no início da automação. Apesar de que o *framework* apresenta uma alternativa para solucionar esse problema. Ambas as limitações não foram observadas para os *frameworks* Selenium e Robot Framework.

REFERÊNCIAS

Apêndice Online. GitHub, 2023. Disponível em: <https://github.com/ViniLopes20/automacao-frameworks>. Acesso em: 14 de maio de 2023.

WebDriver Documentation. Selenium, 2022. Disponível em: <https://www.selenium.dev/documentation/webdriver/>. Acesso em: 02 de dec. de 2022.

WebDriver Getting Started. Selenium, 2022. Disponível em: <https://www.selenium.dev/documentation/webdriver/getting_started/>. Acesso em: 02 de dec. de 2022.

Testing has been broken for too long. Cypress, 2021. Disponível em: <https://www.cypress.io/how-it-works/>. Acesso em: 02 de dec. de 2022.

Why Cypress?. Cypress, 2021. Disponível em: <https://docs.cypress.io/guides/overview/why-cypress#Features>. Acesso em: 02 de dec. de 2022.

Cypress Architecture. Cypress, 2021. Disponível em: <https://docs.cypress.io/guides/overview/key-differences#Architecture>. Acesso em: 02 de dec. de 2022.

Robot Framework Introduction. Robot Framework, 2017. Disponível em: <https://robotframework.org/?tab=0&example=Simple%20Example#getting-started>. Acesso em: 02 de dec. de 2022.

Cross Origin Testing. Cypress, 2023. Disponível em: <https://docs.cypress.io/guides/guides/cross-origin-testing>. Acesso em: 19 de maio de 2023.

Trade-offs. Cypress, 2023. Disponível em: <https://docs.cypress.io/guides/references/trade-offs#Multiple-tabs>. Acesso em: 19 de maio de 2023.

Candido Silva et al. “Automação de Testes Funcionais: Uma Análise Técnica dos Frameworks Cypress e Selenium”, 2021, Even3 Publicações, p. 01-23.

Mobaraya, Fatini et al. “Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern Web Application Testing”, 2019, AIRCC Publishing Corporation, vol. 9, nº 18, p. 01-20.

Aussourd. “Automação de Testes com Selenium IDE e WebDriver: Um Estudo Comparativo”, 2022, UNICAP, p. 01-44.

Clerissi et al., “Towards the Generation of End-to-End Web Test Scripts from Requirements Specifications”, 2017, IEEE.

VILAS BOAS, CAROL et al. Jornada Ágil de Qualidade. Rio de Janeiro: Brasport, 2020.

Noello, T. P, “Melhoria no processo de teste em uma empresa de desenvolvimento de software”, 2016, UNISINOS, p. 01-37.

Dustin et al. Implementing Automated Software Testing. Boston: Addison-Wesley Professional, 2009.

Watkins, John. Agile testing: how to succeed in an extreme testing environment. Reino Unido: Cambridge University Press, 2009.

VALENTE, MARCO. Engenharia de Software Moderna. Belo Horizonte: Independente, 2022.

IZABEL, LEONARDO. “Testes Automatizados no Processo de Desenvolvimento de Softwares”, 2014, UFRJ, p. 01-65.