UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Jéssica Moreira Leal

APLICAÇÃO DE LLMS OPEN-WEIGHT NA AUTOMAÇÃO DE TESTES WEB: ESTUDO COMPARATIVO UTILIZANDO PYTHON E SELENIUM

Recife

2026

Jéssica Moreira Leal

APLICAÇÃO DE LLMS OPEN-WEIGHT NA AUTOMAÇÃO DE TESTES WEB: ESTUDO COMPARATIVO UTILIZANDO PYTHON E SELENIUM

Monografia apresentada ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Estatística e Informática

Curso de Bacharelado em Sistemas de Informação

Orientador: Cleyton Vanut Cordeiro de Magalhães

Recife

2026

Dados Internacionais de Catalogação na Publicação
Sistema Integrado de Bibliotecas da UFRPE
Bibliotecário(a): Auxiliadora Cunha – CRB-4 1134

L434a Leal, Jéssica Moreira.
Aplicação de LLMs "Open-Weight" na automação de testes web: estudo comparativo utilizando Python e Selenium / Jéssica Moreira Leal. – Recife, 2026.
110 f.; il.

Orientador(a): Cleyton Vanut Cordeiro de Magalhães.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal Rural de Pernambuco, Bacharelado em Sistemas da Informação, Recife, BR-PE, 2026.

Inclui referências e apêndice(s).

1. Modelos de Linguagem de Larga Escala. 2. Modelos de Peso Aberto. 3. Teste de Software. 4. Automação de Testes 5. Inteligência Artificial. I. Magalhães, Cleyton Vanut Cordeiro de, orient. II. Título

CDD 004

À minha família e todos os meus professores de Ensino Médio que apoiaram meus estudos e a mim por não desistir deles.

Agradecimentos

Agradeço imensamente à minha família por trabalhar incansavelmente para me permitir estudar e entrar numa Universidade Pública. Agradeço a todos os professores do Ensino Médio da Escola Estadual Dom Bosco que me instigavam a ser melhor a cada dia nos estudos e sempre ter uma perspectiva de futuro e a bibliotecária que sempre me disponibilizava mais livros do que o usual.

Agradeço aos meus amigos da faculdade, que estiveram nessa jornada comigo realizando projetos e estudando para provas de madrugada.

Agradeço também ao meu amor, que esteve comigo desde o início da faculdade e acompanhou todas as diferentes versões da Jéssica, e que continua me apoiando e permanecendo ao meu lado independente da situação.

Agradeço ao meu orientador, Prof. Cleyton Vanut Cordeiro de Magalhães, pela paciência e pelas orientações valiosas ao longo desta pesquisa.

Agradeço a todas as pessoas que conheci na Seed a Bit, CESAR School e NTT Data por contribuírem no meu arcabouço profissional como Engenheira de *Software* e QA.

Por fim, agradeço a todos os professores e profissionais que compõem a Universidade Pública, considerada por mim um mecanismo de progresso e ascensão social em meio a um mundo repleto de injustiças sociais que afetam os menos favorecidos, excepcionalmente mulheres. Mecanismo esse que transformou a minha vida e a da minha família e continua transformando a vida de todos que passam por ela.

“Enquanto estamos vivos, carregamos em nós um futuro de possibilidades multifacetadas.”

(Matt Haig, A Biblioteca da Meia-Noite)

Resumo

A automação de testes de software contribui para reduzir custos, ampliar a cobertura dos testes e aumentar a confiabilidade no desenvolvimento de sistemas. Nesse contexto, modelos de linguagem de grande porte (*Large Language Models* - LLMs) têm sido explorados como apoio à geração de scripts de testes automatizados. Entretanto, o custo de modelos comerciais pode limitar sua adoção, especialmente por equipes e organizações com recursos restritos, tornando os modelos *open-weight* uma alternativa relevante. Este trabalho teve como objetivo comparar a eficácia de quatro modelos LLM *open-weight* na geração de testes automatizados para um ambiente *e-commerce*, utilizando Python, Selenium e Pytest. Foram avaliados os modelos GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B e Qwen3 Coder 480B. A análise considerou métricas de taxa de sucesso de execução (*PASSRATE*), cobertura dos requisitos, qualidade estática do código por meio do Score Pylint, complexidade ciclomática e categorias de defeitos identificadas. Os resultados permitiram observar diferenças entre os modelos quanto à capacidade de gerar scripts executáveis, aderentes aos requisitos e com qualidade estrutural adequada. A pesquisa contribui ao apresentar uma avaliação comparativa de modelos LLM *open-weight* aplicados à automação de testes, fornecendo subsídios para sua adoção em contextos de qualidade de software com recursos limitados.

Palavras-chave: Modelos de Linguagem de Larga Escala, Modelos de Peso Aberto, Teste de Software, Automação de Testes, Inteligência Artificial, Python.

Abstract

Software test automation contributes to reducing costs, increasing test coverage, and improving reliability in software development. In this context, Large Language Models (LLMs) have been explored as a supporting tool for generating automated test scripts. However, the cost of commercial models may limit their adoption, especially by teams and organizations with restricted resources, making open-weight models a relevant alternative. This study aimed to compare the effectiveness of four open-weight LLMs in generating automated tests for an e-commerce environment using Python, Selenium, and Pytest. The evaluated models were GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B, and Qwen3 Coder 480B. The analysis considered execution success rate (PASSRATE), requirements coverage, static code quality measured by the Pylint Score, cyclomatic complexity, and categories of defects identified in the generated scripts. The results showed differences among the models regarding their ability to generate executable scripts, meet the specified requirements, and produce structurally adequate code. This research contributes by presenting a comparative evaluation of open-weight LLMs applied to test automation, providing insights to support their adoption in software quality contexts with limited resources.

Keywords: LLM, Open-weight Models, Software Testing, Test Automation, Artificial Intelligence, Python.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface* (Interface de Programação de Aplicações)
IA	Inteligência Artificial
LLM	*Large Language Model* (Modelo de Linguagem de Grande Escala)
MoE	*Mixture of Experts* (Mistura de Especialistas)
QA	*Quality Assurance* (Garantia de Qualidade)

Sumário

1 Introdução

1.1 Contextualização

O crescimento da área de tecnologia e desenvolvimento de *software* tem sido notável nas últimas décadas, impulsionado pela evolução das tecnologias digitais e pela crescente demanda por soluções inovadoras. Esse avanço reflete-se diretamente nos indicadores do setor: o mercado mundial de TI e o mercado brasileiro registraram uma ascensão de 10,8% e 13,9%, respectivamente, em comparação com o ano de 2023, com investimentos de aproximadamente 59,5 bilhões de dólares, impulsionados principalmente pela adoção de tecnologias emergentes, como IA, aprendizado de máquina e automação de processos (ABES, 2024). Nesse cenário, as tendências do setor indicam: o aumento da utilização e implantação de LLMs no desenvolvimento de sistemas; melhorias na área de cibersegurança por meio de recursos inteligentes baseados em IA; e o impulsionamento de *Data Centers* a partir de IA Generativa. Segundo Pressman (2014), o desenvolvimento de *software* tornou-se um dos principais pilares das economias modernas, com impacto direto em diversos setores, desde a indústria até a saúde.

Diante dessa demanda crescente, o processo de desenvolvimento de *software* passou por transformações profundas. Para Llopis, García e Martínez (2021), a aplicação de técnicas avançadas de automação, incluindo ferramentas baseadas em IA, tem permitido a redução de custos e a melhoria da eficiência dos processos de desenvolvimento. O crescimento acelerado da computação em nuvem e a popularização de ambientes de desenvolvimento ágeis também contribuíram para a inovação e otimização dos fluxos de trabalho, tornando a tecnologia cada vez mais acessível e essencial para o sucesso organizacional (ROSENBERG, 2019). A alta demanda por sistemas complexos e adaptáveis ao modelo de negócio de distintas corporações torna, consequentemente, a atuação de profissionais especializados em qualidade de *software* um fator crucial para o desenvolvimento de sistemas de alto nível (BARTIE, 2002).

Os modelos de linguagem de larga escala (LLMs) são sistemas de IA capazes de processar e gerar textos de forma eficiente e sofisticada, e emergiram como um dos principais vetores dessa transformação no desenvolvimento de *software*. A versatilidade e acessibilidade desses modelos permitem aplicações que vão desde tarefas simples, como agendar compromissos, até o auxílio no desenvolvimento de sistemas escaláveis (AKHMEDOV et al., 2023). Exemplos notáveis incluem o ChatGPT, da OpenAI, e o Gemini, do Google, ambos amplamente utilizados no mercado brasileiro e internacional para auxílio em todas as etapas do processo de desenvolvimento de *software*

(RANE; CHOUDHARY; RANE, 2024).

Diante da adoção progressiva de tecnologias avançadas, como LLMs e IA generativa, a área de testes de *software* assume um papel ainda mais estratégico dentro das organizações (UGWA, 2024). A complexidade crescente dessas soluções exige mecanismos de validação contínua capazes de acompanhar a velocidade das entregas, garantir a integridade dos sistemas e mitigar riscos decorrentes de funcionalidades inteligentes ou altamente integradas (SOARES; COSTA; KULESZA, 2023). Nesse contexto, o trabalho dos profissionais de qualidade torna-se essencial para assegurar que os produtos atendam aos requisitos definidos e mantenham estabilidade operacional, prevenindo falhas que poderiam comprometer todo o ciclo de desenvolvimento (SOMMERVILLE, 2011; AZEVEDO, 2021).

Assim, a implementação de testes automatizados nos ciclos de testes reduz o esforço do time de testadores manuais, otimiza o tempo de execução e proporciona maior confiabilidade no processo de desenvolvimento de *software* (AZEVEDO, 2021). No entanto, cada caso de teste deve ser avaliado a fim de mensurar seu esforço e riscos, que englobam desde custos elevados até a inflexibilidade do *framework* e a alta frequência de manutenção para controle da complexidade do sistema (SOMMERVILLE, 2011). Além disso, quando o sistema é excessivamente complexo ou o *framework* se torna instável após sucessivas tentativas de correção, torna-se inviável manter a automação de casos de teste, pois seus resultados não serão considerados confiáveis (THUMMALAPENTA et al., 2013). Apesar dessas limitações, os LLMs emergem como uma abordagem complementar promissora, capaz de aprimorar o processo de desenvolvimento de *software* em todas as suas etapas, inclusive a de testes (FERINO et al., 2026).

1.2 Problema de Pesquisa

Apesar dos avanços na automação de testes, muitas empresas ainda enfrentam desafios em relação ao alto custo de desenvolvimento, manutenção e confiabilidade dos testes automatizados. A implementação de uma estrutura de automação de testes eficaz requer um investimento inicial significativo, não apenas na aquisição de ferramentas e recursos, mas também na capacitação das equipes (BERNARDO, 2011). Além disso, a manutenção constante dos testes automatizados, especialmente em projetos com mudanças frequentes no código, pode se tornar um processo complexo e dispendioso (SOMMERVILLE, 2011). Quando o código é alterado, os testes automatizados precisam ser adaptados, o que demanda tempo e esforço adicional.

A confiabilidade dos testes automatizados enfrenta desafios significativos devido à sua execução baseada em *scripts* pré-definidos. Qualquer falha nesses *scripts* ou na infraestrutura de automação pode resultar em diagnósticos imprecisos. Além disso, a falta de flexibilidade dos testes automatizados pode limitar sua eficácia em simular interações complexas do usuário ou cenários dinâmicos. De acordo com Strandberg (2021), a automação de testes de *software* em sistemas embarcados industriais enfrenta desafios como a análise de casos de teste que, quando executados ao longo do tempo em *software*, *testware* ou revisões de *hardware* em evolução, parecem falhar aleatoriamente. Isso destaca a dificuldade em garantir a confiabilidade e a precisão dos testes automatizados em ambientes complexos e em constante mudança.

Por outro lado, a automação de testes traz benefícios evidentes, como a redução do tempo de execução de testes repetitivos e a cobertura mais ampla, principalmente em testes de regressão (AMORIM et al., 2017). No entanto, para que esses benefícios sejam aproveitados ao máximo, é necessário que as empresas adotem uma abordagem equilibrada, utilizando automação onde ela é mais eficiente, enquanto preservam os testes manuais para cenários mais complexos e imprevisíveis. A combinação estratégica entre automação e testes manuais pode melhorar a qualidade do *software* e tornar o processo de desenvolvimento mais ágil e eficiente (SILVA; ALVES; BRUNO, 2011).

Diante dos desafios inerentes à automação de testes tradicional, os LLMs emergem como uma tecnologia promissora para suportar e aprimorar essas atividades. No contexto de testes de *software*, os LLMs vêm sendo aplicados em tarefas como geração de casos de teste, reprodução de *bugs*, síntese de *scripts* e criação de documentação, sendo particularmente valiosos na automação e na depuração de código. No entanto, apesar do potencial identificado, a adoção prática dessa tecnologia ainda carece de maturidade. Estudos iniciais apontam direcionamentos relevantes, porém as pesquisas existentes concentram-se predominantemente em testes funcionais no nível de implementação, deixando etapas como planejamento, design e testes não fun-

cionais ainda pouco exploradas (SANTANA; MAGALHAES; SANTOS, 2025). Dessa forma, na prática, ainda não existem diretrizes ou recomendações consolidadas que orientem os engenheiros de testes a tirar proveito efetivo dos LLMs ao longo das etapas do processo de automação de testes, evidenciando uma lacuna relevante a ser investigada.

1.3 Justificativa

A fim de maximizar a eficiência da automação para melhorar o processo de desenvolvimento e a qualidade do software, a utilização de LLMs em conjunto com engenharia de prompt no processo de desenvolvimento de testes automatizados deve ser aprofundada e testada. A aplicação de LLMs no processo de desenvolvimento de casos de testes automatizados pode oferecer uma abordagem inovadora na criação de *scripts* de teste, permitindo que as ferramentas de automação sejam mais adaptáveis e inteligentes. O ChatGPT, por exemplo, é capaz de compreender e gerar código com base em descrições em linguagem natural, o que pode facilitar a criação de testes e reduzir o tempo gasto com a escrita de *scripts* complexos (YU et al., 2023).

Além disso, modelos LLM têm o potencial de melhorar a detecção de falhas e localização de *bugs* (LI et al., 2025), sugerindo modificações ou identificando casos de teste que não haviam sido considerados. Com isso, os testes podem se tornar mais robustos e dinâmicos, cobrindo cenários mais diversos e complexos do que os métodos tradicionais. Segundo (SCHÄFER et al., 2024), a integração dos LLMs com *frameworks* de automação de testes pode não só aumentar a cobertura dos testes, mas também proporcionar uma análise mais detalhada e precisa dos resultados, melhorando a confiabilidade do processo de testes automatizados.

Portanto, a implementação desses modelos no ciclo de desenvolvimento de software é promissora e necessita de mais experimentos e avaliações práticas, a fim de entender seu impacto real no aumento da produtividade e na qualidade dos produtos finais. O uso de modelos LLM nesse contexto poderia, inclusive, reduzir a necessidade de manutenção constante dos testes, tornando os processos de automação mais sustentáveis a longo prazo (LI et al., 2025).

Contudo, a viabilidade prática dessa implementação esbarra em questões econômicas. Modelos comerciais, embora demonstrem capacidades superiores, requerem investimentos contínuos em licenciamento e uso de APIs (*Application Programming Interface*) que podem ser proibitivos para pequenas e médias empresas. Segundo a ABES (2024), esse fator econômico é apontado por 62% das empresas brasileiras de TI como principal barreira à adoção de IA. Diante dessa realidade, modelos *open-weight* apresentam-se como alternativas estratégicas, combinando acessibilidade com desempenho competitivo. Diferentemente de modelos *open-source* completos, que exigem acesso ao código de treinamento e aos dados utilizados, modelos *open-weight* disponibilizam publicamente os pesos do modelo treinado, permitindo sua utilização via API de inferência remota sem a necessidade de infraestrutura de hardware dedicada. A literatura recente indica que esses modelos alcançam níveis de performance próximos a soluções proprietárias em domínios específicos, particularmente em geração de código (CHEN et al., 2025). Essa convergência de capacidades técnicas e viabilidade

econômica justifica a investigação aprofundada de modelos *open-weight* no contexto de automação de testes.

1.4 Objetivos

1.4.1 Perguntas de Pesquisa

Diante disso, este trabalho busca responder à seguinte pergunta de pesquisa:

Qual é a eficácia de LLMs open-weight na geração de scripts de testes automatizados, medida por taxa de sucesso (PASSRATE) e cobertura de requisitos?

Para apoiar essa investigação, definem-se as seguintes perguntas secundárias:

1. Como os modelos LLM *open-weight* avaliados se diferenciam na geração de scripts de testes automatizados para um ambiente *e-commerce*, considerando taxa de sucesso de execução, cobertura de requisitos, qualidade estática do código e complexidade ciclomática?
2. A especialização em código do modelo Qwen3 Coder 480B resulta em *scripts* com maior *PASSRATE*, melhor *Score Pylint* e menor complexidade ciclomática, indicando maior manutenibilidade, em comparação aos modelos generalistas GPT-OSS 120B, Gemma 4 26B e Llama 3.3 70B, considerando o mesmo conjunto de casos de teste?
3. Quais categorias de defeitos, erros sintáticos, erros relacionados ao Selenium ou erros lógicos, predominam nos *scripts* gerados pelos modelos avaliados, e de que forma a qualidade do código, medida por *Score Pylint* e complexidade ciclomática, se associa à ocorrência dessas falhas?

1.4.2 Objetivo Geral

Analisar comparativamente quatro modelos LLM *open-weight* (GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B e Qwen3 Coder 480B), acessados via API de inferência remota, quanto à sua eficácia na geração de *scripts* de testes automatizados em Python com Selenium e Pytest, considerando métricas de qualidade de código, cobertura de cenários, corretude funcional e categorização de defeitos em um ambiente *e-commerce*.

1.4.3 Objetivos Específicos

- Caracterizar as capacidades e limitações dos modelos GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B e Qwen3 Coder 480B na geração de *scripts* de testes automatizados para aplicações web.

- Avaliar a qualidade dos *scripts* gerados por meio de *Score Pylint* e complexidade ciclomática, analisando sua relação com a manutenibilidade do código produzido.
- Mensurar a corretude funcional dos *scripts* gerados por meio de *PASSRATE* e cobertura de requisitos, executando-os em um ambiente *e-commerce* controlado com Selenium e Pytest.
- Classificar e quantificar os defeitos presentes nos *scripts* gerados, categorizando-os em erros sintáticos, erros relacionados ao Selenium e erros lógicos.
- Propor recomendações práticas para equipes de QA com recursos limitados sobre a adoção de modelos LLM *open-weight* na automação de testes em Python.

1.5 Organização da Monografia

Além desta introdução, este trabalho foi organizado a partir da seguinte estrutura:

Capítulo 2 – Fundamentação Teórica

Este capítulo reúne os conceitos essenciais para o entendimento da pesquisa, abordando temas como: Abordagem de testes, tipos de testes de *software*, *Prompt Engineering* (Engenharia de Prompt), teste de *software*, testes automatizados, linguagem Python, *frameworks* de automação, modelos de linguagem de grande escala (LLMs) e trabalhos relacionados à pesquisa. São discutidos os principais fundamentos teóricos que sustentam os experimentos realizados ao longo do estudo.

Capítulo 3 – Metodologia

No presente capítulo, descreve-se o fluxo metodológico adotado para a investigação. São detalhados os critérios utilizados para seleção das ferramentas, dos modelos de linguagem, dos cenários de teste e dos scripts automatizados. Apresenta-se ainda o ambiente de desenvolvimento, o conjunto de dados, as métricas de avaliação e o processo de definição dos experimentos.

Capítulo 4 – Resultados

Este capítulo apresenta os testes realizados e os resultados obtidos, incluindo a comparação entre diferentes modelos de LLMs, a eficácia da engenharia de prompt e o desempenho da automação dos testes em Python. São discutidos os achados relevantes, suas implicações práticas e as limitações observadas durante os experimentos.

Capítulo 5 – Considerações Finais

No capítulo final, são sintetizadas as contribuições do estudo, destacando os resultados mais relevantes. Além disso, apresentam-se sugestões de aprimoramentos e direções para trabalhos futuros que possam aprofundar ou expandir as análises aqui realizadas.

Capítulo 6 – Apêndices

Os apêndices reúnem materiais complementares, tais como trechos de código, detalhes de implementação dos *frameworks* de automação, instruções de configuração do ambiente, exemplos de *prompts* e informações adicionais que auxiliam na reprodutibilidade da pesquisa.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais que sustentam o desenvolvimento deste trabalho, oferecendo uma base teórica para a compreensão dos temas abordados nos capítulos seguintes, compondo o conjunto de referenciais necessários para embasar os experimentos e análises apresentados posteriormente.

2.1 Qualidade de *software*

A qualidade de *software* é um conceito multidimensional que descreve o grau em que um sistema de *software* atende aos requisitos explícitos e implícitos dos seus usuários, bem como as expectativas organizacionais relativas a confiabilidade, eficiência, segurança e manutenção (SOMMERVILLE, 2011). Em sentido amplo, qualidade envolve tanto a conformidade com especificações formais quanto a adequação ao uso em contextos reais, incluindo requisitos funcionais e não funcionais.

A garantia de qualidade (Quality Assurance — QA) compreende o conjunto de atividades processuais e organizacionais destinadas a criar confiança de que os requisitos de qualidade serão atendidos. Entre elas estão definição de processos, auditorias, revisão de requisitos e formação de equipes. O controle de qualidade (Quality Control — QC), por sua vez, refere-se a atividades técnicas de verificação e validação, como inspeção de código, revisões, testes (unitários, integração, sistema e aceitação) e medições de conformidade (PRESSMAN, 2014). Bertolino (2007) reforça que a integração entre QA e práticas de teste contínuo é crucial para detectar defeitos cedo e reduzir custos de correção.

Nesse cenário, modelos LLM *open-weight* emergem como alternativas para aprimorar o processo de geração de casos de testes, caso sejam direcionados apropriadamente (FERREIRA et al., 2025), permitindo que equipes de QA em organizações de diferentes portes possam se beneficiar dos avanços tecnológicos sem comprometer orçamentos e assegurando a qualidade do produto.

2.2 Teste de *software*

O teste de *software* constitui uma atividade essencial no processo de desenvolvimento, cujo propósito central é detectar defeitos, verificar a conformidade com os requisitos especificados e garantir a qualidade do produto entregue. De acordo com Sommerville (2011), testar um sistema consiste na execução controlada de suas fun-

cionalidades com o objetivo de identificar falhas e verificar se o comportamento observado corresponde ao esperado. Essa atividade pode ser realizada de forma manual ou automatizada, envolvendo critérios de aceitação que orientam a validação tanto de requisitos funcionais quanto não funcionais, como desempenho, segurança e usabilidade (MYERS; SANDLER; BADGETT, 2011). Além disso, os testes podem ser estruturados em múltiplos níveis e técnicas, incluindo abordagens de caixa-preta, caixa-branca e mistas, de modo a cobrir diferentes perspectivas de análise do sistema (PRESSMAN, 2014).

A literatura destaca que a adoção sistemática de testes reduz custos ao evitar a propagação de erros para fases mais avançadas do ciclo de vida, mitigando riscos e retrabalho (BERTOLINO, 2007). Paralelamente, a automação de testes tem se mostrado um recurso estratégico para elevar a eficiência e permitir execuções repetitivas, rápidas e confiáveis, contribuindo para ciclos de desenvolvimento contínuos e de maior qualidade (GAROUSI; YILDIRIM, 2018).

Recentemente, a aplicação de LLMs tem emergido como abordagem promissora para automatizar não apenas a execução, mas também a *criação* de casos de teste. Estudos como o de Ferreira et al. (2025) demonstram que LLMs podem gerar cenários de testes de aceitação a partir de histórias de usuário com alta taxa de aprovação (95% em 65 instâncias reais), evidenciando potencial para reduzir significativamente o esforço manual na especificação de testes.

2.2.1 Abordagem de Testes

Abordagem de testes consiste no conjunto de estratégias, técnicas e diretrizes que orientam como o processo de verificação e validação de *software* será conduzido (ISTQB – International Software Testing Qualifications Board, 2024). Ela define os métodos utilizados para identificar defeitos, avaliar a conformidade com requisitos e garantir atributos de qualidade, considerando diferentes perspectivas de análise.

Os testes de caixa-preta avaliam o comportamento externo do *software* sem considerar sua implementação interna. O foco recai sobre requisitos funcionais e não funcionais, verificando se o sistema responde corretamente a entradas específicas (SOMMERVILLE, 2011).

A abordagem de caixa-branca examina a lógica interna, caminhos de execução e estruturas de controle do código. O testador necessita conhecimento detalhado da implementação para garantir cobertura adequada e identificar falhas estruturais (MYERS; SANDLER; BADGETT, 2011).

Os testes de caixa-cinza combinam elementos das abordagens de caixa-preta e caixa-branca, permitindo ao testador utilizar informações parciais da estrutura interna

(PRESSMAN, 2014). Com isso, é possível criar cenários mais precisos sem exigir conhecimento completo do código.

Zimmermann e Koziolek (2023) apresentaram no *Workshop on Automated Testing* (A-TEST), colocalizado com a ASE 2023, uma abordagem que integra o modelo GPT-4 com Selenium WebDriver para automação de testes GUI de aplicações *web*, demonstrando que LLMs pré-treinados podem realizar testes sem necessidade de dados gravados de testadores humanos, alcançando cobertura de *branches* significativamente superior ao *monkey testing*.

A abordagem caixa-preta adotada neste trabalho alinha-se com essa linha de pesquisa, permitindo avaliar a capacidade dos modelos em traduzir requisitos funcionais diretamente em código executável para validação de interfaces *web*, cenário comum em equipes de QA que validam sistemas sem acesso ao código fonte (GAROUSI; YILDIRIM, 2018).

2.2.2 Níveis de Testes

Os níveis de teste constituem uma estrutura metodológica que organiza a verificação e a validação do *software* em etapas progressivas, cada uma com escopo, objetivos e artefatos específicos. Essa divisão sistemática permite que o produto seja examinado desde seus componentes mais elementares até sua avaliação como um sistema completo, favorecendo a identificação gradual de defeitos ao longo do ciclo de desenvolvimento.

Os testes unitários constituem o nível mais básico do processo de verificação, focando na validação de unidades individuais de código, como funções, classes ou módulos. Seu objetivo é assegurar que cada componente opere de forma isolada e conforme especificado (International Software Test Institute, 2025; ISTQB – International Software Testing Qualifications Board, 2024).

Os testes de integração verificam o comportamento emergente gerado pela combinação de diferentes módulos previamente validados individualmente. Nessa etapa, busca-se identificar falhas relacionadas à comunicação, troca de dados e dependências entre componentes(International Software Test Institute, 2025; ISTQB – International Software Testing Qualifications Board, 2024).

Os testes de sistema avaliam o *software* como um produto completo, considerando seus requisitos funcionais e não funcionais sob condições reais ou simuladas de uso. Esse nível verifica desde o desempenho até a segurança e a usabilidade, assegurando que o sistema se comporta como um todo integrado (International Software Test Institute, 2025; ISTQB – International Software Testing Qualifications Board, 2024).

Os testes de aceitação são conduzidos a partir da perspectiva do usuário final

ou do cliente, buscando confirmar se o sistema atende às necessidades do negócio e aos critérios definidos contratualmente. Esses testes podem assumir diferentes formas, como User Acceptance Testing (UAT) ou testes alfa e beta.

No contexto deste trabalho, o foco recai sobre testes de sistema automatizados para aplicações *web*, especificamente testes *end-to-end* que simulam interações completas do usuário. Esses testes, implementados através do *framework* Selenium combinado com Pytest, situam-se na fronteira entre testes de sistema e de aceitação: validam o comportamento completo da aplicação como um todo integrado (característica de testes de sistema) sob a perspectiva de fluxos reais do usuário (característica de testes de aceitação) (GAROUSI; YILDIRIM, 2018).

2.2.3 Testes Automatizados

Testes automatizados referem-se ao uso de ferramentas e *scripts* para a execução de testes de *software* de maneira automática, eliminando a necessidade de intervenção manual na maior parte do tempo. Seu principal objetivo é validar o comportamento do sistema, garantindo que as funcionalidades estejam corretas e que o *software* opere conforme esperado.

O uso de testes automatizados é principalmente motivado pela necessidade de aumentar a frequência e a consistência das validações, reduzindo riscos de regressão e agilizando ciclos de entrega. (PRESSMAN, 2014) destaca que a automação é especialmente eficaz em cenários onde há grande volume de funcionalidades, demanda por validações repetitivas ou atualizações contínuas do código. Ao automatizar a execução desses testes, as equipes asseguram maior cobertura, diminuição de erros humanos e maior previsibilidade na detecção de falhas.

A literatura evidencia ainda que a automação contribui significativamente para a redução de custos e esforços de manutenção no longo prazo, embora requeira investimento inicial para estruturar *scripts* e *frameworks* adequados. Bertolino (2007) aponta que a automação funciona como um mecanismo preventivo, evitando que falhas avancem para etapas posteriores do ciclo de vida, onde o custo de correção tende a ser exponencialmente maior. Em ambientes industriais, estudos empíricos demonstram que a automação de testes, quando bem implementada, não apenas reduz retrabalho, como também aumenta a confiabilidade das entregas (GAROUSI; YILDIRIM, 2018).

Dessa forma, no contexto deste trabalho, é notório que a automação de testes desempenha um papel essencial na garantia de qualidade em sistemas modernos, oferecendo suporte à escalabilidade, estabilidade e velocidade exigidas por produtos digitais robustos e orientados a entregas contínuas. Estudos recentes (SCHÄFER et al., 2024; LI et al., 2025; FERREIRA et al., 2025) demonstram que modelos de linguagem

de grande escala (LLMs) podem automatizar a geração de testes, representando uma evolução promissora no paradigma de automação de qualidade de *software*.

2.3 Modelo de Linguagem de Grande Escala

Modelos de Linguagem de Grande Escala, conhecidos como Large Language Models (LLMs), representam uma classe de modelos de aprendizagem profunda projetados para processar, interpretar e gerar linguagem natural com elevado grau de coerência e fluidez (CHOPRA; PRASHAR; SAIN, 2013). Esses modelos são estruturados, em sua maioria, segundo a arquitetura Transformer, que se fundamenta em mecanismos de *self-attention* capazes de capturar dependências contextuais de curto e longo alcance (VASWANI et al., 2017), e são treinados em grandes volumes de dados textuais que permitem a aprendizagem de padrões linguísticos complexos e relações semânticas amplas.

2.3.1 Modelos *Open-Weight*, *Open-Source* e Proprietários

O grau de abertura de um LLM constitui um critério central para sua seleção em contextos de pesquisa e de aplicação prática, distinguindo-se três paradigmas principais. Em modelos proprietários (ou *closed-source*), tanto os pesos treinados quanto os dados de treinamento permanecem inacessíveis publicamente, e a interação ocorre exclusivamente por meio de APIs controladas pelo provedor, sem possibilidade de auditoria interna do sistema (MANCHANDA et al., 2024).

Já os modelos *open-source* disponibilizam publicamente não apenas os pesos treinados, mas também o código de treinamento e, idealmente, os dados utilizados, permitindo replicação completa do processo de desenvolvimento do modelo. Os modelos *open-weight*, por sua vez, ocupam uma posição intermediária: disponibilizam publicamente apenas os parâmetros (pesos) do modelo já treinado, permitindo que qualquer usuário baixe, execute localmente ou via API de inferência remota, e ajuste o modelo para tarefas específicas, sem que o código de treinamento ou os dados originais sejam revelados (HUNG, 2026).

Essa distinção tem implicações diretas para a reprodutibilidade científica: enquanto avaliações de modelos proprietários acessados via API estão sujeitas a atualizações não documentadas pelo provedor, capazes de invalidar a replicação de um estudo ao longo do tempo, os modelos *open-weight* permitem que pesquisadores implantem exatamente a mesma versão do modelo utilizada no estudo original, favorecendo a estabilidade e a replicabilidade dos resultados (HUNG, 2026). Essa característica, somada à ausência de custos de licenciamento por *token*, fundamenta a escolha metodológica desta pesquisa por modelos *open-weight*, alinhando-se ao contexto de

equipes de qualidade de *software* com recursos limitados discutido na Seção de Justificativa.

2.3.2 Arquiteturas Mixture of Experts e Transformers Densos

Entre os modelos *open-weight* disponíveis, duas abordagens arquiteturais predominam. Nos *transformers* densos, todos os parâmetros do modelo são ativados a cada inferência, o que garante previsibilidade de custo computacional, mas exige maior capacidade de processamento à medida que a escala do modelo aumenta. Já na arquitetura *Mixture of Experts* (MoE), o modelo é composto por múltiplos subconjuntos de parâmetros especializados (*experts*), dos quais apenas uma fração é ativada em cada passagem de inferência, selecionada dinamicamente por um mecanismo de roteamento. Essa abordagem permite que modelos com um número total de parâmetros muito elevado mantenham custo de inferência comparável ao de modelos de menor escala, uma vez que somente os parâmetros ativos por *token* contribuem para o cálculo computacional.

A aplicação de LLMs em engenharia de *software* tem crescido exponencialmente nos últimos anos. Wang et al. (2024) conduziram uma pesquisa abrangente sobre o uso de LLMs em testes de *software*, identificando que esses modelos têm sido aplicados em múltiplas atividades de teste, desde geração de casos de teste até detecção de defeitos e melhoria de cobertura. Adicionalmente, Alshahwan et al. (2024) reportaram a aplicação industrial de LLMs na Meta para melhoria automatizada de testes unitários, demonstrando viabilidade prática em escala industrial. Esses trabalhos evidenciam a maturidade crescente de LLMs como ferramentas auxiliares em processos de QA, embora concentrados predominantemente em modelos proprietários, lacuna que esta pesquisa busca preencher ao investigar especificamente o desempenho de modelos *open-weight* nesse contexto.

Nesta pesquisa, em específico, a opção por modelos exclusivamente *open-weight* decorre diretamente da barreira econômica de acesso a soluções comerciais por disponibilizarem publicamente os pesos treinados e permitirem uso via API de inferência remota sem custos elevados de licenciamento, esses modelos representam a alternativa mais viável para equipes de qualidade de *software* com recursos limitados, público-alvo direto deste estudo. A distinção entre arquiteturas MoE e densas, por sua vez, é utilizada como um dos critérios de caracterização dos quatro modelos avaliados, permitindo investigar se a arquitetura adotada exerce influência sobre o desempenho na geração de *scripts* de teste automatizados.

2.4 Engenharia de Prompt

A engenharia de *prompt* consiste em um conjunto de métodos destinados à elaboração estruturada de instruções para orientar LLMs. Seu objetivo é controlar o comportamento do modelo de maneira previsível, precisa e alinhada aos requisitos da tarefa. O avanço dessa área tornou-se expressivo após os resultados apresentados por Brown et al. (2020), que demonstraram a capacidade dos LLMs de realizar tarefas complexas por meio de instruções cuidadosamente formuladas.

Diversas estratégias de *prompting* são discutidas na literatura, incluindo *zero-shot prompting* e *few-shot prompting*, amplamente estudadas em Liu et al. (2023), que apresentam um panorama sistemático sobre o paradigma "Pre-train, Prompt, Predict". Ademais, técnicas avançadas como *chain-of-thought prompting*, proposta por Wei et al. (2022), incentivam o modelo a explicitar seu raciocínio intermediário, enquanto o método *zero-shot chain-of-thought*, explorado por Kojima et al. (2022), demonstra que instruções simples podem induzir comportamentos de raciocínio estruturado mesmo sem exemplos adicionais.

Outras abordagens, como o *self-consistency prompting*, introduzido por Wang et al. (2023), combinam múltiplas cadeias de raciocínio para obter respostas mais estáveis e robustas. Complementarmente, estratégias como *role prompting* e *prompt templates* vêm sendo discutidas em aplicações práticas, incluindo sua aplicação na engenharia de *software*, conforme apresentado por White et al. (2023).

Nesta pesquisa, em específico, a engenharia de *prompt* é importante na padronização das instruções enviadas aos quatro modelos LLM avaliados, garantindo que todos recebam o mesmo nível de contexto, restrições e exemplos de saída esperada. Estudos recentes demonstram o impacto direto da técnica de *prompting* adotada sobre a corretude sintática, a cobertura de código e a qualidade de testes unitários gerados por LLMs, reforçando a necessidade de um protocolo padronizado de engenharia de *prompt* para garantir a comparabilidade entre diferentes modelos avaliados (OUÉDRAOGO et al., 2026).

2.5 Linguagem de Programação Python

Python é uma linguagem de programação de alto nível, interpretada e multiparadigma, criada por Guido van Rossum no início da década de 1990 e projetada com ênfase na legibilidade do código e na simplicidade sintática (ROSSUM; DRAKE, 2009). Seu modelo de tipagem dinâmica e gerenciamento automático de memória tornam a linguagem acessível a iniciantes, sem abrir mão de recursos avançados para aplicações complexas, e seu suporte a múltiplos paradigmas. Além de ser orientado

a objetos, funcional e imperativo, a linguagem confere flexibilidade a diferentes necessidades de projeto (DOWNEY, 2015). Nesta pesquisa, em específico, a linguagem Python é a base de implementação dos *scripts* de teste automatizados gerados pelos modelos LLM avaliados, integrando os *frameworks* Selenium e Pytest às APIs de inferência remota utilizadas no experimento.

2.6 *Frameworks* de Automação de Teste

Frameworks de automação de testes são estruturas que proporcionam uma base organizada para o desenvolvimento, execução e gerenciamento de testes automatizados. O uso dessas estruturas permite a criação de *scripts* de testes consistentes, reutilizáveis e de baixa manutenção a fim de otimizar o ciclo de testes e obter *feedbacks* sobre a qualidade do *software* de forma contínua (NAGLE, 2000).

2.6.1 Selenium

O Selenium é uma ferramenta que permite a simulação de interações reais do usuário, como cliques, preenchimento de formulários e navegação entre páginas, tornando-o especialmente útil para testes de aceitação e regressão em ambientes de desenvolvimento contínuo (Selenium Project Contributors, 2025). Por causa da sua compatibilidade com diversas linguagens de programação e navegadores, o Selenium facilita a criação de *scripts* reutilizáveis e robustos, proporcionando maior cobertura e eficiência na validação de sistemas *web* (MESZAROS, 2007; GUPTA; CHAUHAN, 2019).

O Selenium trata-se do *framework* de automação de testes *web* mais amplamente documentado e utilizado na indústria, o que maximiza a probabilidade de os modelos LLM avaliados terem sido expostos a exemplos representativos de sua sintaxe durante o treinamento, condição necessária para uma avaliação justa da capacidade de geração de código dos modelos (ZIMMERMANN; KOZIOLEK, 2023). Além disso, sua API orientada a objetos, baseada na localização e manipulação de elementos do DOM (*Document Object Model*), permite mapear de forma direta e granular cada ação prevista nos casos de teste definidos nessa pesquisa, tal como cliques, preenchimento de campos e verificações de estado, o que possibilita comparar objetivamente diferentes implementações de um mesmo cenário entre os quatro modelos avaliados. Também por ser *open-source* e independente de licenciamento, o Selenium está alinhado ao critério de acessibilidade que orienta a seleção de todas as demais tecnologias empregadas neste estudo, coerente com o foco em soluções de baixo custo discutido na Justificativa deste trabalho.

2.6.2 Pytest

O Pytest é um *framework* amplamente utilizado para automação de testes em Python, possui uma sintaxe simples, recursos expressivos e suporte nativo a testes unitários, funcionais e de integração (BARBOSA; HORA, 2022). Por ser uma ferramenta de código aberto e altamente extensível (pytest contributors, 2025), o Pytest permite a criação de suítes de teste flexíveis e escaláveis, com suporte a *plugins* que ampliam sua capacidade e o adaptam a diferentes contextos de desenvolvimento. Essa flexibilidade tem tornado o Pytest o formato de saída padrão adotado por sistemas baseados em LLMs para geração automática de *scripts* de teste, como demonstrado por Han e Zhu (2025), cujo sistema multiagente gera, executa e avalia *scripts* Pytest a partir de especificações de API, reforçando a relevância do *framework* como alvo de integração entre modelos de linguagem e automação de testes.

A adoção do Pytest neste trabalho decorre diretamente da escolha metodológica pela linguagem Python, uma vez que se trata do *framework* de testes nativo mais consolidado desse ecossistema, o que evita a introdução de dependências externas ou adaptações de sintaxe que pudessem introduzir viés na avaliação dos modelos. Além disso, seu sistema de descoberta automática de testes e sua sintaxe baseada em funções, sem a exigência de estruturas orientadas a objetos como em *frameworks* concorrentes (por exemplo, *unittest*), tornam o código gerado pelos LLMs mais direto de padronizar e comparar entre os quatro modelos avaliados, já que reduz a variabilidade estrutural não relacionada à qualidade do teste em si. Por fim, a saída padronizada do Pytest, que reporta de forma clara o status de cada teste como *Passed*, *Failed* ou *Error*, viabiliza diretamente o cálculo automatizado da métrica de *PASSRATE*, central para responder à pergunta principal desta pesquisa, sem a necessidade de *parsers* customizados para interpretação dos resultados de execução.

2.7 Trabalhos Relacionados

Esta seção apresenta uma revisão de trabalhos recentes que investigam a aplicação de LLMs no contexto de automação de testes de *software*, com foco particular em geração automática de casos de teste para aplicações web. A análise desses estudos permite identificar tendências, lacunas e oportunidades de pesquisa que fundamentam e contextualizam os objetivos deste trabalho.

2.7.1 Teste Automatizado de Aplicações Web: Geração de Casos de Teste de Ponta-a-Ponta com Modelos de Linguagem de Grande Escala e Grafos de Transição de Telas

O trabalho apresenta uma abordagem integrada para geração automatizada de casos de teste voltados a aplicações web. A proposta combina LLMs com duas estruturas formais: grafos de transição de telas, responsáveis por mapear possíveis caminhos de navegação, e grafos de estado de formulários, que representam dependências, condicionais e configurações de campos presentes nas interfaces.

O sistema analisa esses modelos para construir automaticamente sequências completas de interação, incluindo ações de navegação, preenchimento de formulários e submissão. Além disso, o método gera *scripts* Selenium prontos para execução, possibilitando validação automática de ponta-a-ponta. Os resultados experimentais indicam que a solução alcança maior cobertura e maior capacidade de lidar com formulários dinâmicos quando comparada a abordagens tradicionais, demonstrando o potencial dos LLMs como ferramenta de apoio na automação de testes em ambientes web.

Este estudo demonstra a viabilidade técnica de utilizar LLMs para geração de scripts Selenium, *framework* também adotado na presente pesquisa. No entanto, o trabalho não especifica quais modelos LLM foram utilizados nem aborda questões de custo e acessibilidade, lacunas que este TCC pretende preencher ao focar especificamente em modelos *open-weight* acessíveis gratuitamente.

2.7.2 AutoQALLMs: Automação de Testes de Aplicações Web com LLMs e Selenium

O trabalho propõe o AutoQALLMs, um *framework* de teste automatizado para aplicações web que integra LLMs comerciais, como GPT-4, Claude 4.5 e Grok, com o Selenium WebDriver e a biblioteca *BeautifulSoup*. Este sistema é projetado para realizar testes com *“one-click testing”*, onde o usuário fornece apenas uma URL como entrada.

O *framework* extrai elementos HTML da página, utiliza a API do LLM para gerar *scripts* de teste baseados em Selenium, e emprega expressões regulares para otimizar e aumentar a manutenibilidade desses *scripts*. Avaliado em 30 websites, o AutoQALLMs demonstrou reduzir drasticamente o tempo de criação de casos de teste, alcançar uma ampla cobertura de testes e permitir a rápida regeneração de testes em resposta a mudanças na estrutura da página.

O estudo conclui que a abordagem oferece um forte equilíbrio entre velocidade

e confiabilidade, posicionando-a como um “acelerador de testes” eficaz para ambientes ágeis. No entanto, o trabalho reconhece limitações na validação de lógicas mais profundas e no tratamento de mudanças de página altamente complexas.

AutoQALLMs fornece evidências empíricas importantes sobre a eficácia de LLMs na geração de testes Selenium, alcançando coberturas próximas a testes manuais (96-98%). Entretanto, o estudo concentra-se exclusivamente em modelos comerciais (GPT-4, Claude 4.5, Grok), cujo custo de uso contínuo pode ser proibitivo para pequenas e médias empresas. Este trabalho complementa essa pesquisa ao investigar se modelos *open-weight* acessíveis podem alcançar níveis comparáveis de desempenho, democratizando o acesso a essas tecnologias. Além disso, enquanto AutoQALLMs foca em automação de geração com “um clique”, este TCC enfatiza análise comparativa sistemática de métricas de qualidade e confiabilidade entre diferentes modelos.

2.7.3 Automated Generation of End-to-End Web Test Cases via a Generic AI Agent: A Comparative Study of DeepSeek V3 and Claude Sonnet 4

O estudo avalia a capacidade do agente de IA genérico Suna em gerar testes de ponta-a-ponta automatizados para aplicações web, utilizando Selenium WebDriver e JUnit 5, com base apenas em um prompt de linha de comando. O *framework* Suna foi configurado com dois modelos LLM: DeepSeek V3 e Claude Sonnet 4, para testar nove websites distintos.

Os resultados agregados mostram que o Claude Sonnet 4 alcançou uma taxa de sucesso de 70,1% (336 de 479 testes executados com sucesso), significativamente superior à taxa de 34,3% (165 de 481 testes) do DeepSeek V3. Uma análise de custo-benefício indicou um custo médio de $0,15 por caso de teste bem-sucedido gerado pelo modelo pago, considerado razoável dada a complexidade do teste de ponta-a-ponta.

O trabalho também detalha a diferença nos tipos de erros observados: DeepSeek V3 apresentou 50% dos erros como `NoSuchElementException`, enquanto Claude Sonnet 4 teve 56,3% dos erros como `TimeoutException`. O estudo conclui que agentes de IA genéricos são mais eficazes do que interações diretas com LLMs para essa tarefa e sugere uma estratégia de teste incremental, começando com modelos gratuitos e complementando com modelos de maior qualidade para otimizar custos.

O estudo faz uma comparação direta entre um modelo *open-source* (DeepSeek V3) e um modelo comercial (Claude Sonnet 4) e identifica que DeepSeek V3, apesar de gratuito, apresenta taxa de sucesso significativamente inferior (34,3% vs 70,1%), sugerindo que a escolha do modelo impacta fortemente a qualidade dos testes gerados. No entanto, o estudo limita-se a apenas dois modelos e utiliza JUnit 5 em vez de Pytest, enquanto este trabalho expande a análise para quatro modelos diferentes e adota a

stack Python/Selenium/Pytest, mais comum em contextos de automação de testes web.

2.7.4 Síntese Comparativa e Posicionamento deste Trabalho

A Tabela 1 sintetiza as principais características dos trabalhos relacionados em comparação com a presente pesquisa.

Tabela 1 – Comparação entre trabalhos relacionados e este TCC

Trabalho	**Modelos Avaliados**	**Frameworks de Teste**	**Foco Principal**	**Limitações**
Grafos de Transição com LLMs (VO et al., 2025)	Não especificado	Selenium	Geração de testes baseada em grafos formais	Modelo LLM não identificado; custos e critérios de avaliação não discutidos
AutoQALLMs (MALLIPEDDI et al., 2025)	GPT-4, Claude 4.5, Grok	Selenium, BeautifulSoup	Automação de testes em abordagem *one-click*	Restrito a modelos comerciais de alto custo
Suna com DeepSeek e Claude (MONTEIRO et al., 2025)	DeepSeek V3, Claude Sonnet 4	Selenium, JUnit 5	Comparação entre agentes baseados em IA	Avaliação limitada a dois modelos e à linguagem Java
Esta Pesquisa	**GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B, Qwen3 Coder 480B**	**Selenium, Pytest**	**Análise comparativa de desempenho e confiabilidade de modelos *open-weight* acessíveis**	**Validação restrita a um único ambiente web**

Fonte: Elaborado pela autora (2026).

2.7.5 Lacunas Identificadas e Contribuições deste Trabalho

A análise dos trabalhos relacionados permite identificar as seguintes lacunas na literatura atual:

1. Ausência de foco em modelos *open-weight*: Os estudos existentes concentram-se predominantemente em modelos comerciais premium (GPT-4o, Claude), com pouca atenção a alternativas com pesos disponíveis publicamente, apesar de seu crescente uso pela comunidade de desenvolvedores.

2. Limitação no número de modelos comparados: Trabalhos que comparam modelos geralmente limitam-se a dois ou três opções, não explorando a diversidade de escalas, arquiteturas e especializações disponíveis no ecossistema *open-weight*.

3. Falta de análise de viabilidade econômica: Poucos estudos discutem explicitamente a relação custo-benefício ou a acessibilidade das soluções propostas para organizações com recursos limitados, lacuna crítica em contextos como o mercado brasileiro de TI.

4. Escassez de métricas de confiabilidade detalhadas: Embora trabalhos reportem taxas de sucesso gerais, há pouca análise sistemática de tipos específicos de erros, manutenibilidade do código gerado e aderência a padrões de *frameworks*.

Este TCC posiciona-se para preencher essas lacunas através das seguintes contribuições:

1. Análise comparativa abrangente de modelos *open-weight*: Avaliação sistemática de quatro modelos LLM com pesos publicamente disponíveis — GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B e Qwen3 Coder 480B —, contemplando diferentes escalas, arquiteturas (MoE e transformers densos) e especializações (generalistas vs. especialistas em código).

2. Foco em acessibilidade e viabilidade prática: Ênfase em soluções que podem ser adotadas por equipes com recursos limitados, mediante acesso via API, refletindo a realidade de 62% das empresas brasileiras de TI que citam custo como barreira (ABES, 2024).

3. Avaliação multidimensional de qualidade: Análise não apenas de taxa de sucesso, mas também de cobertura de cenários, requisitos e defeitos encontrados.

4. Contextualização para *stack* Python: Uso de Pytest e Selenium com Python, *stack* amplamente adotada na comunidade de desenvolvimento web, em contraste com estudos que utilizam JUnit/Java.

5. Transparência metodológica sobre limitações: Reconhecimento explícito de restrições orçamentárias e infraestruturais como parte da metodologia, transformando-as em critérios de seleção válidos que refletem cenários reais de uso.

Dessa forma, este trabalho não apenas complementa os estudos existentes, mas também oferece uma perspectiva prática e acessível sobre a aplicação de LLMs em automação de testes, contribuindo tanto para o avanço do conhecimento científico quanto para orientações aplicáveis a profissionais de qualidade de *software* em contextos de recursos limitados.

3 Metodologia

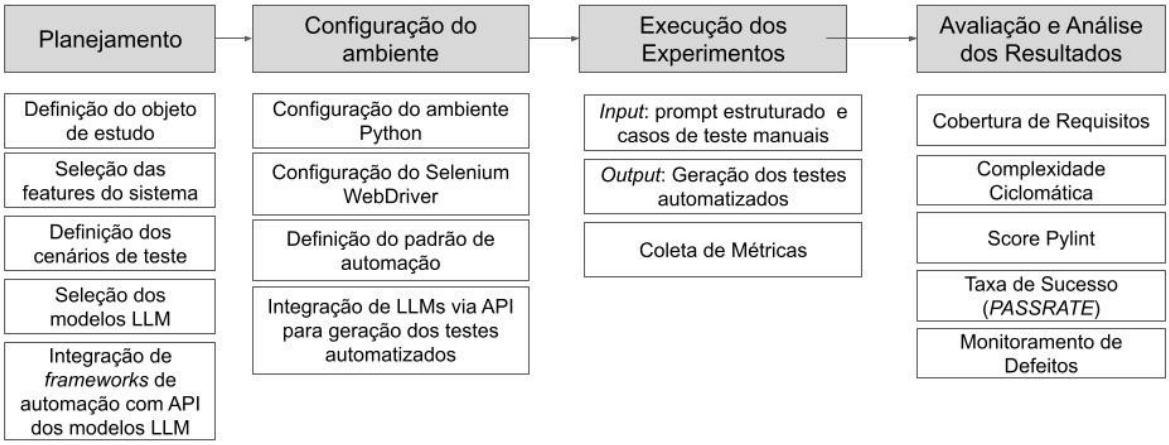

Figura 1 – Diagrama do processo metodológico geral da pesquisa.

Fonte: Elaborado pela autora (2026).

Este estudo consiste em uma abordagem quantitativa e experimental, baseada em *benchmarking*, com o objetivo de coletar e analisar dados por meio da avaliação comparativa de LLMs aplicados à geração de *scripts* de testes automatizados na linguagem Python. A metodologia de *benchmarking* adotada segue diretrizes que enfatizam a comparabilidade, a reprodutibilidade e o uso de métricas objetivas (SIM; EASTERBROOK; HOLT, 2003). Nesse contexto, diferentes modelos de linguagem são avaliados a partir da execução de tarefas padronizadas, utilizando os mesmos dados de entrada e condições experimentais controladas.

A pesquisa foi conduzida em um ambiente controlado que simulava cenários reais de aplicação, no qual os modelos foram integrados ao processo de automação de testes utilizando os *frameworks* Selenium e Pytest. Todos os experimentos foram realizados com o uso de entradas padronizadas, garantindo a consistência na coleta e análise dos resultados.

Conforme ilustrado na Figura 1, o processo metodológico está organizado em quatro etapas macro, executadas de forma sequencial:

1. Planejamento: Definição do objeto de estudo, das *features* do sistema, dos cenários de teste e dos modelos LLM selecionados;

2. Configuração do ambiente: Preparação do ambiente Python, do Selenium WebDriver e integração dos modelos LLM via API;

3. Execução dos Experimentos: Envio dos *prompts* estruturados aos modelos para geração dos testes automatizados e coleta das métricas;

4. Avaliação e Análise dos Resultados: Consolidação e interpretação comparativa das métricas coletadas entre os modelos avaliados.

3.1 Planejamento

3.1.1 Definição do objeto de estudo

O ambiente de testes do estudo consiste na aplicação web de *e-commerce* Amazon Brasil (<https://www.amazon.com.br>), selecionada por ser uma plataforma pública, amplamente conhecida, com comportamento representativo de aplicações reais de comércio eletrônico. Os testes automatizados foram restritos a funcionalidades acessíveis publicamente, sem necessidade de autenticação e sem utilização de conta pessoal, abrangendo exclusivamente navegação, busca, visualização de produtos e manipulação do carrinho de compras, conforme detalhado na Tabela 2. A coleta de dados foi realizada no período de 01/05/2026 a 31/05/2026, utilizando o navegador Google Chrome na versão 149.0.7827.201 em conjunto com o Selenium WebDriver para automatizar a interação com a interface do usuário e o Pytest para gerenciamento do *framework* e da suíte de testes.

Para mitigar variações decorrentes da natureza dinâmica de uma plataforma de produção comercial, os mesmos produtos e termos de busca foram utilizados de forma padronizada em todas as execuções dos quatro modelos LLM avaliados, garantindo que as entradas de teste permanecessem constantes ao longo do experimento. Ainda assim, como as execuções foram realizadas em diferentes momentos ao longo do mês de maio de 2026, não é possível descartar totalmente a ocorrência de variações de elementos, tal como *widgets* de disponibilidade ou alterações bruscas de *layout* da interface entre uma execução e outra. Não foram identificadas, contudo, alterações perceptíveis na estrutura da página ou nos produtos utilizados durante o período de coleta que pudessem comprometer a validade dos resultados obtidos.

3.1.1.1 Nota Ética e Metodológica sobre o Uso de Site Público

Os testes automatizados realizados neste estudo foram conduzidos exclusivamente para fins acadêmicos e de avaliação comparativa dos *scripts* gerados por modelos LLM, não tendo como objetivo auditar, explorar vulnerabilidades, coletar dados sensíveis, realizar *scraping* sistemático ou interferir no funcionamento da plataforma analisada. A aplicação utilizada como ambiente experimental foi acessada apenas por meio

de funcionalidades públicas, sem autenticação, sem criação ou utilização de conta pessoal, sem realização de compras e sem manipulação de dados privados de usuários.

Para reduzir impactos sobre a infraestrutura da plataforma, o experimento foi limitado a um volume reduzido de interações automatizadas, correspondente aos casos de teste definidos na pesquisa, com foco em fluxos comuns de navegação, busca, visualização de produtos e manipulação local do carrinho. As execuções foram conduzidas de forma controlada, evitando ações repetitivas em larga escala, envio massivo de requisições ou qualquer comportamento intencionalmente semelhante a carga, abuso ou varredura automatizada.

3.1.2 Seleção das *features* do sistema

O conjunto de testes desenvolvido neste trabalho se distribui entre sete funcionalidades principais da aplicação web analisada: Navegação, busca de produtos, visualização de produtos, carrinho de compras, disponibilidade, filtros e página inicial. Essa divisão foi definida com base na criticidade e na frequência de uso dessas funcionalidades no fluxo principal do usuário, garantindo uma cobertura adequada dos cenários mais relevantes do sistema.

Tabela 2 – Matriz de rastreabilidade de requisitos por *feature*

ID Req	**Feature**	**Descrição do Requisito**	**Casos de Teste**
REQ-01	Navegação	Permitir que o usuário navegue pelo menu principal da aplicação.	CT-01
REQ-02	Busca de Produtos	Permitir a busca de produtos por palavras-chave, considerando diferentes formatos de entrada.	CT-02 a CT-06
REQ-03	Visualização de Produtos	Exibir informações principais e detalhadas dos produtos.	CT-07, CT-08, CT-09, CT-11
REQ-04	Carrinho de Compras	Permitir adicionar, remover e manter produtos no carrinho sem autenticação.	CT-10, CT-12 a CT-15
REQ-05	Disponibilidade	Impedir a adição de produtos indisponíveis ao carrinho.	CT-16
REQ-06	Filtros	Permitir filtrar produtos por faixa de preço.	CT-17 a CT-19
REQ-07	Página Inicial	Carregar corretamente a página inicial da aplicação.	CT-20

Fonte: Elaborado pela autora (2026).

Conforme apresentado na Tabela 2, os requisitos do sistema foram mapeados às respectivas *features* e casos de teste.

3.1.3 Definição dos cenários de teste

A seleção dos casos de teste adotada nesta pesquisa foi realizada de forma sistemática, com base na identificação das funcionalidades críticas do fluxo principal de navegação e compra em uma aplicação de comércio eletrônico. Foram definidos 20 casos de teste, distribuídos entre sete *features* centrais: navegação, busca de produtos, visualização de produtos, carrinho de compras, disponibilidade, filtros e página inicial. Essa seleção considerou critérios como frequência de uso pelo usuário final, impacto direto na experiência de compra e viabilidade de automação com ferramentas de teste funcional.

Ao todo, foram gerados 80 *scripts* de teste automatizados, resultantes da combinação entre os 20 casos de teste definidos e os quatro modelos LLM avaliados, sem descartes na etapa de geração de código. Cada *script* foi submetido a uma execução inicial; em caso de erro ou falha, o *script* foi executado em até duas tentativas adicionais (totalizando até três execuções), a fim de distinguir falhas genuínas de instabilidades momentâneas na infraestrutura de teste, como oscilações de conectividade ou elementos temporariamente indisponíveis na página.

Apenas um dos 80 *scripts*, correspondente ao CT-15, gerado pelo Qwen3 Coder 480B, foi descartado na etapa de avaliação, por não conter a asserção adequada para validar a exibição da mensagem de confirmação ao adicionar um produto ao carrinho diretamente da página de resultados de busca, inviabilizando a verificação automática do critério de aceitação do requisito testado. Em conformidade com a Categoria 4 (Seção 3.3.3), esse *script* foi contabilizado como reprovação no cálculo do *PASSRATE* do Qwen3 Coder 480B, mantendo o denominador de 20 casos de teste para esse modelo.

A visão geral dos casos de teste é apresentada na Seção A, enquanto a descrição detalhada de cada cenário, incluindo pré-condições, passos e critérios de aceitação, encontra-se na Seção B. Os cenários contemplam tanto casos positivos quanto negativos, além de variações de entrada e persistência de estado.

3.1.4 Seleção dos modelos LLM

A seleção dos modelos de *Large Language Models* (LLMs) utilizados neste trabalho foi conduzida considerando critérios técnicos alinhados aos objetivos de automação de testes de software e restrições de viabilidade prática inerentes ao contexto de uma pesquisa acadêmica. Os seguintes critérios foram estabelecidos para orientar a seleção dos modelos:

3.1.4.0.1 Critérios Técnicos

1. Capacidade demonstrada em *benchmarks* de programação, como HumanEval e MBPP;
2. Disponibilidade por meio de APIs gratuitas ou plataformas *open-source*;
3. Inclusão de modelos generalistas e especializados em código, contemplando diferentes escalas;
4. Disponibilidade de relatórios técnicos e especificações públicas.

3.1.4.0.2 Restrições Práticas

Além disso, a pesquisa foi conduzida considerando as seguintes restrições práticas, que definiram o escopo de modelos acessíveis:

a) Infraestrutura Computacional: Limitações de *hardware* local que inviabilizaram a execução de modelos de grande escala ($>$70B parâmetros) de forma auto-hospedada, direcionando a escolha para modelos disponíveis via plataformas *cloud* com *tiers* gratuitos ou de baixo custo.

b) Implementação Técnica: Necessidade de compatibilidade com ferramentas de orquestração de LLMs que facilitassem a integração com os *frameworks* de automação (Selenium/Pytest), priorizando plataformas como Ollama, Google AI Studio e Groq que oferecem APIs unificadas e bem documentadas.

c) Representatividade Prática: Alinhamento com cenários reais enfrentados por pequenas e médias empresas, *startups* e equipes de desenvolvimento que não dispõem de orçamentos significativos para ferramentas de IA, mas ainda assim buscam incorporar LLMs em seus processos de qualidade.

Segundo o relatório da ABES (2024), aproximadamente 62% das empresas brasileiras de TI de pequeno e médio porte citam o custo como principal barreira para adoção de soluções de IA generativa, tornando as soluções *open-weight* e de baixo custo particularmente relevantes para o mercado nacional.

3.1.4.1 Modelos LLM Selecionados Para a Pesquisa

Dessa forma, foram selecionados quatro modelos para análise comparativa completa:

1. GPT-OSS 120B

   O GPT-OSS 120B é um modelo generalista *open-weight* desenvolvido pela OpenAI, lançado em agosto de 2025 sob licença Apache 2,0, o primeiro modelo de grande escala da empresa com pesos publicamente disponíveis desde o GPT-2, em 2019. Baseia-se em uma arquitetura MoE com aproximadamente 117 bilhões de parâmetros totais, dos quais cerca de 5,1 bilhões são ativados por *token* durante a inferência, o que permite eficiência computacional elevada com desempenho competitivo. O modelo suporta janela de contexto de até 128 mil *tokens* e oferece níveis ajustáveis de raciocínio (*low*, *medium* e *high*), além de suporte nativo a chamadas de ferramentas e execução de código Python. Em *benchmarks* de programação, alcança 88,3% no HumanEval, desempenho comparável ao GPT-4o-mini em tarefas de geração de código. O acesso é realizado via API de inferência remota pela plataforma Ollama.

2. Gemma 4 26B

   O Gemma 4 26B é um modelo generalista *open-weight* desenvolvido pelo Google DeepMind, lançado em abril de 2026 sob licença Apache 2,0. Adota arquitetura MoE com 26 bilhões de parâmetros totais, dos quais aproximadamente 3,8 bilhões são ativados por *token* durante a inferência, conferindo latência e custo computacional equivalentes a modelos de menor escala. O modelo suporta janela de contexto de até 256 mil *tokens* e foi desenvolvido com base na mesma pesquisa que fundamenta a família Gemini 3, incorporando capacidades de raciocínio configurável e suporte nativo a chamadas de função para fluxos agênticos. Em *benchmarks* de programação, alcança 78,5% no HumanEval, posicionando-se entre os modelos *open-weight* de melhor desempenho em sua faixa de parâmetros ativos. O acesso é realizado via API de inferência remota pela plataforma Google AI Studio.

3. Llama 3.3 70B

   O Llama 3.3 70B é um modelo generalista *open-weight* desenvolvido pela Meta AI, lançado em dezembro de 2024. Com 70 bilhões de parâmetros e arquitetura *transformer* densa otimizada com *Grouped-Query Attention* (GQA), o modelo foi ajustado por meio de *Supervised Fine-Tuning* (SFT) e *Reinforcement Learning from Human Feedback* (RLHF) para maior aderência a instruções. O modelo suporta janela de contexto de até 128 mil *tokens* e apresenta capacidades multilíngues em 8 idiomas, incluindo português. Em *benchmarks* de programação, alcança 88,4% no HumanEval, desempenho comparável ao Llama 3.1 405B, modelo com escala aproximadamente seis vezes superior. No MMLU, que avalia compreensão de linguagem em múltiplos domínios, atinge 86,0%. O acesso é realizado via API de inferência remota pela plataforma Groq.

4. Qwen3 Coder 480B

   O Qwen3 Coder 480B é um modelo especialista em programação desenvolvido pela equipe Qwen da Alibaba Cloud, lançado em julho de 2025 sob licença Apache 2,0. Baseia-se em arquitetura MoE com 480 bilhões de parâmetros totais, dos quais 35 bilhões são ativados por *token* durante a inferência (8 de 160 especialistas por passagem). O modelo suporta janela de contexto de até 256 mil *tokens* nativamente, com possibilidade de extensão a 1 milhão de *tokens* via método YaRN. Projetado especificamente para tarefas de codificação agêntica, o modelo apresenta suporte nativo a chamadas de ferramentas, raciocínio de longo horizonte e integração com o protocolo MCP (*Model Context Protocol*). Seus *benchmarks* primários são SWE-Bench Verified e TAU-Bench, focados em engenharia de software real, onde alcança desempenho comparável ao Claude Sonnet 4 entre os modelos *open-weight*. O acesso é realizado via API de inferência remota pela plataforma Ollama.

A Tabela 3 sintetiza as principais características técnicas dos quatro modelos LLM *open-weight* selecionados, evidenciando diferenças em escala, arquitetura, contexto e desempenho em geração de código.

Observa-se uma variação expressiva no número total de parâmetros, do Gemma 4 26B (26B) ao Qwen3 Coder 480B (480B), mas o número de parâmetros *ativos* por inferência revela um padrão distinto: os três modelos de arquitetura MoE, GPT-OSS 120B, Gemma 4 26B e Qwen3 Coder 480B, ativam apenas uma fração de seus parâmetros totais a cada consulta, enquanto o Llama 3.3 70B, de arquitetura densa, ativa a totalidade de seus 70 bilhões de parâmetros. Quanto ao contexto, Gemma 4 26B e Qwen3 Coder 480B suportam até 256 mil tokens, o dobro de GPT-OSS 120B e Llama 3.3 70B (128 mil tokens).

No HumanEval, GPT-OSS 120B (88,3%) e Llama 3.3 70B (88,4%) apresentam desempenho equivalente, superando o Gemma 4 26B (78,5%), enquanto o Qwen3 Coder 480B não reporta pontuação direta nesse *benchmark*, sendo avaliado prioritariamente por meio de SWE-Bench Verified e TAU-Bench, mais alinhados ao seu propósito de codificação agêntica.

3.1.5 Integração de *frameworks* de automação com API dos modelos LLM

Os *frameworks* selecionados para o estudo foram Selenium e Pytest, utilizados para a estruturação do projeto de automação de testes, mapeamento dos elementos da interface do ambiente de testes e desenvolvimento de funções para simulação de interações humanas. A integração de modelos de linguagem baseados em LLMs foi

Tabela 3 – Comparação dos modelos LLM *open-weight* selecionados

Modelo	**Parâmetros**	**Ativos**	**Arquitetura**	**Contexto**	**HumanEval**	**Categoria**
GPT-OSS 120B	117B	5,1B	MoE	128k	88,3%	Generalista
Gemma 4 26B	26B	3,8B	MoE	256k	78,5%	Generalista
Llama 3.3 70B	70B	70B	Dense	128k	88,4%	Generalista
Qwen3 Coder 480B	480B	35B	MoE	256k	N/D*	Especialista

Fonte: Elaborado pela autora (2026).

realizada por meio de API *cloud*, utilizando os clientes Ollama, Groq e Google AI Studio para geração automatizada de testes em Python.

3.2 Configuração do ambiente

A etapa de configuração do ambiente representou a base metodológica sobre a qual toda a automação de testes foi construída.

Inicialmente, realizou-se a configuração do ambiente Python, assegurando que todas as bibliotecas e dependências necessárias estivessem corretamente instaladas, incluindo pacotes específicos para automação de testes, manipulação de dados e integração com APIs externas.

Em seguida, procedeu-se à configuração do Selenium WebDriver, ferramenta responsável pela interação automatizada com o navegador, permitindo a execução de testes *end-to-end* de aplicações *web* de forma programática.

Complementarmente, foi definida a estratégia de automação, que incluiu a utilização de um gerenciador, no caso o Pytest, convenções de nomenclatura, organização dos *scripts*, gerenciamento de casos de teste e tratamento de exceções, a fim de garantir que os testes fossem escaláveis.

Por fim, a integração de LLMs via API permitiu a geração automatizada de *scripts* de teste e casos de uso, transformando descrições textuais de requisitos em implementações de teste estruturadas.

3.3 Métricas de Avaliação

3.3.1 Qualidade do Código

3.3.1.1 Score Pylint

O Pylint é uma ferramenta de análise estática que verifica a conformidade do código Python com o padrão PEP 8 e identifica erros potenciais (Pylint Contributors,

2025). A ferramenta atribui uma nota de 0 a 10, onde valores mais altos indicam melhor qualidade. Conforme apresentado na Tabela 4, os escores obtidos permitem classificar a qualidade do código gerado pelas LLMs.

Tabela 4 – Interpretação da pontuação de qualidade do código

Faixa de Pontuação	**Interpretação**
9,0 – 10,0	Código de excelente qualidade
7,0 – 8,9	Boa qualidade
5,0 – 6,9	Qualidade aceitável
$<$ 5,0	Necessita refatoração

Fonte: Elaborado pela autora (2026).

O comando de execução da medição da qualidade do código gerado é apresentado na listagem 3.1.

```
1 pylint teste_llm.py --output-format=json
```

Listing 3.1 – Execução da ferramenta Pylint

3.3.1.2 Complexidade Ciclomática

A complexidade ciclomática é uma métrica utilizada para mensurar o número de caminhos independentes existentes no fluxo de controle de um programa. Neste trabalho, essa métrica foi obtida por meio da ferramenta *Radon*, amplamente utilizada para análise estática de código em Python. Segundo apresentado na Tabela 5, valores mais baixos de complexidade ciclomática indicam código mais simples, com menor ramificação lógica, o que tende a facilitar a compreensão, a manutenção e a evolução do software (Radon Contributors, 2025).

Tabela 5 – Interpretação da complexidade ciclomática

Faixa de Valores	**Interpretação**
1 – 5	Baixa complexidade (código simples)
6 – 10	Complexidade moderada
11 – 20	Alta complexidade
$>$ 20	Muito complexa (difícil manutenção)

Fonte: Elaborado pela autora (2026).

A medição da complexidade ciclomática foi realizada utilizando o comando apresentado na listagem 3.2:

```
1 radon cc teste_llm.py -a
```

Listing 3.2 – Execução da ferramenta Radon

3.3.1.3 Cobertura de Requisitos

A cobertura de requisitos avalia o grau em que os *scripts* de teste automatizados, gerados por cada modelo LLM, implementam corretamente a lógica e as asserções necessárias para validar cada requisito funcional definido na matriz de rastreabilidade (Tabela 2).

O cálculo foi realizado por requisito e por modelo, segundo o seguinte protocolo:

1. Leitura e análise do código-fonte de cada *script* gerado pelo modelo LLM i, associado a um determinado requisito, conforme o mapeamento da Tabela 2;
2. Verificação, para cada caso de teste associado ao requisito, se o *script* correspondente implementa corretamente as ações e asserções necessárias para validar o comportamento exigido;
3. Cálculo do percentual de cobertura do requisito, considerando a proporção de casos de teste associados que foram corretamente implementados.

Para um requisito j associado a n_j casos de teste, a cobertura foi calculada como:

$$CR_j = \frac{R_{\text{testados},j}}{R_{\text{totais},j}} \times 100\% \tag{3.1}$$

Onde:

- CR_j: Cobertura do requisito j para um dado modelo LLM;
- $R_{\text{testados},j}$: Número de casos de teste associados ao requisito j cujo *script* implementou corretamente a validação exigida;
- $R_{\text{totais},j}$: Número total de casos de teste associados ao requisito j, conforme a Tabela 2.

Por exemplo, o requisito REQ-02 (Busca de Produtos) está associado a cinco casos de teste (CT-02 a CT-06); caso um modelo tenha implementado corretamente a validação em apenas dois desses cinco *scripts*, a cobertura desse requisito, para esse modelo, corresponde a 40%. A cobertura média de requisitos de cada modelo, apresentada nos Resultados, corresponde à média aritmética simples de CR_j entre os sete requisitos definidos. Cada caso de teste foi validado individualmente por meio dessa análise, a fim de assegurar que o sistema estivesse sendo testado de acordo com as necessidades do usuário.

3.3.2 Corretude Funcional

3.3.2.1 Taxa de Sucesso (*PASSRATE*)

A taxa de sucesso dos testes automatizados foi calculada considerando o número de testes executados com sucesso em relação ao número total de testes realizados. Um *script* foi considerado aprovado se pelo menos uma entre as até três execuções realizadas obtivesse *status Passed*; um *script* foi considerado reprovado apenas quando todas as tentativas de execução resultassem em falha, ou quando classificado na categoria *Scripts Descartados*. Uma taxa de sucesso elevada transmite confiabilidade na detecção de falhas do sistema. A Tabela 6 apresenta a interpretação adotada para a taxa de sucesso dos testes automatizados.

Assim, o cálculo realizado a seguir retorna o valor da métrica de *passrate*:

$$\text{Taxa de Sucesso} = \frac{\text{Testes Aprovados}}{\text{Total de Testes}} \times 100\% \tag{3.2}$$

Tabela 6 – Interpretação da taxa de sucesso dos testes

Faixa de Percentual (%)	**Interpretação**
90,0 – 100,0	Taxa de sucesso elevada
70,0 – 89,9	Taxa de sucesso satisfatória
50,0 – 69,9	Taxa de sucesso moderada
$<$ 50,0	Taxa de sucesso baixa

Fonte: Elaborado pela autora (2026).

3.3.3 Monitoramento de Defeitos

O monitoramento de defeitos consiste na identificação, classificação e contabilização dos erros presentes no código gerado pelos modelos LLM. Defeitos são categorizados segundo sua natureza para permitir análise qualitativa dos problemas mais comuns.

3.3.3.1 Categorização de Defeitos

Para fins de análise dos códigos gerados pelos modelos de linguagem, os defeitos identificados foram organizados e categorizados, permitindo a classificação sistemática dos problemas observados durante a execução e avaliação dos testes automatizados.

A categoria SEM_ERRO representa a ausência de qualquer defeito, ou seja, *scripts* executados com sucesso, sem erros sintáticos, erros relacionados ao Selenium ou erros lógicos, cujo comportamento observado corresponde ao critério de aceitação

definido para o caso de teste. As demais quatro categorias, detalhadas a seguir, representam os diferentes tipos de defeito identificados.

É importante distinguir a categoria SEM_ERRO da métrica de *PASSRATE*: enquanto o *PASSRATE* é uma métrica quantitativa, calculada automaticamente a partir do *status* de execução reportado pelo Pytest (*Passed* ou *Failed*), a categoria SEM_ERRO integra a classificação qualitativa de defeitos, atribuída durante a revisão manual do código-fonte de cada *script*. Um *script* aprovado no *PASSRATE* corresponde, em regra, a um *script* classificado como SEM_ERRO; contudo, a categorização de defeitos permite identificar, entre os *scripts* reprovados, a natureza específica da falha (sintática, Selenium, lógica ou descarte por ausência de asserção), informação não capturada pelo *PASSRATE* isoladamente.

3.3.3.1.1 Categoria 1: Erros Sintáticos

Compreendem defeitos que impedem a correta interpretação ou execução do código Python pelo interpretador, inviabilizando a execução dos testes automatizados. Exemplos incluem:

- Indentação incorreta;
- Parênteses ou colchetes não fechados;
- Importações ausentes ou incorretas;
- Uso de variáveis não definidas.

3.3.3.1.2 Categoria 2: Erros Relacionados ao Selenium

Englobam exceções lançadas pelo *framework* Selenium durante a interação com a interface web, geralmente associadas a problemas de sincronização, localização ou estado dos elementos da página. Os principais erros mapeados estão entre os listados a seguir:

- *NoSuchElementException*: Elemento não localizado na página;
- *TimeoutException*: Tempo limite excedido ao aguardar uma condição;
- *StaleElementReferenceException*: Referência a elemento que não está mais presente no DOM;
- *ElementNotInteractableException*: Elemento presente, porém não interagível.

3.3.3.1.3 Categoria 3: Erros Lógicos

Referem-se a situações em que o código é executado sem falhas sintáticas ou de *framework*, porém não valida corretamente o comportamento esperado da aplicação sob teste. Esses defeitos afetam diretamente a eficácia dos testes e incluem:

- Asserções incorretas, imprecisas ou ausentes;
- Fluxo de execução inadequado dos casos de teste;
- Utilização de dados de teste inválidos ou inconsistentes;
- Verificações parciais ou incompletas dos requisitos.

3.3.3.1.4 Categoria 4: Scripts Descartados

Compreende *scripts* que, embora executados até o fim sem erros de sintaxe, *framework* ou lógica evidente, não contêm as asserções necessárias para verificar automaticamente o critério de aceitação do requisito testado, inviabilizando a avaliação objetiva de aprovação ou reprovação. *Scripts* nessa categoria são contabilizados como reprovação para fins de cálculo do *PASSRATE*.

3.3.3.2 Método de Rastreio

O rastreio de defeitos foi adotado neste estudo com o objetivo de identificar, registrar e classificar falhas presentes no código de testes automatizados gerado pelos modelos LLM. O procedimento é composto pelas seguintes etapas:

1. Execução e captura de erros: o código gerado foi executado em ambiente controlado, sendo registradas falhas de compilação, exceções em tempo de execução e comportamentos incorretos durante a execução dos testes;
2. Classificação dos defeitos: cada defeito identificado foi analisado e classificado de acordo com a categorização de defeitos definida neste trabalho;
3. Mapeamento em planilha: os defeitos classificados foram registrados em uma planilha estruturada, contendo informações como modelo LLM, cenário de teste, categoria do defeito e descrição do problema, possibilitando análises quantitativas e comparativas.

3.4 Procedimento de Coleta de Dados

3.4.1 Etapa 1: Geração do Código

Para cada um dos 20 casos de teste definidos, foi solicitado a cada modelo LLM que gerasse o código correspondente usando Selenium e Pytest a partir de um *prompt* estruturado por meio da técnica *few-shot*.

1. Formulação de um único *prompt* composto por três blocos: o primeiro descrevendo objetivo, contexto e restrições; o segundo descrevendo o caso de teste; e o terceiro apresentando um código de exemplo esperado para *output*;
2. Envio do *prompt* para cada um dos quatro modelos LLM através de inferência remota via API;
3. Salvamento do código gerado pelos modelos LLM em arquivo Python;
4. Organização em estrutura de pastas por modelo.

A título de exemplo, o trecho a seguir apresenta os Blocos 1 e 2 do *prompt* completo utilizado para o caso de teste CT-01 (Navegação pelo menu principal). O Bloco 3 (exemplo de *output few-shot*), por ser extenso e idêntico para os 20 casos, é omitido aqui e apresentado uma única vez no Apêndice C, junto aos demais 19 *prompts* completos.

Prompt (Blocos 1 e 2) – CT-01, exemplo ilustrativo

```
Role
Você é um engenheiro de qualidade de software especializado em automação de testes web com Python, Pytest e Selenium.

- Contexto
Deseja-se gerar testes automatizados para uma aplicação web, a Amazon Brasil (https://www.amazon.com.br/).

# Objetivo
Gerar um teste automatizado utilizando Python, Pytest e Selenium baseado no caso de teste fornecido, seguindo os padrões de projeto de automação de testes.

# O projeto já utiliza:

- pytest
- Selenium
- Page Object Model
- seleniumpagefactory.Pagefactory
- O driver(driver) e a validação de internet(is_connected) são fixtures já presentes no conftest, chame-os como parâmetro no teste.

```

```
18
19 # Restrições de geração de código
20 - O código deve ser executável
21 - Retorne apenas o conteúdo bruto do arquivo Python.
22 - Não inclua markdown.
23 - Não inclua explicações.
24
25 # Restrições de desenvolvimento de teste
26 - Utilize:
27     "from seleniumpagefactory.Pagefactory import PageFactory" para mapeamento
           de elementos da tela
28 - Só adicione métodos caso o comportamento necessário não exista nas classes
       existentes.
29 - Não recrie Page Objects existentes.
30 - Reutilize classes, métodos e locators existentes.
31 - Todas as classes Page Object (ex: Home, SearchResults, ProductDetails, Cart)
        devem ser definidas no próprio arquivo de teste gerado, antes da função de
        teste.
32 - Não importe classes Page Object de módulos externos.
33 - Não utilize caminhos de importação como "from your_project..." ou similares.
34 - O fluxo completo deve estar em uma única função de teste.
35 - Métodos auxiliares podem existir apenas dentro das Page Objects.
36 - Utilize apenas os imports necessários para execução.
37 - Utilize WebDriverWait e expected_conditions.
38 - Não utilize time.sleep().
39 - Utilize nomenclatura em snake_case.
40 - Os testes devem seguir o padrão test_<id>.
41 - Utilize logging.info() para todos os passos [STEP] e resultados esperados [
       EXPECTED RESULT] do teste automatizado.
42 - Utilize o padrão Page Object Model.
43 - A lógica de interação com elementos deve permanecer dentro das classes Page
       Object.
44 - O teste deve conter apenas fluxo e validações.
45 - Utilize assertions explícitas com mensagens descritivas.
46
47 # Formato de output esperado
48 1. Criar o teste correspondente à descrição do teste e ao exemplo de output
       fornecido.
49 2. Utilizar asserts com mensagens descritivas para validar o comportamento
       esperado no cenário de teste apresentado.
50
51 # Estrutura esperada do arquivo gerado:
52 import logging
53 from seleniumpagefactory.Pagefactory import PageFactory, By, WebDriverWait
54 from selenium.webdriver.support import expected_conditions as EC
55
56 AMAZON_URL = "https://www.amazon.com.br/"
57
58 class Home(PageFactory):
59     ...
60
61 class SearchResults(PageFactory):
62     ...
63
64 def test_<id>(driver, is_connected):
```

```
    ...
--------------------------------------------------------------------------------


# Caso de Teste
Tipo de teste: Funcional - Teste de Sistema
Título: Navegação pelo menu principal

Objetivo:
- Verificar se o usuário consegue navegar pelo menu principal.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Página inicial carregada

Passos:
1. Acessar a página inicial da Amazon
2. Localizar o menu superior
3. Clicar na opção "Ofertas do Dia"

Resultado esperado:
- O sistema redireciona para a página de ofertas
- A URL é atualizada corretamente
- A lista de ofertas é exibida
```

A escolha e a autoria do exemplo *few-shot* seguiram critérios definidos previamente à execução dos experimentos. O exemplo utilizado (Bloco 3) corresponde à solução do caso de teste CT-18 (Filtrar produtos por faixa de preço máximo) e foi escrito manualmente pela autora, e não gerado por nenhum dos quatro modelos LLM avaliados. Essa escolha foi deliberada: caso o exemplo fosse produzido por um dos modelos em avaliação, o modelo autor do exemplo poderia obter vantagem indevida na comparação, uma vez que tenderia a reproduzir com maior fidelidade um padrão de código estilisticamente próximo ao seu próprio, enquanto os demais modelos precisariam generalizar a partir de um padrão alheio. Ao utilizar um exemplo de autoria humana, nenhum dos quatro modelos parte de uma posição de vantagem estilística em relação aos demais.

O mesmo exemplo (Bloco 3) foi mantido idêntico para os 20 casos de teste e para os quatro modelos avaliados, sem variação. Essa padronização foi adotada para isolar a variável de interesse do experimento, o modelo LLM, controlando o efeito de possíveis diferenças na qualidade ou no estilo do exemplo fornecido, que poderiam introduzir um viés de comparação caso exemplos distintos fossem utilizados para diferentes modelos ou casos de teste.

3.4.2 Etapa 2: Análise de Qualidade

Nesta etapa, os *scripts* de teste gerados por cada modelo LLM foram avaliados por meio de ferramentas de análise automatizada, conforme as métricas definidas na Seção de Métricas de Avaliação. A análise contemplou tanto aspectos relacionados à execução dos testes quanto características de qualidade estática e complexidade estrutural do código.

Inicialmente, os *scripts* foram executados com o Pytest, a fim de verificar a taxa de sucesso de execução dos testes, representada pela métrica *PASSRATE*. Em seguida, foi utilizada a ferramenta Pylint para avaliar a qualidade estática do código gerado, considerando aspectos como legibilidade, padronização, organização, documentação e possíveis problemas de manutenção. Por fim, a ferramenta Radon foi empregada para calcular a complexidade ciclomática dos *scripts*, permitindo analisar a simplicidade estrutural do código produzido pelos modelos.

3.4.3 Etapa 3: Execução de Testes Funcionais

Os *scripts* gerados foram executados contra o ambiente de teste para verificar se funcionavam corretamente.

1. Configuração do *WebDriver* do Selenium;
2. Execução de todos os casos de testes automatizados por intermédio do Pytest;
3. Registro do status de execução dos testes (*Passed*, *Failed* ou *Error*);
4. Execução das ferramentas de *benchmark* além da realização dos cálculos definidos neste estudo;
5. Registro e consolidação dos resultados obtidos.

3.4.4 Etapa 4: Consolidação dos Dados

Nesta etapa, os dados obtidos ao longo das fases anteriores foram organizados e consolidados em arquivos estruturados, com o objetivo de possibilitar a análise comparativa entre os modelos avaliados e garantir a rastreabilidade dos resultados. Foram reunidos os arquivos de validação manual, as métricas de execução dos testes, os relatórios gerados pelo Pytest, os resultados da análise estática com Pylint e as métricas de complexidade ciclomática extraídas pelo Radon.

Os artefatos consolidados foram armazenados em diferentes formatos, conforme sua finalidade. Os arquivos em formato `.csv` foram utilizados para organizar as

métricas quantitativas e classificações dos scripts, como *PASSRATE*, cobertura dos requisitos, categorias de defeitos, Score Pylint e complexidade ciclomática. Os arquivos `.html` correspondem aos relatórios de execução gerados pelo Pytest. Já os arquivos `.txt` e `.json` registram logs e saídas intermediárias das ferramentas utilizadas, permitindo auditoria e posterior reprocessamento dos dados. Os artefatos produzidos podem ser consultados e reutilizados para verificar os procedimentos adotados, reproduzir as análises e gerar novamente as tabelas e figuras apresentadas neste trabalho.

3.5 Reprodutibilidade e Disponibilidade do Código-Fonte

Com o objetivo de favorecer a transparência e a reprodutibilidade da pesquisa, os artefatos produzidos ao longo deste trabalho, incluindo os *scripts* de teste automatizados gerados pelos quatro modelos LLM avaliados, os *prompts* utilizados na geração (Apêndice C), os scripts de coleta e consolidação de métricas, e os dados brutos resultantes da execução dos testes, estão disponíveis publicamente no repositório da pesquisa (LEAL, 2026).

3.6 Análise dos Resultados

A análise dos resultados foi realizada a partir da consolidação das métricas coletadas durante a execução dos testes automatizados. Os dados obtidos foram organizados em planilhas estruturadas, permitindo a comparação direta entre os modelos LLM avaliados em relação às métricas de qualidade do código, cobertura, desempenho e incidência de defeitos.

Os resultados foram analisados de forma quantitativa, por meio da comparação dos valores numéricos obtidos para cada métrica, e qualitativa, considerando a natureza dos defeitos identificados e o comportamento observado nos diferentes cenários de teste. A interpretação dos dados buscou identificar padrões, vantagens e limitações de cada modelo, bem como avaliar sua adequação ao uso em automação de testes *web*.

A discussão detalhada dos resultados obtidos é apresentada na Seção 4.2.

4 Resultados

Esta seção apresenta os resultados obtidos a partir da avaliação dos *scripts* de testes automatizados gerados pelos modelos LLM *open-weight* analisados. A análise considera quatro modelos: GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B e Qwen3 Coder 480B. Os *scripts* foram avaliados em relação à taxa de sucesso de execução, à cobertura dos requisitos propostos, à qualidade estática do código e à complexidade ciclomática.

A apresentação dos resultados está organizada de acordo com as perguntas de pesquisa definidas neste trabalho. Inicialmente, são analisadas a taxa de sucesso e a cobertura média dos requisitos, com o objetivo de responder à pergunta principal sobre a eficácia dos modelos na geração de *scripts* de testes automatizados. Em seguida, são comparadas as métricas de qualidade e manutenibilidade dos códigos gerados. Posteriormente, discute-se o desempenho específico do modelo Qwen3 Coder 480B em relação aos modelos generalistas. Por fim, são analisadas as categorias de defeitos identificadas, bem como sua relação com as métricas de qualidade do código.

4.1 Perguntas de Pesquisa

4.1.1 Pergunta Principal

Ao início da pesquisa, a principal pergunta determinada foi:

Qual é a eficácia de LLMs open-weight na geração de scripts de testes automatizados, medida por taxa de sucesso (PASSRATE) e cobertura de requisitos?

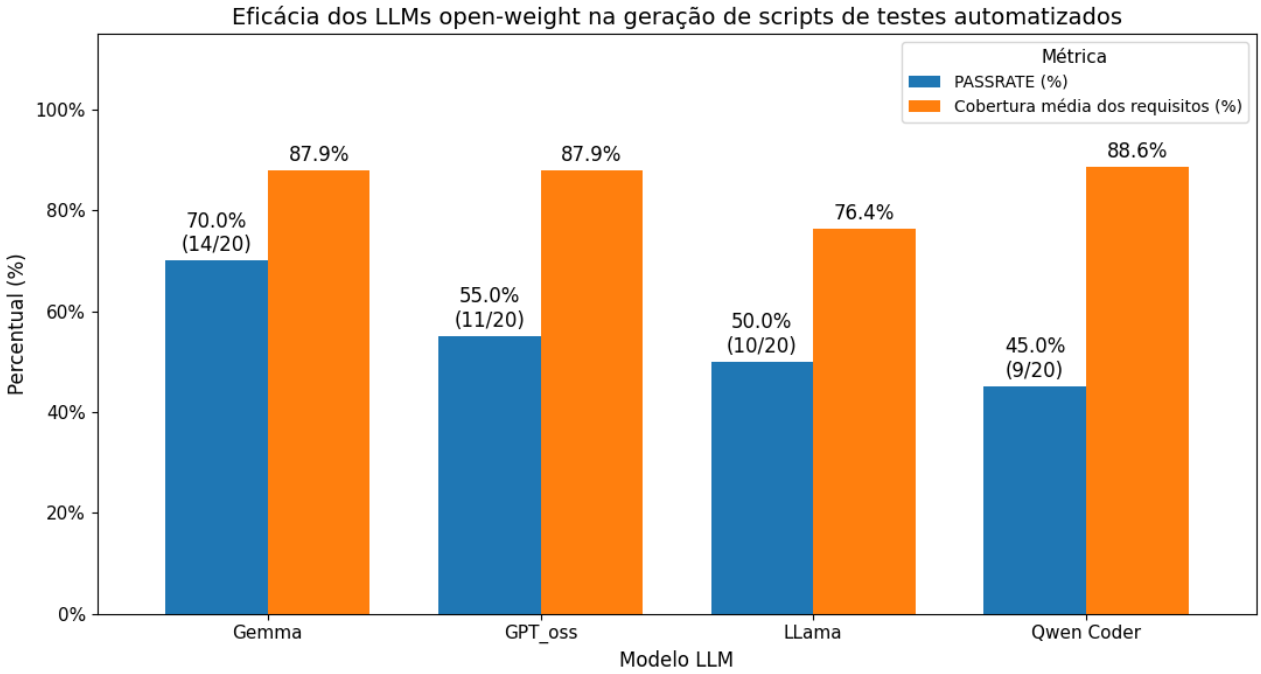

Figura 2 – Comparação entre PASSRATE e cobertura média dos requisitos por modelo LLM.

O gráfico apresentado na Figura 2 compara a taxa de sucesso de execução dos scripts, representada pelo *PASSRATE*, com a cobertura média dos requisitos por modelo LLM. Observa-se que a cobertura média dos requisitos variou entre 76,4% e 88,6%, indicando que, de modo geral, os scripts gerados apresentaram boa aderência aos requisitos descritos nos *prompts*. O modelo Qwen3 Coder obteve a maior cobertura média dos requisitos, com 88,6%, seguido pelos modelos Gemma e GPT-OSS, ambos com 87,9%. O modelo Llama apresentou a menor cobertura média, com 76,4%.

Apesar de o Qwen3 Coder ter apresentado a maior cobertura média dos requisitos, esse modelo obteve o menor *PASSRATE*, com 45%. Esse resultado indica que seus scripts tenderam a representar os requisitos solicitados, mas apresentaram maior dificuldade de execução bem-sucedida. Em contraste, o modelo Gemma obteve o maior *PASSRATE*, com 70%, além de manter uma cobertura média elevada, de

87,9%. Os modelos GPT-OSS e Llama apresentaram taxas intermediárias de execução, com 55% e 50%, respectivamente.

Esses resultados sugerem que a eficácia dos modelos não deve ser avaliada apenas pela capacidade de gerar scripts executáveis. Embora a cobertura média dos requisitos tenha sido elevada para a maioria dos modelos, o *PASSRATE* apresentou maior variação, indicando que parte dos scripts, mesmo próximos dos requisitos solicitados, falhou durante a execução. Dessa forma, observa-se uma distinção entre a adequação funcional dos testes aos requisitos e a robustez técnica necessária para que os scripts sejam executados com sucesso.

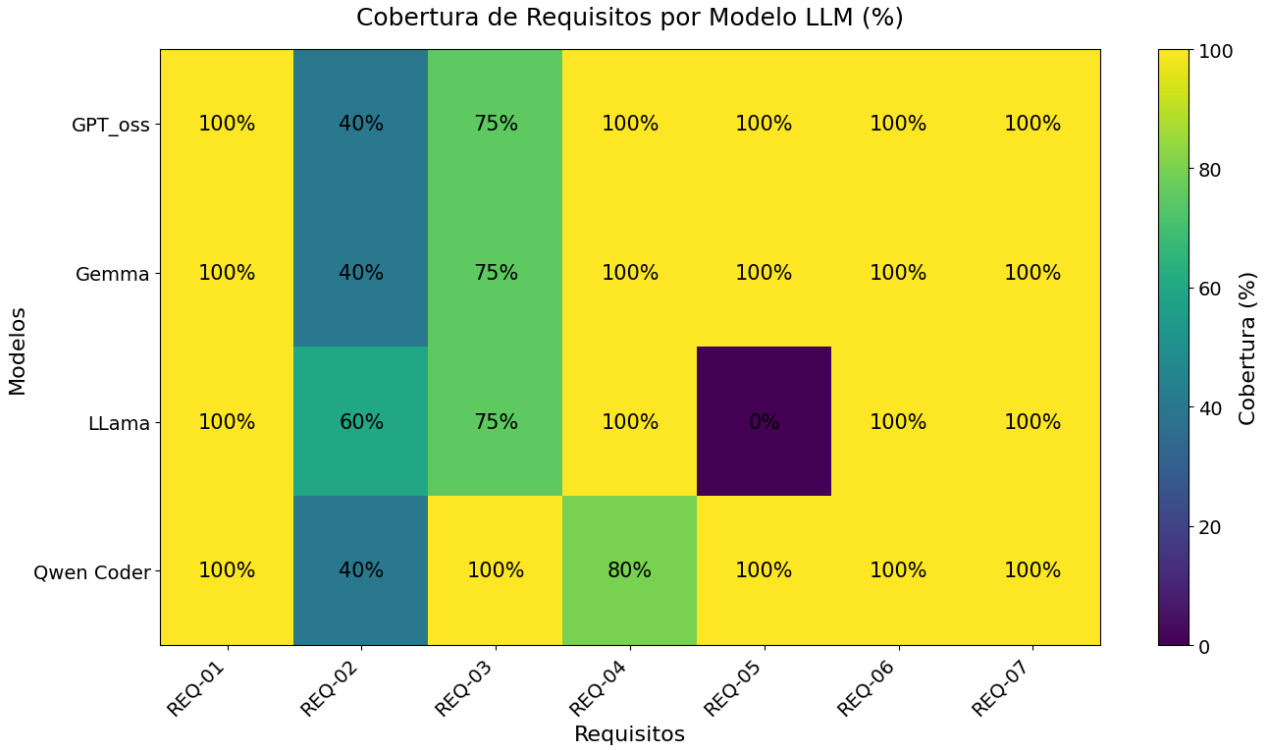

Figura 3 – Gráfico de Heatmap entre cobertura e requisitos.

Da mesma forma, segundo o gráfico apresentado na Figura 3, os requisitos 2 (Busca de Produtos) e 3 (Visualização de Produtos) foram os requisitos menos cobertos pelos modelos de LLMs. O requisito 2, relacionado a Busca de Produtos, (que compõe 25% dos cenários de teste e é fundamental em contexto de *e-commerce*), foi o menos coberto pelos modelos de LLM avaliados. Os modelos GPT-OSS, Gemma e Qwen3 Coder apresentaram apenas 40% de cobertura neste requisito, enquanto Llama alcançou 60%.

A uniformidade desta fraqueza em três modelos distintos sugere que o problema não reside em limitações específicas de cada modelo, mas possivelmente na ambiguidade ou incompletude das instruções do *prompt*. Esta observação indica uma oportunidade clara de melhoria através do refinamento da engenharia de *prompt*, particular-

mente na especificação de padrões de busca e estrutura esperada dos casos de teste para este requisito.

Logo, o Qwen3 Coder teve a melhor performance em relação a cobrir os requisitos solicitados, principalmente nos requisitos com maior complexidade técnica de execução, sendo eles requisito 4 (Carrinho de Compras), 5 (Disponibilidade de Produtos) e 6 (Filtros). Apesar da taxa de sucesso ser a mais baixa dentre todos os outros modelos, a análise de cobertura de requisitos demonstra que este modelo compreende adequadamente as regras de negócio e consegue gerar testes que abrangem *features* com maior complexidade técnica. Isto sugere que a especialização do modelo em geração de código (Qwen3 Coder) fornece vantagem na compreensão de lógica de negócio e estrutura de testes, apesar das limitações na integração com *frameworks* externos.

Em contraposição, Llama apresentou a pior performance relativa, com destaque no requisito 5 (Disponibilidade de Produtos), onde alcançou 0% de cobertura. Esta falha crítica revela uma incapacidade fundamental deste modelo em gerar validações que envolvem lógica condicional de estado de produtos. Tal limitação é particularmente preocupante em contextos de *e-commerce*, onde impedir a adição de itens indisponíveis ao carrinho constitui funcionalidade essencial para a integridade da aplicação.

4.1.2 Pergunta de Pesquisa 1

A primeira pergunta secundária buscou verificar em que medida os modelos LLM *open-weight* avaliados diferem entre si quanto ao *PASSRATE*, à cobertura de requisitos e à qualidade estática do código, medida por Score Pylint e complexidade ciclomática, nos scripts de testes automatizados produzidos para o ambiente *e-commerce* com Selenium e Pytest. As dimensões de *PASSRATE* e cobertura de requisitos já foram apresentadas na Seção ”Perguntas de Pesquisa”, no contexto da resposta à pergunta principal; esta seção complementa essa análise ao examinar a qualidade estática e a complexidade estrutural dos *scripts* gerados, permitindo uma comparação mais completa entre os quatro modelos nas quatro frentes propostas pela pergunta.

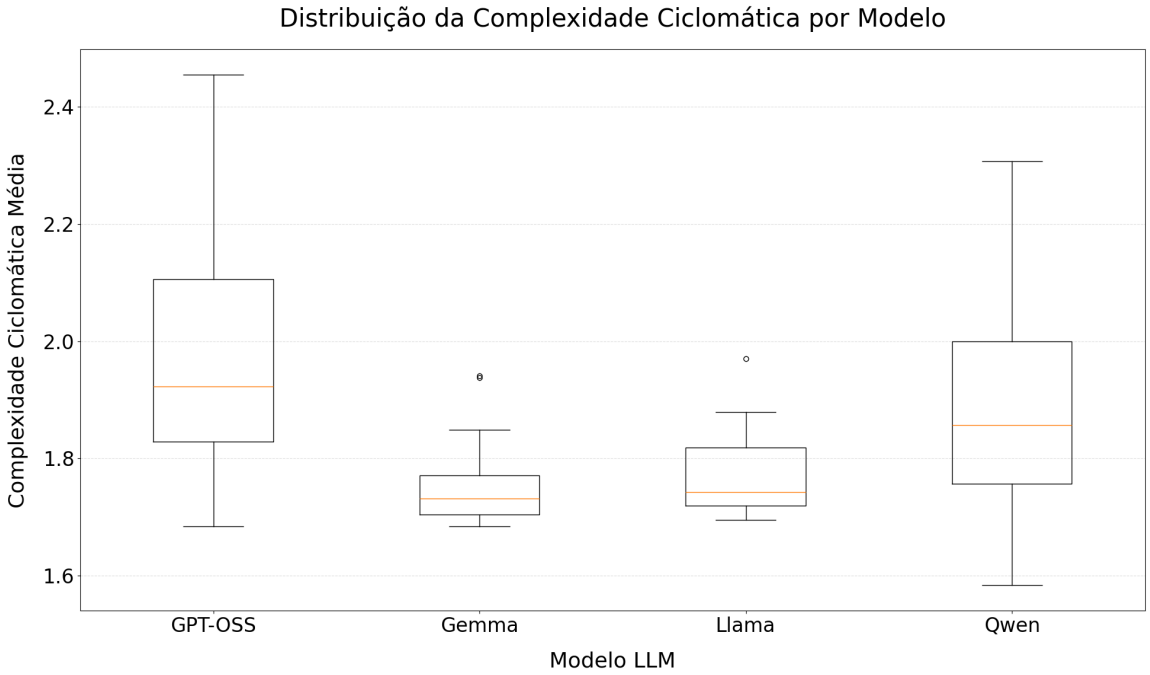

Figura 4 – Complexidade Ciclomática dos Testes Automatizados gerados por modelo de LLM.

A Figura 4 apresenta a distribuição da complexidade ciclomática média dos *scripts* gerados por cada modelo. Observa-se que todos os modelos apresentaram valores baixos de complexidade, concentrados aproximadamente entre 1,58 e 2,45 e mediana entre 1,73 e 1,86. Esse resultado indica que, de modo geral, os scripts possuem estruturas simples, com baixa quantidade de ramificações lógicas, o que é positivo num contexto de reutilização e manutenção de um *framework* de automação voltado para testes.

Entre os modelos avaliados, Gemma e Llama apresentaram distribuições mais concentradas e poucos *outliers* com, respectivamente, medianas próximas de 1,73 e 1,74, sugerindo maior uniformidade na estrutura dos *scripts* gerados. O Qwen3 Coder apresentou mediana de 1,86, ligeiramente superior e maior variação em relação a Gemma e Llama, enquanto o GPT-OSS apresentou a maior dispersão, com medianas entre 1,68 e 2,45, com alguns *scripts* atingindo os maiores valores de complexidade observados.

Apesar das diferenças de dispersão e medianas, os valores absolutos de complexidade permaneceram baixos para todos os modelos de LLM, o que sugere que LLMs, seguindo as regras de negócio apresentadas em casos de testes com complemento de código referencial para o desenvolvimento do teste automatizado criam *scripts* simples, o que facilita a manutenção de casos de teste automatizados na medida que os requisitos mudam com o tempo.

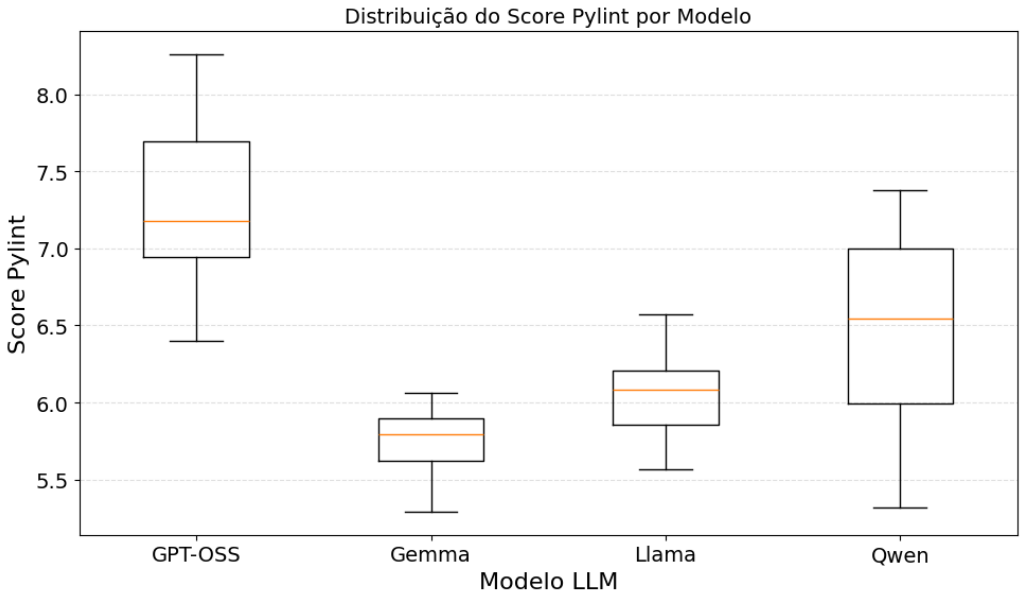

Figura 5 – Score Pylint por Testes Automatizados gerados modelo de LLM.

A Figura 5 apresenta a distribuição do Score Pylint dos *scripts* gerados por cada modelo. Observa-se que o GPT-OSS apresentou os maiores valores de qualidade estática, com mediana de 7,18, superior aos demais modelos e maior concentração em faixas elevadas de pontuação. O Qwen3 Coder apresentou desempenho intermediário, com mediana próxima de 6,54, porém com maior dispersão, indicando maior variabilidade na qualidade estática dos *scripts* gerados. O Llama também apresentou desempenho intermediário, com mediana de 6,08, enquanto o Gemma obteve os menores valores de Score Pylint entre os modelos avaliados, com mediana de 5,80 e pouca dispersão de pontos do pylint. Dessa forma, os modelos variam as pontuações entre código de boa qualidade e de qualidade aceitável, segundo as métricas de avaliação de qualidade de código determinadas nessa pesquisa.

Ao comparar esse resultado com a distribuição da complexidade ciclomática apresentada na Figura 4, observa-se uma correlação interessante, ainda que não confundível com ”melhor desempenho”: os mesmos modelos que apresentaram maior complexidade ciclomática, GPT-OSS (1,97) e, em seguida, Qwen3 Coder (1,89), também obtiveram os maiores Scores Pylint (7,26 e 6,47, respectivamente), enquanto os modelos com menor complexidade, Gemma (1,76) e Llama (1,77), apresentaram os menores Scores Pylint (5,80 e 6,07). É importante notar que essas duas métricas avaliam aspectos distintos e não devem ser somadas em um único julgamento de ”melhor modelo”: o GPT-OSS obteve o melhor desempenho em qualidade estática (Score Pylint), mas, simultaneamente, a maior complexidade ciclomática entre os quatro modelos, ou seja, o pior resultado nessa métrica específica, uma vez que valores mais baixos indicam scripts estruturalmente mais simples e de mais fácil manutenção. Já Gemma e Llama, apesar do Score Pylint mais baixo, produziram os scripts estruturalmente

mais simples do experimento. Essa correlação sugere que as diferenças de qualidade capturadas pelo Pylint estão mais relacionadas a aspectos como estilo, organização e convenções de código do que à quantidade de ramificações lógicas presentes nos *scripts*, mas não implica que maior complexidade seja, em si, desejável.

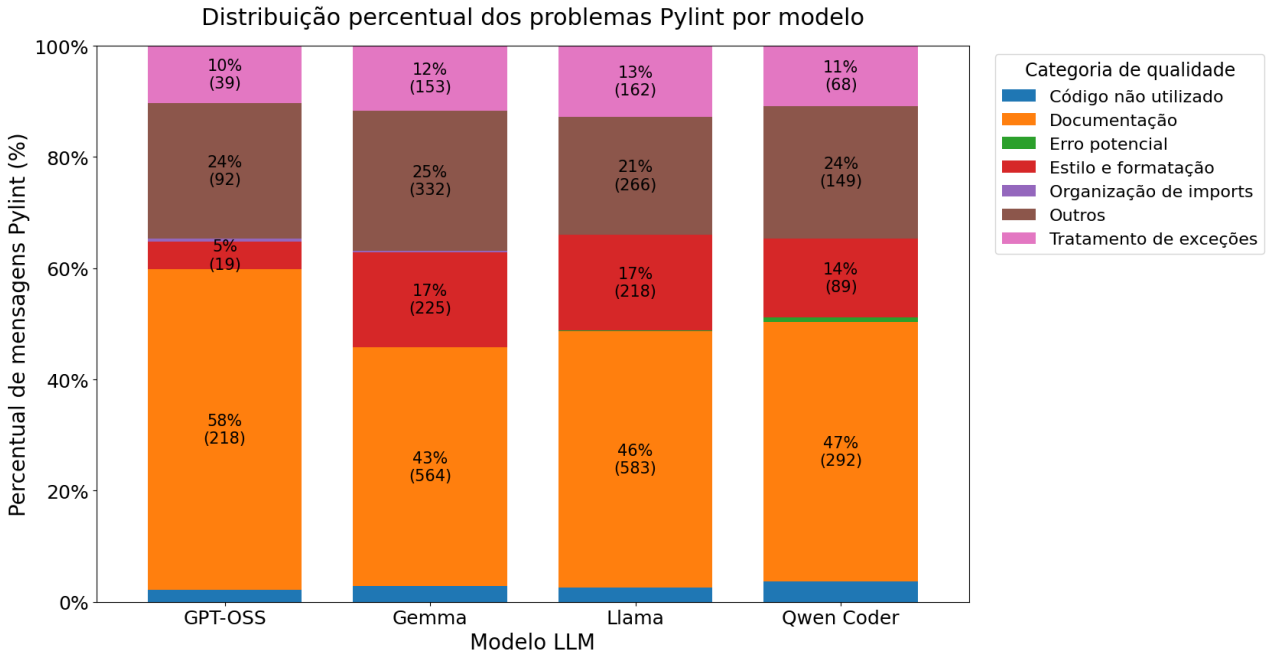

Figura 6 – Gráfico de distribuição percentual dos problemas Identificados pelo Pylint por Modelo.

A Figura 6 apresenta a distribuição percentual das categorias de problemas identificadas pelo Pylint nos *scripts* gerados por cada modelo. Observa-se que, em todos os modelos, a categoria Documentação foi predominante, representando 58% das mensagens no GPT-OSS, 43% no Gemma, 46% no Llama e 47% no Qwen3 Coder. Esse resultado indica que uma parcela expressiva das penalizações do Pylint esteve associada à ausência de documentação no código, como *docstrings* em módulos, classes ou funções.

Esse achado, contudo, não deve ser atribuído a uma limitação dos modelos avaliados. Nem o *prompt* utilizado nem o exemplo *few-shot* fornecido (Apêndice C) especificavam a inclusão de *docstrings* entre as restrições de desenvolvimento do teste, de modo que os modelos reproduziram fielmente o padrão demonstrado. A predominância dessa categoria reflete, portanto, uma lacuna na especificação do *prompt*, e não uma deficiência de geração de código: os modelos foram penalizados pelo Pylint por não produzirem um artefato que não lhes foi solicitado.

Além disso, a categoria Outros apresentou participação relevante em todos os modelos, variando entre 21% e 25% das mensagens. Problemas de estilo e formatação também foram observados, especialmente nos modelos Gemma e Llama, ambos

com 17%, seguidos pelo Qwen3 Coder, com 14%, e pelo GPT-OSS, com 5%. Essa diferença sugere que parte da variação no Score Pylint entre os modelos está associada não apenas à ausência de documentação, mas também à conformidade com padrões de estilo e organização do código.

A categoria Tratamento de exceções apareceu em todos os modelos, com percentuais próximos, variando entre 10% e 13%. Esse tipo de problema é relevante para a análise de qualidade, pois pode afetar a robustez dos testes automatizados, principalmente quando há uso de tratamentos genéricos de exceções. Por outro lado, categorias como Código não utilizado, Organização de *imports* e Erro potencial apresentaram baixa participação percentual.

Quadro 1 – Erro sintático (Qwen3 Coder 480B, CT-01)

```
# import re  <-- omitido pelo modelo

filters = re.findall(r'R\$\d[\d\.]*', active)  # E0602: Undefined variable 're
    '
```

O modelo reutilizou o método do exemplo *few-shot* que depende do módulo `re`, mas omitiu a importação. Ocorreu em 5 *scripts* do Qwen3 Coder 480B e 1 do Llama 3.3 70B.

Quadro 2 – Erro relacionado ao Selenium (Qwen3 Coder 480B, CT-01)

```
[STEP] Clicar na opção 'Ofertas␣do␣Dia'
[EXPECTED RESULT] O sistema redireciona para a página de ofertas

E  selenium.common.exceptions.TimeoutException: Message:
   selenium/webdriver/support/wait.py:121: TimeoutException
```

O *locator* definido pelo modelo não correspondeu a nenhum elemento da página, e o `WebDriverWait` excedeu o tempo limite aguardando a condição. A `TimeoutException` foi o defeito mais frequente do experimento (17 das 39 falhas), concentrando-se no Qwen3 Coder 480B (8 ocorrências).

Quadro 3 – Erro lógico (Gemma 4 26B, CT-14)

```
details.add_to_cart()
assert details.is_cart_confirmation_visible(), \
    "A␣confirmação␣de␣item␣adicionado␣ao␣carrinho␣não␣apareceu"

E  AssertionError: A confirmação de item adicionado ao carrinho não apareceu
E  assert False
```

O *script* executou o fluxo completo sem erros de sintaxe ou de *framework*, mas

a asserção falhou: o método de verificação implementado pelo modelo não localizou o elemento de confirmação esperado. Erros de asserção (`AssertionError`) representaram 11 das 39 falhas registradas.

De modo geral, os resultados indicam que os problemas identificados pelo Pylint estão concentrados majoritariamente em aspectos de documentação, estilo, formatação e manutenibilidade. Assim, as penalizações observadas no Score Pylint parecem estar mais associadas à qualidade estática e à legibilidade do código do que a erros diretamente relacionados à execução funcional dos testes.

Consolidando os resultados apresentados nesta seção com os discutidos anteriormente na resposta à pergunta principal, observa-se que os quatro modelos LLM *open-weight* avaliados apresentam diferenças relevantes em todas as dimensões investigadas pela P.P. 1. Em termos de *PASSRATE*, o Gemma se destacou com o maior valor (70%), seguido por GPT-OSS (55%), Llama (50%) e Qwen3 Coder (45%). Quanto à cobertura de requisitos, a ordem se inverte parcialmente: Qwen3 Coder liderou (88,6%), seguido por GPT-OSS e Gemma (87,9% cada) e Llama (76,4%). Já na qualidade estática do código, medida pelo Score Pylint, o GPT-OSS obteve a melhor pontuação (7,26), seguido por Qwen3 Coder (6,47), Llama (6,07) e Gemma (5,80); e, na complexidade ciclomática, Gemma e Llama produziram os scripts estruturalmente mais simples (1,76 e 1,77, respectivamente), enquanto Qwen3 Coder (1,89) e GPT-OSS (1,97) apresentaram os valores mais elevados.

Esse panorama evidencia que nenhum dos quatro modelos se destaca de forma consistente em todas as métricas simultaneamente: o modelo com maior taxa de sucesso de execução (Gemma) não é o que produz o código de melhor qualidade estática, e o modelo com maior cobertura de requisitos (Qwen3 Coder) não é o mais confiável em termos de execução nem o que gera o código mais simples. Esse resultado confirma a hipótese subjacente à P.P. 1 de que os modelos LLM *open-weight* avaliados diferem entre si no conjunto observado, mas revela que essas diferenças se manifestam de forma heterogênea entre as métricas, não sendo possível apontar um único modelo como superior em todas as dimensões de corretude funcional, cobertura e manutenibilidade consideradas nesta pesquisa.

4.1.3 Pergunta de Pesquisa 2

A segunda pergunta secundária investigou se a especialização em código do modelo Qwen3 Coder 480B resultaria em scripts com maior *PASSRATE*, melhor Score Pylint e menor complexidade ciclomática, indicando maior manutenibilidade em comparação aos modelos generalistas GPT-OSS 120B, Gemma 4 26B e Llama 3.3 70B, considerando o mesmo conjunto de casos de teste.

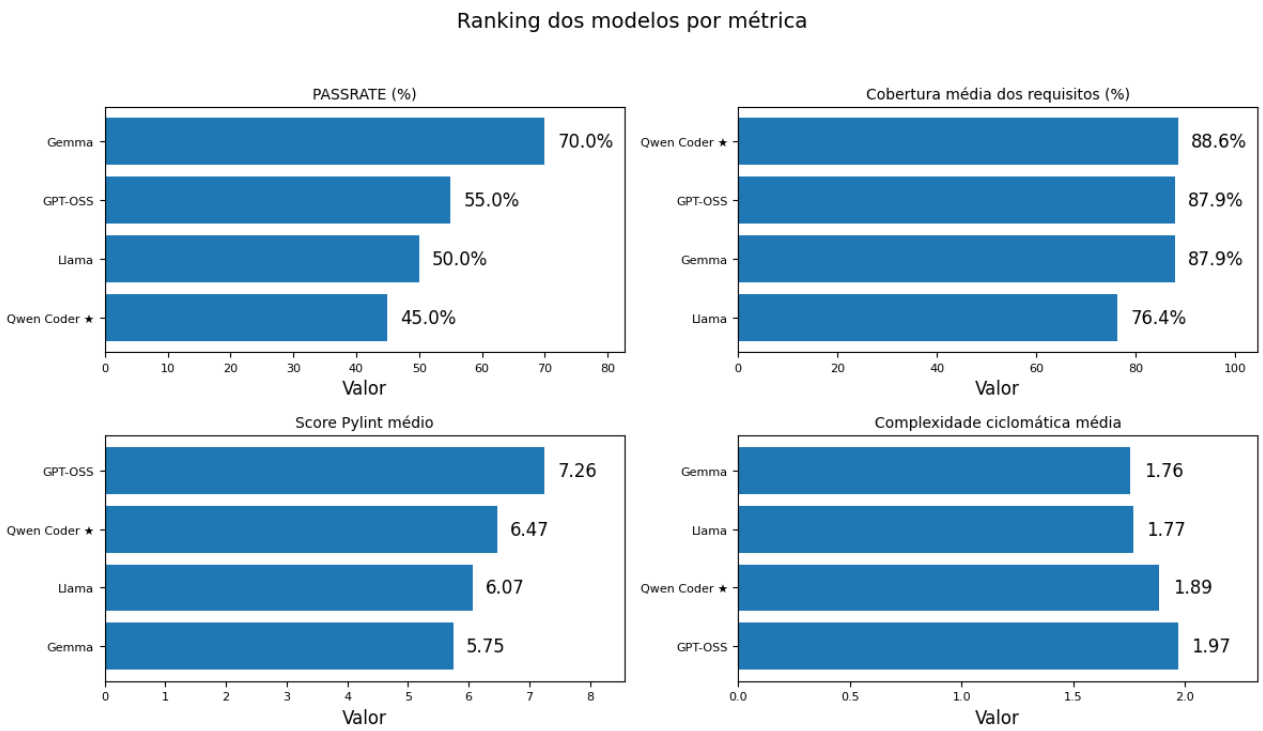

Figura 7 – Ranking de Modelos LLM por Métrica.

A Figura 7 sintetiza o desempenho dos quatro modelos nas quatro métricas avaliadas, permitindo uma comparação direta da posição do Qwen3 Coder frente aos modelos generalistas em cada uma delas. Em relação ao *PASSRATE*, o Qwen3 Coder obteve o menor valor entre os quatro modelos (45%), posicionando-se atrás de Gemma (70%), GPT-OSS (55%) e Llama (50%). Esse resultado já contraria diretamente a hipótese da pergunta, uma vez que o modelo especialista em código apresentou a taxa de sucesso de execução mais baixa do experimento, e não a mais alta como seria esperado dada sua especialização. Já em relação à complexidade ciclomática média, na qual valores mais baixos indicam maior simplicidade e manutenibilidade, o Qwen3 Coder apresentou 1,89, sendo superado apenas pelo GPT-OSS (1,97), ou seja, o Qwen3 Coder produziu a segunda maior complexidade entre os modelos, enquanto Gemma (1,76) e Llama (1,77) geraram scripts estruturalmente mais simples. Assim, tampouco neste critério o modelo especialista demonstrou a vantagem esperada. O único indicador em que o Qwen3 Coder se destacou positivamente foi a cobertura média de requisitos, na qual obteve o maior valor entre os modelos (88,6%), à frente de GPT-OSS e Gemma (87,9% cada) e de Llama (76,4%).

Diante desses resultados, conclui-se que a hipótese não se confirma para os três indicadores centrais da pergunta, *PASSRATE*, Score Pylint e complexidade ciclomática. A especialização em código do Qwen3 Coder 480B não se traduziu em scripts mais bem-sucedidos na execução, com melhor qualidade estática ou estruturalmente mais simples que os dos modelos generalistas avaliados. Pelo contrário, o

modelo generalista Gemma 4 26B, de menor escala de parâmetros ativos entre os avaliados, obteve o melhor *PASSRATE* do experimento, enquanto o GPT-OSS 120B liderou tanto o Score Pylint quanto (por margem) a menor complexidade ciclomática relativa ao Qwen3 Coder.

Por outro lado, o desempenho superior do Qwen3 Coder na cobertura de requisitos sugere que sua especialização em geração de código oferece vantagem na compreensão semântica das regras de negócio e na tradução de requisitos funcionais em cenários de teste mais abrangentes, mas essa vantagem não se estende à robustez de execução nem à qualidade estilística do código produzido.

4.1.4 Pergunta de Pesquisa 3

A terceira pergunta secundária buscou identificar quais categorias de defeitos, erros sintáticos, erros relacionados ao Selenium ou erros lógicos, predominam nos scripts gerados pelos modelos avaliados, e de que forma a qualidade do código se associa à ocorrência dessas falhas.

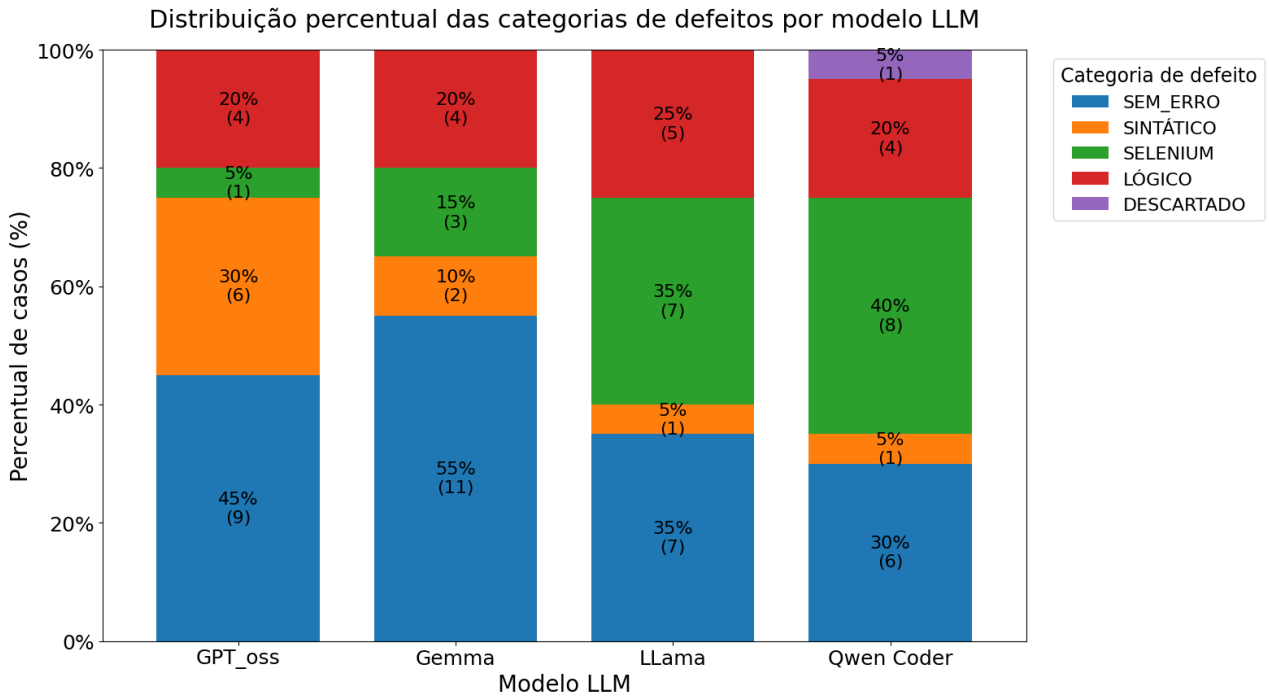

Figura 8 – Categorização de Defeitos por Modelo de LLM.

A Figura 8 apresenta a distribuição percentual das categorias de defeitos por modelo. Observa-se que os quatro modelos apresentam perfis de erro distintos. O Gemma obteve a maior proporção de scripts sem erro (55%), porém concentrou a maior taxa de erros sintáticos entre todos os modelos (30%), sugerindo que, apesar de gerar a maioria dos scripts funcionalmente corretos, uma parcela relevante de suas falhas está associada a problemas básicos de interpretação do código pelo interpre-

tador Python. O GPT-OSS apresentou perfil semelhante em menor intensidade, com 45% de scripts sem erro e 30% de erros sintáticos, além de 20% de erros lógicos e apenas 5% de erros relacionados ao Selenium, a menor incidência dessa categoria entre os quatro modelos.

Em contraste, Llama e Qwen3 Coder apresentaram uma inversão desse padrão: a categoria predominante de defeito nesses dois modelos foi erros relacionados ao Selenium, correspondendo a 35% e 40% dos casos, respectivamente, enquanto erros sintáticos praticamente desapareceram (5% em ambos). O Qwen3 Coder, especificamente, apresentou a menor proporção de scripts sem erro (30%) combinada com a maior incidência de falhas de Selenium (40%) entre todos os modelos avaliados.

Esse resultado, confrontado com o achado da P.P. 2, expõe a seguinte informação: o baixo *PASSRATE* do Qwen3 Coder (45%, o menor entre os quatro modelos) não decorre de deficiências sintáticas ou de raciocínio lógico, nessas duas categorias o modelo apresentou desempenho comparável ou superior aos demais, mas sim de dificuldades específicas na interação programática com a interface web via Selenium (por exemplo, seletores incorretos, problemas de sincronização ou tempo de espera inadequado). Isso reforça a hipótese levantada anteriormente de que a especialização em geração de código do Qwen3 Coder favorece a compreensão da lógica de negócio e da estrutura sintática dos testes, mas não necessariamente a integração robusta com *frameworks* externos de automação, que dependem de conhecimento situacional sobre o comportamento dinâmico do DOM e do navegador.

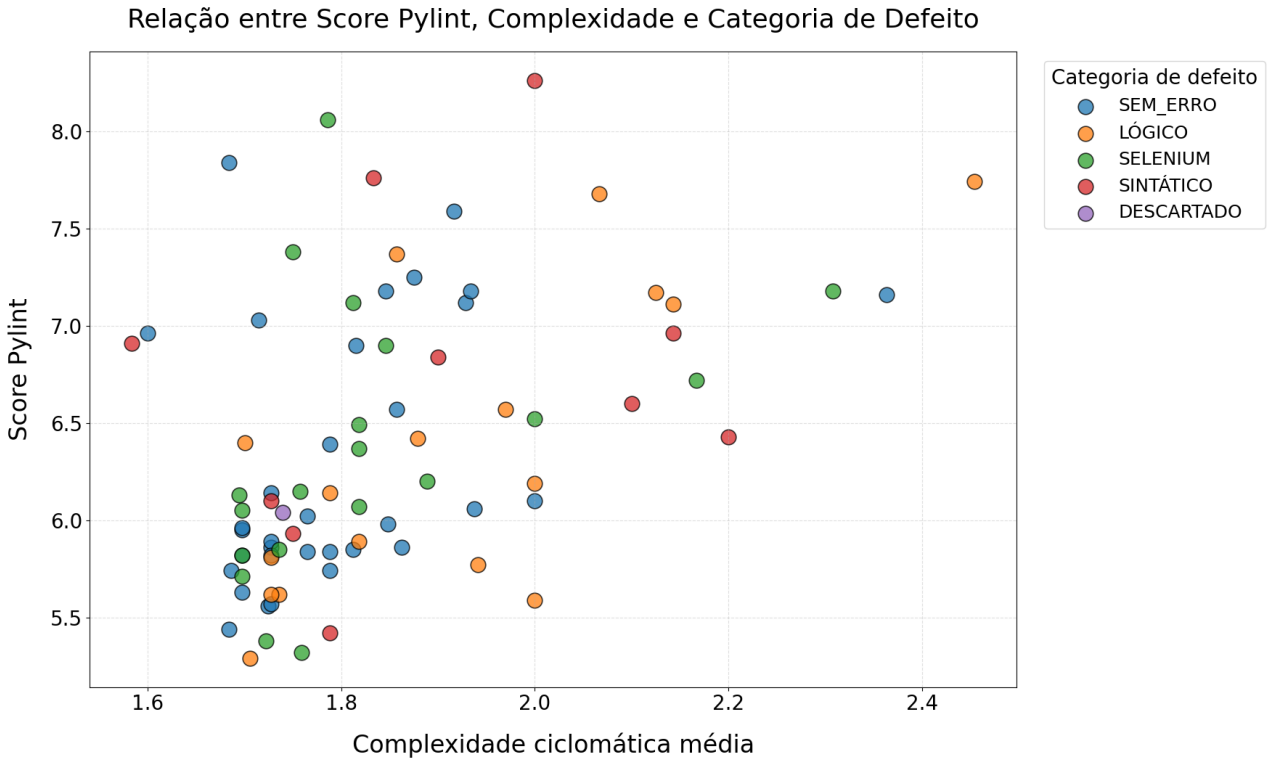

Figura 9 – Relação entre defeitos e complexidade ciclomática e qualidade de código.

A Figura 9 relaciona o Score Pylint, a complexidade ciclomática média e a categoria de defeito de cada script gerado, com o objetivo de investigar se scripts de maior complexidade ou menor qualidade estática concentram tipos específicos de erro. Diferentemente do que se poderia esperar, os pontos referentes às diferentes categorias de defeito (SEM_ERRO, LÓGICO, SELENIUM, SINTÁTICO e DESCARTADO) aparecem distribuídos de forma relativamente homogênea ao longo de toda a faixa observada de complexidade ciclomática (aproximadamente 1,6 a 2,4) e de Score Pylint (aproximadamente 5,5 a 8,5), sem a formação de agrupamentos visuais nítidos por categoria. Scripts com Score Pylint elevado (acima de 7,5) apresentam tanto casos sem erro quanto erros lógicos e de Selenium, e o mesmo padrão de mistura se repete nas faixas intermediárias e inferiores de pontuação.

Esse resultado indica que, para o conjunto de dados analisado, não foi observada associação visual clara entre a qualidade estática do código (Score Pylint), sua complexidade estrutural (complexidade ciclomática) e o tipo de defeito predominante. Em outras palavras, um script sintaticamente bem organizado e pouco ramificado não é necessariamente mais propenso a estar livre de erros de execução, e scripts mais complexos não concentram, de forma sistemática, um tipo específico de falha. Isso sugere que os defeitos observados, especialmente os relacionados ao Selenium, têm origem menos ligada ao estilo ou à estrutura lógica do código gerado, e mais relacionada à precisão factual do modelo sobre a interface real da aplicação sob teste (por exemplo, seletores de elementos incorretos ou hipóteses equivocadas sobre o comportamento assíncrono da página), fatores não capturados pelas métricas estáticas de qualidade utilizadas neste estudo.

4.2 Discussão

Esta seção posiciona os resultados obtidos por este trabalho em relação à literatura de automação de testes com LLMs discutida na Seção 2.7, e discute as implicações desses achados tanto para a pesquisa acadêmica quanto para a prática industrial de qualidade de software.

4.2.1 Comparação com Trabalhos Relacionados

Os resultados obtidos neste estudo dialogam diretamente com os três trabalhos relacionados discutidos na Seção 2.7 (Tabela 1), ao mesmo tempo em que revelam particularidades específicas do uso de modelos *open-weight*.

O primeiro trabalho relacionado, Grafos de Transição com LLMs, que propõe a geração automatizada de casos de teste de ponta-a-ponta combinando LLMs com grafos de transição de telas e grafos de estado de formulários, demonstra a viabilidade

técnica de utilizar LLMs para geração de scripts Selenium, abordagem também adotada neste estudo. No entanto, por não especificar quais modelos LLM foram utilizados nem discutir questões de custo e acessibilidade, aquele trabalho não permite comparação direta de métricas de desempenho com os resultados aqui obtidos. Ainda assim, a arquitetura baseada em grafos de transição proposta naquele estudo representa uma direção complementar à abordagem adotada neste trabalho, que se baseia em casos de teste estruturados manualmente em vez de mapeamento automático de navegação, distinção que pode explicar, em parte, por que a cobertura do requisito de Busca de Produtos (REQ-02) foi a menor entre os requisitos avaliados neste estudo: a ausência de um mapeamento formal dos caminhos de navegação, como o empregado naquele trabalho, pode ter limitado a capacidade dos modelos de inferir variações de busca a partir apenas da descrição textual dos casos de teste.

O estudo Suna com DeepSeek e Claude, que comparou DeepSeek V3 e Claude Sonnet 4 na geração de testes end-to-end, reportou PASSRATE de 34,3% para o modelo open-source e 70,1% para o modelo comercial, uma diferença de 35,8 pontos percentuais favorável ao modelo pago. Neste trabalho, o PASSRATE dos quatro modelos *open-weight* avaliados variou entre 45% e 70%, com o Gemma 4 26B atingindo 70%, valor equivalente ao do modelo comercial Claude Sonnet 4 no estudo comparativo mencionado. Esse resultado sugere que, ao menos para o modelo Gemma 4 26B e para o ambiente de teste utilizado neste estudo, a lacuna de desempenho entre modelos *open-weight* e modelos comerciais premium pode ser menor do que a observada em estudos anteriores, corroborando a tendência de aproximação de desempenho relatada na literatura recente sobre modelos *open-weight*.

Em relação ao AutoQALLMs, que reportou cobertura de testes entre 96% e 98% utilizando exclusivamente modelos comerciais (GPT-4, Claude 4.5, Grok), a cobertura média de requisitos obtida pelos modelos *open-weight* avaliados neste trabalho (76,4% a 88,6%) ficou abaixo desses valores. Essa diferença pode ser atribuída tanto à natureza distinta das métricas comparadas, cobertura de testes versus cobertura de requisitos, quanto ao fato de o AutoQALLMs empregar exclusivamente modelos comerciais de ponta, reforçando que, embora os modelos *open-weight* avaliados apresentem desempenho competitivo em corretude funcional, ainda há espaço para melhoria na abrangência dos cenários gerados automaticamente.

Por fim, este trabalho amplia a literatura ao avaliar quatro modelos *open-weight* simultaneamente, indo além da comparação binária adotada nos estudos de Monteiro e colaboradores e da abordagem restrita a modelos comerciais do AutoQALLMs, além de investigar especificamente se a especialização em código (Qwen3 Coder 480B) confere vantagem sobre modelos generalistas, questão não explorada nos trabalhos relacionados analisados.

4.2.2 Implicações para a Pesquisa

Os achados deste trabalho contribuem para a literatura de automação de testes de software com LLMs em três frentes. Primeiro, ao demonstrar que a especialização em geração de código não se traduz automaticamente em maior robustez de execução, este estudo evidencia a necessidade de avaliações multidimensionais de modelos LLM aplicados a testes de software, que não se limitem a *benchmarks* tradicionais de geração de código, como o HumanEval, mas incluam métricas específicas de corretude funcional e integração com frameworks de automação.

Segundo, a identificação de que os defeitos observados nos scripts gerados concentram-se predominantemente em erros de integração com o Selenium, e não em erros sintáticos ou lógicos, aponta uma direção de pesquisa ainda pouco explorada na literatura: o desenvolvimento de técnicas de *prompting* ou de pós-processamento voltadas especificamente à correção de interações com frameworks de automação web, complementando as abordagens atuais de geração de código, majoritariamente focadas na correção sintática e lógica.

Terceiro, este trabalho preenche uma lacuna metodológica ao aplicar uma metodologia de *benchmarking* sistemática e reprodutível a um conjunto amplo de modelos *open-weight*, contemplando diferentes arquiteturas (MoE e transformer denso) e graus de especialização, contribuindo com um protocolo experimental que pode ser replicado e estendido por pesquisas futuras que avaliem novos modelos *open-weight* à medida que forem lançados.

4.2.3 Implicações para a Indústria

Do ponto de vista prático, os resultados deste trabalho oferecem subsídios concretos para equipes de qualidade de software que consideram adotar LLMs *open-weight* na automação de testes, especialmente em contextos de recursos limitados. A constatação de que o Gemma 4 26B, o menor modelo em número de parâmetros ativos entre os avaliados, obteve o maior *PASSRATE* do experimento sugere que, para tarefas de geração de testes end-to-end, modelos de menor escala podem ser suficientes e até preferíveis a alternativas maiores e mais custosas computacionalmente, favorecendo a adoção por equipes com infraestrutura limitada.

Adicionalmente, a constatação de que o baixo *PASSRATE* do Qwen3 Coder 480B decorre predominantemente de erros de integração com o Selenium, e não de falhas de compreensão dos requisitos, indica que equipes que optarem por esse modelo devem direcionar sua revisão manual prioritariamente aos trechos de interação com a interface web, otimizando o esforço de validação humana em vez de revisar o *script* por completo.

Por fim, a ausência de associação clara entre qualidade estática do código (Score Pylint) e ocorrência de defeitos sugere que ferramentas de análise estática, embora úteis para garantir aderência a padrões de estilo, não substituem a execução real dos testes como critério de validação da corretude funcional, uma constatação relevante para equipes que eventualmente utilizem apenas métricas estáticas como *gate* de qualidade em pipelines de integração contínua envolvendo *scripts* gerados por LLMs.

5 Considerações Finais

Este trabalho teve como objetivo analisar comparativamente quatro modelos LLM *open-weight*, GPT-OSS 120B, Gemma 4 26B, Llama 3.3 70B e Qwen3 Coder 480B, quanto à sua eficácia na geração de *scripts* de testes automatizados em Python com Selenium e Pytest, considerando métricas de qualidade de código, cobertura de cenários, corretude funcional e categorização de defeitos em um ambiente *e-commerce*. Esta seção retoma a pergunta principal de pesquisa e as três perguntas secundárias, sintetizando os principais achados obtidos, discutindo as contribuições do estudo, suas limitações e apontando direções para trabalhos futuros.

5.1 Síntese dos Resultados

Em relação à pergunta principal, *qual é a eficácia de LLMs open-weight na geração de* scripts *de testes automatizados, medida por taxa de sucesso (PASSRATE) e cobertura de requisitos?*, os resultados demonstraram que os quatro modelos avaliados apresentam boa capacidade de compreender e traduzir requisitos funcionais em *scripts* de teste, com cobertura média de requisitos variando entre 76,4% e 88,6%. Entretanto, essa capacidade de representação dos requisitos não se traduziu necessariamente em execução bem-sucedida: o *PASSRATE* variou de forma mais acentuada entre os modelos (45% a 70%), revelando uma distinção relevante entre a adequação funcional dos testes gerados e a robustez técnica necessária para sua execução sem falhas. O modelo Gemma 4 26B destacou-se como o mais equilibrado entre essas duas dimensões, combinando o maior *PASSRATE* (70%) com uma cobertura de requisitos elevada (87,9%).

A primeira pergunta secundária, voltada à comparação geral entre os modelos nas quatro métricas de corretude funcional e qualidade de código, confirmou que os modelos diferem de forma consistente entre si, mas de forma heterogênea: nenhum modelo se destacou simultaneamente em todas as métricas avaliadas. O Gemma liderou em *PASSRATE*, o Qwen3 Coder em cobertura de requisitos, o GPT-OSS em qualidade estática de código (Score Pylint), e Gemma e Llama produziram os scripts estruturalmente mais simples (menor complexidade ciclomática). Esse resultado indica que a escolha do modelo mais adequado depende diretamente da prioridade da equipe de QA, confiabilidade de execução, abrangência de cenários ou manutenibilidade do código, não havendo uma solução universalmente superior entre as opções *open-weight* avaliadas.

Além disso, a segunda pergunta secundária, que investigou se a especializa-

ção em código do Qwen3 Coder 480B resultaria em vantagem consistente sobre os modelos generalistas, não foi confirmada pelos dados. O Qwen3 Coder apresentou o menor *PASSRATE* entre os quatro modelos (45%) e não liderou nem o Score Pylint nem a complexidade ciclomática, ficando atrás do GPT-OSS 120B nessas duas métricas. Isso sugere que a especialização em geração de código, embora vantajosa para a compreensão semântica de requisitos e regras de negócio, refletida na maior cobertura de requisitos do modelo, não garante, por si só, maior robustez na integração com *frameworks* de automação de testes nem melhor aderência a boas práticas de estilo e organização de código.

E por fim, a terceira pergunta secundária, relativa à categorização dos defeitos, revelou que os erros predominantes variam conforme o modelo: Gemma e GPT-OSS concentraram maior incidência de erros sintáticos, enquanto Llama e, sobretudo, Qwen3 Coder apresentaram maior proporção de erros relacionados ao Selenium. Esse achado explica, em parte, o baixo *PASSRATE* do Qwen3 Coder identificado na segunda pergunta secundária: suas falhas não decorrem de problemas sintáticos ou lógicos, mas de dificuldades específicas na interação programática com a interface web. Adicionalmente, não foi observada associação visual clara entre a qualidade estática do código (Score Pylint), sua complexidade ciclomática e o tipo de defeito predominante, sugerindo que os erros de execução têm origem menos ligada ao estilo do código gerado e mais relacionada à precisão do modelo sobre o comportamento real da interface sob teste, fator não capturado pelas métricas estáticas utilizadas neste estudo.

5.2 Limitações do Estudo

Este trabalho apresenta limitações que devem ser consideradas na interpretação dos resultados. Em primeiro lugar, a validação experimental foi restrita a um único ambiente de teste, uma aplicação web do tipo *e-commerce*, e a um conjunto de 20 casos de teste distribuídos entre sete funcionalidades, o que limita a generalização dos achados para outros domínios de aplicação, com interfaces, fluxos de negócio ou níveis de complexidade distintos.

Em segundo lugar, os experimentos foram conduzidos utilizando exclusivamente a técnica de *prompting few-shot*, não tendo sido realizada a comparação com outras estratégias existentes, tal qual a *zero-shot*. Dessa forma, não é possível afirmar em que medida os resultados obtidos seriam distintos caso os modelos fossem avaliados sem exemplos de referência no prompt, o que representa uma lacuna a ser explorada em trabalhos futuros.

Adicionalmente, a concentração das penalizações do Score Pylint em ausência

de *docstrings* evidencia a sensibilidade das métricas de qualidade estática à completude da especificação do *prompt*: Requisitos não explicitados tendem a não ser atendidos pelos modelos, ainda que sejam avaliados pelas ferramentas de análise. Esse achado reforça a necessidade de alinhamento explícito entre os requisitos declarados no *prompt* e os critérios utilizados na avaliação do código gerado, sob risco de penalizar os modelos por omissões atribuíveis ao desenho do experimento.

Por fim, a classificação dos defeitos identificados nos *scripts* gerados foi realizada de forma manual, conforme descrito na metodologia, o que introduz um grau de subjetividade inerente ao processo de categorização, ainda que mitigado pela definição prévia de critérios objetivos para cada categoria de erro.

5.3 Declaração de Uso de Inteligência Artificial Generativa

Este Trabalho de Conclusão de Curso utiliza modelos de linguagem de grande escala (LLMs) como objeto de investigação científica, constituindo o cerne da pesquisa aqui apresentada.

Além disso, durante a elaboração textual deste trabalho, ferramentas de inteligência artificial generativa foram utilizadas exclusivamente como apoio auxiliar à revisão textual, incluindo correção gramatical, sugestões de clareza e coesão, e verificação de consistência terminológica. A concepção da pesquisa, o desenho experimental, a coleta e a análise dos dados, a interpretação dos resultados e as conclusões apresentadas são de inteira responsabilidade da autora.

5.4 Trabalhos Futuros

A partir das limitações identificadas e dos resultados obtidos, sugerem-se as seguintes direções para trabalhos futuros:

- Ampliar a validação experimental para múltiplos ambientes web de naturezas distintas (por exemplo, aplicações de conteúdo, sistemas administrativos e plataformas *SaaS*), a fim de verificar se os padrões de desempenho observados se mantêm em diferentes contextos de negócio;
- Realizar a comparação entre as estratégias de *prompting zero-shot* e *few-shot*, investigando o impacto da presença de exemplos de referência sobre o *PASS-RATE*, a cobertura de requisitos e a qualidade do código gerado;
- Investigar estatisticamente, por meio de testes de correlação, a relação entre qualidade estática do código, complexidade ciclomática e incidência de categorias

específicas de defeitos, de modo a validar ou refutar formalmente a ausência de associação observada visualmente neste estudo;

- Avaliar estratégias híbridas de geração de testes, combinando modelos distintos para diferentes etapas do processo por exemplo, utilizando um modelo especialista em código para a etapa de planejamento e cobertura de requisitos, e um modelo generalista para a geração do código de interação com o Selenium, explorando se essa combinação supera o desempenho de cada modelo isoladamente;
- Estender a avaliação a modelos *open-weight* adicionais lançados posteriormente a este estudo, dado o ritmo acelerado de evolução dessa categoria de soluções.
- Refinamento de *prompt* a fim de reduzir penalidades no pylint por documentação.

Referências

ABES. *Mercado Brasileiro de Software: panorama e tendências 2024*. São Paulo: [s.n.], 2024. Acesso em: 7 fev. 2026. Disponível em: <https://www.abessoftware.com.br>. Citado 4 vezes nas páginas 14, 18, 35 e 40.

AKHMEDOV, A. B. et al. The impact of artificial intelligence on human life in the future. *International Multidisciplinary Journal for Research & Development*, v. 10, n. 10, 2023. Disponível em: <https://www.ijmrd.in/index.php/imjrd/article/view/239>. Citado na página 14.

ALSHAHWAN, N. et al. Automated unit test improvement using large language models at meta. In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. [S.l.]: ACM, 2024. (FSE 2024), p. 185–196. Citado na página 28.

AMORIM, B. P. d. et al. *GESTÃO DE QUALIDADE NO PROCESSO DE DESENVOLVIMENTO DE SOFTWARE*. 2017. Disponível em: <https://portal.unisepe.com.br/unifia/wp-content/uploads/sites/10001/2018/06/060_gestao.pdf>. Citado na página 16.

AZEVEDO, M. S. d. *Automação de testes funcionais em uma esteira de desenvolvimento de software*. 101 f. p. Dissertação (Mestrado) — Universidade Federal da Paraíba, João Pessoa, 2021. Trabalho de Conclusão de Curso (Graduação em Ciências da Computação). Disponível em: <https://repositorio.ufpb.br/jspui/bitstream/123456789/29068/1/MaxwandellS.deAzevedo_TCC.pdf>. Citado na página 15.

BARBOSA, L.; HORA, A. How and why developers migrate python tests. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2022. p. 538–548. Citado na página 31.

BARTIE, A. *Garantia da qualidade de software: adquirindo maturidade organizacional*. 1. ed. Rio de Janeiro: Elsevier, 2002. ISBN 978-8535211245. Citado na página 14.

BERNARDO, P. C. *Padrões de testes automatizados*. Tese (Dissertação de Mestrado) — Universidade de São Paulo, 2011. Citado na página 16.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. *Future of Software Engineering (FOSE)*, IEEE, p. 85–103, 2007. Citado 3 vezes nas páginas 23, 24 e 26.

BROWN, T. et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, v. 33, 2020. Disponível em: <https://arxiv.org/abs/2005.14165>. Citado na página 29.

CHEN, K. et al. *CODEMENV: Benchmarking Large Language Models on Code Migration*. 2025. Disponível em: <https://arxiv.org/abs/2506.00894>. Citado na página 18.

CHOPRA, A.; PRASHAR, A.; SAIN, C. Natural language processing. *International Journal of Technology Enhancements and Emerging Engineering Research*, v. 1, p. 131–134, 2013. Disponível em: <https://api.semanticscholar.org/CorpusID:196052257>. Citado na página 27.

DOWNEY, A. B. *Think Python: How to Think Like a Computer Scientist*. 2. ed. O'Reilly Media, 2015. Disponível em: <https://greenteapress.com/wp/think-python-2e/>. Citado na página 30.

FERINO, S. et al. Novice developers' perspectives on adopting llms for software development: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, mar. 2026. ISSN 1049-331X. Just Accepted. Disponível em: <https://doi.org/10.1145/3800580>. Citado na página 15.

FERREIRA, M. et al. Acceptance Test Generation with Large Language Models: An Industrial Case Study . In: *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 1–11. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/AST66626.2025.00007>. Citado 3 vezes nas páginas 23, 24 e 26.

GAROUSI, V.; YILDIRIM, S. Introducing automated gui testing and observing its benefits: An industrial case study. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. [S.l.: s.n.], 2018. Citado 3 vezes nas páginas 24, 25 e 26.

GUPTA, N.; CHAUHAN, R. A comparative study of automated testing tools: Selenium, qtp and loadrunner. *International Journal of Computer Applications*, v. 182, n. 5, p. 1–5, 2019. Apresenta revisão sobre ferramentas de testes automatizados, destacando as capacidades do Selenium para aplicações web. Citado na página 30.

HAN, X.; ZHU, H. MASTEST: A LLM-based multi-agent system for RESTful API tests. *arXiv preprint arXiv:2511.18038*, 2025. Disponível em: <https://arxiv.org/pdf/2511.18038>. Citado na página 31.

HUNG, J. Benchmarking open-weight foundation models for global AI technical governance. *arXiv preprint arXiv:2606.26099*, 2026. Disponível em: <https://arxiv.org/pdf/2606.26099>. Citado na página 27.

International Software Test Institute. *Software Testing Levels: Unit, Integration, System, Acceptance*. 2025. <https://www.test-institute.org/Software_Testing_Levels.php>. Citado na página 25.

ISTQB – International Software Testing Qualifications Board. *ISTQB Glossary of Testing Terms*. 2024. <https://istqb-glossary.page/br/>. Citado 2 vezes nas páginas 24 e 25.

KOJIMA, T. et al. Large language models are zero-shot reasoners. *arXiv*, 2022. Disponível em: <https://arxiv.org/abs/2205.11916>. Citado na página 29.

LEAL, J. M. Trabalho de Conclusão de Curso, *Aplicação de LLMs Open-Weight na Automação de Testes Web: Estudo Comparativo Utilizando*

Python e Selenium. 2026. Disponível em: <https://github.com/JessyLeal/LLM-test-generator>. Citado na página 53.

LI, Y. et al. Evaluating large language models for software testing. *Computer Standards & Interfaces*, v. 93, p. 103942, 2025. ISSN 0920-5489. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0920548924001119>. Citado 2 vezes nas páginas 18 e 26.

LIU, P. et al. Pre-train, prompt, predict: A systematic survey of prompting methods in nlp. *ACM Computing Surveys*, v. 55, n. 9, p. 1–35, 2023. Citado na página 29.

LLOPIS, I.; GARCÍA, A.; MARTÍNEZ, O. Automation in software development and testing: state of the art and future trends. *Journal of Software Engineering*, v. 35, p. 58–75, 2021. Citado na página 14.

MALLIPEDDI, S. et al. AutoQALLMs: Automating web application testing using large language models (LLMs) and selenium. *Computers*, v. 14, n. 11, p. 501, 2025. Disponível em: <https://doi.org/10.3390/computers14110501>. Citado na página 34.

MANCHANDA, J. et al. The open-source advantage in large language models (LLMs). *arXiv preprint arXiv:2412.12004*, 2024. Disponível em: <https://arxiv.org/pdf/2412.12004>. Citado na página 27.

MESZAROS, G. *xUnit Test Patterns: Refactoring Test Code*. [S.l.]: Addison-Wesley, 2007. Discute ferramentas e padrões de automação amplamente aplicados com frameworks como o Selenium. ISBN 978-0131495050. Citado na página 30.

MONTEIRO, C. E. O. et al. Automated generation of end-to-end web test cases via a generic AI agent: A comparative study of DeepSeek V3 and Claude Sonnet 4. In: *Anais do XXXI Brazilian Symposium on Multimedia and the Web (WebMedia 2025)*. Rio de Janeiro, RJ: Sociedade Brasileira de Computação, 2025. p. 57–66. Citado na página 34.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. 3. ed. [S.l.]: Wiley, 2011. Citado na página 24.

NAGLE, C. J. *Design for Test Automation – Data Driven Test Automation Frameworks*. 2000. <https://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.html>. Consultado em: 24 nov. 2025. Citado na página 30.

OUÉDRAOGO, W. C. et al. Prompt engineering in LLMs for automated unit test generation: A large-scale study. *Empirical Software Engineering*, v. 31, n. 4, p. 103, 2026. Citado na página 29.

PRESSMAN, R. S. *Engenharia de software: uma abordagem profissional*. 8. ed. São Paulo: McGraw-Hill, 2014. Citado 5 vezes nas páginas 14, 23, 24, 25 e 26.

Pylint Contributors. *Pylint: It's Not Just a Linter that Annoys You*. 2025. <https://pylint.readthedocs.io/>. Acesso em: 9 jul. 2026. Citado na página 44.

pytest contributors. *pytest Documentation*. 2025. <https://docs.pytest.org/en/stable/>. Citado na página 31.

Radon Contributors. *Radon: Various Code Metrics for Python Code*. 2025. <https://radon.readthedocs.io/>. Acesso em: 9 jul. 2026. Citado na página 44.

RANE, N.; CHOUDHARY, S.; RANE, J. Gemini versus chatgpt: applications, performance, architecture, capabilities, and implementation. *Journal of Applied Artificial Intelligence*, v. 5, n. 1, p. 69–93, 2024. Disponível em: <https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4723687>. Citado na página 15.

ROSENBERG, J. Cloud computing: opportunities and challenges for software development. *Computer Science Review*, v. 45, p. 50–68, 2019. Citado na página 14.

ROSSUM, G. V.; DRAKE, F. L. *Python 3 Reference Manual*. CreateSpace Independent Publishing Platform, 2009. Disponível em: <https://docs.python.org/3/reference/>. Citado na página 29.

SANTANA, D.; MAGALHAES, C.; SANTOS, R. de S. Software testing with large language models: An interview study with practitioners. In: *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. [S.l.: s.n.], 2025. p. 96–104. Citado na página 17.

SCHÄFER, M. et al. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, v. 50, n. 1, p. 85–105, jan 2024. Citado 2 vezes nas páginas 18 e 26.

Selenium Project Contributors. *Selenium Documentation*. 2025. <https://www.selenium.dev/documentation/>. Citado na página 30.

SILVA, P. C. B. da; ALVES, T. S.; BRUNO, E. A. AutomaÇão de testes funcionais. In: *Testes funcionais automatizados de software*. Revista de Ciências Exatas e Tecnologia, 2011. p. 95–112. Disponível em: <https://exatastecnologias.pgsscogna.com.br/rcext/article/view/2311#:~:text=Automa%C3%A7%C3%A3o%20de%20testes%20funcionais%3A%20testes%20funcionais%20automatizados%20de%20software>. Citado na página 16.

SIM, S.; EASTERBROOK, S.; HOLT, R. Using benchmarking to advance research: a challenge to software engineering. In: *25th International Conference on Software Engineering, 2003. Proceedings.* [S.l.: s.n.], 2003. p. 74–83. Citado na página 36.

SOARES, E.; COSTA, D. A. d.; KULESZA, U. Continuous integration and software quality: A causal explanatory study. *arXiv*, 2023. Disponível em: <https://arxiv.org/abs/2309.10205>. Citado na página 15.

SOMMERVILLE, I. *Software Engineering*. 9. ed. [S.l.]: Pearson, 2011. Citado 4 vezes nas páginas 15, 16, 23 e 24.

STRANDBERG, P. E. Automated system-level software testing of industrial networked embedded systems. *arXiv preprint arXiv:2111.08312*, 2021. Disponível em: <https://arxiv.org/abs/2111.08312>. Citado na página 16.

THUMMALAPENTA, S. et al. Efficient and change-resilient test automation: An industrial case study. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 1002–1011. Citado na página 15.

UGWA, S. From scripts to intelligence: How ai is reshaping the future of software testing. *World Journal of Advanced Engineering Technology and Sciences*, v. 13, n. 01, p. 1167–1179, 2024. Citado na página 15.

VASWANI, A. et al. Attention is all you need. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [s.n.], 2017. v. 30. Disponível em: <https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf>. Citado na página 27.

VO, T. et al. *Automated Web Application Testing: End-to-End Test Case Generation with Large Language Models and Screen Transition Graphs*. 2025. Disponível em: <https://arxiv.org/abs/2506.02529>. Citado na página 34.

WANG, J. et al. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, v. 50, n. 4, p. 911–936, 2024. Citado na página 28.

WANG, X. et al. Self-consistency improves chain of thought reasoning in language models. *arXiv*, 2023. Disponível em: <https://arxiv.org/abs/2203.11171>. Citado na página 29.

WEI, J. et al. Chain-of-thought prompting elicits reasoning in large language models. *arXiv*, 2022. Disponível em: <https://arxiv.org/abs/2201.11903>. Citado na página 29.

WHITE, J. et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv*, 2023. Disponível em: <https://arxiv.org/abs/2302.11382>. Citado na página 29.

YU, S. et al. LLM for test script generation and migration: Challenges, capabilities, and opportunities. In: *IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. Chiang Mai, Thailand: IEEE, 2023. p. 206–217. Citado na página 18.

ZIMMERMANN, D.; KOZIOLEK, A. Gui-based software testing: An automated approach using gpt-4 and selenium webdriver. In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering Workshops*. Luxembourg: IEEE, 2023. (ASE 2023 Workshop A-TEST). Citado 2 vezes nas páginas 25 e 30.

APÊNDICE A – Visão Geral de Casos de Testes

Tabela 7 – Resumo dos casos de testes

ID	Título do Teste	Tipo de Teste	Objetivo
CT-01	Navegação pelo menu principal	Funcional – Teste de Sistema	Verificar se o usuário consegue navegar pelo menu principal da aplicação.
CT-02	Buscar por produto pela barra de pesquisa	Funcional – Teste de Sistema	Validar se o sistema retorna resultados relevantes ao realizar uma busca.
CT-03	Buscar produto com letras maiúsculas	Funcional – Teste de Sistema	Verificar se a funcionalidade de busca é *case-insensitive*.
CT-04	Buscar produto com termo parcial	Funcional – Teste de Sistema	Verificar se o sistema retorna produtos relevantes com termos incompletos.
CT-05	Buscar produto com caracteres especiais	Funcional – Teste de Sistema	Validar o comportamento do sistema ao receber caracteres especiais na busca.
CT-06	Buscar produto com termo inexistente	Funcional – Teste de Sistema	Verificar como o sistema lida com buscas que não retornam resultados.
CT-07	Exibição das informações principais do produto	Funcional – Teste de Sistema	Verificar se as informações essenciais do produto são exibidas corretamente na página de detalhes.
CT-08	Validar informação de disponibilidade do produto	Funcional – Teste de Sistema	Garantir que o sistema informa corretamente a disponibilidade do produto.
CT-09	Visualizar detalhes de um produto	Funcional – Teste de Sistema	Garantir que o usuário consiga acessar as informações detalhadas de um produto.

Continua na próxima página

ID	Título do Teste	Tipo de Teste	Objetivo
CT-10	Adicionar produto ao carrinho a partir da página do produto	Funcional – Teste de Sistema	Verificar se o produto pode ser adicionado ao carrinho com sucesso.
CT-11	Exibição da descrição e características do produto	Funcional – Teste de Sistema	Verificar se as informações detalhadas do produto estão acessíveis ao usuário.
CT-12	Adicionar produto ao carrinho de compras	Funcional – Teste de Sistema	Verificar se um produto pode ser adicionado ao carrinho sem autenticação.
CT-13	Remover produto do carrinho de compras	Funcional – Teste de Sistema	Verificar se um produto pode ser removido do carrinho sem autenticação.
CT-14	Validar persistência de produto no carrinho após atualização da página	Funcional – Teste de Sistema	Verificar se os produtos adicionados ao carrinho permanecem armazenados após a atualização da página do navegador.
CT-15	Adicionar produto ao carrinho diretamente dos resultados de busca	Funcional – Teste de Sistema	Verificar se o produto pode ser adicionado ao carrinho diretamente da lista de resultados.
CT-16	Impedir adição de produto indisponível ao carrinho	Funcional – Teste de Sistema	Verificar se o sistema impede a adição de produtos indisponíveis ao carrinho.
CT-17	Filtrar produtos por faixa de preço mínimo	Funcional – Teste de Sistema	Verificar se o filtro de preço refina corretamente os resultados da busca.
CT-18	Filtrar produtos por faixa de preço máximo	Funcional – Teste de Sistema	Verificar se o filtro de preço refina corretamente os resultados da busca.
CT-19	Remover filtro de preço aplicado	Funcional – Teste de Sistema	Verificar se o sistema remove corretamente o filtro de preço aplicado e atualiza os resultados da busca.
CT-20	Acesso à página inicial da Amazon Brasil	Funcional – Teste de Sistema	Verificar se a página inicial da Amazon Brasil carrega corretamente.

Fonte: Elaborado pela autora (2026).

APÊNDICE B – Casos de Testes Detalhados

Tabela 8 – Caso de Teste CT-01 – Navegação pelo menu principal

Campo	**Descrição**
ID	CT-01
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se o usuário consegue navegar pelo menu principal da aplicação
Pré-condições	Página inicial do e-commerce carregada corretamente
Passos	1. Acessar a página inicial do e-commerce 2. Localizar o menu superior 3. Clicar na opção “Ofertas do Dia”
Resultado Esperado	O sistema redireciona para a página de ofertas, a URL é atualizada corretamente e a lista de ofertas é exibida
Framework	Selenium + Pytest

Tabela 9 – Caso de Teste CT-02 – Busca por produto

Campo	**Descrição**
ID	CT-02
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Validar se o sistema retorna resultados relevantes ao realizar uma busca
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Localizar a barra de busca 3. Digitar o termo “notebook” 4. Executar a busca
Resultado Esperado	O sistema redireciona para a página de resultados e exibe produtos relacionados ao termo pesquisado
Framework	Selenium + Pytest

Tabela 10 – Caso de Teste CT-03 – Busca com letras maiúsculas

Campo	**Descrição**
ID	CT-03
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se a busca é *case-insensitive*
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Inserir o termo “WHEY PROTEIN” 3. Executar a busca
Resultado Esperado	O sistema retorna resultados normalmente, sem impacto pelo uso de letras maiúsculas
Framework	Selenium + Pytest

Tabela 11 – Caso de Teste CT-04 – Busca com termo parcial

Campo	**Descrição**
ID	CT-04
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Avaliar se o sistema retorna produtos relevantes a partir de termos incompletos
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Inserir o termo “note” na barra de busca 3. Executar a busca
Resultado Esperado	O sistema retorna produtos relacionados, como notebooks
Framework	Selenium + Pytest

Tabela 12 – Caso de Teste CT-05 – Busca com caracteres especiais

Campo	**Descrição**
ID	CT-05
Tipo de Teste	Funcional – Teste de Sistema

Campo	**Descrição**
Objetivo	Validar o comportamento do sistema ao receber caracteres especiais
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Inserir o termo “@biscoito*#” na barra de busca 3. Executar a busca
Resultado Esperado	O sistema não apresenta erro e exibe mensagem de nenhum resultado encontrado
Framework	Selenium + Pytest

Tabela 13 – Caso de Teste CT-06 – Busca com termo inexistente

Campo	**Descrição**
ID	CT-06
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar o tratamento do sistema para buscas sem resultados
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Digitar o termo “zxqv987” 3. Executar a busca
Resultado Esperado	O sistema exibe mensagem informando que não foram encontrados resultados
Framework	Selenium + Pytest

Tabela 14 – Caso de Teste CT-07 – Exibição das informações do produto

Campo	**Descrição**
ID	CT-07
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Garantir que informações essenciais do produto sejam exibidas corretamente
Pré-condições	Página válida de produto acessada

Campo	**Descrição**
Passos	1. Acessar a página de detalhes do produto 2. Localizar título, preço e imagem
Resultado Esperado	Título visível, preço no formato monetário brasileiro e imagem carregada
Framework	Selenium + Pytest

Tabela 15 – Caso de Teste CT-08 – Disponibilidade do produto

Campo	**Descrição**
ID	CT-08
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Assegurar que a disponibilidade do produto seja informada claramente
Pré-condições	Produto acessado via página de resultados
Passos	1. Acessar a página de detalhes do produto 2. Verificar a seção de disponibilidade
Resultado Esperado	Mensagem clara indicando disponibilidade ou indisponibilidade
Framework	Selenium + Pytest

Tabela 16 – Caso de Teste CT-09 – Visualização de detalhes do produto

Campo	**Descrição**
ID	CT-09
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Garantir o acesso à página de detalhes do produto
Pré-condições	Conexão ativa com a internet
Passos	1. Buscar por um produto 2. Selecionar o primeiro item listado
Resultado Esperado	O sistema redireciona para a página de detalhes do produto
Framework	Selenium + Pytest

Tabela 17 – Caso de Teste CT-10 – Adição de produto ao carrinho

Campo	**Descrição**
ID	CT-10
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se um produto pode ser adicionado ao carrinho com sucesso
Pré-condições	Produto disponível para compra
Passos	1. Acessar a página do produto 2. Clicar em “Adicionar ao carrinho”
Resultado Esperado	Mensagem de confirmação exibida e contador do carrinho incrementado
Framework	Selenium + Pytest

Tabela 18 – Caso de Teste CT-11 – Exibição da descrição e características do produto

Campo	**Descrição**
ID	CT-11
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se as informações detalhadas do produto estão acessíveis ao usuário.
Pré-condições	Página de produto carregada
Passos	1. Acessar a página do produto 2. Rolar a página até a seção de descrição 3. Localizar a área de características técnicas ou descrição detalhada
Resultado Esperado	- A seção de descrição do produto está visível - O texto não está vazio - As informações são apresentadas de forma legível
Framework	Selenium + Pytest

Tabela 19 – Caso de Teste CT-12 – Adicionar produto ao carrinho de compras

Campo	**Descrição**
ID	CT-12
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se um produto pode ser adicionado ao carrinho sem autenticação.
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Localizar a barra de busca 3. Procurar pelo produto “whey protein baunilha” 4. Clicar no título ou imagem do produto 5. Clicar no botão “Add to Cart” (Adicionar ao carrinho)
Resultado Esperado	- O sistema redireciona o usuário para a página de detalhes do produto selecionado - A URL corresponde à página de detalhes do produto - O produto é adicionado ao carrinho - O sistema exibe confirmação visual da ação - O contador do carrinho é atualizado
Framework	Selenium + Pytest

Tabela 20 – Caso de Teste CT-13 – Remover produto do carrinho de compras

Campo	**Descrição**
ID	CT-13
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se um produto pode ser removido do carrinho sem autenticação.
Pré-condições	Conexão ativa com a internet
Passos	1. Acessar o site do e-commerce 2. Localizar a barra de busca 3. Procurar pelo produto “whey protein baunilha” 4. Clicar no título ou imagem do produto 5. Clicar no botão “Add to Cart” (Adicionar ao carrinho) 6. Acessar o carrinho 7. Clicar na opção “Remover” associada ao produto 8. Aguardar a atualização da página

Campo	Descrição
Resultado Esperado	- O sistema redireciona o usuário para a página de detalhes do produto selecionado - A URL corresponde à página de detalhes do produto - O produto é removido da lista - O carrinho reflete o estado atualizado - Caso não haja mais itens, é exibida mensagem de carrinho vazio
Framework	Selenium + Pytest

Tabela 21 – Caso de Teste CT-14 – Validar persistência de produto no carrinho após atualização da página

Campo	Descrição
ID	CT-14
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se os produtos adicionados ao carrinho permanecem armazenados após a atualização da página do navegador.
Pré-condições	Produto adicionado ao carrinho
Passos	1. Atualizar a página do navegador 2. Aguardar o recarregamento 3. Clicar no ícone do carrinho
Resultado Esperado	- O produto permanece listado no carrinho - A quantidade adicionada é mantida - Nenhuma perda de dados ocorre
Framework	Selenium + Pytest

Tabela 22 – Caso de Teste CT-15 – Adicionar produto ao carrinho diretamente dos resultados de busca

Campo	Descrição
ID	CT-15
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se o produto pode ser adicionado ao carrinho diretamente da lista de resultados.

Campo	**Descrição**
Pré-condições	- Página de resultados de busca carregada - Produto disponível para compra
Passos	1. Acessar a página de resultados de busca 2. Localizar um produto que possua botão “Adicionar ao carrinho” 3. Clicar no botão correspondente 4. Aguardar a resposta do sistema
Resultado Esperado	- O sistema exibe mensagem de confirmação - O produto é adicionado ao carrinho - O contador do carrinho é incrementado
Framework	Selenium + Pytest

Tabela 23 – Caso de Teste CT-16 – Impedir adição de produto indisponível ao carrinho

Campo	**Descrição**
ID	CT-16
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se o sistema impede a adição de produtos indisponíveis ao carrinho.
Pré-condições	Usuário acessa a página de um produto indisponível
Passos	1. Acessar a página de um produto sem estoque 2. Localizar a área de compra 3. Verificar a presença ou ausência do botão “Adicionar ao carrinho”
Resultado Esperado	- O botão “Adicionar ao carrinho” não está disponível ou está desabilitado - Uma mensagem de indisponibilidade é exibida - O produto não é adicionado ao carrinho
Framework	Selenium + Pytest

Tabela 24 – Caso de Teste CT-17 – Filtrar produtos por faixa de preço mínimo

Campo	**Descrição**
ID	CT-17

Campo	**Descrição**
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se o filtro de preço refina corretamente os resultados da busca.
Pré-condições	- Conexão ativa com a internet - Usuário acessa o site do e-commerce
Passos	1. Acessar o site do e-commerce 2. Localizar a barra de busca 3. Pesquisar pelo produto “whey” 4. Na página de resultados, localizar o filtro de preço 5. Selecionar a opção de menor valor
Resultado Esperado	- A lista de produtos é atualizada após a aplicação do filtro - Todos os produtos exibidos possuem preço até a opção de menor valor - Produtos fora da faixa definida não são exibidos - O filtro de preço aplicado permanece visível na interface
Framework	Selenium + Pytest

Tabela 25 – Caso de Teste CT-18 – Filtrar produtos por faixa de preço máximo

Campo	**Descrição**
ID	CT-18
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se o filtro de preço refina corretamente os resultados da busca.
Pré-condições	- Conexão ativa com a internet - Usuário acessa o site do e-commerce
Passos	1. Acessar o site do e-commerce 2. Localizar a barra de busca 3. Pesquisar pelo produto “whey” 4. Na página de resultados, localizar o filtro de preço 5. Selecionar a opção de maior valor
Resultado Esperado	- A lista de produtos é atualizada após a aplicação do filtro - Todos os produtos exibidos possuem preço até a opção de maior valor - Produtos fora da faixa definida não são exibidos - O filtro de preço aplicado permanece visível na interface

Campo	Descrição
Framework	Selenium + Pytest

Tabela 26 – Caso de Teste CT-19 – Remover filtro de preço aplicado

Campo	Descrição
ID	CT-19
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se o sistema remove corretamente o filtro de preço aplicado e atualiza os resultados da busca.
Pré-condições	Filtro de preço aplicado
Passos	1. Localizar o filtro ativo 2. Clicar na opção de remover filtro
Resultado Esperado	- A lista de produtos é atualizada - O filtro não aparece mais como ativo
Framework	Selenium + Pytest

Tabela 27 – Caso de Teste CT-20 – Acesso à página inicial da Amazon Brasil

Campo	Descrição
ID	CT-20
Tipo de Teste	Funcional – Teste de Sistema
Objetivo	Verificar se a página inicial da Amazon Brasil carrega corretamente.
Pré-condições	Conexão ativa com a internet
Passos	1. Abrir o navegador 2. Acessar o site da Amazon Brasil (<https://www.amazon.com.br>)
Resultado Esperado	- A página inicial é carregada com sucesso - O logotipo da Amazon é exibido - O menu superior está visível
Framework	Selenium + Pytest

APÊNDICE C – Prompts Utilizados na Geração dos Testes

Este apêndice apresenta a íntegra dos *prompts* utilizados para a geração dos *scripts* de teste automatizados por meio da técnica *few-shot*. Cada *prompt* enviado aos modelos resulta da concatenação de três blocos, nesta ordem:

1. A estrutura de instruções, contexto e restrições, fixa para os 20 casos de teste e apresentada na Seção C.1;
2. A descrição do caso de teste específico, variável para cada um dos 20 casos e apresentada na Seção C.2;
3. Um exemplo de *output* no padrão *few-shot*, também fixo para os 20 casos e apresentado na Seção C.3. Os blocos fixos são exibidos uma única vez a seguir, evitando a repetição do mesmo conteúdo em cada um dos 20 casos de teste.

C.1 Bloco 1 - Estrutura fixa do *prompt*

Estrutura fixa (role, contexto, restrições e formato esperado)

```
Role
Você é um engenheiro de qualidade de software especializado em automação de testes web com Python, pytest e Selenium.

- Contexto
Deseja-se gerar testes automatizados para uma aplicação web, a Amazon Brasil (https://www.amazon.com.br/).

# Objetivo
Gerar um teste automatizado utilizando Python, pytest e Selenium baseado no caso de teste fornecido, seguindo os padrões de projeto de automação de testes.

# O projeto já utiliza:

- pytest
- Selenium
- Page Object Model
- seleniumpagefactory.Pagefactory
- O driver(driver) e a validação de internet(is_connected) são fixtures já presentes no conftest, chame-os como parâmetro no teste.


# Restrições de geração de código
- O código deve ser executável
- Retorne apenas o conteúdo bruto do arquivo Python.
```

```
- Não inclua markdown.
- Não inclua explicações.

# Restrições de desenvolvimento de teste
- Utilize:
    "from seleniumpagefactory.Pagefactory import PageFactory" para
        mapeamento de elementos da tela
- Só adicione métodos caso o comportamento necessário não exista nas
    classes existentes.
- Não recrie Page Objects existentes.
- Reutilize classes, métodos e locators existentes.
- Todas as classes Page Object (ex: Home, SearchResults, ProductDetails,
    Cart) devem ser definidas no próprio arquivo de teste gerado, antes
    da função de teste.
- Não importe classes Page Object de módulos externos.
- Não utilize caminhos de importação como "from your_project..." ou
    similares.
- O fluxo completo deve estar em uma única função de teste.
- Métodos auxiliares podem existir apenas dentro das Page Objects.
- Utilize apenas os imports necessários para execução.
- Utilize WebDriverWait e expected_conditions.
- Não utilize time.sleep().
- Utilize nomenclatura em snake_case.
- Os testes devem seguir o padrão test_<id>.
- Utilize logging.info() para todos os passos [STEP] e resultados
    esperados [EXPECTED RESULT] do teste automatizado.
- Utilize o padrão Page Object Model.
- A lógica de interação com elementos deve permanecer dentro das classes
    Page Object.
- O teste deve conter apenas fluxo e validações.
- Utilize assertions explícitas com mensagens descritivas.

# Formato de output esperado
1. Criar o teste correspondente à descrição do teste e ao exemplo de
    output fornecido.
2. Utilizar asserts com mensagens descritivas para validar o
    comportamento esperado no cenário de teste apresentado.

# Estrutura esperada do arquivo gerado:
import logging
from seleniumpagefactory.Pagefactory import PageFactory, By,
    WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

AMAZON_URL = "https://www.amazon.com.br/"

class Home(PageFactory):
    ...

class SearchResults(PageFactory):
    ...

def test_<id>(driver, is_connected):
    ...
--------------------------------------------------------------------------------
```

C.2 Bloco 2 – Casos de teste (parte variável do prompt)

Para cada um dos 20 casos de teste a seguir, o *prompt* completo enviado ao modelo corresponde à concatenação do Bloco 1 (Seção C.1) + o bloco variável apresentado abaixo + o Bloco 3 (Seção C.3), nesta ordem.

C.2.1 CT-01 – Navegação pelo menu principal

Caso de teste – CT-01

```
Tipo de teste: Funcional - Teste de Sistema
Título: Navegação pelo menu principal

Objetivo:
- Verificar se o usuário consegue navegar pelo menu principal.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Página inicial carregada

Passos:
1. Acessar a página inicial da Amazon
2. Localizar o menu superior
3. Clicar na opção "Ofertas do "Dia

Resultado esperado:
- O sistema redireciona para a página de ofertas
- A URL é atualizada corretamente
- A lista de ofertas é exibida
```

C.2.2 CT-02 – Buscar por produto pela barra de pesquisa

Caso de teste – CT-02

```
Tipo de teste: Funcional - Teste de Sistema
Título: Buscar por produto utilizando a barra de pesquisa

Objetivo:
- Validar se o sistema retorna resultados relevantes ao realizar uma
    busca

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet

Passos:
1. Acessar o site da Amazon
```

```
2. Localizar a barra de busca no topo da página
3. Digitar o termo ""notebook
4. Clicar no botão de busca ou pressionar Enter

Resultado esperado:
- O sistema redireciona para a página de resultados de busca
- A página exibe uma lista de produtos relacionados ao termo ""notebook
```

C.2.3 CT-03 – Buscar produto com letras maiúsculas

Caso de teste – CT-03

```
Tipo de teste: Funcional - Teste de Sistema
Título: Buscar produto com letras maiúsculas

Objetivo:
- Validar se a busca é case-insensitive.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet

Passos:
1. Acessar a Amazon
2. Localizar o campo de busca
3. Digitar o termo "WHEY PROTEIN" em letras maiúsculas
4. Executar a busca

Resultado esperado:
- A página de resultados é exibida corretamente
- Produtos relacionados ao termo pesquisado são apresentados
```

C.2.4 CT-04 – Buscar produto com termo parcial

Caso de teste – CT-04

```
Tipo de teste: Funcional - Teste de Sistema
Título: Buscar produto com termo parcial

Objetivo:
- Verificar se o sistema retorna produtos relevantes com termos
    incompletos.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet

Passos:
1. Acessar o site da Amazon
2. Inserir o termo "note" na barra de busca
3. Executar a busca
```

```
15
16 Resultado esperado:
17 - A página de resultados é exibida corretamente
18 - Produtos relacionados ao termo pesquisado são apresentados
```

C.2.5 CT-05 – Buscar produto com caracteres especiais

Caso de teste – CT-05

```
1  Tipo de teste: Funcional - Teste de Sistema
2  Título: Buscar produto utilizando caracteres especiais
3
4  Objetivo:
5  - Validar o comportamento da busca ao receber caracteres especiais.
6
7  Pré-condições:
8  - Usuário acessando a Amazon Brasil
9  - Conexão ativa com a internet
10
11 Passos:
12 1. Acessar o site da Amazon
13 2. Inserir o termo "@biscoito*#" na barra de busca
14 3. Executar a busca
15
16 Resultado esperado:
17 - O sistema não apresenta erro
18 - Opções relacionadas a biscoito devem aparecer na tela de resultados de
       busca
```

C.2.6 CT-06 – Buscar produto com termo inexistente

Caso de teste – CT-06

```
1  Tipo de teste: Funcional - Teste de Sistema
2  Título: Buscar produto com termo inexistente
3
4  Objetivo:
5  - Verificar como o sistema lida com buscas que não retornam resultados.
6
7  Pré-condições:
8  - Usuário acessando a Amazon Brasil
9  - Conexão ativa com a internet
10
11 Passos:
12 1. Acessar o site da Amazon
13 2. Localizar a barra de busca
14 3. Digitar o termo "zxqv987"
15 4. Executar a busca clicando no botão de pesquisa ou pressionando Enter
16
17 Resultado esperado:
18 - O sistema redireciona para a página de resultados de busca
```

```
- É exibida uma mensagem informando que não foram encontrados resultados
    para o termo pesquisado
```

C.2.7 CT-07 – Exibição das informações principais do produto

Caso de teste – CT-07

```
Tipo de teste: Funcional - Teste de Sistema
Título: Exibição das informações principais do produto

Objetivo:
Verificar se as informações essenciais do produto são exibidas
    corretamente na página de detalhes.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet
- Pesquisar e acessar a página de um produto válido da Amazon (whey)

Passos:
1. Acessar a página do produto
2. Aguardar o carregamento completo da página
3. Localizar o título do produto
4. Localizar o preço principal do produto
5. Localizar a imagem principal do produto

Resultado esperado:
- O título do produto está visível e não está vaziow
- O preço do produto é exibido no formato monetário brasileiro
- Pelo menos uma imagem do produto é carregada
- Nenhuma mensagem de erro é apresentada
```

C.2.8 CT-08 – Validar informação de disponibilidade do produto

Caso de teste – CT-08

```
Tipo de teste: Funcional - Teste de Sistema

Título: Validar informação de disponibilidade do produto

Objetivo:
Verificar se o sistema exibe corretamente a disponibilidade do produto na
     página de detalhes.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Usuário localizado na página de resultados de busca
- Produto disponível para acesso na listagem

Passos:
1. Selecionar um produto na página de resultados
```

```
2. Acessar a página de detalhes do produto
3. Localizar a seção de disponibilidade ou estoque
4. Verificar a mensagem exibida sobre disponibilidade do produto

Resultado esperado:
- O sistema exibe claramente o status de disponibilidade do produto
- A mensagem apresentada informa corretamente se o produto está
    disponível, indisponível ou temporariamente sem estoque
```

C.2.9 CT-09 – Visualizar detalhes de um produto

Caso de teste – CT-09

```
Tipo de teste: Funcional - Teste de Sistema
Título: Visualizar detalhes de um produto

Objetivo:
- Garantir que o usuário consiga acessar as informações detalhadas de um
    produto.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet

Passos:
1. Acessar o site da Amazon Brasil
2. Localizar a barra de busca
3. Digitar o termo ""notebook e executar a busca
4. Na página de resultados, selecionar o primeiro produto listado
5. Clicar no título ou na imagem do produto

Resultado esperado:
- O sistema redireciona para a página de detalhes do produto selecionado
```

C.2.10 CT-10 – Adicionar produto ao carrinho a partir da página do produto

Caso de teste – CT-10

```
Tipo de teste: Funcional - Teste de Sistema
Título: Adicionar produto ao carrinho a partir da página do produto

Objetivo:
- Verificar se um produto pode ser adicionado ao carrinho com sucesso.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Produto disponível para compra

Passos:
```

```
1. Acessar a Amazon
2. Pesquisar por "BATOM"
3. Clicar no primeiro produto disponível
4. Acessar a página do produto
5. Localizar o botão "Adicionar ao "carrinho
6. Clicar no botão
7. Aguardar a resposta do sistema

Resultado esperado:
- O sistema exibe uma mensagem de confirmação "(Adicionado ao "carrinho)
- O contador do carrinho é incrementado
```

C.2.11 CT-11 – Exibição da descrição e características do produto

Caso de teste – CT-11

```
Tipo de teste: Funcional - Teste de Sistema
Título: Exibição da descrição e características do produto

Objetivo:
- Verificar se as informações detalhadas do produto estão acessíveis ao
    usuário.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Página de produto carregada

Passos:
1. Acessar a página do produto
2. Rolar a página até a seção de descrição
3. Validar se a descrição está visível

Resultado esperado:
- A seção de descrição do produto está visível
- O texto não está vazio
- As informações são apresentadas de forma legível
```

C.2.12 CT-12 – Adicionar produto ao carrinho de compras

Caso de teste – CT-12

```
Tipo de teste: Funcional - Teste de Sistema
Título: Adicionar produto ao carrinho de compras

Objetivo:
- Verificar se um produto pode ser adicionado ao carrinho sem
    autenticação.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet
```

```
10
11 Passos:
12 1. Acessar o site da Amazon
13 2. Localizar a barra de busca
14 3. Procurar pelo produto "whey protein baunilha"
15 4. Clicar no título ou imagem do produto
16 5. Clicar no botão "Add to "Cart (Adicionar ao carrinho)
17
18 Resultado esperado:
19 - O sistema redireciona o usuário para a página de detalhes do produto
       selecionado
20 - A URL corresponde à página de detalhes do produto
21 - O produto é adicionado ao carrinho
22 - O sistema exibe confirmação visual da ação
23 - O contador do carrinho é atualizado
```

C.2.13 CT-13 – Remover produto do carrinho de compras

Caso de teste – CT-13

```
1  Tipo de teste: Funcional - Teste de Sistema
2  Título: Remover produto do carrinho de compras
3
4  Objetivo:
5  - Verificar se um produto pode ser removido do carrinho sem autenticação.
6
7  Pré-condições:
8  - Usuário acessando a Amazon Brasil
9  - Conexão ativa com a internet
10
11 Passos:
12 1. Acessar o site da Amazon
13 2. Localizar a barra de busca
14 3. Procurar pelo produto "whey protein baunilha"
15 4. Clicar no título ou imagem do produto
16 5. Clicar no botão "Add to "Cart (Adicionar ao carrinho)
17 6. Acessar o carrinho
18 7. Clicar na opção ""Remover associada ao produto
19 8. Aguardar a atualização da página
20
21 Resultado esperado:
22 - O sistema redireciona o usuário para a página de detalhes do produto
       selecionado
23 - A URL corresponde à página de detalhes do produto
24 - O produto é removido da lista
25 - O carrinho reflete o estado atualizado
26 - Caso não haja mais itens, é exibida mensagem de carrinho vazio
```

C.2.14 CT-14 – Manter produto no carrinho após atualização da página

Caso de teste – CT-14

```
Tipo de teste: Funcional - Teste de Sistema

Título: Validar persistência de produto no carrinho após atualização da
    página

Objetivo:
Verificar se os produtos adicionados ao carrinho permanecem armazenados
    após a atualização da página do navegador

Pré-condições:
- Usuário acessando a Amazon Brasil
- Pelo menos 1 produto adicionado ao carrinho
- Ícone do carrinho exibindo a quantidade correta de itens

Passos:

1. Adicionar um produto ao carrinho
2. Validar que o produto foi adicionado com sucesso
3. Atualizar a página do navegador
4. Aguardar o carregamento completo da página
5. Clicar no ícone do carrinho
6. Visualizar os produtos presentes no carrinho

Resultado esperado:

- O produto previamente adicionado permanece listado no carrinho
- A quantidade do produto permanece inalterada após a atualização da
    página
```

C.2.15 CT-15 – Adicionar produto ao carrinho diretamente dos resultados de busca

Caso de teste – CT-15

```
Tipo de teste: Funcional - Teste de Sistema
Título: Adicionar produto ao carrinho diretamente dos resultados de busca

Objetivo:
- Verificar se o produto pode ser adicionado ao carrinho diretamente da
    lista de resultados

Pré-condições:
- Usuário acessando a Amazon Brasil
- Página de resultados de busca carregada
- Produto disponível para compra

Passos:
1. Acessar a página de resultados de busca
2. Localizar um produto que possua botão "Adicionar ao "carrinho
```

```
3. Clicar no botão correspondente
4. Aguardar a resposta do sistema

Resultado esperado:
- O sistema exibe mensagem de confirmação
- O produto é adicionado ao carrinho
- O contador do carrinho é incrementado
```

C.2.16 CT-16 – Impedir adição de produto indisponível ao carrinho

Caso de teste – CT-16

```
Tipo de teste: Funcional - Teste de Sistema
Título: Impedir adição de produto indisponível ao carrinho

Objetivo:
- Verificar se o sistema impede a adição de produtos indisponíveis ao
    carrinho.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Produto indisponível disponível para consulta na plataforma
- Produto alvo identificado pelo ASIN: B08K89GNFD

Passos:
1. Pesquisar o produto pelo ASIN na barra de pesquisa
1. Acessar a página do produto
2. Aguardar o carregamento completo da página de detalhes do produto
3. Verificar a disponibilidade do botão "Adicionar ao "carrinho
4. Verificar a exibição da mensagem de indisponibilidade ou ausência de
    oferta disponível

Resultado esperado:
- O botão "Adicionar ao "carrinho não está disponível ou está
    desabilitado
- Uma mensagem de indisponibilidade é exibida
- O produto não é adicionado ao carrinho
```

C.2.17 CT-17 – Filtrar produtos por faixa de preço mínimo

Caso de teste – CT-17

```
Tipo de teste: Funcional - Teste de Sistema
Título: Filtrar produtos por faixa de preço mínimo

Objetivo:
- Verificar se o filtro de preço refina corretamente os resultados da
    busca.

Pré-condições:
- Usuário acessando a Amazon Brasil
```

```
- Conexão ativa com a internet

Passos:
1. Acessar o site da Amazon
2. Localizar a barra de busca
3. Pesquisar pelo produto "whey"
4. Na página de resultados, localizar o filtro de preço
5. Selecionar a opção de menor valor

Resultado esperado:
- O filtro está ativo
- Todos os produtos exibidos possuem preço até a opção de menor valor
- O filtro de preço aplicado permanece visível na interface
```

C.2.18 CT-18 – Filtrar produtos por faixa de preço máximo

Caso de teste – CT-18

```
Tipo de teste: Funcional - Teste de Sistema
Título: Filtrar produtos por faixa de preço máximo

Objetivo:
- Verificar se o filtro de preço refina corretamente os resultados da
    busca.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet

Passos:
1. Acessar o site da Amazon
2. Localizar a barra de busca
3. Pesquisar pelo produto "whey"
4. Na página de resultados, localizar o filtro de preço
5. Selecionar a opção de maior valor

Resultado esperado:
- O filtro está ativo
- Todos os produtos exibidos possuem preço até a opção de maior valor
- O filtro de preço aplicado permanece visível na interface
```

C.2.19 CT-19 – Remover filtro de preço aplicado

Caso de teste – CT-19

```
Tipo de teste: Funcional - Teste de Sistema
Título: Remover filtro de preço aplicado

Objetivo:

```

```
Verificar se o sistema remove corretamente o filtro de preço aplicado e
    atualiza os resultados da busca.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Usuário acessou a página de resultados de busca
- Um filtro de preço foi aplicado previamente
- O botão ou opção ""Limpar está visível na área de filtros ativos

Passos:
1. Localizar o filtro de preço ativo na página
2. Clicar na opção ""Limpar para remover o filtro
3. Aguardar a atualização da lista de produtos

Resultado esperado:
- O filtro de preço deixa de aparecer como ativo
- O botão ""Limpar não é mais exibido
- A lista de produtos é atualizada corretamente
- Produtos fora da faixa anteriormente filtrada voltam a ser exibidos
- Nenhum erro visual ou funcional ocorre durante a atualização da página
```

C.2.20 CT-20 – Acesso à página inicial

Caso de teste – CT-20

```
Tipo de teste: Funcional - Teste de Sistema
Título: Acesso à página inicial da Amazon

Objetivo:
- Verificar se a página inicial da Amazon carrega corretamente.

Pré-condições:
- Usuário acessando a Amazon Brasil
- Conexão ativa com a internet

Passos:
1. Abrir o navegador
2. Acessar o site https://www.amazon.com.br

Resultado esperado:
- A página inicial é carregada com sucesso
- O logotipo da Amazon é exibido
- O menu superior está visível
```

C.3 Bloco 3 – Exemplo de *output few-shot* (fixo)

Exemplo de output few-shot, referente ao CT-18

```
import re
from typing import Optional

from selenium.webdriver.support import expected_conditions as EC
from seleniumpagefactory.Pagefactory import By, PageFactory,
    WebDriverWait
import logging

AMAZON_URL = "https://www.amazon.com.br/"

class Home(PageFactory):
    locators = {
        "search_field":     ("ID", "twotabsearchtextbox"),
        "btn_search":       ("ID", "nav-search-submit-button"),
        "logo":             ("ID", "nav-logo"),
        "nav_menu":         ("ID", "nav-hamburger-menu"),
        "ofertas_do_dia":   ("CSS", "a[href*='/deals']")
    }
    def __init__(self, driver):
        self.driver = driver
        self.timeout = 15

    def search(self, term: str):
        self.search_field.set_text(term)
        self.btn_search.click_button()

    def go_to_daily_deals(self) -> None:
        self.ofertas_do_dia.click_button()

    def is_in_screen(self) -> bool:
        return self.search_field.is_displayed()

class Cart(PageFactory):
    locators = {
        "cart_items": ("ID", "sc-active-cart"),
        "btn_remove": ("CSS", "button[data-a-selector='decrement']"),
        "empty_message": ("CSS", "div[data-feature-id='delete-success-
            message']"),
    }
    def __init__(self, driver):
        self.driver = driver
        self.timeout = 15

    def remove_first_item(self) -> None:
        self.btn_remove.click_button()

    def is_empty(self) -> bool:
        try:
            return self.empty_message.is_displayed()
        except Exception:
            return False
```

```
50
51 class SearchResults(PageFactory):
52     locators = {
53         "first_product_add_to_cart": ("XPATH", "(//button[@name='submit.
               addToCart'])[1]"),
54         "no_results_message": ("XPATH", "//span[contains(text(),'Nenhum
               resultado para sua consulta')]"),
55         "price_filter_section": ("ID", "priceRefinements"),
56         "min_price_filter_option": ("XPATH", "(//ul[@id='filter-p_36']//
               li//a)[1]"),
57         "max_price_filter_option": ("XPATH", "(//ul[@id='filter-p_36']//
               li//a)[last()]"),
58         "btn_clean_filters": ("XPATH", '//a[.//span[normalize-space()="
               Limpar"]]')
59     }
60
61     def __init__(self, driver):
62         self.driver = driver
63         self.timeout = 15
64
65     def select_first_product(self):
66         products = self.driver.find_elements(By.CSS_SELECTOR, 'div[data-
               component-type="s-search-result"]')
67         first_product = products[0]
68         if not first_product.is_displayed():
69             raise Exception("Nenhum produto visível nos resultados")
70
71         first_product.click()
72
73     def add_to_cart(self):
74         if not self.first_product_add_to_cart.is_displayed():
75             raise Exception("Botão de adicionar ao carrinho não está
                   visível para o primeiro produto")
76         self.first_product_add_to_cart.click_button()
77
78     def has_no_results(self) -> bool:
79         try:
80             return self.no_results_message.is_displayed()
81         except Exception:
82             return False
83
84     def has_price_filter(self) -> bool:
85         try:
86             return self.price_filter_section.is_displayed()
87         except Exception:
88             return False
89
90     def apply_min_price_filter(self) -> None:
91         self.driver.execute_script(
92         "arguments[0].scrollIntoView({block: 'center'});",
93         self.min_price_filter_option
94     )
95         self.min_price_filter_option.click()
96
97     def apply_max_price_filter(self) -> None:
```

```
98          self.driver.execute_script(
99          "arguments[0].scrollIntoView({block: 'center'});",
100         self.max_price_filter_option
101     )
102         self.max_price_filter_option.click()
103
104     def get_active_price_filter(self) -> Optional[list[float]]:
105         try:
106             WebDriverWait(self.driver, self.timeout).until(
107                 EC.presence_of_element_located(
108                     (By.ID, "s-refinements-a11y-summary")
109                 )
110             )
111
112             element = self.driver.find_element(
113                 By.ID,
114                 "s-refinements-a11y-summary"
115             )
116             active = element.get_attribute("textContent").strip()
117             filters = re.findall(r'R\$\d[\d\.]*', active)
118             print(filters)
119             prices = [
120                 float(filter.replace("R$", "").replace(".", "").replace("
                        ,", "."))
121                 for filter in filters
122             ]
123             return prices
124         except Exception as e:
125             print(f"Erro ao obter filtro de preço ativo: {e}")
126             return None
127     def validate_price_filter_applied(self, expected_price: float) ->
            bool:
128         products = self.driver.find_elements(
129             "xpath",
130             '//div[@data-component-type="s-search-result"]'
131         )
132
133         if not products:
134             logging.info("Nenhum produto encontrado")
135             return False
136
137         valid_products = 0
138
139         for index, product in enumerate(products, start=1):
140             try:
141                 whole = product.find_element(
142                     "xpath",
143                     './/span[contains(@class,"a-price-whole")]'
144                 ).text
145
146                 fraction = product.find_element(
147                     "xpath",
148                     './/span[contains(@class,"a-price-fraction")]'
149                 ).text
150
```

```
                whole = (
                    whole
                    .replace(".", "")
                    .replace(",", "")
                    .strip()
                )

                fraction = fraction.strip()

                price = float(f"{whole}.{fraction}")

                logging.info(
                    f"Produto {index} | Preço encontrado: {price}"
                )

                valid_products += 1

                if price > expected_price:
                    logging.info(
                        f"Preço inválido encontrado: {price} "
                        f"(máximo esperado: {expected_price})"
                    )
                    return False

            except Exception as e:
                logging.info(
                    f"Produto {index} ignorado. "
                    f"Não foi possível obter o preço: {e}"
                )
                continue

        if valid_products == 0:
            logging.info("Nenhum produto válido com preço encontrado")
            return False

        return True

    def clean_filters(self) -> None:
        self.btn_clean_filters.click_button()


class ProductDetails(PageFactory):
    locators = {
        "title":            ("ID", 'productTitle'),
        "price_container":   ("CSS",'div.apex-core-price-identifier'),
        "main_image":       ("ID", "landingImage"),
        "availability":     ("ID", "availability"),
        "unavailable_item": ("XPATH", "//span[contains(text(),'Nenhuma
            opção de compra em destaque')]"),
        "description":      ("ID", "productDescription"),
        "btn_add_to_cart":  ("ID", "add-to-cart-button"),
        "btn_cart_count":   ("ID", "nav-cart-count"),
        "cart_confirmation":("XPATH", "//h1[contains(text(),'Adicionado
            ao carrinho')]")
    }
```

```
204     def __init__(self, driver):
205         self.driver = driver
206         self.timeout = 15
207 
208     def _get_product_price(self) -> str:
209         price_element = self.price_container.find_element(
210             By.CSS_SELECTOR,
211             'span.a-price span.a-offscreen'
212         )
213 
214         return price_element.get_attribute("textContent").replace("R$", "
                ").replace(".", "").replace(",", ".").strip()
215 
216     def is_in_screen(self) -> bool:
217         return self.title.is_displayed()
218 
219     def check_availability(self) -> bool:
220         try:
221             self.availability.is_displayed()
222             return  "em estoque" in self.availability.get_text().strip().
                    lower()
223         except Exception as e:
224             return False
225 
226     def is_item_unavailable(self) -> bool:
227         try:
228             return self.unavailable_item.is_displayed()
229         except Exception:
230             return False
231 
232     def get_product_name(self) -> str:
233         return self.title.text.strip()
234 
235     def get_price(self) -> float:
236         try:
237             return float(self._get_product_price())
238         except Exception:
239             raise Exception("Preço não encontrado na página.")
240 
241     def get_url(self) -> str:
242         return self.driver.current_url
243 
244     def add_to_cart(self) -> None:
245         self.btn_add_to_cart.click_button()
246 
247     def get_cart_count(self) -> str:
248         return self.btn_cart_count.text.strip()
249 
250     def is_cart_confirmation_visible(self) -> bool:
251         try:
252             return self.cart_confirmation.is_displayed()
253         except Exception:
254             return False
255 
256 def test_apply_max_price_filter(driver):
```

```
257     home = Home(driver)
258     results = SearchResults(driver)
259
260     logging.info("TESTE: Filtrar produtos por faixa de preço máximo")
261
262     logging.info("[STEP] Acessar a página inicial da Amazon")
263     driver.get(AMAZON_URL)
264     assert home.is_in_screen(), (
265         "A página inicial da Amazon não foi acessada"
266     )
267     logging.info("[EXPECTED RESULT] A página inicial da Amazon acessada
            com sucesso")
268
269
270     logging.info("[STEP] Pesquisar pelo produto 'whey'")
271     home.search("whey")
272
273     logging.info("[STEP] Aplicar filtro de preço máximo")
274     results.apply_max_price_filter()
275
276     after_filters = results.get_active_price_filter()
277     assert after_filters is not None and len(after_filters) > 0, (
278         "Nenhum filtro de preço ativo encontrado após aplicar o filtro
                máximo"
279     )
280     logging.info("[EXPECTED RESULT] O filtro de preço máximo está ativo")
281
282     logging.info("[STEP] Validar se os produtos exibidos estão dentro da
            faixa de preço máximo aplicada")
283
284     after = after_filters[-1]
285     assert results.validate_price_filter_applied(after), "Produtos for do
             valor esperado encontrados"
286
287     logging.info(f"[EXPECTED RESULT] Todos os produtos válidos possuem
            preço até {after}")
```