Universidade Federal de Goiás
Instituto de Informática

Gabriel Porto do Carmo

Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Goiânia
2025

Universidade Federal de Goiás
Instituto de Informática

Autorização para Publicação de Trabalho de Conclusão de Curso em Formato Eletrônico

Na qualidade de titular dos direitos de autor, **AUTORIZO** a Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos I e VI, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulte, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Autor(a): Gabriel Porto do Carmo

Goiânia, de Novembro de 2025.

Gabriel Porto do Carmo – Autor

Gilmar Ferreira Arantes – Orientador

Gabriel Porto do Carmo

Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Trabalho de Conclusão apresentado à Coordenação do Curso de Bacharelado em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Bacharelado em Ciência da Computação.

Área de concentração: Qualidade de Software.

Orientador: Prof. Gilmar Ferreira Arantes

Goiânia
2025

Gabriel Porto do Carmo

Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Trabalho de Conclusão apresentado à Coordenação do Curso de Bacharelado em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Bacharel em Bacharelado em Ciência da Computação, aprovada em de Novembro de 2025, pela Banca Examinadora constituída pelos professores:

Prof. Gilmar Ferreira Arantes
Instituto de Informática – UFG
Presidente da Banca

Prof. Alessandro Cruvinel Machado de Araújo
Instituto de Informática – UFG

Gabriel Porto do Carmo

Graduando em Ciência da Computação pela Universidade Federal de Goiás, iniciou sua trajetória profissional como estagiário na área de desenvolvimento de software, atuando principalmente com Java e tecnologias relacionadas ao ecossistema da linguagem. Participou de atividades de implementação de funcionalidades, correção de falhas e manutenção de aplicações corporativas. Atualmente trabalha na área de tecnologia, voltado ao desenvolvimento e aprimoramento de sistemas, com interesse crescente em automação de testes, integração contínua e qualidade de software.

Dedico este trabalho à minha família, pelo amor, apoio e incentivo em todos os momentos da minha vida.

Agradecimentos

Agradeço primeiramente aos meus pais, por todo amor, apoio e dedicação ao longo da minha vida. Cada conquista minha é reflexo do esforço, dos valores e da força que sempre me transmitiram.

Aos meus irmãos, deixo um agradecimento especial por estarem ao meu lado em todos os momentos, compartilhando companheirismo, incentivo e aprendizado. O apoio sincero de vocês e a força que me ofereceram, muitas vezes quando eu nem sabia que precisava, foram fundamentais em toda a minha trajetória.

Agradeço também à minha namorada, pela compreensão, paciência e apoio constante durante esta etapa. Sua presença e incentivo foram essenciais para que eu pudesse seguir firme até a conclusão deste trabalho.

Ao meu orientador, expresso meu sincero agradecimento pelas orientações, pela confiança e por todo o conhecimento compartilhado. Sua contribuição foi indispensável para a execução e o amadurecimento deste projeto.

Por fim, agradeço ao restante da minha família, que sempre torceu por mim e ofereceu suporte incondicional. A todos que contribuíram de alguma forma para esta caminhada, deixo registrado meu profundo reconhecimento e gratidão.

"Dificuldades preparam pessoas comuns para destinos extraordinários."

C. S. Lewis,
Mere Christianity.

Resumo

Carmo, Gabriel Porto do. **Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial**. Goiânia, 2025. 43p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

Este trabalho apresenta o desenvolvimento e a avaliação de uma solução de automação de testes funcionais integrada a mecanismos de inteligência artificial. A proposta utiliza Robot Framework e Selenium WebDriver para realizar a detecção automática de falhas em aplicações web, complementada por um módulo de análise inteligente que interpreta erros durante a execução dos testes por meio da API Gemini. Além disso, a solução integra diretamente com o GitHub Actions e GitHub Issues, permitindo a abertura automática de registros de falhas com evidências detalhadas, como capturas de tela, logs e hipóteses geradas pela IA. A pesquisa tem caráter aplicado e explora como a combinação entre automação, inteligência artificial e práticas de integração contínua pode aprimorar a rastreabilidade, a eficiência e a confiabilidade no processo de garantia da qualidade de software.

Palavras–chave

Automação de Testes; Inteligência Artificial; Robot Framework; Selenium WebDriver; GitHub Actions

Abstract

Carmo, Gabriel Porto do. **Automatic Test Case Generation for Robot Framework: An Artificial Intelligence–Based Approach**. Goiânia, 2025. 43p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

This work presents the development and evaluation of a functional test automation solution integrated with artificial intelligence mechanisms. The proposed approach employs Robot Framework and Selenium WebDriver to perform automatic failure detection in web applications, complemented by an intelligent analysis module that interprets errors during test execution using the Gemini API. Additionally, the solution integrates directly with GitHub Actions and GitHub Issues, enabling the automatic creation of defect reports containing detailed evidence such as screenshots, logs, and AI-generated diagnostic hypotheses. This applied research explores how the combination of test automation, artificial intelligence, and continuous integration practices can enhance traceability, efficiency, and reliability within the software quality assurance process.

Keywords

Test Automation; Artificial Intelligence; Robot Framework; Selenium WebDriver; Continuous Integration

Contents

List of Figures

List of Tables

CHAPTER 1

Introdução

Os testes de software são essenciais para a garantia e validação das funcionalidades de um projeto, desde a concepção inicial até sua versão final. Eles terão como objetivo validar funcionalidades, identificar falhas e assegurar a qualidade do software, evitando que erros cheguem ao usuário.

No âmbito dos testes, existirão diversos tipos e técnicas que poderão ser aplicados de acordo com as funcionalidades e a estrutura do projeto. Os testes poderão ser manuais ou automatizados, abrangendo desde testes unitários, que validarão individualmente uma parte do código, até testes de integração e de sistema, que avaliarão a interação entre módulos e o funcionamento completo do software. Cada abordagem terá seu papel na construção de um processo confiável de desenvolvimento e entrega ao usuário.

Em alguns casos, os testes tornam-se ainda mais indispensáveis, especialmente quando envolvem funcionalidades críticas que podem comprometer completamente o software e causar falhas catastróficas no mundo real. Nesse cenário, a validação contínua da qualidade e da segurança torna-se uma exigência constante. Para isso, os processos e as técnicas de teste devem ser cuidadosamente planejados, priorizando as melhores estratégias para cobrir todas as funcionalidades essenciais do software.

Durante o desenvolvimento e a manutenção do software, mudanças e melhorias serão necessárias, e os testes deverão acompanhar essas alterações. Uma das alternativas para esse monitoramento e aperfeiçoamento será a aplicação de testes automatizados, que surgirão como uma solução eficaz para manter o processo de validação sempre atualizado.

Para que essa estratégia seja realmente eficaz, os testes não poderão ser conduzidos como uma etapa isolada ou final do desenvolvimento. Eles precisarão ser planejados desde o início do projeto e integrados ao fluxo de trabalho da equipe, tornando-se parte natural e contínua do ciclo de vida do software. Esse paradigma é fortemente defendido por metodologias ágeis e pela cultura DevOps, que promoverão práticas como Integração Contínua (CI) e Entrega Contínua (CD), estimulando que

a validação da qualidade ocorra em todas as fases do desenvolvimento, de forma automatizada e iterativa.

Nesse contexto, formula-se a questão que guiará o desenvolvimento deste trabalho: de que maneira a integração entre automação de testes e mecanismos de inteligência artificial poderá aprimorar o processo de análise de falhas, ampliar a rastreabilidade e tornar o ciclo de garantia da qualidade mais eficiente?

Assim, insere-se neste cenário a proposta deste trabalho, que explorará a automação de testes em conjunto com mecanismos modernos de inteligência artificial e pipelines de CI/CD, demonstrando como essas abordagens poderão elevar a confiabilidade, a rastreabilidade e a eficiência do processo de desenvolvimento de software.

1.1 Justificativa

Este trabalho surgiu do interesse pessoal do autor, pela área de testes de software e pela busca por soluções que automatizem estes testes. A validação automatizada exige uma abordagem constante e personalizada. O grande desafio é garantir que todas as funcionalidades do software continuem funcionando corretamente, especialmente após alterações no seu código-fonte, o que é fundamental para garantir a qualidade e a segurança do software.

Com foco principal na melhoria dos testes de software por meio da automação, este trabalho também utiliza Inteligência Artificial (IA) para validar e apoiar a tomada de decisões. Isso não apenas reduz o risco de problemas não detectados, mas também contribui para um ciclo de desenvolvimento mais ágil e seguro.

Além disso, este trabalho também busca fomentar discussões sobre como a automação de testes e o uso de IA podem melhorar a validação de sistemas de software. Ao detectar defeitos, o quanto antes, agiliza o ciclo de desenvolvimento, beneficiando os envolvidos no processo de desenvolvimento de software.

Em resumo, a ideia é o aprimoramento contínuo e a ampliação do conhecimento sobre todos os aspectos relacionados à automação de testes, beneficiando assim a comunidade de profissionais e entusiastas da área.

1.2 Objetivos

1.2.1 Objetivo Geral

Esse estudo tem como objetivo analisar como a automação de teste, combinado com o uso de inteligencia artificial podem melhorar o teste de software, com

foco na validação contínua das funcionalidades do software, promovendo maior qualidade, segurança e eficiência no ciclo de desenvolvimento.

1.2.2 Objetivos Específicos

- Compreender como a automação de testes pode ser aplicada de forma eficaz em todas as partes de um software;
- Analisar como a Inteligência Artificial pode ser utilizada para identificar falhas e auxiliar na tomada de decisões durante os ciclos de testes;
- Apresentar casos práticos onde o uso do *Robot Framework*[1] e do *GitHub Actions*[2] contribui para a detecção de defeitos e a integração contínua do código;
- Avaliar a eficácia da validação automatizada.

1.3 Metodologia

Este trabalho se caracteriza como uma pesquisa aplicada, de natureza qualitativa e exploratória. Sua abordagem metodológica concentra-se na investigação e experimentação prática de técnicas voltadas à detecção e análise de falhas em sistemas web, utilizando ferramentas de automação de testes e plataformas de versionamento de código como elementos centrais do estudo.

Na fase inicial, realizou-se um levantamento teórico sobre conceitos fundamentais de testes de software, automação e qualidade, apoiado em obras clássicas e materiais contemporâneos da área, incluindo Myers (MYERS; SANDLER; BADGETT, 2011), Pressman (PRESSMAN, 2002), McCabe (MCCABE, 1976), bem como artigos, guias técnicos e documentações oficiais relacionadas ao Robot Framework, Selenium WebDriver, integração contínua e técnicas funcionais e estruturais de teste. Esse levantamento teve como finalidade fornecer a base conceitual necessária para orientar o desenvolvimento do experimento.

Em seguida, foram desenvolvidos testes automatizados utilizando Robot Framework e *Selenium WebDriver*. Os primeiros testes foram aplicados em um site com cenários de erro proposital[3]. Esse ambiente foi ideal para validar o funcionamento da estratégia de testes automatizados.

Depois dessa fase inicial, o projeto será implementada em um sistema real para testar sua eficácia em um ambiente de produção. Os testes foram configurados

[1] https://robotframework.org/
[2] https://github.com/features/actions
[3] Disponível em: https://the-internet.herokuapp.com/login. Acesso em: 22 jun. 2025.

para rodar automaticamente sempre que um novo *commit* é feito no repositório do projeto no GitHub, ou em horários definidos, com o uso do *GitHub Actions*. Quando uma falha é detectada, o sistema gera automaticamente uma *issue* no próprio *GitHub*, contendo *prints* da tela, *logs* e uma descrição resumida com a ajuda de uma inteligência artificial que interpreta o erro.

Por fim, os resultados são analisados, observando os tipos de erros mais frequentes, o tempo ate encontrar a falha, e a utilidade das informações geradas para os desenvolvedores.

1.4 Organização do Trabalho

Para facilitar o alcance dos objetivos especificados, além desta Introdução, o restante deste trabalho está organzado da seguinte forma:

- No Capítulo 2 é apresentada a fundamentação teórica, que embasa todo este trabalho.
- No Capítulo 3 é apresentado o projeto propriamente dito...
- No Capítulo 4 são apresentadas as considerações finais e os possíveis desdobramentos futuros deste trabalho.

CHAPTER 2

Fundamentação Teórica

Para esta dissertação, é fundamental apresentar os conceitos essenciais que baseiam o processo de teste de software, destacando sua relevância para assegurar a qualidade em projetos de desenvolvimento. Os testes assumem um papel central na verificação de requisitos e na validação de funcionalidades, permitindo a identificação de falhas e a prevenção de erros que poderiam comprometer tanto a experiência do usuário quanto a segurança do software.

Com base nisso, diferentes técnicas de teste foram desenvolvidas e aprimoradas ao longo do tempo. Entre elas, destacam-se as técnicas funcionais e estruturais, que oferecem visões complementares para a avaliação do software. Enquanto os testes funcionais verificam se o sistema atende às especificações definidas, os estruturais exploram a lógica interna do código, permitindo uma análise mais detalhada do seu comportamento.

Já os níveis de teste representam as diferentes etapas em que o software é avaliado ao longo do desenvolvimento. Eles vão desde a verificação de partes menores do sistema até a análise do produto final, pronto para uso. A ideia é que cada nível contribua para aumentar a confiança no software, começando com verificações mais específicas e chegando a testes mais amplos, que simulam o uso real.

2.1 Técnicas de Teste de Software

Com o avanço nas práticas de desenvolvimento de software, diversas novas abordagens de teste foram surgindo. No entanto, algumas técnicas clássicas, desenvolvidas ainda nas primeiras décadas da computação, permanecem fundamentais até os dias de hoje, e com o mesmo propósito, de revelar a presença de defeitos no software.

As principais técnicas de teste ganharam destaque a partir das publicações de Glenford Myers (MYERS; SANDLER; BADGETT, 2011), que propôs a divisão entre testes funcionais e estruturais. Essas duas abordagens analisam de forma diferente o software. Os testes funcionais verificam se o software atende aos seus

requisitos, enquanto os testes estruturais avaliam o código por dentro. Dessa forma, é possível validar o software de maneira mais completa.

Essa relação demonstra que a aplicação de uma única técnica não é suficiente para garantir a segurança e a qualidade do software. É justamente a combinação entre diferentes abordagens que amplia a cobertura dos testes e fortalece a detecção de falhas. Além disso, a consolidação desses conceitos possibilitou o surgimento de métodos derivados e de ferramentas que são amplamente utilizadas em todo o ecossistema de testes.

2.1.1 Técnicas de Teste Funcional

A técnica de teste funcional ou teste caixa-preta, Figura 2.1, se baseia em verificar o funcionamento do software, sem ter que testar necessariamente o código que foi implementado. Nesse método, são definidos dados de entrada e os resultados esperados para cada situação. O teste é aprovado quando as respostas obtidas correspondem as respostas esperadas.

Da forma que a técnica foi desenvolvida, ela consegue ser aplicada em diferentes níveis do software, desde um método isolado até a aplicação completa. Além disso, esse modelo permite ao testador ter uma perspectiva mais próxima da experiência do usuário final, avaliando o comportamento do software como ele seria percebido na prática.

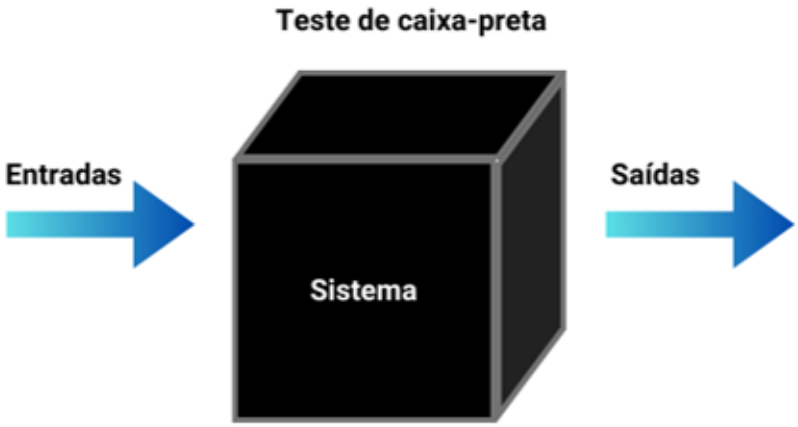

Figure 2.1: *Ilustração do conceito de Teste de Caixa Preta (OLIVEIRA, 2024).*

2.1.2 Técnicas de Teste Estrutural

A técnica de teste estrutural, Figura 2.2, também conhecida como teste de caixa-branca, foca na validação do software a partir da análise direta do código-fonte. Diferentemente da abordagem funcional, esse modelo exige que o testador examine as estruturas internas, a lógica de execução e os métodos implementados, garantindo que todos os caminhos possíveis do código sejam devidamente testados.

Com base nesse código, são elaborados casos de teste que buscam cobrir toda a execução do requisito avaliado.

Os testes de unidade e os testes de integração são exemplos dessa abordagem, sendo amplamente aplicados pelos desenvolvedores ao longo do processo de desenvolvimento. Em um teste de unidade, por exemplo, pode-se verificar se um método responsável por calcular descontos e retorna o valor esperado para diferentes entradas. Já em um teste de integração, a atenção pode estar na interação entre módulos, como a comunicação entre o módulo de autenticação e o módulo de geração de relatórios. Esses testes permitem revelar a presença de defeitos em partes específicas do software e assegurar que cada elemento funcione corretamente em diferentes cenários.

É importante lembrar que a técnica estrutural não substitui a técnica funcional, elas devem se complementar. Enquanto o teste funcional valida o software a partir da visão do usuário, o estrutural garante confiabilidade e segurança do código, contribuindo para uma cobertura de testes mais completa e eficaz.

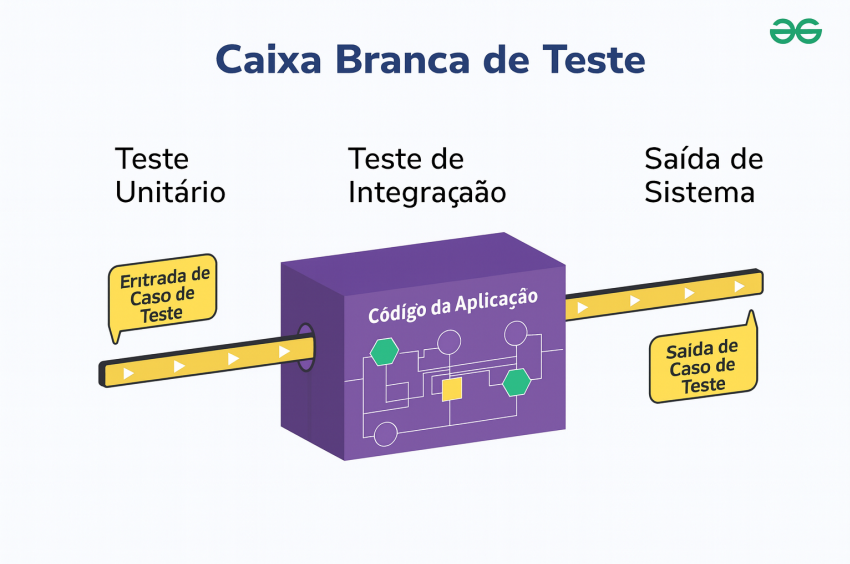

Figure 2.2: *Ilustração do conceito de Teste de Caixa Branca (GEEKSFORGEEKS, 2024).*

2.1.3 Técnicas de Teste Baseada em Defeitos

A técnica baseada em defeitos é uma abordagem que visa identificar vulnerabilidades no software por meio da introdução intencional de falhas. Em vez de se basear apenas em requisitos funcionais, essa técnica busca avaliar como o software se comporta diante de erros, explorando suas fragilidades e limites. Para isso, são utilizadas técnicas como *Mutation Testing*, conforme Figura 2.3, que realiza pequenas alterações no código, simulando erros comuns e *Fuzz Testing*, que fornece entradas aleatórias e inesperadas ao software. O objetivo é verificar se os testes conseguem detectar esses problemas e garantir que o softaware continue operando corretamente mesmo em situações adversas.

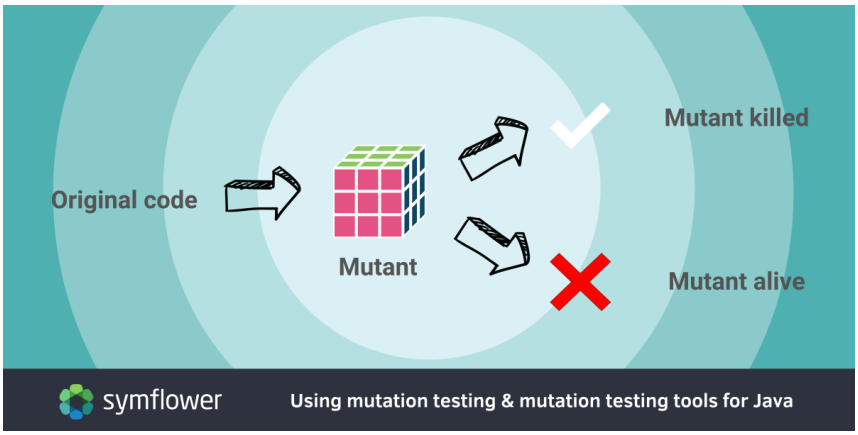

Figure 2.3: *Ilustração do conceito de Teste de Mutação (SYMFLOWER, 2023).*

2.1.4 Critérios de Teste

Os critérios de teste desempenham um papel fundamental no processo de validação de software, pois são responsáveis por definir as regras e diretrizes que orientarão a análise da qualidade. Segundo Myers, Sandler e Badgett (MYERS; SANDLER; BADGETT, 2011), os critérios permitem avaliar se o conjunto de testes elaborado é suficiente para verificar adequadamente o componente ou funcionalidade em questão, aumentando assim a probabilidade de revelar a presença de defeitos no sistema.

Existem diferentes tipos de critérios, que variam conforme a técnica de teste utilizada. Entre os principais estão os critérios funcionais, que se concentram no comportamento externo da aplicação, avaliando as entradas e saídas observáveis, e os critérios estruturais, que consideram a lógica interna do código-fonte. Cada categoria possui técnicas específicas que auxiliam o testador no planejamento, na

seleção e na execução dos casos de teste, garantindo maior rigor e cobertura no processo de verificação.

Critérios de Teste Funcional

Para validar se o software atende aos requisitos, os critérios de testes funcional exercem papel fundamental. Segundo Pressman (PRESSMAN, 2002), o teste funcional procura mostrar que os requisitos funcionais do software são satisfeitos, que a entrada é adequadamente aceita, que a saída esperada é produzida e que a integridade das informações externas é mantida. Essa abordagem permite que o testador simule diferentes cenários de uso real, avaliando se as funcionalidades respondem corretamente de acordo com o que foi especificado.

1. **Particionamento por Equivalência**;
   O particionamento por equivalência, Figura 2.4, é uma técnica que busca simplificar os testes, dividindo os dados de entrada em grupos que devem ser tratados da mesma forma pelo software. A ideia é que, se um valor de um grupo especifico funciona corretamente, os demais também funcionarão, esses grupos são conhecidos como classe de equivalência, e podem conter dados validos e inválidos.

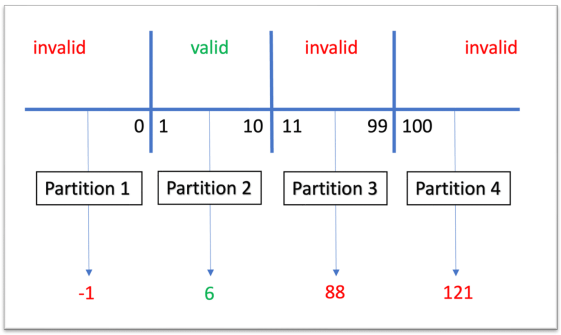

Figure 2.4: *Partição de equivalência. (ACAR, 2024).*

2. **Análise do Valor Limite**;
   A analise do valor limite, Figura 2.5, é uma técnica que se baseia em pontos limites de uma faixa de dados, apresentando um abordagem simples, onde os erros costumam ocorrer nos pontos extremos das entradas. A partir disso, em vez de testar os valores típicos, o testador analisa os valores mais altos e baixos da amostra de dados. Essa técnica pode ser considerada um complemento para o particionamento por equivalência.
3. **Grafo de Causa e Efeito**
   O Grafo de Causa e Efeito, Figura 2.6, é uma técnica funcional utilizada para representar relações lógicas entre condições de entrada (causas) e os compor-

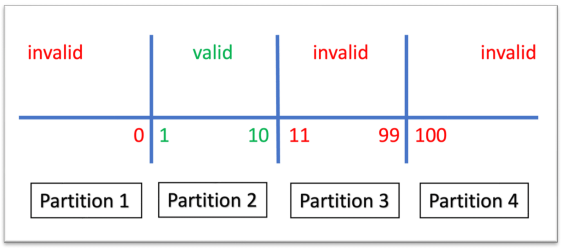

Figure 2.5: *Análise do Valor Limite. (ACAR, 2024).*

tamentos esperados do sistema (efeitos). Segundo Tutorialspoint (POINTS, 2025), essa abordagem transforma requisitos escritos em um modelo gráfico que evidencia como diferentes combinações de causas podem ativar ou inibir determinados efeitos, permitindo uma compreensão clara de regras de negócio complexas.

Cada causa representa uma condição ou situação de entrada, enquanto cada efeito corresponde a uma ação, resposta ou resultado esperado do sistema. As ligações entre causas e efeitos são definidas por operadores lógicos, como *AND*, *OR* e *NOT*, o que possibilita estruturar cenários em que múltiplas condições precisam ocorrer simultaneamente para que um determinado comportamento seja produzido.

Após a construção do grafo, o modelo é convertido em uma tabela de decisão, procedimento também destacado por Tutorialspoint (POINTS, 2025). Essa etapa garante que todas as combinações relevantes de causas sejam analisadas e que os casos de teste derivados cubram integralmente as regras descritas. Dessa forma, a técnica auxilia na identificação de inconsistências, omissões ou ambiguidades nos requisitos, aumentando a precisão e abrangência dos testes funcionais.

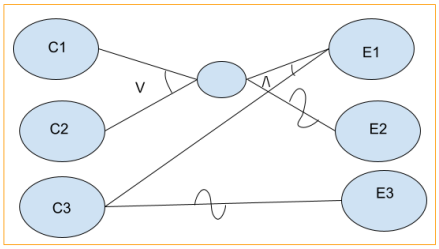

Figure 2.6: *Grafo de Causa e Efeito. (POINTS, 2025)*

4. **Tabela de Decisão**.

   A Tabela de decisão, Figura 2.7, é um técnica conhecida por testar softwares que envolvem varias condições lógicas e regras de negocio mais complexas. Ela ajuda a organizar e visualizar de maneira clara como o software deve se comportar com diferentes combinação de entradas.

A técnica consiste em apresentar as ações e condições de entrada em um tabela, onde cada linha corresponde a um regra especifica, e as condições são avaliadas como verdadeiras ou falsas. Essa estrutura é bastante útil em cenários com diversas variações de comportamento, pois auxilia na identificação de todos os casos relevantes a serem testados.

Conditions	**R1**	**R2**	**R3**
Withdrawal Amount <= Balance	T	F	F
Credit granted	-	T	F
Actions			
Withdrawal granted	T	T	F

Figure 2.7: *Tabela de Decisão (ALBUQUERQUE; FERNANDES, 2017)*

Critérios de Teste Estrutural

Para uma análise mais precisa do código-fonte, são aplicados os critérios de teste estrutural. Eles são utilizados quando o objetivo é verificar o funcionamento interno do software, analisando diretamente a lógica implementada, como condições, fluxos e estruturas de decisão presentes no código.

Alguns exemplos comuns desses critérios, referencia são:

- **Cobertura de instruções** , que garante que cada linha de código seja executada ao menos uma vez. Por exemplo, em um método que soma dois números, o critério assegura que a linha de retorno seja testada em diferentes situações.

- **Cobertura de caminhos** , que busca percorrer todos os caminhos possíveis de execução do código. Esse critério é útil em sistemas com múltiplas regras de negócio encadeadas.
- **Cobertura de decisões** , que verifica se todas as ramificações de um comando condicional, como if ou switch, foram testadas, e se os resultados estão de acordo com o esperado.

Para viabilizar a aplicação desses critérios, utiliza-se o Grafo de Fluxo de Controle (GFC), Figura 2.8, constitui uma representação gráfica que ilustra o fluxo de execução de programas ou aplicações, permitindo a análise detalhada do comportamento interno do software. Desenvolvido originalmente por Frances E. Allen, esse grafo é direcionado e orientado a processos, no qual os nós representam blocos básicos de instruções e as arestas indicam os possíveis caminhos de fluxo de

controle. Essa estrutura facilita a compreensão, otimização e verificação da lógica de programas, sendo essencial para o desenvolvimento de software mais seguro e eficiente.

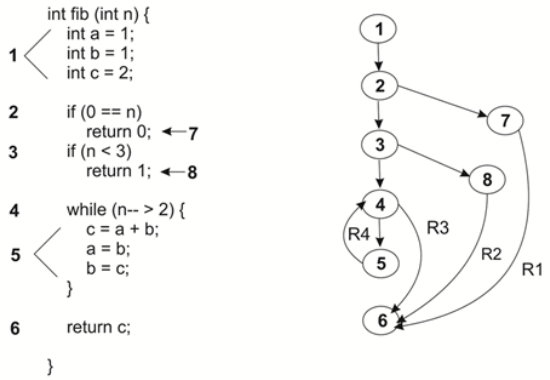

Figure 2.8: *Grafo de Fluxo de Controle. (DEVMEDIA, 2014)*

Entre as métricas derivadas do Grafo de Fluxo de Controle, a Complexidade Ciclomática destaca-se como uma medida fundamental na Engenharia de Software. Proposta por Thomas McCabe em 1976, essa métrica quantifica o número de caminhos linearmente independentes existentes no código, permitindo estimar seu nível de complexidade lógica (MCCABE, 1976). A análise baseia-se na contagem de estruturas condicionais, desvios e loops, que influenciam diretamente o esforço necessário para testar e manter o software.

Quanto maior o valor da complexidade ciclomática, maior a quantidade de ramificações lógicas (como estruturas condicionais e loops), o que tende a aumentar a dificuldade de compreensão, teste e manutenção do software. Por outro lado, valores mais baixos indicam código mais simples, estruturado e confiável, reduzindo a probabilidade de defeitos e facilitando a evolução do sistema.

Considerando essas características, os testes estruturais são geralmente aplicados nos níveis de teste unitário e de integração. Sua implementação requer conhecimento detalhado do código-fonte e o uso de ferramentas específicas, sendo portanto de responsabilidade primária do desenvolvedor. Essa abordagem assegura que a verificação da qualidade interna do software ocorra desde as etapas iniciais de desenvolvimento, contribuindo significativamente para a construção de sistemas mais robustos e confiáveis.

2.1.5 Níveis de Teste

A descrição dos quatro níveis de teste de software (unitário, integração, sistema e aceitação), conforme consta da Figura 2.9, foi baseada no artigo publicado por (DALL'OCA, 2015).

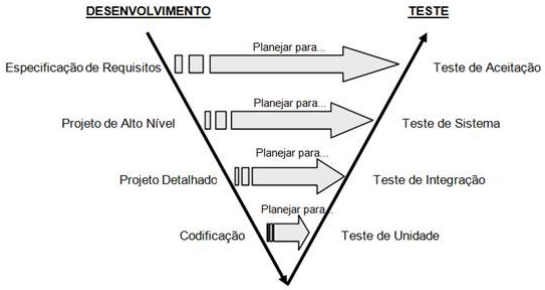

Figure 2.9: *Modelo V de desenvolvimento de software (DALL'OCA, 2015).*

Teste Unitário

O testes unitários representam o nível básicos no processo de software, eles são responsáveis por verificar pequenos trechos de código, como métodos, funções e módulos isolados. A principal função é encontrar erros lógicos ou de implementação ainda nas etapas iniciais do desenvolvimento

Teste de Integração

Após a validação individual dos módulos, o teste de integração busca encontrar falhas que podem surgir quando esses módulos são integrados. Essa etapa é essencial para garantir que a estrutura do projeto esteja funcional e bem acoplada.

Teste de Sistema

No teste de sistema, o software como um todo é validado, já em sua versão integrada. Os testes são executados em condições semelhantes às do ambiente real de uso, simulando o comportamento de um usuário final e validando os requisitos de forma geral.

Teste de Aceitação

Os testes de aceitação são realizados por um grupo restrito de usuários finais, buscando confirma se o software esta pronto para ser colocado em produção. Esses devem simular tarefas e rotinas que serão realizadas quando o projeto estiver entregue

2.1.6 Tipos de Teste

Teste de Performance

O teste de performance tem como objetivo avaliar a eficiência de um software em termos de tempo de resposta, uso de recursos e estabilidade sob diferentes

condições de operação. Esse tipo de teste é fundamental para assegurar que o sistema atenda a requisitos de qualidade relacionados a rapidez, confiabilidade e escalabilidade, fatores determinantes para a experiência do usuário e para a competitividade da aplicação no mercado.

Por meio do teste de performance, é possível identificar gargalos que impactam diretamente o funcionamento do sistema, como consultas lentas ao banco de dados, consumo excessivo de memória ou problemas de rede. Além disso, esses testes permitem definir parâmetros de aceitação, tais como o tempo máximo aceitável de resposta, a quantidade de transações suportadas por segundo e a eficiência na utilização de recursos do servidor.

Teste de Carga

O teste de carga consiste em simular a demanda real de usuários em um software, com o objetivo de analisar seu comportamento sob diferentes condições de tráfego, desde situações de uso leve até picos de acesso intenso. Esse método é geralmente aplicado nas fases finais do ciclo de desenvolvimento, permitindo avaliar a robustez e a estabilidade do sistema perante cenários operacionais previsíveis. Por meio dessa técnica, é possível assegurar que a aplicação suportará o volume esperado de usuários, além de identificar e corrigir possíveis problemas de desempenho antes que o software seja disponibilizado para o público geral

Teste de Estresse

O teste de estresse tem como objetivo avaliar até onde um sistema consegue suportar situações extremas, indo além das condições normais de uso. Ele é essencial porque não basta que um software funcione bem em cenários comuns, é preciso garantir também que ele se mantenha estável quando submetido a sobrecargas ou pressões inesperadas. Esse tipo de teste mostra o quão robusto e confiável o sistema realmente é, revelando se ele resiste sem travar ou falhar quando colocado em seu limite.

Teste de Regressão

O teste de regressão é uma técnica essencial no desenvolvimento de software, aplicada sempre que uma nova versão do sistema é lançada ou quando ele passa por ciclos de evolução contínua. O objetivo principal dessa abordagem é assegurar que as modificações implementadas não introduzam falhas em funcionalidades que já estavam sólidas.

Para isso, todos os testes que foram previamente executados nas versões ou ciclos anteriores são reaplicados à nova versão em avaliação. Além disso, leva-se em consideração as fases e técnicas de teste mais adequadas, de acordo com o impacto causado pelas alterações recentes.

2.2 Ferramentas de Teste

À medida que sistemas se tornam mais complexos e exigem maior rapidez em sua entrega, o uso de ferramentas de teste passa a ser essencial para apoiar equipes na criação, execução e gestão das atividades de verificação e validação.

Essas ferramentas oferecem recursos que vão desde a elaboração de casos de teste até a automação da execução e o acompanhamento dos resultados, possibilitando maior controle sobre o processo e reduzindo falhas que poderiam comprometer a experiência do usuário final. Além disso, permitem padronizar práticas, otimizar o tempo das equipes e fornecer métricas importantes para a tomada de decisão.

Outro aspecto importante é a capacidade dessas ferramentas de se integrarem a diferentes fases do ciclo de desenvolvimento, especialmente em metodologias ágeis e em práticas de integração e entrega contínuas (CI/CD). Isso possibilita que a verificação de qualidade seja incorporada de forma contínua e incremental, acompanhando a evolução do software e prevenindo problemas em estágios mais avançados do projeto.

2.3 Teste de Software e Inteligência Artificial

A inteligência artificial está revolucionando a área de testes de software, trazendo um novo patamar de eficiência para processos que antes dependiam quase que exclusivamente do trabalho manual e demandavam tempo significativo. O uso de técnicas de aprendizado de máquina, mineração de dados e análise preditiva permite acelerar atividades fundamentais como a geração de casos de teste, a priorização de cenários e a detecção de falhas, e ate mesmo propor soluções para os erros encontrados .

Entre os benefícios mais relevantes está a capacidade da IA em automatizar a criação de casos de teste, a partir da análise de requisitos, históricos de execução e até do comportamento dos usuários em produção. Isso amplia a cobertura dos testes e reduz a possibilidade de lacunas que poderiam comprometer a confiabilidade do sistema. Além disso, algoritmos inteligentes podem identificar padrões de falhas recorrentes e priorizar a execução dos testes mais críticos, direcionando os recursos para as áreas de maior risco.

Outro ponto de destaque é o suporte da IA na gestão de resultados de teste. Em projetos de grande porte, a quantidade de dados gerados pode dificultar a análise manual. Nesse cenário, sistemas baseados em IA auxiliam na categorização automática de falhas, na identificação de falsos positivos e na sugestão de correções mais prováveis, agilizando a tomada de decisão das equipes de desenvolvimento e qualidade.

CHAPTER 3

Projeto de Teste Automatizado

O projeto em questão, visa demonstrar a aplicação pratica do *Robot Framework*[1] na automação de testes funcionais em uma aplicação *web* real. A base do trabalho é avaliar como a utilização de um *framework* de automação baseado em *keywords* pode contribuir para a melhoria do processo de garantia da qualidade de software, ao proporcionar maior organização, reutilização de componentes e geração automática de evidências de execução.

3.1 Descrição do Projeto

Para a execução dos testes de interface, o projeto faz uso da *SeleniumLibrary*[2], que integra o *Robot Framework* ao *Selenium WebDriver*[3]. Essa combinação permite simular o comportamento real de um usuário no navegador. O *Selenium* é um dos *frameworks* mais consolidados para automação de navegadores, oferecendo suporte a diferentes navegadores e sistemas operacionais, o que garante flexibilidade e maior realismo nos cenários de teste.

Além do foco na automação de testes, o trabalho busca através da integração com a inteligencia artificial (IA) com a *api* do *gemini*[4], gerar um ambiente muito eficiente onde as falhas encontradas durante a execução dos testes não apenas são detectadas, mas também analisadas automaticamente. A IA sugere hipóteses sobre as possíveis causas e até caminhos para uma depuração mais rápida, servindo como um apoio extra para as equipes de *quality assurance* (QA) e de desenvolvimento.

Outro ponto importante é a conexão com o *GitHub Issues*[5]. Sempre que um defeito é identificado e diagnosticado, ele pode ser registrado automaticamente em um repositório de versionamento. Isso torna o fluxo de trabalho mais ágil e

[1] <https://robotframework.org/>
[2] <SeleniumLibrary>
[3] <https://www.selenium.dev/documentation/webdriver/>
[4] <https://ai.google.dev/gemini-api/docs>
[5] <https://github.com/features/issues>

colaborativo, reduzindo atividades manuais e garantindo a rastreabilidade entre o defeito, sua análise e o acompanhamento da correção.

Sendo assim, o objetivo deste estudo de caso vai além de apenas validar a execução de casos de teste. Ele busca mostrar o potencial da automação inteligente, unindo o *Robot Framework*, o *Selenium* e recursos de IA para apoiar diagnósticos e tornar o desenvolvimento de software ainda mais orientado à qualidade.

3.2 Contexto da Aplicação de Testes

Os testes automatizados desenvolvidos neste projeto foram aplicados sobre uma aplicação web pública e amplamente utilizada em treinamentos de automação: o portal *The Internet Herokuapp*[6]. Essa plataforma foi escolhida por oferecer diversos cenários controlados de teste, como formulários de login, *upload* de arquivos, botões dinâmicos e elementos interativos, que permitem validar diferentes aspectos do comportamento de uma aplicação *web*.

Durante a execução, o *Robot Framework*, em conjunto com a *SeleniumLibrary*, foi responsável por controlar o navegador e reproduzir as ações do usuário, como preencher campos, clicar em botões e validar mensagens de erro. Esse processo possibilitou avaliar a estabilidade da aplicação e a capacidade do *framework* em detectar e registrar falhas de interface de forma confiável.

Além disso, o contexto do teste foi ampliado com a integração de ferramentas complementares. A *API Gemini* foi utilizada para interpretar automaticamente as falhas registradas, fornecendo análises e hipóteses sobre suas possíveis causas, enquanto a integração com o *GitHub Issues* garantiu que cada erro detectado fosse documentado diretamente em um repositório de versionamento, permitindo o acompanhamento e rastreabilidade do ciclo de correção.

Dessa forma, o contexto da aplicação de testes reflete um ambiente que simula situações reais de uso e falhas comuns em sistemas web, demonstrando como a automação aliada à inteligência artificial pode contribuir para tornar o processo de validação mais inteligente, ágil e integrado ao fluxo de desenvolvimento de software.

3.3 Arquitetura e Estrutura do Projeto

A arquitetura do projeto foi concebida para integrar, de forma contínua e automatizada, diferentes componentes responsáveis pela execução dos testes funcionais, análise inteligente de falhas e abertura de *issues* no repositório de desenvolvimento.

[6] https://the-internet.herokuapp.com

Essa integração ocorre dentro de um fluxo de *CI/CD* (Integração Contínua/Entrega Contínua), garantindo que cada nova alteração no código seja validada, analisada e rastreada.

A Figura 3.1 apresenta uma visão geral do processo, modelado em notação *BPMN* (Business Process Model and Notation). Esse diagrama ilustra o ciclo completo de automação, desde o disparo dos testes até a abertura automática de uma *issue* em caso de falha.

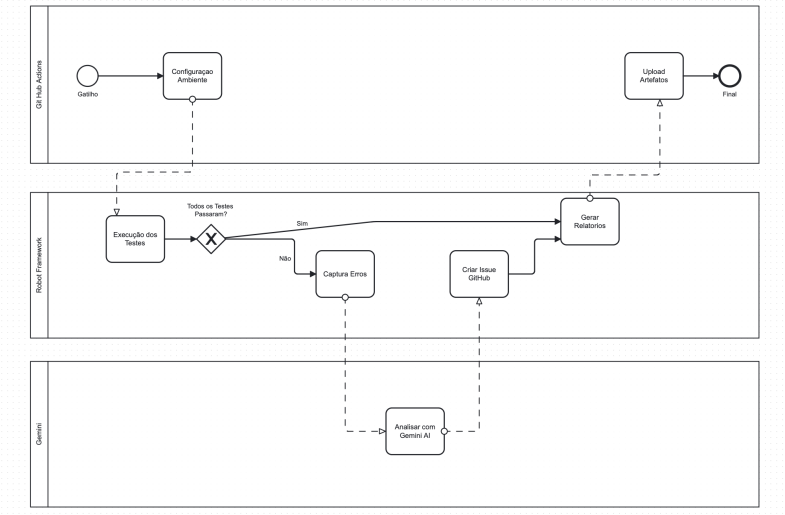

Figure 3.1: *Fluxo BPMN da automação proposta*

O fluxo inicia-se quando o repositório recebe um novo *commit* do usuário, acionando automaticamente a rotina de testes. Além disso, a pipeline também é executada em horários pré-determinados, permitindo verificações periódicas mesmo sem alterações recentes no código. A suíte de testes, implementada em *Robot Framework*, utiliza a biblioteca *SeleniumLibrary* para interação com a interface web, realizando validações funcionais e simulando a rotina de uso do sistema.

Em caso de falha, as evidências são capturadas automaticamente (capturas de tela, logs e relatórios detalhados). Esse conjunto de informações é então enviado à *API Gemini*, que analisa o erro, identifica possíveis causas e sugere ações de depuração. Com base nesse diagnóstico, é criada automaticamente uma *issue* no GitHub, contendo título, descrição, evidências e o parecer gerado pela IA.

A estrutura modular do projeto, separada em arquivos de teste e arquivos de palavras-chave (*resource files*), permite organização clara e reutilização de comandos. Esse padrão facilita a manutenção, amplia a escalabilidade da suíte de testes e torna o fluxo automatizado mais robusto e confiável.

Em síntese, a arquitetura proposta combina ferramentas de automação, inteligência artificial e versionamento de forma integrada, garantindo rastreabilidade completa entre código, testes, falhas, análises e correções.

3.4 Tecnologias Utilizadas

O desenvolvimento deste projeto envolveu o uso de diversas tecnologias que, em conjunto, permitiram construir um ambiente de automação robusto, integrado e inteligente. Cada uma delas desempenha um papel específico na execução dos testes e na análise automatizada dos resultados.

3.4.1 *Robot Framework*

O **Robot Framework** é a base do projeto, sendo um *framework* de automação de testes de código aberto amplamente utilizado para testes funcionais, de aceitação e de integração. Seu principal diferencial é o uso de uma linguagem baseada em *keywords*, que torna a escrita dos testes mais legível e acessível, mesmo para profissionais que não possuem conhecimento avançado em programação. Além disso, sua arquitetura modular permite a integração com diferentes bibliotecas e ferramentas externas, o que facilita a expansão das funcionalidades.

3.4.2 *Selenium WebDriver* e *SeleniumLibrary*

A automação da interface foi implementada por meio da ***SeleniumLibrary***, que faz a ponte entre o *Robot Framework* e o ***Selenium WebDriver***. Essa integração permite controlar o navegador da mesma forma que um usuário real, simulando ações como cliques, preenchimento de campos e navegação entre páginas. O *Selenium* é uma das tecnologias mais consolidadas para automação de testes em aplicações *web*, oferecendo suporte a múltiplos navegadores e sistemas operacionais, o que garante flexibilidade e portabilidade aos testes.

3.4.3 API Gemini

A integração com a **API Gemini**, da Google, representa o elemento de IA do projeto. Sempre que um defeito é detectado durante a execução dos testes, a IA é acionada para interpretar o contexto do defeito e gerar uma análise automática, sugerindo causas técnicas prováveis e ações de depuração. Essa abordagem agrega um nível de automação cognitiva ao processo, reduzindo o tempo de diagnóstico e apoiando as equipes de QA e desenvolvimento na identificação de problemas.

3.4.4 GitHub e GitHub Issues

O **GitHub** foi utilizado tanto como repositório para controle de versão do código quanto como ferramenta de rastreabilidade de falhas. A integração direta com

o **GitHub Issues** permite que cada defeito identificado seja automaticamente registrado como uma ocorrência documentada, contendo detalhes do teste, a mensagem de erro e a análise gerada pela IA. Esse processo automatizado facilita o acompanhamento das correções e promove maior colaboração entre os times envolvidos.

Dessa forma, a combinação dessas tecnologias possibilitou a criação de um ecossistema de automação completo, no qual a execução de testes, a detecção de falhas, a análise inteligente e o registro de ocorrências estão integrados em um único fluxo automatizado e eficiente.

3.5 Fluxo de Execução dos Testes com IA

O fluxo de execução dos testes automatizados foi projetado para demonstrar como a integração entre o *Robot Framework*, a IA (API Gemini) e o *GitHub Issues* pode criar um ciclo de validação e análise contínua, tornando o processo de garantia da qualidade mais inteligente e autônomo.

3.5.1 Etapa 1: Inicialização do Teste

A execução se inicia no *Robot Framework*, que orquestra todos os componentes do processo. O ambiente de testes é configurado com as bibliotecas necessárias, como *SeleniumLibrary* (para controle do navegador), *RequestsLibrary* (para comunicação HTTP) e *AllureLibrary* (para geração de relatórios visuais). Em seguida, o *framework* abre o navegador definido e acessa a aplicação alvo — neste caso, o portal *The Internet Herokuapp* — para iniciar os cenários de teste funcionais.

3.5.2 Etapa 2: Execução das Ações e Validação

Durante os testes, o *Selenium* simula as interações de um usuário real, realizando ações como preencher formulários, submeter dados e validar respostas da interface. Caso a aplicação se comporte conforme o esperado, o teste é marcado como bem-sucedido e registrado no relatório de execução. No entanto, se uma inconsistência for identificada (como um erro de autenticação ou falha de carregamento), o *framework* aciona automaticamente o módulo de análise com IA.

3.5.3 Etapa 3: Análise Automática da Falha com IA

Quando uma falha é detectada durante a execução da suíte de testes, o erro correspondente é capturado e enviado à *API Gemini*, da Google. A execução desses testes ocorre de duas formas: automaticamente a cada novo *commit* realizado pelo

usuário no repositório e, adicionalmente, em horários pré-determinados definidos na rotina de integração contínua. Dessa forma, cada falha registrada pode ser associada tanto a uma alteração recente no código quanto a verificações periódicas do estado geral do sistema.

A *API Gemini* recebe como entrada o contexto completo do evento, incluindo a mensagem de erro retornada, o nome do teste afetado, o *commit* em execução (quando aplicável) e o autor responsável pela modificação, e gera uma resposta textual contendo:

- três possíveis causas técnicas para a falha;
- duas sugestões de ações de depuração rápida.

Essa análise automática permite identificar a origem do problema de forma imediata, reduzindo significativamente o tempo necessário para diagnóstico manual e fornecendo insights técnicos diretamente para a equipe de desenvolvimento. Além disso, a combinação entre disparo por *commit* e execuções agendadas aumenta a cobertura, a rastreabilidade e a confiabilidade do processo de validação contínua.

3.5.4 Etapa 4: Registro da Falha no GitHub Issues

Após o retorno da análise da IA, o sistema utiliza a **API do GitHub** para registrar automaticamente uma *issue* no repositório do projeto. Este registro contém:

- título descritivo da falha;
- mensagem de erro capturada;
- diagnóstico gerado pela IA (causas e ações recomendadas);
- informações sobre o commit e o responsável pela execução.

Essa integração garante rastreabilidade entre o erro ocorrido, sua análise e o acompanhamento da correção, permitindo que o fluxo de trabalho entre QA e desenvolvimento se torne mais colaborativo e eficiente.

3.5.5 Etapa 5: Relatórios e Evidências

Por fim, os resultados completos são consolidados pelo **Allure Report**, que gera relatórios visuais contendo o status de cada teste, as evidências de execução (como capturas de tela) e o histórico das execuções anteriores. Dessa forma, é possível acompanhar a evolução da qualidade da aplicação ao longo do tempo.

Em síntese, o fluxo de execução dos testes combina automação funcional, análise inteligente e rastreabilidade contínua, demonstrando como a união entre **Robot Framework**, **Selenium**, **IA Gemini** e **GitHub Issues** pode criar um ecossistema de validação moderno, ágil e orientado à qualidade.

CHAPTER 4

Considerações Finais

Este capítulo apresenta os resultados obtidos durante a execução do estudo de caso, destacando o impacto real do uso do Robot Framework integrado à Inteligência Artificial na automação de testes funcionais. Mais do que validar apenas a execução de suites automatizadas, o objetivo deste trabalho foi compreender se a IA pode agir como um agente complementar de análise, diagnóstico e apoio ao fluxo de QA, reduzindo esforço manual, acelerando investigação de falhas e tornando o ciclo de desenvolvimento mais orientado à evidência

Ao longo dos testes, avaliamos alguns pontos importantes: a capacidade de detectar erros, a qualidade das análises feitas pela IA, a consistência do rastreamento automático no GitHub Issues e o quanto a ferramenta agregou de valor, seja pelo reaproveitamento de componentes, seja pela facilidade de manter os testes.

Com isso, os resultados que serão apresentados aqui mostram não só que o modelo proposto é tecnicamente viável, mas também que ele tem aplicação real, agrega relevância ao processo de garantia de qualidade e tem potencial para evoluir em direção a um contexto de automação inteligente de testes.

4.1 Resultados Obtidos nos Testes

A suíte de automação desenvolvida contemplou tanto os cenários de autenticação quanto funcionalidades adicionais disponibilizadas pelo ambiente de testes *The Internet*, abrangendo componentes como *checkboxes*, *dropdown menus*, upload de arquivos, carregamento dinâmico, detecção de imagens quebradas e identificação de erros ortográficos. Ao todo, foram executados X casos de teste, cobrindo tanto cenários positivos quanto negativos e comportamentos inesperados.

4.1.1 Volume total de casos executados

A suíte contemplou 8 testes automatizados, distribuídos em:

- Testes de login (fluxos válidos e inválidos);

- Testes de validação funcional (checkboxes, dropdown, dynamic loading);
- Testes de robustez (upload, typos, imagens quebradas).

Esse conjunto permitiu validar tanto o funcionamento essencial dos componentes quanto cenários extremos ou propositalmente defeituosos, disponíveis no ambiente de testes.

4.1.2 Execução com sucesso e falhas detectadas

A execução completa gerou:

- 5 testes bem-sucedidos;
- 3 testes com falhas detectadas automaticamente.

Algumas falhas identificadas não foram geradas pela automação, mas representam erros reais do ambiente, como:

- imagens quebradas na rota `/broken_images`;
- variações textuais e erros ortográficos em `/typos`;
- possíveis perdas de estado no carregamento dinâmico.

4.1.3 Tempo médio e ganho proporcionado pela IA

A suíte apresentou um tempo médio de execução de aproximadamente 1 minuto, com tempo individual de execução variando de acordo com a complexidade de cada cenário. A integração com a IA reduziu significativamente o esforço de análise das falhas, uma vez que o modelo Gemini foi capaz de gerar diagnósticos técnicos em poucos segundos.

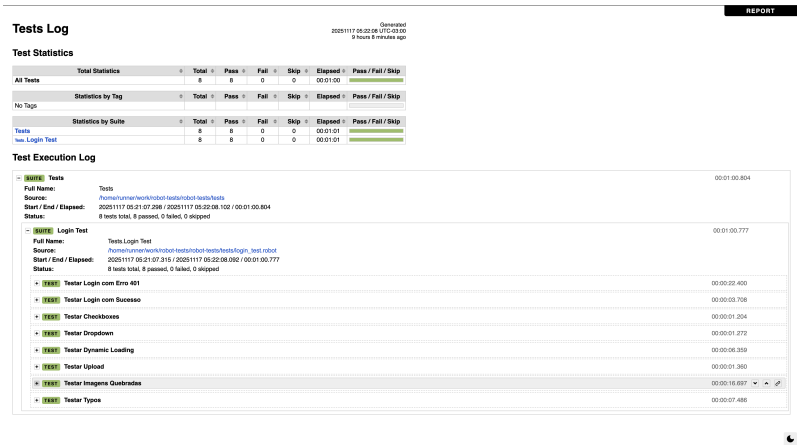

Figure 4.1: *Relatório gerado automaticamente pelo Robot Framework contendo o resumo da execução dos testes.*

4.1.4 Evidências de execução

Durante as execuções, foram coletadas evidências como:

- capturas de tela automáticas nos pontos de falha;
- logs detalhados gerados pelo Robot Framework;
- relatórios HTML (`log.html` e `report.html`);
- registros das análises feitas pela IA;
- issues geradas automaticamente no GitHub.

4.2 Análise da Contribuição da IA no Diagnóstico das Falhas

A integração da Inteligência Artificial no processo de automação trouxe benefícios relevantes para a triagem e diagnóstico de falhas, reduzindo o tempo de análise humana e evitando retrabalho. O modelo Gemini se mostrou eficiente ao interpretar erros, propor causas possíveis e sugerir ações corretivas com base na execução do teste.

4.2.1 Impacto real na identificação das falhas

A IA foi capaz de analisar mensagens de erro capturadas pela automação, correlacionar dados do ambiente de execução e gerar diagnósticos consistentes. Isso permitiu acelerar significativamente o processo de detecção de falhas e gerar documentação clara para suporte e desenvolvimento.

4.2.2 Exemplos reais de falhas analisadas pela IA

Entre os exemplos observados, destacam-se:

- imagens quebradas, onde a IA sugeriu revisão dos caminhos de recursos estáticos;
- erro ortográfico na página `/typos`, interpretado corretamente como parte do comportamento do ambiente de testes;
- inconsistência no carregamento do *dynamic loading*, onde a IA recomendou revisão de timeout e inspeção do DOM;

4.2.3 Comparação com abordagem manual

Antes da integração da IA, cada falha exigia análise manual, como inspeção de logs, reprodução do teste e abertura manual de issues. Com a IA, esse processo tornou-se automático, reduzindo esforço humano e tempo de resposta.

Dessa forma, observou-se uma redução significativa no ciclo de detecção–documentação–correção, aumentando a eficiência do fluxo entre QA e desenvolvimento.

4.2.4 Limitações observadas

Apesar da eficácia, algumas limitações foram identificadas:

- dependência da clareza da mensagem de erro extraída dos testes;
- diagnósticos menos precisos em falhas muito técnicas ou contextos complexos;
- necessidade de interpretação humana final antes de decisões críticas.

4.3 Integração com GitHub Issues e Rastreabilidade

A integração direta entre a automação, a IA e o GitHub permitiu que cada falha identificada fosse transformada automaticamente em uma issue. Este fluxo garantiu maior rastreabilidade, melhor organização do backlog e comunicação mais eficiente entre testers e desenvolvedores.

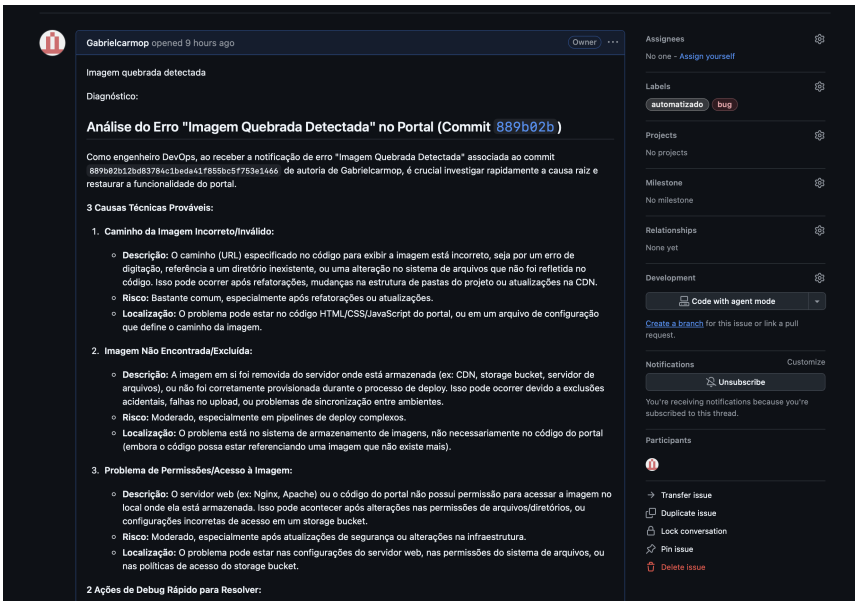

Figure 4.2: *Issue criada automaticamente no GitHub contendo o diagnóstico gerado pela API Gemini.*

4.3.1 Agilidade no fluxo de QA e Desenvolvimento

O processo reduziu a latência entre a ocorrência do erro e sua formalização, uma vez que:

- a detecção da falha ocorre no teste;
- a IA analisa a falha imediatamente;
- a issue é criada automaticamente com título, corpo, labels e diagnóstico técnico.

4.3.2 Número de issues geradas

Durante as execuções da suíte, foram geradas:

- 3 issues referentes a erros reais;
- 1 issues provenientes de testes negativos intencionais.

Isso evidenciou a capacidade da suíte em fornecer documentação automatizada para cada falha detectada.

4.3.3 Benefícios diretos para o time

Entre os benefícios concretos, destacam-se:

- maior rastreabilidade entre falha, evidência, diagnóstico e correção;
- histórico organizado para auditoria e melhoria contínua;
- eliminação do trabalho manual de documentação;
- agilidade no fluxo de manutenção.

4.4 Discussão sobre o Uso do Robot Framework para Automação Funcional

O Robot Framework demonstrou ser uma ferramenta adequada para automação funcional devido à sua sintaxe simples, modularidade e grande ecossistema de bibliotecas.

4.4.1 Pontos fortes

Entre as características positivas, destacam-se:

- sintaxe altamente legível;
- separação clara entre lógica, keywords e testes;

- suporte nativo a logs e relatórios automáticos;
- integração fácil com Selenium, Requests e APIs externas;
- compatibilidade com CI/CD.

4.4.2 Pontos fracos

Algumas limitações também foram identificadas:

- forte dependência de espaçamento e formatação;
- debugging limitado em comparação com linguagens tradicionais;
- dependência da estabilidade do Selenium WebDriver;
- necessidade de maior controle em execuções headless.

4.4.3 Benefícios reais para times e escala

O uso do Robot Framework possibilitou:

- aumento da produtividade no desenvolvimento de testes;
- redução da curva de aprendizado para novos integrantes do time;
- manutenção facilitada pela modularidade das keywords;
- escalabilidade do conjunto de testes;
- integração fluida com processos de CI/CD e ferramentas de rastreabilidade.

Referências

ACAR, G. *Unleashing the Power of Equivalence Partitioning and Boundary Value Analysis in Software Testing*. 2024. Disponível em: <https://www.commencis.com/thoughts/unleashing-the-power-of-equivalence-partitioning-and-boundary-value-analysis-in-software-testing/>. Acesso em 23/11/2025.

ALBUQUERQUE, V. A.; FERNANDES, B. J. T. *Fall Detection: Sensor Data for Classification*. 2017. Disponível em: <https://www.ecomp.poli.br/ListaTCC/20171/vinicius-albuquerque-fall-detection-final.pdf>. Acesso em 23/11/2025.

DALL'OCA, T. *Níveis de teste, Modelo V de desenvolvimento de software e Testes de software*. 2015. Disponível em <https://taisdalloca.blogspot.com/2015/10/niveis-de-teste-modelo-v-de.html>. Acesso em 23/11/2025.

DEVMEDIA. *Teste unitário com JUnit e ComplexGraph*. 2014. Disponível em <https://www.devmedia.com.br/teste-unitario-com-junit-e-complexgraph/31382>. Acesso em 23/11/2025.

GEEKSFORGEEKS. *White Box Testing — Illustration*. 2024. Disponível em: <https://www.geeksforgeeks.org/software-testing/software-engineering-white-box-testing/>. Acesso em: 20/02/2025.

MCCABE, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, 1976.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962.

OLIVEIRA, A. N. *Testes de Caixa Preta*. 2024. Disponível em: <https://www.linkedin.com/pulse/testes-de-caixa-preta-anselmo-n-de-oliveira-ygdgf/>. Acesso em: 20/02/2025.

POINTS, T. *Software Testing - Cause-Effect Graph*. [S.l.]: Disponível em: <https://www.tutorialspoint.com/software_testing_dictionary/cause_effect_graph.htm>. Acesso em 23/11/2025., 2025.

PRESSMAN, R. S. *Engenharia de Software*. 5. ed. Rio de Janeiro: McGraw Hill, 2002.

SYMFLOWER. *How to test your tests? A guide to mutation testing & mutation testing tools*. 2023. Disponível em: <https://symflower.com/en/company/blog/2023/using-mutation-testing/>. Acesso em: 20/02/2025.