Fabricio Augusto Miranda dos Santos

TGen - Testes automatizados utilizando inteligência artificial generativa

Ouro Branco, 08 de julho de 2026

Fabricio Augusto Miranda dos Santos

TGen - Testes automatizados utilizando inteligência artificial generativa

Projeto de Pesquisa apresentado ao curso de Sistemas de Informação do Instituto Federal de Minas Gerais – Campus Ouro Branco, como requisito parcial para aprovação na Disciplina Trabalho de Conclusão de Curso I.

Orientador: Prof. Jânio Rosa

Ouro Branco, 08 de julho de 2026

S237t Santos, Fabrício Augusto Miranda dos.

TGen - testes automatizados utilizando inteligência artificial generativa. / Fabrício Augusto Miranda dos Santos. – 2026.

16 f. il.col.

Orientador: Jânio Rosa da Silva.
Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Instituto Federal de Minas Gerais. *Campus* Ouro Branco, 2026.

1. Inteligência artificial. 2. Large Language Models. 3. Testes unitários. 4. Automação de testes. I. Santos, Fabrício Augusto Miranda dos.II. Silva, Jânio Rosa da. III. Instituto Federal de Minas Gerais. *Campus* Ouro Branco. IV. Título.

CDU. 65.011.56

Catalogação: Márcia Margarida Vilaça - CRB-6/2235
Biblioteca do Instituto Federal de Minas Gerais, *Campus* Ouro Branco

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS
Campus Ouro Branco
Diretoria de Ensino
Docentes Computação
Rua Afonso Sardinha, número 90 - Bairro Minas Talco - CEP 36494-018 - Ouro Branco - MG - www.ifmg.edu.br

ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO

Aos 08 (oito) dias do mês de julho de 2026, às 17h00 (dezessete horas), Fabrício Augusto Miranda dos Santos, aluno regularmente matriculado no Curso de Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais, campus Ouro Branco, matrícula 0039952, concluiu o seu Trabalho de Conclusão de Curso por meio de defesa em sessão pública realizada às 17h00 (dezessete horas), via Google Meet, na presença da banca examinadora composta pelos docentes:

1 – Prof. Me. Janio Rosa da Silva (Presidente/Orientador)
2 – Prof. Me. Márcio Assis Miranda (Avaliador)
3 – Prof. Me. Ederson Naves Fernandes Gonçalves Júnior (Avaliador)

do artigo intitulado: TGen - Testes automatizados utilizando inteligência artificial generativa.

A banca examinadora, após reunião em sessão reservada, deliberou pela Aprovação do referido trabalho, atribuindo a nota 8,0 (oito vírgula zero), em uma escala de 0 (zero) a 10 (dez), correspondente a 80% (oitenta por cento) de aproveitamento, condicionada à realização, pelo estudante, das alterações sugeridas pela banca examinadora, no prazo estabelecido pelo Regulamento do TCC do Curso.

Condicionantes de aprovação: a aprovação do trabalho fica condicionada à entrega, pelo estudante, de versão final do TCC contemplando as correções e sugestões apontadas pela banca examinadora durante a sessão de defesa, a ser encaminhada ao orientador no prazo de até 30 (trinta) dias corridos, contados a partir desta data. A não entrega da versão corrigida no prazo estabelecido implicará a reavaliação da aprovação concedida, conforme normativas vigentes.

Ouro Branco, 8 de julho de 2026.

Documento assinado eletronicamente por **EDERSON NAVES FERNANDES GONÇALVES JÚNIOR**, **Professor Substituto**, em 11/07/2026, às 00:53, conforme Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcio Assis Miranda**, **Professor**, em 11/07/2026, às 09:00, conforme Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Janio Rosa da Silva**, **Professor**, em 11/07/2026, às 16:37, conforme Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade do documento pode ser conferida no site https://sei.ifmg.edu.br/consultadocs informando o código verificador **2805461** e o código CRC **AC945ED6**.

23712.001009/2026-54	2805461v1

Documento assinado digitalmente
FABRICIO AUGUSTO MIRANDA DOS SANTOS
Data: 04/08/2026 21:06:10-0300
Verifique em https://validar.iti.gov.br

RESUMO

O avanço das práticas de Engenharia de Software tem destacado a automação de testes como um elemento essencial para a garantia da qualidade, confiabilidade e manutenibilidade de sistemas computacionais. Este trabalho propõe o desenvolvimento de uma ferramenta automatizada capaz de gerar, validar e executar testes unitários para projetos Python utilizando um modelo de linguagem generativa. A ferramenta realiza a leitura de repositórios, gera testes compatíveis com pytest, executa-os automaticamente e registra métricas. A aplicação prática da solução em um repositório real resultou na geração autônoma de 343 casos de teste, alcançando uma notável taxa de sucesso de 86,0% em sua execução. Estes resultados evidenciam a eficácia empírica da Inteligência Artificial na ampliação da cobertura de testes e redução do esforço manual, convidando o leitor a explorar como as falhas e os sucessos mapeados impactam diretamente a produtividade e a qualidade no desenvolvimento de software.

ABSTRACT

The advancement of Software Engineering practices has highlighted test automation as an essential element for ensuring the quality, reliability, and maintainability of computer systems. This work proposes the development of an automated tool capable of generating, validating, and executing unit tests for Python projects using a generative language model. The tool reads repositories, generates tests compatible with pytest, executes them automatically, and records metrics. When applied to a real-world repository, the solution autonomously generated 343 test cases, achieving a remarkable execution success rate of 86.0%. These results demonstrate the empirical effectiveness of Artificial Intelligence in expanding test coverage and reducing manual effort, inviting the reader to explore how the mapped successes and failures directly impact productivity and quality in software development.

Palavras-chave: Inteligência Artificial; Large Language Models; Testes Unitários; Automação de Testes; Engenharia de Software; Geração Automática de Testes; ChatGPT; Gemini; Python; Qualidade de Software

SUMÁRIO

RESUMO 6
1 – INTRODUÇÃO 8
2 – PROBLEMA DE PESQUISA 9
3 – JUSTIFICATIVA 9
4 – OBJETIVO 9
5 – REFERENCIAL TEÓRICO 10
6 – METODOLOGIA 12
6.1. Seleção do Repositório 12
6.2. Desenho Experimental 13
6.3. Arquitetura e funcionamento da ferramenta 14
6.4. Modelo de Linguagem Utilizado 14
6.5. Geração automática dos testes 15
6.6. Execução e validação dos testes 15
6.7. Critérios de avaliação 16
7 – RESULTADOS 17
7.1. Caracterização do Repositório Analisado 18
7.2. Geração Automática de Testes Unitários 18
7.3. Execução e Análise dos Resultados dos Testes 19
7.4. Discussão dos Resultados 20
7.5. Interface da Ferramenta 21
7.6. Persistência dos Resultados 22
7.7. Considerações Finais sobre os Resultados 24
8 – CONCLUSÃO 25
9 – TRABALHOS FUTUROS 26
10 – REFERENCIAS 26

1 – INTRODUÇÃO

O avanço das práticas de Engenharia de Software tem consolidado a automação de testes como um dos principais mecanismos para apoiar a qualidade, a confiabilidade e a manutenção de sistemas computacionais. Em ambientes caracterizados por ciclos curtos de desenvolvimento, integração contínua e frequentes atualizações de software, a utilização de testes automatizados pode auxiliar na identificação precoce de defeitos e na redução do retrabalho, contribuindo para a evolução segura do código ao longo do tempo (MARTIN, 2018; WANG et al., 2024).

Apesar desses benefícios, a criação e manutenção de testes automatizados ainda representam um desafio para muitas equipes de desenvolvimento. A elaboração manual de casos de teste demanda tempo, conhecimento sobre o domínio da aplicação e compreensão detalhada do código-fonte, tornando-se uma atividade custosa, principalmente em projetos extensos ou desenvolvidos de forma colaborativa (MARTIN, 2018; ALSHAHWAN et al., 2024).

Nos últimos anos, os modelos de linguagem de grande porte (Large Language Models – LLMs) passaram a demonstrar desempenho promissor em atividades relacionadas ao desenvolvimento de software, incluindo geração de código, assistência à programação e criação de testes automatizados. Estudos recentes indicam que esses modelos podem apoiar a geração de casos de teste e sugerir melhorias em testes existentes, embora ainda existam desafios relacionados à precisão, consistência e validação dos resultados produzidos (ALSHAHWAN et al., 2024; WANG et al., 2024; LI et al., 2025).

Nesse contexto, a aplicação de Inteligência Artificial Generativa na geração automática de testes apresenta potencial para apoiar desenvolvedores durante o processo de validação de software. Entretanto, a efetividade dessa abordagem depende de fatores como a qualidade do código analisado, a engenharia de prompts, a capacidade do modelo de compreender o contexto da aplicação e a validação dos testes produzidos antes de sua utilização em ambientes reais (PROMPT ENGINEERING GUIDE, 2023; DAKHEL et al., 2024).

Diante desse cenário, este trabalho propõe o desenvolvimento de uma ferramenta capaz de gerar, validar e executar testes unitários para projetos desenvolvidos em Python utilizando um modelo de linguagem generativa. A solução realiza automaticamente a leitura de repositórios, identifica arquivos elegíveis para análise, gera testes compatíveis com o framework pytest, executa os casos produzidos e registra métricas relacionadas ao processo de geração e execução.

Além da implementação da ferramenta, este trabalho busca avaliar sua aplicabilidade por meio da execução dos testes gerados em um repositório público, permitindo analisar indicadores como taxa de execução bem-sucedida, limitações encontradas e aspectos que influenciam a utilização de modelos de linguagem como apoio às atividades de teste de software. Dessa forma, pretende-se contribuir para a discussão sobre o uso de Inteligência Artificial Generativa na automação de testes, oferecendo evidências experimentais sobre seu potencial e suas limitações.

2 – PROBLEMA DE PESQUISA

Embora estudos recentes demonstrem resultados promissores no uso de Large Language Models para apoio às atividades de teste de software (ALSHAHWAN et al., 2024; WANG et al., 2024), ainda permanecem desafios relacionados à confiabilidade, à consistência e à validação dos testes gerados automaticamente, especialmente quando aplicados a projetos reais com diferentes estruturas de código e dependências externas. Além disso, observa-se uma quantidade reduzida de estudos que avaliam ferramentas capazes de integrar, em um único fluxo automatizado, a geração, execução e registro sistemático dos resultados dos testes.

Diante disso, o problema que norteia este trabalho pode ser definido da seguinte forma:

Como modelos de Inteligência Artificial Generativa podem ser utilizados para apoiar a geração automática de testes unitários em projetos Python, considerando não apenas a geração dos testes, mas também sua execução e avaliação em repositórios reais?

3 – JUSTIFICATIVA

A realização deste trabalho justifica-se pela crescente demanda por sistemas de informação confiáveis, escaláveis e com alta qualidade, especialmente em ambientes organizacionais que dependem fortemente de software para suporte às suas atividades. A automação de testes é um dos principais mecanismos para garantir essa qualidade, porém sua adoção plena ainda enfrenta barreiras práticas relacionadas à complexidade e ao custo de implementação.

A aplicação de Inteligência Artificial Generativa na geração de testes automatizados apresenta-se como uma alternativa inovadora e relevante, capaz de reduzir o esforço manual e acelerar o ciclo de desenvolvimento. Além disso, ao permitir a validação e execução real dos testes gerados, a solução proposta possibilita uma avaliação objetiva da eficácia dessa abordagem, contribuindo para o avanço do conhecimento na área de Sistemas de Informação.

Do ponto de vista acadêmico, o trabalho amplia a discussão sobre o uso prático de Inteligência Artificial no desenvolvimento de software, enquanto, sob a perspectiva profissional, oferece uma ferramenta que pode ser adaptada e aplicada em ambientes reais de desenvolvimento e manutenção de sistemas.

4 – OBJETIVO

Desenvolver uma ferramenta automatizada, baseada em Inteligência Artificial Generativa, capaz de gerar, validar e executar testes unitários para sistemas desenvolvidos em Python, contribuindo para a melhoria da qualidade de sistemas de informação.

- Implementar um mecanismo de leitura e análise de repositórios contendo código-fonte em Python;
- Desenvolver um processo automatizado para geração de testes unitários utilizando Inteligência Artificial Generativa;
- Validar sintaticamente os testes gerados antes de sua execução;
- Executar automaticamente os testes utilizando o framework pytest;
- Registrar métricas de execução, como tempo e status dos testes, de forma acumulativa;
- Avaliar a aplicabilidade e a eficácia da abordagem proposta.

5 – REFERENCIAL TEÓRICO

O referencial teórico deste trabalho fundamenta-se em pesquisas relacionadas à Engenharia de Software, qualidade de software, compreensão de código-fonte, testes automatizados e ao uso de modelos de linguagem generativa aplicados ao desenvolvimento de software. A evolução desses estudos evidencia como os avanços em Inteligência Artificial têm ampliado as possibilidades de automação de atividades tradicionalmente executadas por desenvolvedores, especialmente no contexto da geração de testes de software.

Os estudos sobre compreensão de código-fonte constituem uma das bases para o desenvolvimento de técnicas de geração automática de testes. Avidan e Feitelson (2017) demonstraram que a escolha adequada de nomes de variáveis influencia diretamente a compreensão do código pelos desenvolvedores, favorecendo atividades de manutenção e reduzindo ambiguidades durante a interpretação das funcionalidades implementadas. No mesmo ano, Gopstein et al. (2017) investigaram fatores que dificultam a leitura de programas, introduzindo o conceito de *Atoms of Confusion*, pequenas estruturas sintáticas que podem aumentar a carga cognitiva dos programadores e favorecer interpretações equivocadas do comportamento do código.

Complementando esses estudos, Martin (2018) destaca que a qualidade estrutural de um software está diretamente relacionada à sua manutenibilidade, legibilidade e facilidade de evolução. Segundo o autor, sistemas organizados de acordo com boas práticas arquiteturais tendem a apresentar maior facilidade para manutenção, evolução e implementação de testes automatizados, tornando a testabilidade uma característica importante da qualidade de software.

Posteriormente, Langhout e Aniche (2021) aprofundaram a investigação sobre os *Atoms of Confusion*, analisando especificamente sua ocorrência em programas Java. Os resultados indicam que determinados padrões sintáticos aumentam significativamente a dificuldade de compreensão do código, impactando atividades como depuração, manutenção e desenvolvimento de testes. Em complemento, o conjunto de dados *Atoms of Confusion Dataset in Java Programs* (2022) disponibilizou exemplos dessas estruturas, contribuindo para pesquisas voltadas à análise estática de código e compreensão de programas.

Com o avanço dos modelos de linguagem generativa, surgiram novos estudos voltados à utilização de Inteligência Artificial no desenvolvimento de software. Nesse contexto, o *Prompt Engineering Guide* (2023) apresenta técnicas para construção de instruções destinadas a modelos generativos, descrevendo estratégias como *zero-shot prompting*, *few-shot prompting* e *chain-of-thought prompting*. Essas técnicas buscam melhorar a qualidade das respostas produzidas pelos modelos e têm sido amplamente empregadas em tarefas relacionadas à geração automática de código e testes.

A partir de 2024 observa-se um crescimento significativo das pesquisas envolvendo Large Language Models aplicados à Engenharia de Software. Alshahwan et al. (2024) investigaram a utilização desses modelos para aprimoramento automático de testes unitários, observando que eles podem sugerir melhorias em testes existentes e ampliar os cenários avaliados. Embora os resultados sejam promissores, os autores destacam que a validação dos testes produzidos continua sendo uma etapa importante do processo.

No mesmo período, Wang et al. (2024) realizaram um amplo levantamento sobre a aplicação de Large Language Models em testes de software. Os autores identificam aplicações relacionadas à geração automática de casos de teste, detecção de falhas e aumento da cobertura de testes, destacando que desafios relacionados à precisão das respostas, interpretação contextual e validação dos testes ainda representam limitações importantes para sua adoção em ambientes reais.

Buscando ampliar a qualidade dos testes produzidos automaticamente, Dakhel et al. (2024) investigaram a combinação entre modelos de linguagem e técnicas de *mutation testing*. Os resultados indicam que essa integração pode aumentar a capacidade de identificação de cenários de falha quando comparada à utilização isolada dos modelos generativos.

Ainda em 2024, Silva et al. investigaram o uso do ChatGPT na identificação de *code smells*, demonstrando que modelos generativos podem auxiliar atividades relacionadas à inspeção e qualidade de software. Entretanto, os autores ressaltam que os resultados obtidos dependem tanto da qualidade dos prompts quanto da validação realizada por especialistas.

Também em 2024, He et al. apresentaram o conjunto de dados UniTSyn, desenvolvido especificamente para treinamento e avaliação de modelos voltados à geração automática de testes unitários. A disponibilização desse dataset contribui para o desenvolvimento de pesquisas destinadas ao aprimoramento dos modelos de linguagem em tarefas relacionadas aos testes de software.

Outra contribuição relevante foi apresentada por Rane, Choudhary e Rane (2024), que realizaram uma comparação entre os modelos Gemini e ChatGPT considerando aspectos como arquitetura, desempenho e aplicações em desenvolvimento de software. Os autores observaram que ambos apresentam potencial para tarefas relacionadas à geração de código e automação, embora existam diferenças quanto ao desempenho e à interpretação contextual.

Ainda em 2024, Wang et al. propuseram o benchmark TESTEVAL, destinado à comparação de diferentes Large Language Models na geração automática de casos de teste. Os experimentos evidenciaram diferenças entre os modelos quanto à cobertura alcançada, precisão e capacidade de produzir cenários de teste mais complexos, reforçando a importância da escolha adequada do modelo e da engenharia de prompts.

Mais recentemente, Li et al. (2025) avaliaram o desempenho de Large Language Models em atividades relacionadas aos testes de software, incluindo geração automática de testes e detecção de falhas. Os resultados reforçam o potencial desses modelos como ferramentas de apoio ao desenvolvimento, mas também evidenciam limitações relacionadas à consistência das respostas produzidas e à dependência da qualidade dos prompts utilizados.

Em conjunto, os estudos apresentados demonstram a evolução das pesquisas desde aspectos relacionados à qualidade e compreensão do código-fonte até a aplicação de modelos de linguagem generativa na automação de testes. Apesar dos avanços observados, a literatura aponta que ainda existem desafios relacionados à confiabilidade dos testes gerados, à adaptação dos modelos a diferentes contextos de software e à necessidade de validação humana dos resultados. Nesse cenário, este trabalho busca contribuir por meio do desenvolvimento e da avaliação experimental de uma ferramenta capaz de integrar geração, execução e registro de testes unitários para projetos desenvolvidos em Python.

6 – METODOLOGIA

Este trabalho caracteriza-se como uma pesquisa aplicada, de natureza experimental, cujo objetivo foi desenvolver e avaliar uma ferramenta para geração automática de testes unitários utilizando um modelo de linguagem generativa.

A avaliação experimental foi conduzida sobre um único repositório público desenvolvido na linguagem Python. A escolha de um único objeto de estudo permitiu analisar detalhadamente o comportamento da ferramenta durante todas as etapas do processo, desde a geração dos testes até sua execução e registro

6.1. Seleção do Repositório

Para validação da ferramenta foi selecionado um único repositório público hospedado na plataforma GitHub.

A seleção ocorreu por amostragem intencional (não probabilística), considerando critérios definidos previamente para garantir que o experimento pudesse ser executado de forma reprodutível.

Os critérios utilizados foram:

- projeto desenvolvido integralmente em Python;
- código-fonte público;
- organização em múltiplos arquivos;
- implementação de algoritmos e estruturas de dados;
- facilidade de reprodução dos experimentos.

Com base nesses critérios foi selecionado o repositório joeyajames/Python, disponível publicamente no GitHub.

A escolha desse projeto permitiu avaliar a ferramenta em diferentes funções e estruturas de código presentes em um único ambiente controlado.

6.2. Desenho Experimental

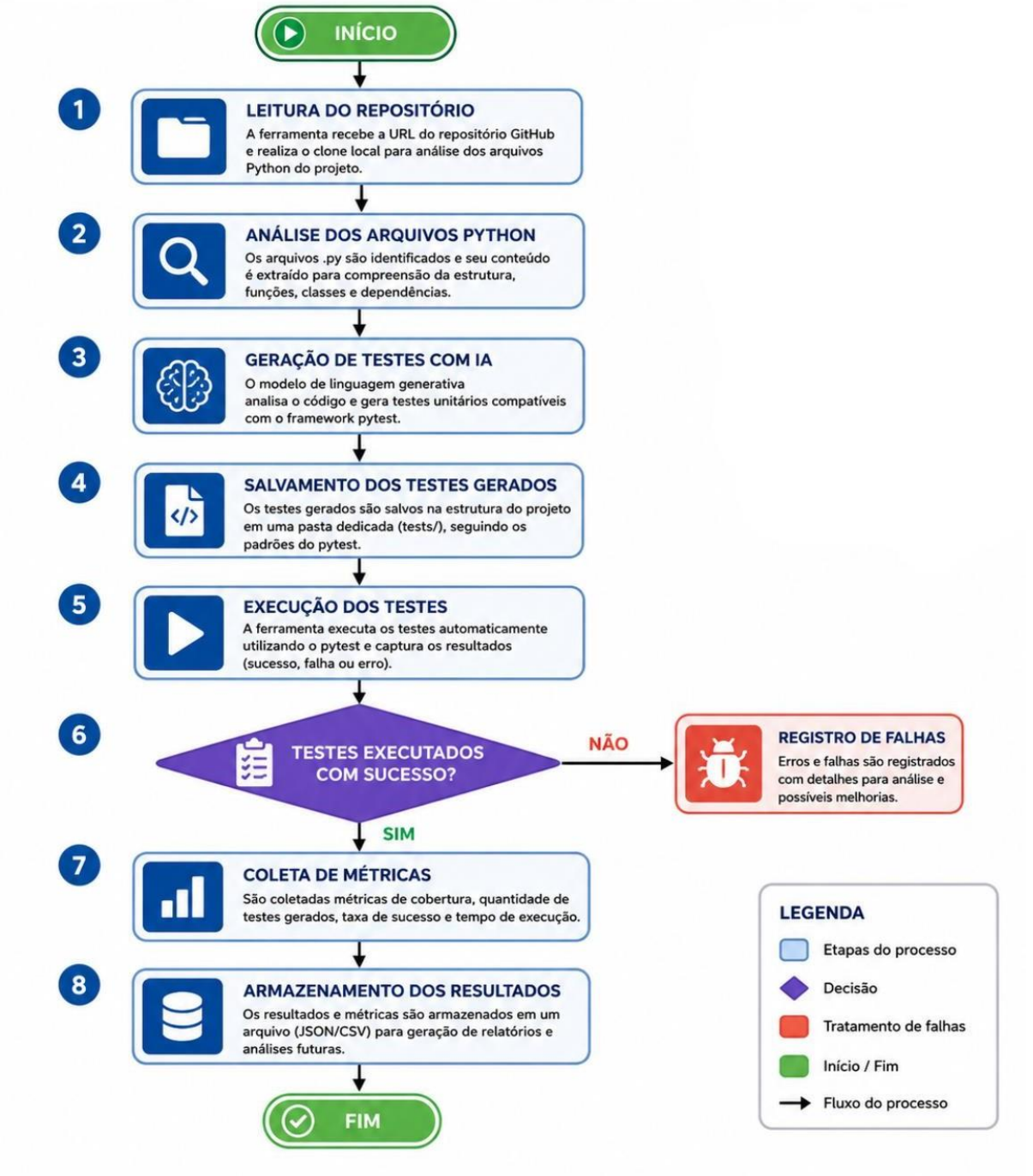


Figura 1. Fluxograma TGen

6.3. Arquitetura e funcionamento da ferramenta

A ferramenta foi implementada em Python e estruturada de forma modular, permitindo facilitar a manutenção, a compreensão do código e a evolução futura da solução. Seu funcionamento baseia-se na leitura e análise automática dos arquivos-fonte do repositório, seguida da geração de testes unitários utilizando um modelo de linguagem de grande porte (LLM), acessado por meio de uma API de IA generativa.

O processo inicia-se com a varredura do diretório do repositório, identificando arquivos com extensão .py. Arquivos irrelevantes, como scripts de configuração, ambientes virtuais ou dependências externas, são desconsiderados para evitar a geração de testes inadequados ou redundantes.

Em seguida, cada arquivo Python é analisado utilizando a biblioteca ast (Abstract Syntax Tree), que permite extrair informações estruturais do código-fonte, como:

- Definições de funções;
- Métodos de classes;
- Parâmetros de entrada;
- Estruturas condicionais e retornos.

Essa análise estrutural possibilita compreender a lógica do código sem a necessidade de execução, fornecendo informações suficientes para a criação de casos de teste coerentes e alinhados à implementação original.

6.4. Modelo de Linguagem Utilizado

A geração automática dos testes unitários foi realizada utilizando o modelo de linguagem Gemini 3.5 Flash, disponibilizado pela Google por meio da API Gemini.

O modelo foi escolhido por apresentar suporte à geração de código, capacidade de compreensão de linguagem natural e código-fonte, além de permitir integração por meio de chamadas HTTP, facilitando sua incorporação à ferramenta desenvolvida.

Durante a execução do experimento, cada arquivo Python identificado como elegível foi enviado individualmente ao modelo, acompanhado por um prompt contendo instruções específicas para geração de testes unitários compatíveis com o framework *pytest*. O modelo retornava exclusivamente o código-fonte dos testes, que posteriormente era validado, armazenado e executado automaticamente pela ferramenta.

6.5. Geração automática dos testes

Após a análise do código, as informações extraídas são organizadas e enviadas como entrada para o modelo de linguagem por meio de uma requisição HTTP à API de IA generativa. O prompt é cuidadosamente construído para instruir o modelo a gerar testes unitários no padrão do framework pytest, incluindo:

- Nomeação adequada das funções de teste;
- Uso de asserts compatíveis com o comportamento esperado;
- Cobertura de diferentes cenários de execução, como casos válidos, inválidos e de exceção.

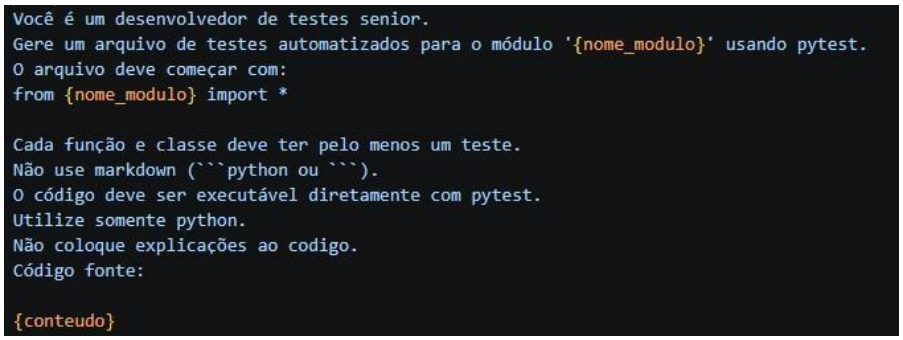


Figura 2 - Trecho do prompt enviado ao modelo

A Figura 2 apresenta um trecho do prompt utilizado durante os experimentos. O prompt foi elaborado para orientar o modelo quanto ao framework utilizado, formato esperado da resposta e restrições necessárias para permitir a execução automática dos testes gerados.

Os testes gerados são então processados e salvos automaticamente em arquivos específicos, posicionados na mesma pasta do código-fonte original, seguindo convenções de nomenclatura amplamente adotadas, como o prefixo test. Essa abordagem facilita a execução posterior dos testes e mantém a organização do projeto.

6.6. Execução e validação dos testes

Após a geração dos arquivos de teste, a ferramenta executa automaticamente o framework pytest por meio de chamadas ao sistema operacional. Os resultados da execução são capturados e analisados, permitindo identificar:

- Testes aprovados;
- Testes com falha;

- Erros de execução;
- Problemas de importação ou dependências.

Essas informações são registradas para fins de avaliação da eficácia da ferramenta, possibilitando verificar se os testes gerados são executáveis e se refletem corretamente o comportamento esperado do código analisado.

6.7. Critérios de avaliação

A avaliação da solução proposta foi realizada com base em critérios qualitativos e quantitativos, incluindo:

- quantidade de arquivos processados.
- quantidade de arquivos incompatíveis.
- número de testes gerados.
- número de testes executados.
- quantidade de testes aprovados.
- quantidade de testes com falha.
- tempo de execução da ferramenta.

Esses critérios permitem analisar o potencial da ferramenta como apoio ao desenvolvimento de software, bem como identificar limitações relacionadas à confiabilidade e à necessidade de revisão humana dos testes gerados.

A metodologia descrita estabelece um fluxo reprodutível para avaliação da ferramenta, permitindo observar sua capacidade de gerar, executar e registrar testes unitários em um repositório Python selecionado para o estudo de caso.

7- Resultados

Este capítulo apresenta os resultados obtidos a partir da aplicação da ferramenta desenvolvida neste trabalho em um repositório real de código-fonte escrito em Python, com o objetivo de avaliar sua capacidade de gerar automaticamente testes unitários utilizando um modelo de linguagem, executá-los e registrar métricas sobre sua eficácia.

Para a validação foi utilizado o repositório público joeyajames/Python, disponível no GitHub, escolhido por possuir diversos exemplos de algoritmos e estruturas de dados implementados em Python, permitindo avaliar a ferramenta em diferentes cenários de código.

7.1. Caracterização do Repositório Analisado

Durante a etapa inicial do experimento, a ferramenta realizou automaticamente a clonagem do repositório e realizou a varredura completa do mesmo, identificando os arquivos de código-fonte elegíveis para o processo de geração automática de testes.

Ao todo, foram analisados 43 arquivos de código-fonte Python, os quais serviram como base para a geração dos testes unitários. Esse quantitativo demonstra que a ferramenta foi capaz de processar integralmente o repositório selecionado, sem a necessidade de ajustes manuais prévios em sua estrutura.

Durante essa etapa, verificou-se que 7 arquivos apresentavam código inválido ou incompatível com o processo de análise, impossibilitando a geração automática dos testes para esses casos. Os demais arquivos foram processados normalmente.

A Figura 3 apresenta o resumo da classificação dos arquivos analisados.

Figura 3 - Distribuição dos arquivos analisados

7.2. Geração Automática de Testes Unitários

Após a leitura dos arquivos válidos, a ferramenta enviou automaticamente o código-fonte ao modelo Gemini, responsável pela geração dos testes unitários no padrão do framework pytest.

Todo o processo ocorreu de forma automatizada, sem necessidade de intervenção do usuário. Para cada arquivo válido, foram gerados casos de teste considerando diferentes cenários de execução, incluindo casos positivos, negativos e condições de erro quando identificadas pelo modelo. Ao final do processo, foram produzidos 343 casos de teste, distribuídos entre todos os arquivos processados.

Esse resultado demonstra que a ferramenta é capaz de produzir múltiplos casos de teste para um mesmo arquivo, aumentando significativamente a cobertura inicial do software analisado.

7.3. Execução e Análise dos Resultados dos Testes

Após a geração dos testes, todos os casos produzidos foram executados automaticamente utilizando o framework *pytest*, possibilitando a verificação do comportamento do código sob teste. Durante a execução foram obtidos os seguintes resultados:

Métrica	Quantidade	Percentual
Casos de teste executados	343	100,0%
Casos executados com sucesso	295	86,0%
Casos com falha	48	14,0%

Tabela 1 - Resumo da execução dos casos de teste

Os resultados representam uma taxa de sucesso de 86,0%, enquanto 14,0

As falhas observadas ocorreram principalmente devido a limitações presentes no código original do repositório, dependências externas, comportamentos específicos de determinadas bibliotecas e inconsistências entre o comportamento esperado pelo modelo de linguagem e a implementação real do software analisado.

Apesar dessas ocorrências, a elevada taxa de sucesso demonstra que a abordagem proposta é capaz de gerar testes funcionais para a maior parte dos cenários avaliados. A Figura 4 apresenta a distribuição dos casos de teste executados.

Figura 4 - Distribuição dos casos de teste executados

7.4. Discussão dos Resultados

Os resultados obtidos demonstram a viabilidade da abordagem proposta tanto do ponto de vista acadêmico quanto prático. Dos 343 casos de teste executados, 295 foram concluídos com sucesso, correspondendo a uma taxa de aprovação de aproximadamente 86,0%. Esse resultado evidencia que a ferramenta é capaz de gerar automaticamente casos de teste funcionais para a maior parte dos cenários avaliados, mesmo quando aplicada a um repositório desenvolvido sem foco específico em testabilidade.

Os 48 casos de teste que apresentaram falha não representam necessariamente limitações da ferramenta, mas refletem características presentes no próprio repositório analisado. Entre os principais fatores observados estão:

- Dependências externas necessárias para a correta execução de determinadas funcionalidades;
- Necessidade de configurações específicas de ambiente ou de arquivos auxiliares;
- Implementações que utilizam estados globais ou recursos compartilhados;
- Comportamentos dependentes de bibliotecas gráficas ou interação com o sistema operacional;
- Particularidades da lógica implementada que exigem conhecimento contextual não disponível durante a geração automática dos testes.

Além disso, durante a etapa de análise foram identificados sete arquivos contendo código inválido ou incompatível com o processo de geração automática de testes, impossibilitando seu processamento. Esse resultado demonstra que a qualidade e a consistência do código-fonte influenciam diretamente a efetividade da geração automática de testes.

De forma geral, os resultados obtidos indicam que a ferramenta apresenta desempenho satisfatório na automatização da geração de testes unitários, produzindo casos de teste válidos para a maior parte dos arquivos analisados. As limitações identificadas representam oportunidades para trabalhos futuros, especialmente no desenvolvimento de mecanismos que permitam adaptar automaticamente os testes ao contexto específico de cada projeto, ampliando ainda mais a taxa de sucesso da abordagem proposta.

7.5. Interface da Ferramenta

Além da avaliação quantitativa, foi desenvolvida uma interface gráfica para facilitar a utilização da ferramenta durante o processo de experimentação.

A interface permite ao usuário informar a URL do repositório GitHub, selecionar o diretório de trabalho, acompanhar em tempo real a geração dos testes e visualizar o progresso da execução.

Após a conclusão do processamento, a aplicação apresenta automaticamente um resumo contendo os casos de teste gerados, testes executados com sucesso, falhas encontradas e arquivos que não puderam ser processados.

As Figuras 5 e 6 ilustram as principais telas da aplicação.

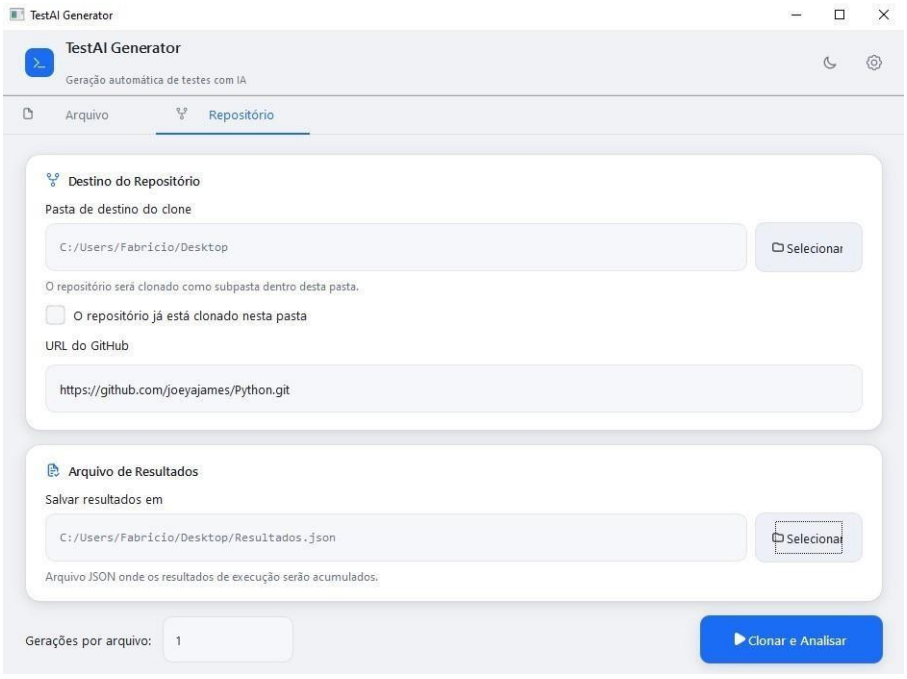
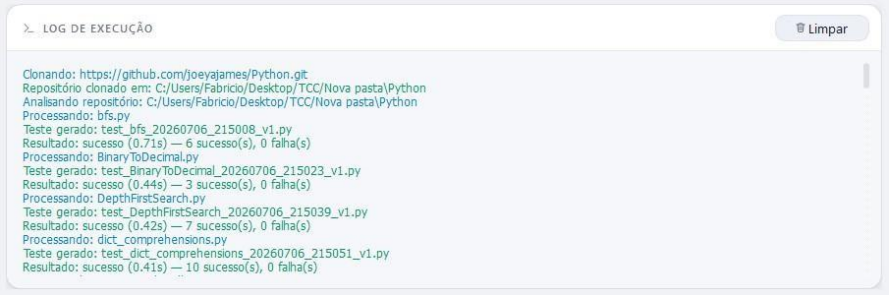


Figura 5 - Interface principal da ferramenta



Fugira 6 - Log de Execução da ferramenta

7.6. Persistência dos Resultados

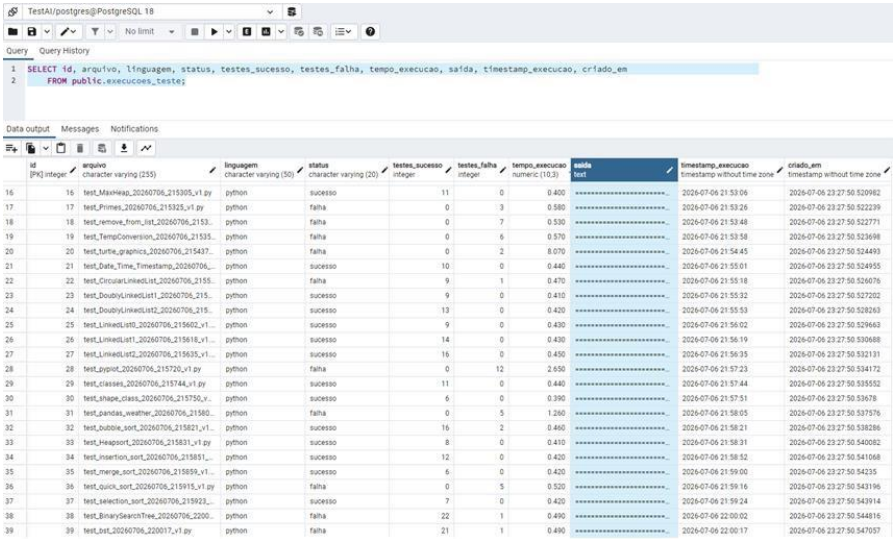
Além da geração e execução automática dos testes unitários, a ferramenta foi desenvolvida para registrar permanentemente todas as informações obtidas durante o processamento. Para isso, os resultados são armazenados em dois formatos distintos: um arquivo JSON, utilizado para registro estruturado das execuções, e um banco de dados PostgreSQL, responsável pelo armazenamento persistente das métricas e informações dos testes.

O arquivo JSON funciona como um histórico detalhado das execuções realizadas pela ferramenta. Nele são registrados dados como o repositório analisado, o arquivo processado, o nome do teste gerado, o status da execução, o tempo de processamento, a quantidade de casos de teste executados com sucesso e com falha, além de outras informações relevantes para auditoria e análise posterior.

Figura 7 - Exemplo do arquivo JSON contendo os resultados das execuções

Paralelamente, todas essas informações também são persistidas em um banco de dados PostgreSQL. Essa abordagem permite consultas estruturadas, geração de relatórios, comparação entre diferentes execuções e integração com outras aplicações, tornando a solução mais escalável e adequada para ambientes de desenvolvimento reais.

O banco de dados foi modelado para armazenar informações referentes aos repositórios analisados, arquivos processados, testes gerados, resultados da execução e métricas de desempenho, possibilitando a construção de indicadores sobre a eficiência da ferramenta ao longo do tempo.



	id [PK] integer	arquivo character varying (255)	linguagem character varying (50)	status character varying (20)	testes_sucesso integer	testes_falha integer	tempo_execucao numeric (10,3)	saida text	timestamp_execucao timestamp without time zone	criado_em timestamp without time zone
16	16	test_MaxHeap_20260706_215305_v1.py	python	sucesso	11	0	0.400	**********	2026-07-06 21:53:06	2026-07-06 23:27:50.520982
17	17	test_Primes_20260706_215325_v1.py	python	falha	0	3	0.580	**********	2026-07-06 21:53:26	2026-07-06 23:27:50.522239
18	18	test_remove_from_list_20260706_2153...	python	falha	0	7	0.530	**********	2026-07-06 21:53:48	2026-07-06 23:27:50.522771
19	19	test_TempConversion_20260706_21535...	python	falha	0	6	0.570	**********	2026-07-06 21:53:58	2026-07-06 23:27:50.523698
20	20	test_turtle_graphics_20260706_215437...	python	falha	0	2	8.070	**********	2026-07-06 21:54:45	2026-07-06 23:27:50.524493
21	21	test_Date_Time_Timestamp_20260706_...	python	sucesso	10	0	0.440	**********	2026-07-06 21:55:01	2026-07-06 23:27:50.524955
22	22	test_CircularLinkedList_20260706_2155...	python	falha	9	1	0.470	**********	2026-07-06 21:55:18	2026-07-06 23:27:50.526076
23	23	test_DoublyLinkedList1_20260706_215...	python	sucesso	9	0	0.410	**********	2026-07-06 21:55:32	2026-07-06 23:27:50.527202
24	24	test_DoublyLinkedList2_20260706_215...	python	sucesso	13	0	0.420	**********	2026-07-06 21:55:53	2026-07-06 23:27:50.528263
25	25	test_LinkedList0_20260706_215602_v1...	python	sucesso	9	0	0.430	**********	2026-07-06 21:56:02	2026-07-06 23:27:50.529663
26	26	test_LinkedList1_20260706_215618_v1...	python	sucesso	14	0	0.430	**********	2026-07-06 21:56:19	2026-07-06 23:27:50.530688
27	27	test_LinkedList2_20260706_215635_v1...	python	sucesso	16	0	0.450	**********	2026-07-06 21:56:35	2026-07-06 23:27:50.532131
28	28	test_pyplot_20260706_215720_v1.py	python	falha	0	12	2.650	**********	2026-07-06 21:57:23	2026-07-06 23:27:50.534172
29	29	test_classes_20260706_215744_v1.py	python	sucesso	11	0	0.440	**********	2026-07-06 21:57:44	2026-07-06 23:27:50.535552
30	30	test_shape_class_20260706_215750_v...	python	sucesso	6	0	0.390	**********	2026-07-06 21:57:51	2026-07-06 23:27:50.53678
31	31	test_pandas_weather_20260706_21580...	python	falha	0	5	1.260	**********	2026-07-06 21:58:05	2026-07-06 23:27:50.537576
32	32	test_bubble_sort_20260706_215821_v1...	python	sucesso	16	2	0.460	**********	2026-07-06 21:58:21	2026-07-06 23:27:50.538286
33	33	test_Heapsort_20260706_215831_v1.py	python	sucesso	8	0	0.410	**********	2026-07-06 21:58:31	2026-07-06 23:27:50.540082
34	34	test_insertion_sort_20260706_215851_...	python	sucesso	12	0	0.420	**********	2026-07-06 21:58:52	2026-07-06 23:27:50.541068
35	35	test_merge_sort_20260706_215859_v1...	python	sucesso	6	0	0.420	**********	2026-07-06 21:59:00	2026-07-06 23:27:50.54235
36	36	test_quick_sort_20260706_215915_v1.py	python	falha	0	5	0.520	**********	2026-07-06 21:59:16	2026-07-06 23:27:50.543196
37	37	test_selection_sort_20260706_215923_...	python	sucesso	7	0	0.420	**********	2026-07-06 21:59:24	2026-07-06 23:27:50.543914
38	38	test_BinarySearchTree_20260706_2200...	python	falha	22	1	0.490	**********	2026-07-06 22:00:02	2026-07-06 23:27:50.544816
39	39	test_bst_20260706_220017_v1.py	python	falha	21	1	0.490	**********	2026-07-06 22:00:17	2026-07-06 23:27:50.547057

Figura 8 - Estrutura da tabela de resultados no banco de dados PostgreSQL

A utilização conjunta do arquivo JSON e do banco de dados proporciona maior confiabilidade ao processo de armazenamento. Enquanto o JSON facilita a inspeção rápida dos resultados e sua utilização em outras ferramentas, o banco de dados oferece persistência, organização e capacidade de consulta, permitindo análises estatísticas e acompanhamento histórico das execuções realizadas. Dessa forma, a ferramenta não apenas automatiza a geração e execução de testes unitários, mas também disponibiliza uma base consistente de informações que pode ser utilizada para avaliação da qualidade do software e evolução contínua do processo de testes.

7.7. Considerações Finais sobre os Resultados

Os resultados obtidos demonstram que a utilização de modelos de linguagem para geração automática de testes unitários apresenta elevado potencial para auxiliar o processo de desenvolvimento de software.

A ferramenta foi capaz de analisar integralmente um repositório contendo 43 arquivos, identificar automaticamente arquivos incompatíveis, gerar 343 casos de teste e executar esses testes de forma totalmente automatizada.

A taxa de 86,0% de sucesso evidencia que o modelo consegue produzir testes corretos para a maior parte dos cenários analisados. As falhas identificadas não representam necessariamente limitações da ferramenta, mas refletem características do próprio repositório avaliado, como códigos incompletos, arquivos incompatíveis, dependências externas ou implementações específicas que dificultam a geração automática.

Os gráficos apresentados ao longo deste capítulo facilitam a interpretação dos resultados, permitindo visualizar tanto a classificação dos arquivos analisados quanto o desempenho dos casos de teste gerados. Em conjunto com as capturas da interface da ferramenta, esses elementos demonstram de forma objetiva o funcionamento da

solução proposta e reforçam sua viabilidade para apoiar o desenvolvimento e a manutenção de aplicações em Python.

8 – CONCLUSÃO

O avanço da Inteligência Artificial aplicada à Engenharia de Software tem proporcionado novas possibilidades para automação de atividades tradicionalmente realizadas de forma manual, especialmente no contexto de testes de software. Neste trabalho, foi proposta e desenvolvida uma ferramenta capaz de gerar, validar e executar testes unitários automaticamente em projetos escritos na linguagem Python, utilizando modelos de linguagem generativa como apoio no processo de criação dos testes.

A solução desenvolvida demonstrou que Large Language Models podem contribuir significativamente para a automação de tarefas relacionadas à qualidade de software, reduzindo o esforço necessário para criação manual de testes unitários e auxiliando na identificação de cenários de validação para funções presentes em repositórios reais.

Durante os experimentos realizados, a ferramenta efetuou a análise automática de 43 arquivos de código-fonte. Desses, 36 arquivos puderam ser processados, enquanto 7 apresentaram código inválido ou incompatível, impossibilitando a geração automática de testes. Ao final do processo, foram gerados e executados 343 casos de teste, dos quais 295 foram concluídos com sucesso e 48 apresentaram falhas, resultando em uma taxa de sucesso de 86,0%.

Os resultados obtidos evidenciam que modelos generativos apresentam capacidade relevante de interpretação de código-fonte e geração automática de testes, especialmente em estruturas mais simples e organizadas. Entretanto, também foram observadas limitações relacionadas à consistência dos testes produzidos, dependência da qualidade dos prompts utilizados e dificuldades na interpretação de códigos mais complexos ou com estruturas ambíguas.

Além disso, verificou-se que fatores relacionados à legibilidade do código, organização estrutural e clareza semântica influenciam diretamente na qualidade dos testes gerados automaticamente. Esse comportamento está alinhado com os estudos apresentados na fundamentação teórica, os quais demonstram que códigos mais claros e menos ambíguos favorecem tanto a compreensão humana quanto a interpretação realizada por modelos de linguagem.

Outro aspecto relevante identificado foi a importância da engenharia de prompt no desempenho da ferramenta. A definição adequada das instruções enviadas ao modelo generativo impactou diretamente na precisão, cobertura e validade dos testes produzidos, evidenciando que a qualidade das respostas obtidas está fortemente associada à construção dos prompts utilizados.

Embora os resultados obtidos não garantam substituição completa da atuação humana no processo de testes de software, o trabalho demonstra que modelos de linguagem podem atuar como ferramentas de apoio ao desenvolvedor, auxiliando na

geração inicial de testes unitários e contribuindo para aumento da produtividade em atividades relacionadas à validação de software.

Dessa forma, conclui-se que a utilização de Inteligência Artificial generativa aplicada à automação de testes unitários representa uma abordagem promissora para a Engenharia de Software, principalmente diante do crescimento da complexidade dos sistemas computacionais e da necessidade constante de aumento da qualidade e confiabilidade das aplicações.

9 – TRABALHOS FUTUROS

Apesar dos resultados satisfatórios obtidos com a ferramenta desenvolvida, este trabalho, em função das limitações de escopo e tempo disponíveis, concentrou-se na implementação de um protótipo funcional voltado à geração automática de testes unitários em repositórios Python. Ainda assim, a análise dos testes gerados e a execução em projetos reais evidenciaram oportunidades de aprimoramento, indicando que a solução pode ser expandida e refinada para atender de forma mais abrangente às demandas do desenvolvimento de software moderno. Diante desse cenário, recomenda-se que trabalhos futuros explorem as seguintes propostas:

- Ampliação do suporte a diferentes linguagens de programação, permitindo a aplicação da ferramenta em projetos escritos em linguagens como Java, JavaScript e C#, aumentando sua aplicabilidade em ambientes corporativos heterogêneos.
- Integração de métricas avançadas de cobertura de testes, possibilitando a análise quantitativa do impacto dos testes gerados automaticamente sobre o código-fonte, bem como a identificação de trechos não cobertos.
- Aprimoramento do processo de validação dos testes gerados, por meio da inclusão de mecanismos automáticos de verificação de confiabilidade, redução de falsos positivos e adaptação dinâmica dos casos de teste com base nos resultados de execução.
- Integração da solução a pipelines de Integração Contínua (CI/CD), possibilitando a geração e execução automática de testes a cada atualização do repositório, fortalecendo práticas de DevOps e garantindo maior qualidade contínua do software.

10 – REFERENCIAS

ALSHAHWAN, Nadia et al. *Automated unit test improvement using large language models: an exploratory study*. In: Proceedings of the 18th International Conference on Software and System Processes, 2024.

ATOMS OF CONFUSION DATASET IN JAVA PROGRAMS. Zenodo, 2022. Disponível em: https://zenodo.org/record/7065842. Acesso em: 12 jul. 2024.

AVIDAN, E.; FEITELSON, D. G. *Effects of variable names on comprehension: an empirical study*. In: IEEE/ACM International Conference on Program Comprehension (ICPC), 25., Buenos Aires, 2017. Anais [...]. IEEE, 2017. p. 55–65. Disponível em: https://doi.org/10.1109/ICPC.2017.27. Acesso em: 12 jul. 2024.

DAKHEL, Arghavan Moradi et al. *Effective test generation using pre-trained Large Language Models and mutation testing*. Information and Software Technology, 2024. Disponível em: https://doi.org/10.1016/j.infsof.2024.107468. Acesso em: 12 jul. 2024.

GOPSTEIN, D. et al. *Understanding Misunderstandings in Source Code*. In: Joint Meeting on Foundations of Software Engineering, 11., Paderborn, 2017. Proceedings [...]. ACM, 2017.

HE, Yifeng et al. *UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing*. 2024. Disponível em: https://arxiv.org/abs/2402.03396. Acesso em: 12 jul. 2024.

LANGHOUT, Chris; ANICHE, Maurício. *Atoms of confusion in Java*. Preprint, 2021. Disponível em: https://arxiv.org/abs/2103.05424. Acesso em: 12 jul. 2024.

LI, Yihao et al. *Evaluating Large Language Models for Software Testing*. Computer Standards & Interfaces, 2025. Disponível em: https://www.researchgate.net/publication/385779829_Evaluating_Large_Language_Models_for_Software_Testing. Acesso em: 12 jul. 2024.

MARTIN, Robert C. *Arquitetura Limpa: o guia do artesão para estrutura e design de software*. Rio de Janeiro: Alta Books, 2018.

PROMPT ENGINEERING GUIDE. Prompting Guide, 2023. Disponível em: https://www.promptingguide.ai/pt. Acesso em: 12 jul. 2024.

RANE, Nitin Liladhar; CHOUDHARY, Saurabh P.; RANE, Jayesh. *Gemini versus ChatGPT: applications, performance, architecture, capabilities, and implementation*. Journal of Applied Artificial Intelligence, Mumbai, v. 5, n. 1, p. 69–93, 2024. Disponível em: https://doi.org/10.48185/jaai.v5i1.1052. Acesso em: 12 jul. 2024.

SILVA, Luciana Lourdes et al. *Detecting Code Smells using ChatGPT: Initial Insights*. Preprint, 2024. Disponível em: https://doi.org/10.13140/RG.2.2.36634.04802. Acesso em: 12 jul. 2024.

WANG, Junjie et al. *Software Testing With Large Language Models: Survey, Landscape, and Vision*. IEEE Transactions on Software Engineering, 2024.

WANG, Wenhan et al. *TESTEVAL: Benchmarking Large Language Models for Test Case Generation*. 2024. Disponível em: https://arxiv.org/abs/2406.04531. Acesso em: 12 jul. 2024.