Universidade Federal de São Carlos – UFSCar
Centro de Ciências Exatas e de Tecnologia – CCET
Departamento de Computação – DC
Bacharelado em Engenharia da Computação – ECC

Lucca Renato Guerino

Estudo do desempenho de LLMs como geradores automáticos de teste unitário para programas Python

São Carlos
2025

Lucca Renato Guerino

Estudo do desempenho de LLMs como geradores automáticos de teste unitário para programas Python

Trabalho de Conclusão de Curso apresentado ao Bacharelado em Engenharia da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia da Computação.

Área de concentração: Metodologias e Técnicas de Computação

Orientador: Prof. Dr. André Takeshi Endo
Coorientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

São Carlos
2025

Dedico este trabalho à minha mãe, Haline, a qual sempre esteve ao meu lado não importa a situação, garantindo carinho, apoio e amor.

Agradecimentos

Agradeço a todas as pessoas que contribuíram na minha vida universitária, não importa a maneira. Sou grato à minha mãe, Haline, a qual nada teria sido possível sem o seu incentivo aos estudos e apoio desde cedo. Agradeço à minha namorada, Ana, que sempre me incentivou a ser a minha melhor versão. Também agradeço aos meus amigos que acompanharam minha jornada universitária em todos os períodos. Também sou grato ao professor Auri, que durante toda minha graduação me orientou e hoje, já em outra universidade, pode ser co-orientador deste trabalho. Por fim, ao professor André, que se possibilitou que esse trabalho se concretizasse sendo meu orientador.

Resumo

O teste de software é uma etapa fundamental no desenvolvimento de sistemas, sendo essencial para a detecção de falhas no desenvolvimento e a garantia da qualidade do produto final, o que continua sendo um desafio para linguagens dinamicamente tipadas, como o Python. O estudo examina a capacidade do modelo de linguagem ChatGPT-3.5-turbo de gerar conjuntos de testes para programas em Python, comparando seu desempenho ao de ferramentas como o Pynguin e a conjuntos de testes já existentes. Para essa avaliação, foram utilizados 40 programas em Python, a partir dos quais se geraram testes compatíveis com Pytest por meio da API da OpenAI, variando níveis de temperatura de geração. Os testes foram validados com Pytest, enquanto métricas de cobertura e pontuação de mutação foram obtidas com as ferramentas Coverage, MutPy e Cosmic-Ray. Os resultados mostram que o ChatGPT-3.5-turbo gera testes válidos para programas simples, mas mantém média geral abaixo de 28%, embora com baixo custo computacional. Temperaturas mais altas melhoram o desempenho, e combinar testes de diferentes temperaturas aumenta a diversidade, permitindo superar Pynguin e testes existentes em cobertura de decisão e pontuação de mutação.

Palavras-chave: teste de software, teste de mutação, linguagem python, automatização de teste, modelos de linguagem de larga escala.

Abstract

Software testing is a fundamental stage in system development, essential for detecting faults and ensuring the quality of the final product, yet it remains challenging for dynamically typed languages such as Python. This study examines the ability of the ChatGPT-3.5-turbo language model to generate test sets for Python programs, comparing its performance with tools such as Pynguin and with existing test suites. For this evaluation, 40 Python programs were used, from which Pytest-compliant tests were generated through the OpenAI API under varying temperature settings. The tests were validated using Pytest, and coverage and mutation scores were obtained with Coverage, MutPy, and Cosmic-Ray. The results show that ChatGPT-3.5-turbo produces valid tests for simple programs but maintains an overall average below 28%, despite its low computational cost. Higher temperatures improve performance, and combining tests generated at different temperatures increases diversity, allowing the model to outperform both Pynguin and existing test suites in decision coverage and mutation score.

Keywords: software test, mutation test, python language, test automation, large language models.

Lista de ilustrações

Lista de tabelas

Sumário

Capítulo 1

Introdução

Nos últimos anos, os grandes modelos de linguagem (LLMs, do inglês Large Language Models) têm se consolidado como ferramentas transformadoras em diversos domínios tecnológicos. Entre esses modelos, a série ChatGPT[1] da OpenAI tem ganhado destaque devido à sua ampla aplicabilidade em tarefas de processamento de linguagem natural e em sistemas de inteligência artificial conversacional. O ChatGPT tem sido amplamente adotado em contextos profissionais e acadêmicos, apesar de seus custos, e tem figurado em diversas publicações recentes. Essa rápida adoção tem despertado interesse na avaliação de seu potencial para apoiar e aprimorar atividades críticas da engenharia de software, em especial no contexto de testes de software (SCHMIDT, 2025).

Além disso, o avanço de serviços de roteamento de modelos linguísticos, como o OpenRouter[2], tem ampliado significativamente o acesso a diferentes LLMs em uma única plataforma integrada. Esses serviços permitem que, em um mesmo ambiente, múltiplos modelos proprietários e abertos sejam utilizados, otimizando tanto a comparação empírica quanto a aplicabilidade prática de suas respostas. Além dos modelos disponibilizados pela OpenAI, destacam-se outros sistemas avançados, como o DeepSeek[3], desenvolvido pela DeepSeek AI; o Qwen[4], criado pelo Alibaba Group; e o Claude[5], desenvolvido pela Anthropic. Essa diversidade de modelos acessíveis por meio de serviços de roteamento favorece uma experimentação mais abrangente e o desenvolvimento de análises comparativas em diferentes contextos de pesquisa.

Com o aumento da complexidade dos sistemas e a crescente demanda por desen-

[1] <https://chat.openai.com/>
[2] <https://openrouter.ai/>
[3] <https://www.deepseek.com/>
[4] <https://qwen.ai/>
[5] <https://www.anthropic.com/claude>

volvimento ágil, a automação de testes torna-se essencial para assegurar a qualidade do software. Apesar do uso de ferramentas automatizadas, desafios ainda persistem, como no caso do Python, uma linguagem dinamicamente tipada, na qual a diversidade estrutural do código exige estratégias de teste mais adaptativas, dificultando a geração automática de testes (VOGL et al., 2021).

Este estudo investiga se os LLMs podem contribuir para superar tais desafios e oferecer suporte a processos de teste de software mais rigorosos. Será realizado um experimento profundo com um modelo específico, o ChatGPT-3.5-turbo, avaliando sua capacidade de gerar testes unitários em diferentes configurações de temperaturas.

A proeminência do Python como a linguagem de programação mais popular, de acordo com o Índice Tiobe (TIOBE Software BV, 2024), torna-o um objeto de estudo particularmente relevante. Sua versatilidade e ampla adoção em diferentes setores asseguram sua aplicabilidade em múltiplos contextos. Este estudo explora o potencial dos LLMs para aprimorar os processos de teste de software em Python. A geração automatizada de testes por meio de LLMs pode trazer ganhos em eficiência, escalabilidade e confiabilidade, impactando diretamente o desenvolvimento e a manutenção de sistemas.

Os conjuntos de testes gerados são avaliados utilizando o Coverage[6] para análise de cobertura de decisão, e as ferramentas MutPy (HOSSNER et al., 2021) e Cosmic-Ray (BINGHAM, 2021) para teste de mutação. O teste de mutação avalia a robustez de um conjunto de testes frente a falhas potenciais, assegurando sua confiabilidade. Conjuntos de testes pré-existentes são empregados como referência para a avaliação comparativa dos testes gerados pelos LLMs.

É de relevância ressaltar que este trabalho foi desenvolvido, submetido e apresentado no XXIV Simpósio Brasileiro de Qualidade de Software – SBQS 2025[7], recebendo um prêmio de *Distinguished Paper*, pelo seu destaque no evento.

A estrutura deste documento está organizada da seguinte forma: a Seção 2 apresenta informações de fundamentação teórica e trabalhos relacionados; a Seção 3 descreve o delineamento experimental e a coleta de dados; a Seção 4 aborda o processo de análise dos dados e discute os resultados obtidos; e, por fim, a Seção 5 expõe as conclusões e as direções para pesquisas futuras.

[6] <https://pypi.org/project/coverage/>
[7] <https://sbqs.sbc.org.br/2025>

Capítulo 2

Fundamentação Teórica

2.1 Geração automática de teste de software

Teste automatizado de software é o processo de verificação da qualidade e do comportamento de sistemas computacionais por meio da execução de scripts ou ferramentas que substituem a intervenção manual. Essa prática busca aumentar a eficiência, a confiabilidade e a repetibilidade da avaliação do software, reduzindo custos e tempo de desenvolvimento.

Os testes automatizados podem abranger diferentes níveis, como testes de unidade, de integração, de sistema e de aceitação, permitindo validar desde componentes individuais até o funcionamento global da aplicação. Além disso, a automação possibilita a execução contínua em diferentes ciclos de desenvolvimento, sendo um elemento central em abordagens modernas como Integração Contínua e Entrega Contínua (CI/CD).

No contexto dos tipos de testes, destacam-se principalmente: testes unitários, testes de integração, testes de sistema e testes de aceitação. O teste unitário, foco desta pesquisa, refere-se à verificação isolada de unidades mínimas do software, geralmente funções, métodos ou classes, com o objetivo de assegurar que cada componente atenda corretamente às especificações previstas. Esse tipo de teste é caracterizado por sua granularidade fina, baixo custo de execução e alta velocidade, permitindo identificar falhas de forma precoce durante o desenvolvimento.

Considerando o contexto de Python, a Pynguin é uma biblioteca voltada para a geração automática de testes que combina técnicas baseadas em mutação e algoritmos evolucionários para aumentar a eficácia dos testes unitários (LUKASCZYK; KROIß; FRASER, 2023). Ao interpretar o bytecode da linguagem, a ferramenta busca maximizar a cobertura de código e, consequentemente, a confiabilidade do software. Conforme destacado

por Guerino e Vincenzi (2023), a escolha pela Pynguin em nosso estudo se justifica por sua adequação como a ferramenta mais indicada para testes automatizados em Python, uma vez que opera de forma independente, sem intervenção humana, baseando-se exclusivamente no programa em teste.

2.2 Teste de mutação

Teste de mutação é uma técnica de avaliação da qualidade de testes de software que consiste em introduzir pequenas alterações artificiais, denominadas mutantes, no código-fonte original de um programa. Essas modificações simulam potenciais erros de implementação, como a substituição de operadores lógicos ou aritméticos, a alteração de valores constantes ou a modificação de condições de controle. O objetivo é verificar se o conjunto de testes existente é capaz de identificar e "matar"esses mutantes, isto é, produzir falhas que evidenciem o comportamento incorreto introduzido. Caso os mutantes não sejam detectados, infere-se que o conjunto de testes pode ser insuficiente ou pouco robusto (ANDREWS; BRIAND; LABICHE, 2005; PETROVIć et al., 2021).

Dessa forma, o teste de mutação é considerado uma das métricas mais rigorosas para mensurar a eficácia de casos de teste, fornecendo uma avaliação precisa sobre a capacidade de detecção de defeitos no software.

Guerino et al. (2024) conduziram um estudo comparativo sobre ferramentas de teste de mutação em Python, especificamente Cosmic-Ray, MutPy, MutMut e Mutatest, avaliando-as sob perspectivas estática e dinâmica. O estudo apresenta um framework de avaliação adaptado de ferramentas em Java, utilizado tanto para a análise estática de documentação e funcionalidades quanto para a análise dinâmica por meio de testes cruzados entre ferramentas, a fim de mensurar a eficácia de cada uma na detecção de mutantes. Os resultados indicam que o Cosmic-Ray oferece o maior nível de personalização, enquanto o MutPy demonstra capacidades superiores de detecção de mutantes, superando, em conjunto, MutMut e Mutatest.

2.3 Trabalhos relacionados

A literatura de teste de software tem se desenvolvido na direção de estudos das possibilidades e limitações do uso de LLMs para geração de teste, o trabalho de Wang et al. (2024) apresenta um panorama abrangente sobre o uso de LLMs em testes de software. Os autores analisam 102 estudos recentes que exploram o uso de LLMs em tarefas que vão desde a geração de testes unitários, geração de oráculos de teste, até atividades mais avançadas como debugging e reparo de programas. A partir dessa análise, são discutidas as estratégias de uso desses modelos — como pretreinamento, fine-tuning e engenharia de prompts — e os diferentes tipos de tarefas de teste em que são aplicados, destacando

os avanços obtidos na geração automática de entradas, cobertura de testes e detecção de defeitos. O trabalho aqui desenvolvido se insere diretamente nesse cenário ao conduzir uma análise experimental sistemática do uso de um LLM específico para geração de testes unitários em Python, contribuindo para suprir parte dessas lacunas identificadas na literatura.

Em relação à geração de testes unitários, o artigo de Zhang et al. (2025) investiga como a presença de ruído em datasets compromete a eficácia de modelos baseados em aprendizado de máquina, especialmente LLMs, na geração de testes unitários. Os autores propõem o CleanTest, um método automático de filtragem que combina três estratégias: filtros sintáticos baseados em regras, filtros de relevância e um filtro de cobertura baseado em modelo, para eliminar ruídos como erros de sintaxe, métodos irrelevantes e baixa cobertura de código. A principal contribuição do estudo está na ênfase à qualidade dos dados como fator determinante no desempenho dos modelos de geração de testes, demonstrando que a remoção de instâncias ruidosas não apenas melhora métricas como cobertura de código e taxa de compilação, mas também reduz significativamente o tempo de treinamento.

O artigo de Schäfer et al. (2024) apresenta uma avaliação empírica do uso de LLMs especificamente para a geração automática de testes unitários. O estudo propõe uma técnica que utiliza LLMs sem necessidade de treinamento adicional, explorando apenas a elaboração cuidadosa de prompts que incluem a especificação e implementação das funções a serem testadas, bem como exemplos de uso extraídos da documentação. Os autores implementaram essa abordagem, a qual foi avaliada em larga escala em projetos JavaScript, alcançando cobertura de código superior às técnicas tradicionais. Os resultados indicam que LLMs possuem grande potencial para automatizar a geração de testes unitários, produzindo testes que não apenas são sintaticamente corretos, mas também próximos do estilo humano, contendo asserções relevantes.

Outras pesquisas também se aprofundam nessa especificidade de testes unitários utilizando LLMs, como no caso do trabalho de Konuk, Baglum e Yayan (2024), no qual foi realizado um extenso estudo empírico sobre a eficácia de LLMs na geração automática de testes unitários, com foco especial em modelos open-source. Os autores avaliam cinco LLMs distintos, utilizando diversos projetos Java e analisando múltiplas variáveis, como estilo de prompt, seleção de características de código e uso de técnicas de aprendizado em contexto (ICL), um mecanismo pelo qual o modelo é capaz de aprender uma tarefa nova durante a inferência ao receber exemplos diretamente no prompt, sem necessidade de ajuste nos seus pesos internos, permitindo assim a adaptação a diferentes contextos de geração de testes com maior flexibilidade.

Capítulo 3

Metodologia

3.1 Design do experimento

O design experimental deste trabalho é orientado pela metodologia Goal-Question-Metric (GQM), com cada seção do GQM detalhada a seguir.

Objetivo

Aprimorar e avaliar a eficácia, a capacidade de detecção de falhas e a relação custo-benefício do uso do ChatGPT-3.5-turbo para geração automática de testes em Python, comparando seu desempenho ao da ferramenta Pynguin e a conjuntos de testes pré-existentes. Os objetivos englobam a validade dos testes, cobertura, score de mutação e consumo de recursos.

Questões

RQ1. Quão capaz é o ChatGPT-3.5-turbo de gerar, de forma autônoma, casos de teste válidos para módulos em Python utilizando sua API?

RQ2. Como a adequação dos testes (cobertura, score de mutação) gerados pelo ChatGPT-3.5-turbo se compara à de uma ferramenta consolidada de geração automática de testes em Python (Pynguin) e a conjuntos de testes pré-existentes?

RQ3. Em que medida os conjuntos de testes gerados por LLMs detectam diferentes categorias de falhas, conforme representadas pelos operadores de mutação do MutPy e do Cosmic-Ray?

Métricas

- Para RQ1 (Capacidade):
  - Taxa de sucesso dos conjuntos de teste gerados (percentual de testes aprovados na validação via Pytest por programa e temperatura).
  - Número de tentativas de correção necessárias para obtenção de um conjunto de teste válido.
  - Custo total e médio de geração de suítes de teste bem-sucedidas e completas.
- Para RQ2 (Adequação):
  - Percentual médio de cobertura de decisão por programa e temperatura; comparado à Pynguin e aos conjuntos de teste pré-existentes.
  - Pontuação média de mutação obtida pelo MutPy e pelo Cosmic-Ray por programa e temperatura, comparada às pontuações de referência (baseline).
  - Desvio padrão das métricas de cobertura e score de mutação entre diferentes configurações de temperatura.
- Para RQ3 (Detecção de Falhas):
  - Score de mutação por operador do MutPy e do Cosmic-Ray (razão entre mutantes mortos e o total de mutantes por operador).
  - Identificação de operadores com score de mutação alta ($\geq$0.90) e baixa ($\leq$0.10) para conjuntos de teste gerados por LLM.
  - Taxas de cobertura e detecção para falhas de difícil identificação (por exemplo, mutações de operadores de comparação e de controle de fluxo).

O objetivo principal deste experimento é avaliar a capacidade dos LLMs de gerar casos de teste de forma autônoma. Optou-se pelo uso principal do ChatGPT, especificamente do modelo 3.5-turbo, devido à sua popularidade, uso em outros estudos (WANG et al., 2024) e baixo custo. Outros modelos mais avançados, como o ChatGPT 4.1[1] ou suas variantes, e o Claude Sonnet 4[2], podem apresentar desempenho superior, mas têm um custo consideravelmente mais elevado em relação ao 3.5-turbo. Além disso, nossos recursos computacionais limitados impossibilitam o uso local de modelos de código aberto, como o Llama 4[3].

Para este propósito, foram selecionados 40 projetos em Python provenientes de repositórios no GitHub[4,5,6]. Os critérios de seleção foram: 1) implementação em Python e 2)

[1] <https://platform.openai.com/docs/models/gpt-4.1>
[2] <https://www.anthropic.com/claude/sonnet>
[3] <https://ollama.com/library/llama4>
[4] <https://github.com/clair3st/Data-Structures>
[5] <https://github.com/keon/algorithms>
[6] <https://github.com/exterkamp/Python-Data-Structures>

presença de um conjunto de testes pré-existente utilizando UnitTest[7] ou Pytest[8]. Após a padronização dos projetos para o experimento, o conjunto de dados foi disponibilizado em um repositório.

A Tabela 1 lista os programas utilizados neste estudo, atribuindo uma identificação única (ID) a cada um. As colunas denominadas PROGRAM, PA, CL/MO, ME/FU, CC e SLOC representam, respectivamente, o nome de cada programa, o paradigma de desenvolvimento (estrutural — ST ou orientado a objetos — OO), o número de classes ou módulos, o número de métodos ou funções, a complexidade ciclomática máxima e o número de linhas de código-fonte sem comentários. Além disso, esses projetos incluem conjuntos de testes pré-existentes criados pelos próprios autores dos repositórios, que servirão como baseline para análises comparativas subsequentes.

De modo geral, observa-se que o conjunto de dados contém 22 programas Python implementados segundo o paradigma de Programação Estrutural e 18 segundo o paradigma de Programação Orientada a Objetos. Esses programas possuem, em média, 4,3 métodos ou funções, com complexidade ciclomática média de 4,6 e 32,2 linhas de código. Pode-se argumentar que os programas selecionados não representam completamente aqueles encontrados em aplicações reais.

No entanto, é importante salientar que o foco deste estudo está na avaliação da eficiência de geração de testes unitários (em nível de método ou função). Esses programas implementam diversas estruturas de dados e algoritmos de ordenação clássicos, o que os torna adequados para fins educacionais e experimentais no contexto de testes de software e análise de mutação. Tais programas, com níveis variados de complexidade, servem como base para avaliar o desempenho de LLMs na geração de testes. Parte-se do pressuposto de que, se o desempenho na geração de testes for insatisfatório nesse conjunto, dificilmente será superior em programas maiores e mais complexos.

A Figura 1 ilustra o fluxo do expeirmento. A geração de testes será realizada utilizando a API da OpenAI, com prompts projetados para solicitar a criação de programas de teste em Python no formato Pytest[9]. Cada teste gerado será executado para verificação de sua correção. Caso o teste gerado falhe ou produza erros, o modelo receberá feedback para revisar e corrigir os casos de teste. Esse processo será repetido até três vezes. Se, após a terceira tentativa, o teste ainda falhar na validação, o resultado será registrado como erro e incluído na análise estatística final das taxas de sucesso na geração de testes. O mesmo processo iterativo de validação aplica-se à verificação das saídas de execução via Pytest. Casos de teste bem-sucedidos sairão do ciclo de feedback assim que a validação for confirmada.

Além disso, o experimento avaliará diferentes valores de temperatura para o ChatGPT (temperatura refere-se ao grau de aleatoriedade na resposta, em que valores mais altos

[7] <https://docs.python.org/3/library/unittest.html>
[8] <https://pytest.org/>
[9] <https://docs.pytest.org>

Tabela 1 – Conjunto de programas Python usados no experimento

ID	PROGRAM	TY	CL/MO	ME/FU	CC_{MAX}	SLOC
P01	`breadth_first_search.py`	ST	1	2	11	39
P02	`binheap.py`	OO	1	5	5	38
P03	`binary_search_tree.py`	OO	1	7	4	54
P04	`bst2.py`	OO	2	21	7	152
P05	`combinations.py`	ST	1	2	5	26
P06	`complement.py`	ST	1	1	3	8
P07	`convert_base.py`	ST	1	2	7	14
P08	`deque.py`	OO	1	8	2	22
P09	`depth_first_search.py`	ST	1	1	11	24
P10	`dijkstras.py`	ST	1	1	7	27
P11	`dll.py`	OO	2	9	8	83
P12	`edit_distance.py`	ST	1	1	16	46
P13	`factorial.py`	ST	1	1	2	5
P14	`fenwick_tree.py`	OO	1	4	3	23
P15	`fibonacci.py`	ST	1	2	3	11
P16	`find_duplicates.py`	ST	1	3	5	30
P17	`first_missing_positive.py`	ST	1	1	6	13
P18	`fizz_buzz.py`	ST	1	1	5	2
P19	`graph1.py`	OO	4	16	4	85
P20	`graph2.py`	OO	1	10	3	28
P21	`hamming_ops.py`	ST	1	2	2	9
P22	`hash_map.py`	ST	1	7	4	42
P23	`heap.py`	OO	1	9	4	50
P24	`heapsort.py`	ST	1	3	4	24
P25	`linked_list1.py`	OO	1	5	8	57
P26	`linked_list2.py`	OO	2	8	6	50
P27	`longest_common_prefix.py`	ST	1	2	3	21
P28	`lru_cache.py`	OO	2	6	3	47
P29	`mergesort.py`	ST	1	1	1	24
P30	`naive_tree.py`	OO	1	6	5	49
P31	`permutations.py`	ST	1	1	1	10
P32	`priority_queue1.py`	OO	1	5	3	32
P33	`priority_queue2.py`	OO	1	4	2	17
P34	`a_queue.py`	OO	1	5	2	15
P35	`quicksort.py`	ST	1	1	1	17
P36	`rabin_karp_substring_search.py`	ST	1	1	8	18
P37	`radixsort.py`	ST	1	1	2	19
P38	`stack.py`	OO	1	3	2	8
P39	`trie.py`	OO	1	3	4	31
P40	`word_squares.py`	ST	1	1	2	18
AVG			1.2	4.3	4.6	32.2
SD			0.5	4.3	3.1	27.4

resultam em saídas mais variadas e valores mais baixos produzem respostas mais determinísticas). Serão testadas onze configurações de temperatura distintas, variando de 0.0 a 1.0, com intervalos de 0.1, a fim de identificar a configuração ideal para a geração de testes. Para cada valor de temperatura, serão gerados 30 conjuntos de testes, garantindo relevância estatística, totalizando 330 conjuntos de testes por programa em Python.

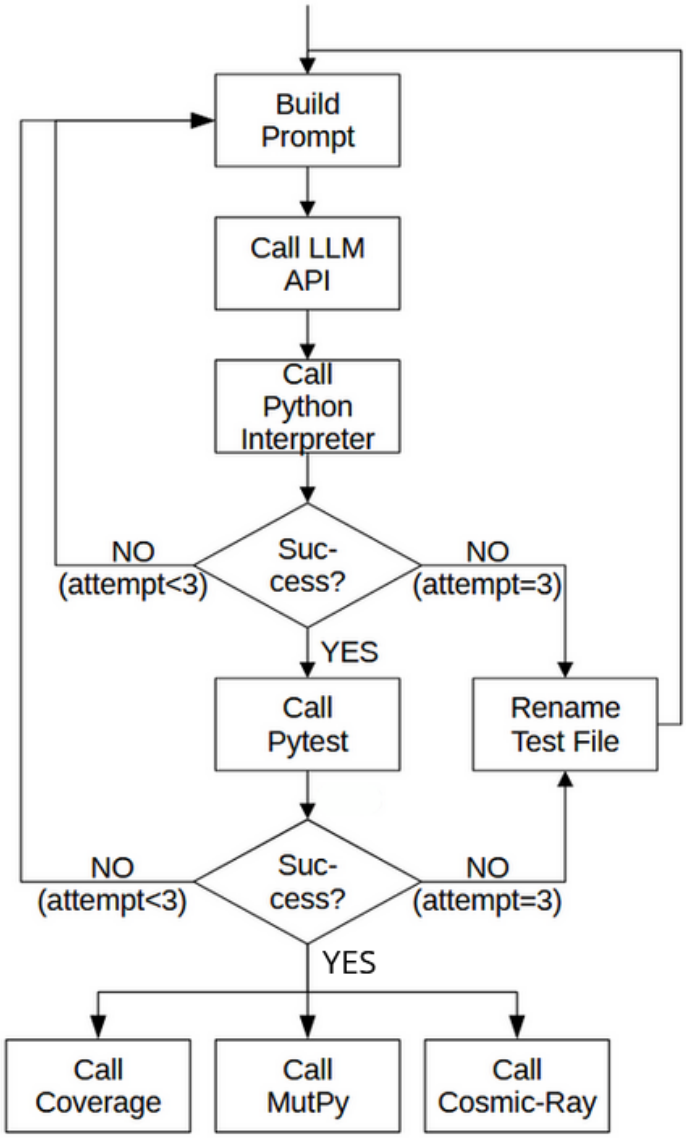

Figura 1 – Fluxo do experimento

A primeira métrica a ser avaliada será a capacidade do LLM de gerar casos de teste que sejam aprovados na validação com o Pytest. A segunda métrica corresponderá à cobertura de decisão de cada conjunto de teste, medida com o uso da ferramenta coverage do Python. A terceira métrica será a score de mutação, obtida por meio das ferramentas de mutação MutPy e Cosmic-Ray. Essas ferramentas foram selecionadas com base no estudo de Guerino et al. (2024), que realizou uma análise estática e dinâmica dessas e de outras duas ferramentas de teste de mutação para Python.

Para estabelecer comparações com as ferramentas atuais de geração automática de testes em Python, serão obtidas tanto a score de mutação quanto a cobertura dos testes gerados pelo Pynguin. Isso envolverá o uso de um conjunto de testes que combina aqueles gerados pelos algoritmos do Pynguin, os quais empregam algumas técnicas de inteligência artificial: Dynamosa, Mio, Mosa e Whole-Suite. Adicionalmente, serão coletados os dados

de mutação e cobertura dos conjuntos de testes originais que acompanham os programas selecionados.

3.2 Coleta de dados

Em relação à coleta de dados, foram utilizados scripts Python em nosso repositório no GitHub[10] para gerar casos de teste por meio da API da OpenAI. O sistema consistiu em enviar um prompt para a API e receber um conjunto de testes como resposta, o qual foi então convertido em um arquivo `.py`. A Listagem 3.1 apresenta o prompt utilizado para a geração do conjunto de testes. As variáveis empregadas no prompt são: `{clazz}` representa o módulo a ser testado, `{message_path_import}` inclui qualquer caminho adicional necessário para o teste funcionar e `{code}` possui o código em si.

```
{"role": "system", "content": "You are a senior software tester
 specialized in mutation testing."},
{"role": "user", "content": f"I am using mutation testing on the Python
 code below. I want to create tests that cover every module of the
 code. Do not leave any method untested. Give me more than one test per
 method. As a senior software tester, give me a set of test cases in
 the Pytest format. Correctly import the file {clazz} and its modules
 being tested. Utilize {message_path_import} to correctly import the
 file. Please, no additional information. Just the test cases.
 \n\n{code}"}
```

Listing 3.1 – Prompt do LLM para geração inicial do conjunto de testes

Para corrigir automaticamente erros de execução dos casos de teste, foi utilizado o prompt apresentado na Listagem 3.2, em que `{generated_tests}` representa os testes iniciais gerados pelo LLM, e `{error_message}` corresponde à saída retornada pelo terminal quando ocorre um erro.

```
{"role": "user", "content": f"The following Pytest test set results in
 these error messages. \n##########\n {generated_tests} \n##########\n.
 You must keep all the correct tests and fix the assertion in the
 failed tests according to the error below. \n\nError Message:
 {error_message} \n Give me the full test set with no comments, only
 the full Python code."}
```

Listing 3.2 – Prompt do LLM para feedback de erro

O arquivo gerado foi, posteriormente, executado utilizando o comando padrão do Python. Caso o código gerado apresentasse erros, um feedback era enviado de volta à API solicitando correções, com até três tentativas de ajuste utilizando o mesmo prompt. Se essas tentativas não fossem bem-sucedidas, os testes eram renomeados e salvos com

[10] <https://github.com/aurimrv/python_experiments2/>

a extensão `.py.err`, sendo registrados como erros em uma tabela. Em caso de sucesso, o conjunto de testes era executado com o Pytest para verificar se todas as execuções passavam integralmente; o mesmo procedimento de correção era aplicado caso essa etapa falhasse.

Esse ciclo de geração foi repetido 30 vezes para cada configuração de temperatura, variando de 0.0 a 1.0 em intervalos de 0.1, resultando em 330 conjuntos de teste por programa. Para a métrica de comparação, os 4 modelos selecionados foram passados pelo mesmo ciclo de geração com os mesmos prompts, sendo avaliados pelas mesmas métricas, formando mais uma base de comparação ao modelo principal do estudo. Os resultados obtidos serão apresentados na seção a seguir.

Capítulo 4

Resultado, análise e discussão

4.1 Resultados

A seguir, os resultados são apresentados após a coleta dos dados, dispostos em forma de tabelas para serem comparados com os critérios previamente estabelecidos.

A taxa de sucesso da geração de conjuntos de teste utilizando a API do ChatGPT 3.5 é apresentada na Tabela 2. Cada linha da tabela representa um programa, enquanto as colunas contêm os valores percentuais de testes gerados corretamente (aqueles que passam na etapa de verificação do Pytest) para cada temperatura. As duas últimas colunas mostram, respectivamente, a taxa média de sucesso para esse programa e a taxa média de sucesso desconsiderando valores nulos. É relevante observar que, em todas as tabelas de resultados, os valores iguais a zero e 100% foram destacados em vermelho e amarelo, respectivamente.

Também foram coletados dados referentes ao número de tokens[1], ao número de tentativas de correção para cada arquivo de conjunto de testes (bem-sucedidas ou não) e ao tempo de execução, todos disponíveis no repositório GitHub da pesquisa. Uma visualização parcial dos custos em tokens das chamadas à API é apresentada na Tabela 3 (os dados completos estão disponíveis no repositório). Os dados seguem a mesma lógica da tabela anterior, em que cada linha representa um programa e cada coluna indica o custo total em tokens para cada configuração de temperatura, considerando todas as mensagens enviadas e recebidas durante o processo de geração dos testes.

Com os testes gerados pelo LLM disponíveis, a etapa seguinte consistiu na utilização de diversos scripts em Python e Bash (também disponíveis no repositório) para executar

[1] Utilizamos a API da OpenAI para obter a estimativa de uso de tokens <https://help.openai.com/en/articles/6614209-how-do-i-check-my-token-usage>.

Tabela 2 – Taxa de sucesso da geração de testes para o ChatGPT-3.5-Turbo

ID	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	Suc. Médio	Suc. Médio s/ zero
P01	96.67	93.33	86.67	96.67	86.67	83.33	90.00	73.33	76.67	76.67	86.67	86.06	86.06
P02	0.00	0.00	0.00	0.00	3.33	3.33	6.67	6.67	3.33	10.00	0.00	3.03	5.56
P03	0.00	0.00	0.00	0.00	0.00	10.00	3.33	6.67	10.00	10.00	3.33	3.94	7.22
P04	100	100	100	100	100	96.67	96.67	90.00	80.00	73.33	93.33	93.64	93.64
P05	96.67	83.33	76.67	80.00	66.67	63.33	63.33	56.67	53.33	46.67	43.33	66.36	66.36
P06	63.33	73.33	83.33	50.00	60.00	46.67	40.00	60.00	50.00	50.00	56.67	57.58	57.58
P07	0.00	3.33	6.67	13.33	13.33	13.33	16.67	13.33	23.33	10.00	10.00	11.21	12.33
P08	6.67	3.33	13.33	13.33	10.00	10.00	6.67	13.33	13.33	13.33	16.67	10.91	10.91
P09	96.67	93.33	90.00	93.33	96.67	96.67	90.00	70.00	63.33	76.67	66.67	84.85	84.85
P10	0.00	0.00	0.00	0.00	0.00	3.33	3.33	3.33	6.67	3.33	10.00	2.73	5.00
P11	23.33	23.33	16.67	23.33	26.67	46.67	30.00	16.67	33.33	16.67	30.00	26.06	26.06
P12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
P13	100	100	100	96.67	100	100	100	100	93.33	93.33	90.00	97.58	97.58
P14	0.00	3.33	3.33	6.67	3.33	6.67	3.33	3.33	0.00	0.00	6.67	3.33	4.58
P15	100	100	100	100	100	100	100	93.33	96.67	93.33	93.33	97.88	97.88
P16	100	100	100	100	96.67	90.00	76.67	56.67	36.67	56.67	63.33	79.70	79.70
P17	100	100	100	100	100	100	96.67	100	100	96.67	100	99.39	99.39
P18	100	100	100	100	96.67	100	100	96.67	100	100	90.00	96.03	96.03
P19	36.67	36.67	16.67	16.67	20.00	26.67	23.33	26.67	36.67	20.00	20.00	25.45	25.45
P20	0.00	0.00	3.33	3.33	6.67	13.33	13.33	3.33	16.67	13.33	10.00	7.58	9.26
P21	60.00	83.33	63.33	66.67	73.33	70.00	73.33	76.67	76.67	63.33	66.67	70.30	70.30
P22	30.00	20.00	23.33	20.00	30.00	43.33	56.67	36.67	56.67	63.33	50.00	39.09	39.09
P23	36.67	33.33	23.33	30.00	33.33	26.67	30.00	10.00	23.33	6.67	30.00	25.76	25.76
P24	0.00	0.00	0.00	0.00	3.33	6.67	33.33	33.33	40.00	26.67	43.33	16.97	26.67
P25	100	93.33	70.00	60.00	36.67	43.33	53.33	43.33	43.33	40.00	50.00	57.58	57.58
P26	6.67	23.33	10.00	13.33	26.67	23.33	43.33	53.33	50.00	33.33	40.00	29.39	29.39
P27	96.67	100	100	100	96.67	100	96.67	100	100	93.33	96.67	98.18	98.18
P28	93.33	80.00	73.33	60.00	63.33	46.67	50.00	53.33	36.67	60.00	63.33	61.82	61.82
P29	43.33	23.33	46.67	43.33	56.67	56.67	56.67	56.67	56.67	46.67	46.67	48.48	48.48
P30	0.00	0.00	0.00	0.00	0.00	0.00	10.00	0.00	6.67	3.33	10.00	2.73	7.50
P31	3.33	3.33	20.00	16.67	26.67	30.00	26.67	26.67	40.00	16.67	26.67	21.52	21.52
P32	0.00	0.00	0.00	3.33	3.33	10.00	6.67	3.33	13.33	13.33	6.67	5.45	7.50
P33	96.67	93.33	86.67	80.00	76.67	56.67	76.67	63.33	83.33	56.67	56.67	75.15	75.15
P34	100	100	100	93.33	86.67	86.67	90.00	90.00	80.00	86.67	83.33	90.61	90.61
P35	96.67	100	96.67	96.67	86.67	100	86.67	90.00	73.33	83.33	66.67	88.79	88.79
P36	6.67	6.67	20.00	6.67	13.33	40.00	26.67	10.00	23.33	33.33	46.67	21.21	21.21
P37	0.00	0.00	0.00	0.00	0.00	0.00	3.33	0.00	0.00	3.33	0.00	0.61	3.33
P38	0.00	0.00	3.33	10.00	0.00	0.00	6.67	6.67	3.33	6.67	3.33	3.64	5.33
P39	0.00	0.00	0.00	0.00	0.00	3.33	6.67	6.67	3.33	13.33	20.00	4.85	8.89
P40	0.00	0.00	0.00	0.00	0.00	0.00	3.33	0.00	0.00	3.33	0.00	0.61	3.33
AVG	33.34	23.33	21.67	21.67	28.34	41.67	36.67	35.00	38.34	33.33	43.33	27.73	43.97
SD	44.85	44.28	42.2	41.06	39.43	37.77	36.25	35.87	33.35	33.41	32.83	37.26	35.85

individualmente cada teste gerado por meio das ferramentas Coverage, MutPy e Cosmic-Ray. Esse processo permitiu obter as métricas individuais de cobertura e score de mutação para cada conjunto de testes, possibilitando a comparação com a baseline. No caso dos conjuntos de teste gerados pelo LLM, os dados apresentados nas tabelas representam a média dos resultados obtidos entre todos os conjuntos de teste bem-sucedidos.

Conforme mencionado, os testes de baseline são gerados pelos algoritmos inteligentes do Pynguin, bem como pelos testes pré-existentes nos programas. Essas linhas de base também foram analisadas por meio das ferramentas de cobertura e mutação, tendo suas pontuações posteriormente comparadas com as médias dos conjuntos de teste gerados pelo LLM.

As informações de cobertura, assim como as pontuações de mutação obtidas pelas ferramentas MutPy e Cosmic-Ray, são apresentadas, respectivamente, nas Tabelas 4, 5 e 6. Cada linha da seção de cobertura exibe os valores médios de cobertura de ramos para

Tabela 3 – Custo total de tokens para geração de testes do ChatGPT-3.5-Turbo

.

ID	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	1,0	SOMA
P01	37,654	41,493	48,441	37,280	48,871	52,200	44,164	61,390	56,862	57,092	47,296	532,743
P02	117,984	119,915	120,832	120,464	118,813	130,534	124,920	125,676	119,327	116,866	126,034	1,341,365
P03	122,036	125,327	124,792	121,635	123,597	114,534	121,006	109,026	112,058	114,723	120,862	1,309,596
P04	76,410	76,268	76,156	80,126	76,287	85,698	91,644	97,793	121,442	127,973	107,003	1,016,800
P05	31,964	43,884	49,308	49,319	55,324	61,589	66,262	67,033	75,940	66,382	78,804	645,809
...						...						...
P12	154,017	215,690	163,955	146,097	185,866	147,638	137,502	158,579	131,509	151,758	135,194	1,727,805
P13	12,227	12,207	12,333	14,478	12,482	12,491	12,627	13,378	16,671	15,978	19,641	154,513
...						...						...
P28	66,951	84,859	90,918	120,261	106,519	129,173	128,404	119,671	141,580	111,556	114,048	1,213,940
P29	110,408	132,269	106,433	100,042	88,492	87,168	91,178	90,632	89,440	93,424	97,458	1,086,944
P30	142,685	148,663	150,004	145,668	145,456	148,806	130,862	130,355	135,922	146,731	133,870	1,559,022
P31	420,332	433,404	218,054	198,090	214,811	205,959	106,458	116,770	99,189	103,064	103,074	2,219,205
P32	144,507	134,558	128,340	118,286	119,909	118,067	118,839	130,309	117,661	106,077	119,895	1,356,448
P33	23,179	26,142	29,324	36,817	37,872	58,351	38,226	54,181	34,705	58,080	62,382	459,259
P34	20,622	20,615	20,692	26,107	31,548	32,166	30,363	28,634	36,126	32,907	33,536	313,316
P35	31,756	27,497	31,224	30,671	43,435	26,699	43,882	38,084	59,740	46,377	65,843	445,208
P36	104,324	96,941	90,173	100,329	91,226	78,821	83,048	90,724	90,948	86,741	74,179	987,454
P37	128,793	130,451	129,370	130,996	135,692	133,256	130,332	124,633	127,874	123,464	128,704	1,423,565
P38	112,440	109,416	106,142	80,128	94,434	95,744	87,254	94,865	92,124	90,180	92,428	1,055,155
P39	119,219	114,696	116,654	116,766	118,873	117,267	112,429	118,365	116,652	108,801	105,592	1,265,314
P40	133,871	138,090	136,055	128,402	129,288	118,699	118,554	115,005	117,461	109,288	114,763	1,359,476
SUM	3,813,172	3,894,842	3,710,459	3,680,756	3,723,089	3,640,839	3,473,508	3,644,665	3,595,037	3,686,263	3,641,439	40,504,069
AVG	95,329.3	97,371.1	92,761.5	92,018.9	93,077.2	91,021.0	86,837.7	91,116.6	89,875.9	92,156.6	91,036.0	1,012,601.7
SD	74,096.3	76,492.2	54,389.9	52,451.8	52,183.9	48,841.2	42,508.5	43,706.4	40,174.4	42,759.8	39,348.2	530,024.8

cada temperatura. No caso do Pynguin e dos testes pré-existentes, apenas o valor absoluto de cobertura é fornecido, uma vez que apenas um conjunto de testes é considerado. Para as ferramentas de mutação, as linhas apresentam as médias das pontuações de mutação por ferramenta, seguindo a mesma lógica de uso de médias para os testes gerados pelo LLM e valores absolutos para o Pynguin e os conjuntos de testes pré-existentes.

É importante observar que algumas das médias referentes aos conjuntos de teste gerados pelo LLM podem apresentar falta de precisão, especialmente nos casos em que apenas um número reduzido de testes foi bem-sucedido, ou seja, aqueles com baixos percentuais na Tabela 2.

Vale destacar que ocorreram alguns casos excepcionais durante o processo de geração de testes. Primeiramente, o ChatGPT não conseguiu gerar testes válidos para o programa `edit_distance.py` (P12), sendo este o único caso em que tal situação ocorreu, embora outros programas tenham apresentado um número menor de conjuntos de teste gerados. Outra exceção envolveu o programa `stack.py` (P38), para o qual a ferramenta Cosmic-Ray não conseguiu criar mutantes. Após a análise do programa, verificou-se que isso provavelmente se deve ao fato de essa implementação de pilha ser uma extensão do programa `linked_list2.py`, o que implica que não há linhas de código consideradas adequadas pela Cosmic-Ray para a geração de mutantes.

Tabela 4 – Cobertura de decisão para ChatGPT-3.5-turbo

ID	LLM - Temperaturas											LLM	Não-LLM	
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	ALL	Pynguin	Pre-existente
P01	90.3	89.9	90.2	89.5	89.3	90.4	89.1	87.8	87.2	88.7	86.7	100	63.6	63.6
P02	0.0	0.0	0.0	0.0	94.4	94.4	83.3	66.7	72.2	79.6	0.0	96.0	94.4	94.4
P03	0.0	0.0	0.0	0.0	0.0	90.0	90.0	82.5	88.3	85.0	90.0	97.0	100	100
P04	74.0	73.0	72.6	72.3	73.0	74.0	71.1	72.5	74.3	70.6	64.9	79.0	88.8	88.8
P05	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P06	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P07	0.0	100	100	93.8	84.4	100	95.0	71.9	87.5	75.0	87.5	100	100	100
P08	100	100	100	100	100	83.3	100	100	100	81.3	90.0	100	100	100
P09	100	99.5	99.7	99.2	99.5	99.5	99.5	100	98.5	97.8	96.4	100	100	100
P10	0.0	0.0	0.0	0.0	0.0	91.7	91.7	91.7	87.5	91.7	83.3	97.0	0.0	92.9
P11	62.7	63.9	60.6	64.7	60.8	66.7	63.9	68.9	61.9	65.6	67.6	99.0	97.2	97.2
P12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	86.4	88.5
P13	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P14	0.0	100	100	100	100	100	100	100	0.0	0.0	100	100	100	100
P15	100	100	100	100	100	100	100	100	100	100	100	100	83.3	100
P16	87.5	87.5	87.5	87.5	87.5	87.7	88.0	87.9	88.6	89.3	90.8	100	100	100
P17	100	100	100	100	100	100	100	100	99.4	100	99.4	100	100	100
P18	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P19	67.8	63.2	67.7	66.8	70.1	74.2	64.3	71.0	65.5	68.6	70.1	95.0	92.1	97.7
P20	0.0	0.0	25.0	25.0	25.0	25.0	37.5	25.0	47.5	46.9	25.0	100	100	100
P21	75.0	75.0	76.3	76.3	80.7	83.3	79.6	85.9	90.2	92.1	88.8	100	100	100
P22	81.3	81.8	78.6	80.3	81.8	80.1	80.0	78.1	76.5	79.4	79.4	98.0	88.9	95.5
P23	72.7	73.6	76.0	76.8	79.5	78.4	79.8	77.3	77.9	86.4	80.3	100	100	100
P24	0.0	0.0	0.0	0.0	100	100	100	100	100	100	100	100	100	100
P25	73.1	73.1	72.5	72.4	73.1	72.8	71.2	75.7	73.1	74.7	74.9	98.0	80.8	92.3
P26	72.2	75.4	77.8	76.4	73.6	77.0	74.8	78.5	75.6	77.2	75.9	100	100	100
P27	91.8	91.3	93.8	92.5	94.4	94.2	95.3	95.0	96.3	96.9	97.0	100	100	100
P28	64.3	67.2	72.2	74.3	75.0	81.3	77.5	81.3	83.0	79.9	84.2	100	100	100
P29	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P30	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	100	100	100	100	21.4	100
P31	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P32	0.0	0.0	0.0	70.0	70.0	70.0	70.0	70.0	70.0	70.0	70.0	88.0	90.0	90.0
P33	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	89.0	50.0	50.0
P34	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P35	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P36	91.7	91.7	93.1	91.7	93.8	93.1	92.7	94.5	91.7	92.5	92.3	100	100	100
P37	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	100	0.0	100	100	100
P38	0.0	0.0	100	100	0.0	0.0	100	100	100	100	100	100	100	100
P39	0.0	0.0	0.0	0.0	0.0	100	100	96.4	100	100	97.6	100	92.9	42.9
P40	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	100	0.0	100	100	100
AVG	56.4	61.4	64.8	66.5	68.9	76.4	86.1	77.7	78.6	83.5	78.6	95.9	90.8	94.8
SD	43.7	42.6	41.1	39.6	38.3	33.1	20.9	30.8	30.1	24.1	30.9	16.1	21.4	13.0

4.2 Análise e discussão

Sucesso e custo da geração dos conjuntos de teste

Em relação à taxa de sucesso da geração de testes pelo modelo ChatGPT-3.5-turbo, observou-se uma considerável variabilidade nos resultados, conforme apresentado na Tabela 2. Para alguns programas, foram obtidas diversas instâncias de 100% de eficiência na criação de testes que passaram na verificação do Pytest em diferentes configurações

Tabela 5 – Score de mutação MutPy para ChatGPT-3.5-turbo

ID	LLM - Temperaturas											LLM	Não-LLM	
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	ALL	Pynguin	Pre-existente
P01	72.5	72.5	73.0	73.6	71.1	70.2	72.4	73.1	70.5	65.9	69.4	92.1	34.9	67.4
P02	0.0	0.0	0.0	0.0	94.9	94.9	90.7	85.6	85.6	88.3	0.0	97.2	96.9	95.0
P03	0.0	0.0	0.0	0.0	0.0	88.7	88.7	85.2	86.9	84.5	88.7	88.7	77.5	91.6
P04	80.4	79.2	78.7	78.4	79.3	80.3	77.4	78.6	80.9	77.6	71.6	91.8	81.6	93.6
P05	99.8	98.9	98.8	98.2	97.7	98.1	97.6	95.8	97.6	95.2	96.7	100	32.1	45.6
P06	92.3	93.7	93.5	92.8	95.3	95.6	96.2	97.4	98.5	96.9	97.7	100	100	93.8
P07	0.0	92.9	92.9	88.1	81.6	91.7	89.1	71.4	82.7	74.6	80.2	97.5	97.6	90.9
P08	92.1	92.1	92.1	92.1	92.1	92.1	92.1	92.1	69.7	91.5	92.1	93.1	89.5	94.7
P09	80.2	79.9	80.5	79.1	77.9	79.9	79.2	80.1	80.0	77.0	77.2	95.6	71.1	82.9
P10	0.0	0.0	0.0	0.0	0.0	92.5	92.5	92.5	90.0	95.0	86.7	95.0	0.0	89.4
P11	77.1	79.4	76.3	77.9	74.4	78.4	76.7	79.4	72.7	74.8	77.1	96.7	87.7	90.9
P12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.9	83.7
P13	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P14	0.0	98.4	96.8	98.4	98.4	98.4	98.4	98.4	0.0	0.0	98.4	98.0	96.8	95.9
P15	94.7	93.5	94.0	94.2	94.5	93.8	94.3	93.9	93.6	93.6	94.1	95.0	75.0	96.7
P16	86.4	86.4	86.4	86.4	86.4	86.3	86.8	86.9	86.6	86.9	87.8	97.7	43.2	86.3
P17	92.9	92.9	92.8	91.9	92.4	92.3	91.2	91.5	91.5	91.7	89.7	96.0	88.0	90.6
P18	100	100	100	100	100	100	100	100	100	100	100	100	12.5	100
P19	76.1	74.3	72.7	72.2	71.0	78.5	73.5	74.5	71.4	75.2	78.9	93.1	84.2	92.9
P20	0.0	0.0	63.2	70.2	70.2	68.4	69.7	63.2	76.5	75.0	62.0	90.2	63.2	96.5
P21	87.5	87.5	88.2	88.1	90.3	91.7	89.8	92.9	94.8	95.7	94.1	100	100	100
P22	89.2	89.5	88.4	89.5	87.6	89.3	84.8	80.1	76.3	83.2	82.6	92.5	92.1	93.1
P23	75.7	77.3	80.8	80.9	83.0	79.7	82.4	81.8	83.3	84.9	81.5	94.0	92.1	93.9
P24	0.0	0.0	0.0	0.0	80.0	80.0	80.0	79.8	79.6	79.7	79.6	80.0	52.5	80.0
P25	77.5	77.5	77.5	76.7	77.4	77.5	78.0	78.8	79.3	80.9	80.7	97.3	90.0	95.0
P26	85.4	87.2	90.3	88.0	86.5	87.7	88.1	89.4	87.6	89.0	87.7	100	91.7	98.2
P27	93.5	93.5	93.5	93.8	94.2	93.6	93.0	93.3	92.9	93.2	91.8	96.2	92.9	90.6
P28	84.1	84.9	86.1	85.9	86.4	87.8	86.3	86.9	87.2	85.9	87.1	94.7	86.7	84.5
P29	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	96.9	69.4	95.1
P30	0.0	0.0	0.0	0.0	0.0	0.0	84.4	0.0	83.9	83.9	83.9	83.9	43.6	97.1
P31	93.8	93.8	93.8	92.5	93.8	93.8	91.4	90.6	92.7	92.5	93.0	92.9	37.5	94.7
P32	0.0	0.0	0.0	86.8	86.8	87.4	86.8	86.8	85.4	87.3	86.8	89.4	77.4	77.8
P33	87.4	87.5	86.7	85.9	85.9	85.8	85.3	85.5	86.0	85.5	86.8	82.4	83.3	80.7
P34	87.1	87.0	87.5	87.7	88.1	88.8	88.1	87.3	87.5	88.1	87.7	88.2	69.6	91.3
P35	91.3	91.3	91.3	91.3	91.3	91.3	91.3	91.3	91.1	91.1	91.1	91.3	52.2	50.0
P36	94.6	94.6	95.2	94.6	95.5	95.1	95.1	95.2	94.6	94.8	94.8	98.2	60.7	92.1
P37	0.0	0.0	0.0	0.0	0.0	0.0	93.6	0.0	0.0	93.6	0.0	92.9	61.3	91.4
P38	0.0	0.0	91.7	91.7	0.0	0.0	91.7	91.7	91.7	91.7	91.7	87.5	16.7	100
P39	0.0	0.0	0.0	0.0	0.0	98.2	98.2	91.1	96.4	92.9	95.2	98.0	82.1	40.0
P40	0.0	0.0	0.0	0.0	0.0	0.0	82.6	0.0	0.0	87.0	0.0	85.0	56.5	92.0
AVG	57.2	62.1	66.0	68.1	70.0	77.6	85.9	78.5	78.1	83.0	78.5	91.5	69.8	87.9
SD	43.0	41.7	39.4	37.9	36.3	30.7	15.9	27.7	27.6	20.8	27.8	15.7	26.3	14.1

de temperatura. No entanto, ao analisar os valores médios de geração, esses geralmente permanecem abaixo de 28% de sucesso. Isso se deve, principalmente, à presença de inúmeros programas para os quais o LLM não conseguiu gerar testes válidos, resultando em múltiplos casos com 0% de sucesso ou percentuais muito baixos.

Considerando as diferentes configurações de temperatura para a geração dos testes, observa-se que respostas menos determinísticas (temperaturas iguais ou superiores a 0.5) tendem a produzir, em média, um número maior de testes bem-sucedidos. Destacam-se,

Tabela 6 – Score de mutação Cosmic-Ray para ChatGPT-3.5-turbo

ID	LLM - Temperaturas											LLM	Não-LLM	
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	ALL	Pynguin	Pre-existente
P01	53.8	54.8	55.1	56.1	53.7	54.5	56.5	56.7	53.3	49.1	53.9	85.0	41.1	82.8
P02	0.0	0.0	0.0	0.0	94.2	95.3	92.1	88.3	89.5	91.4	0.0	97.7	98.3	97.1
P03	0.0	0.0	0.0	0.0	0.0	87.3	91.2	83.8	86.3	79.4	85.3	91.2	88.2	91.2
P04	69.6	69.2	69.2	69.1	69.2	69.7	68.6	69.0	70.4	68.3	63.7	86.8	77.5	86.0
P05	98.8	98.4	98.3	98.4	98.5	98.4	98.4	98.4	98.4	98.2	98.4	100	48.3	100
P06	96.3	96.3	96.7	96.3	97.2	100	100	98.6	100	100	99.1	100	100	96.3
P07	0.0	90.4	92.0	87.0	81.8	88.6	87.8	74.6	83.0	76.8	80.3	96.0	94.4	91.2
P08	100	100	100	100	100	100	100	100	75.0	100	100	100	100	100
P09	62.0	62.5	62.5	60.2	59.5	61.0	60.4	61.2	62.5	60.9	61.9	94.0	60.8	85.5
P10	0.0	0.0	0.0	0.0	0.0	82.9	80.0	82.9	78.6	85.7	79.1	91.4	0.0	88.6
P11	55.0	55.3	53.2	57.3	52.9	56.4	53.3	54.8	51.2	54.2	53.2	91.9	87.4	83.8
P12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	60.8	85.6
P13	94.6	94.4	95.7	97.3	97.0	97.4	97.4	98.2	96.6	96.6	98.6	100	100	97.2
P14	0.0	99.3	84.5	99.3	99.3	98.6	97.9	98.6	0.0	0.0	97.2	99.3	96.5	96.5
P15	99.8	99.2	99.4	99.3	99.2	98.9	98.7	98.5	98.6	98.4	98.6	100	56.8	100
P16	69.4	69.4	69.4	69.4	69.5	69.6	70.6	70.7	71.3	72.2	71.8	87.1	43.6	71.0
P17	88.0	88.0	88.9	90.5	85.5	91.7	84.5	87.2	89.9	89.2	86.8	96.4	83.3	88.1
P18	97.1	97.1	97.1	97.1	97.1	97.1	97.1	97.0	96.9	97.1	97.1	97.1	17.4	94.2
P19	65.2	59.4	56.5	56.1	55.1	67.0	57.1	60.8	52.9	63.9	66.8	89.8	89.8	89.8
P20	0.0	0.0	33.3	33.3	33.3	33.3	33.3	33.3	53.3	41.7	33.3	100	66.7	100
P21	84.6	85.5	84.6	86.7	89.1	94.9	88.1	92.3	94.0	93.9	89.1	100	100	96.6
P22	91.1	91.3	90.3	91.3	86.9	90.8	83.1	73.5	67.9	78.5	82.6	94.8	97.1	93.6
P23	62.6	62.6	66.4	74.8	68.9	57.0	80.7	0.0	82.2	70.7	71.6	91.6	87.9	86.6
P24	0.0	0.0	0.0	0.0	85.7	85.7	85.7	85.7	85.7	85.9	85.9	85.7	66.7	85.7
P25	75.0	75.0	75.1	73.9	74.6	73.6	72.4	76.1	76.2	79.6	80.0	93.2	77.3	93.2
P26	85.4	85.1	85.7	85.4	84.9	85.1	83.9	87.4	85.7	86.1	83.9	95.8	84.7	95.8
P27	84.6	85.5	85.0	85.6	86.1	84.9	83.3	82.5	83.0	82.3	81.3	93.3	83.3	86.7
P28	70.9	73.1	75.0	74.8	75.9	80.5	78.0	77.2	79.7	77.8	79.2	100	100	73.1
P29	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	56.0	96.0
P30	0.0	0.0	0.0	0.0	0.0	0.0	24.1	0.0	24.1	24.1	24.1	24.1	86.2	100
P31	100	100	100	100	100	100	100	87.5	100	100	100	100	87.5	100
P32	0.0	0.0	0.0	47.5	47.5	46.7	47.5	47.5	46.3	47.5	47.5	50.0	40.0	47.5
P33	84.6	84.6	84.6	84.6	84.6	84.6	84.6	84.2	84.3	84.6	85.5	92.3	61.5	84.6
P34	100	100	100	100	96.2	100	96.3	92.6	83.3	88.5	92.0	100	100	100
P35	88.3	88.3	88.2	88.0	88.0	88.1	88.0	87.8	87.6	87.7	87.7	88.3	60.0	88.3
P36	85.0	86.5	86.5	85.9	87.3	87.0	86.4	87.2	86.4	86.3	86.0	92.9	53.0	86.1
P37	0.0	0.0	0.0	0.0	0.0	0.0	92.1	0.0	0.0	92.1	0.0	92.1	60.3	85.7
P38	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
P39	0.0	0.0	0.0	0.0	0.0	93.1	93.1	91.4	89.7	92.2	93.1	93.1	41.4	10.3
P40	0.0	0.0	0.0	0.0	0.0	0.0	87.5	0.0	0.0	87.5	0.0	87.5	79.2	87.5
AVG	54.0	58.7	59.3	61.1	64.9	72.4	76.9	69.1	69.0	74.2	69.8	86.9	70.8	85.6
SD	41.9	40.9	39.8	39.0	36.6	32.2	25.6	33.1	31.3	27.7	32.1	24.4	26.9	21.1

especialmente, as temperaturas 0.5 e 1.0, que superaram a média em 41,67% e 43,33%, respectivamente.

As cobranças da API da OpenAI são calculadas por milhão de tokens, e os preços variam conforme o modelo. Para o ChatGPT-3.5-turbo no momento do experimento, o custo é de $0,50 por milhão de tokens de entrada e $1,50 por milhão de tokens de saída, o que resulta em um custo médio de $0.0000005 por token de entrada e $0.0000015 por token de saída. Assim, adotou-se um custo médio de $0.0000010 por token nas estimativas

de custo.

Considerando o custo total do experimento, foram utilizados 40.504.069 tokens para gerar todos os testes, resultando em um custo aproximado de $40,50. Para estimar o custo de um conjunto de teste bem-sucedido, pode-se considerar um dos programas com taxa de sucesso de 100%, como o `factorial.py` (P13) na temperatura 0.5. Dividindo-se o total de tokens utilizados para esse caso (12.491/30), obtém-se aproximadamente 416 tokens por conjunto de teste bem-sucedido, correspondendo a um custo de cerca de $0.000416.

Em um cenário de pior caso, seria necessário gerar os conjuntos de teste 330 x 6 vezes (devido a falhas na geração dos testes em Python e rejeições de conjuntos pelo Pytest), o que acarretaria um custo significativamente mais alto. Embora isso não tenha ocorrido neste experimento, pode-se citar o programa `edit_distances.py` (P12) como exemplo, uma vez que ele não produziu nenhum teste gerado com sucesso. Nesse caso, considerando a temperatura 0.5, o custo por teste foi de $0.004921, levando em conta todas as tentativas de correção.

Respondendo à RQ1, o modelo ChatGPT-3.5-turbo permitiu a criação de conjuntos de teste válidos para 39 dos 40 programas em pelo menos um parâmetro de temperatura, com custo razoável, especialmente ao se considerar a geração de um único conjunto de teste por temperatura. A taxa média de sucesso ficou abaixo de 28%, o que indica que, em geral, o modelo criou dois casos de teste válidos a cada dez gerados. Ainda assim, o custo total é consideravelmente baixo. Mesmo no pior cenário, considerando a geração incremental dos conjuntos de teste e o uso de todas as temperaturas (330 conjuntos no experimento), o custo total foi de aproximadamente $40,50.

Adequação dos conjuntos de teste

Em relação à cobertura de decisão (Tabela 4), quanto maior a taxa de sucesso dos conjuntos de teste, melhor tende a ser a cobertura obtida. O alto desvio padrão evidencia uma variação significativa nas médias entre diferentes temperaturas e programas. A temperatura com melhor desempenho em termos de cobertura de decisão foi 0.6, com média de 86,10% e desvio padrão de 20,88.

Com as linhas de base disponíveis para comparação, pode-se inferir que, no escopo deste experimento, o conjunto de testes gerado pelo ChatGPT-3.5-turbo para cada temperatura individualmente não atinge o desempenho da ferramenta automática de geração de testes para Python, nem dos testes personalizados originais de cada programa. A cobertura de decisão obtida pelo Pynguin e pelos testes pré-existentes se destaca, sendo 4,65% e 8,74% superiores à melhor média obtida (86,10%), respectivamente, e 34,39% e 38,48% superiores à pior média (56,36%).

Entretanto, conforme mostrado na coluna “LLM ALL”, que reúne todos os conjuntos de teste gerados em todas as temperaturas, a cobertura melhora para um nível superior

ao do Pynguin e dos testes pré-existentes em quase todos os programas analisados, com poucas exceções. Isso sugere que variar a temperatura no modelo LLM é fundamental para gerar casos de teste diversificados, aumentando a cobertura e a capacidade de detecção de falhas. A cobertura média final desse conjunto foi de 95,9%, 1,1% acima da média dos testes pré-existentes.

No que se refere ao score de mutação, é importante observar que não foram determinados os mutantes equivalentes entre os mutantes vivos. Nesse sentido, as pontuações de mutação poderiam ser maiores para todos os conjuntos após a identificação dos mutantes equivalentes. Entretanto, como foi utilizado o mesmo conjunto de mutantes para todas as execuções, a comparação permanece justa, desde que os possíveis mutantes equivalentes permaneçam vivos em todos os casos.

A análise inicia-se com o MutPy (Tabela 5), onde se observa uma redução acentuada nos casos de 100%, embora as médias se tornem mais próximas entre os testes gerados pelo LLM e as linhas de base. Os testes pré-existentes superam a melhor média do LLM em apenas 2,03%, enquanto o Pynguin apresenta resultados intermediários. Isso indica que, apesar das dificuldades na geração de testes válidos e na obtenção de cobertura total de decisão, os conjuntos de teste gerados pelo LLM para cada temperatura individualmente ainda são eficazes na eliminação de mutantes do MutPy.

Tendência semelhante foi observada para os mutantes gerados pela ferramenta Cosmic-Ray (Tabela 6); porém, as pontuações gerais de mutação tendem a ser menores. Essa diferença pode estar relacionada à natureza dos mutantes criados pela ferramenta, conforme apontado no estudo de Guerino e Vincenzi (2023), no qual os resultados da Cosmic-Ray também foram inferiores aos do MutPy, sendo esse um aspecto que pode ser explorado em análises futuras para avaliar a capacidade do LLM em eliminar mutantes específicos ou identificar mutantes equivalentes.

A mesma observação feita para a cobertura de decisão aplica-se ao score de mutação: a temperatura 0.6 produz os melhores resultados médios tanto para o MutPy quanto para o Cosmic-Ray, com o menor desvio padrão. Ao combinar todos os conjuntos gerados pelo LLM, também se observou uma melhora de aproximadamente 10% em relação ao melhor score de mutação obtida na temperatura 0.6, superando, em média, tanto o Pynguin quanto os testes pré-existentes.

Respondendo à RQ2, o LLM apresenta desempenho significativamente superior ao Pynguin na detecção de falhas no modelo MutPy e desempenho moderadamente superior no modelo Cosmic-Ray, considerando os conjuntos de teste de cada temperatura individualmente. Esses resultados sugerem que o Pynguin exibe comportamento semelhante ao de geradores automáticos de testes para Java, conforme observado em outros estudos, especialmente quanto à sua limitação na eliminação de mutantes (VINCENZI et al., 2016; ARAUJO; VINCENZI, 2020). O Pynguin é mais eficiente em gerar con-

juntos de teste com alta cobertura de código e decisão, mas menos eficaz em selecionar testes capazes de detectar falhas modeladas por operadores de mutação. A combinação de casos de teste gerados em diferentes temperaturas também se mostra importante para aprimorar a detecção de falhas, considerando os modelos de mutação do MutPy e do Cosmic-Ray.

Capacidade de detecção de falhas

Por fim, avaliou-se a capacidade de detecção de falhas do conjunto completo de testes gerados com base no LLM. As Tabelas 7 e 8 apresentam os dados coletados para o MutPy e o Cosmic-Ray, respectivamente. Devido a limitações de espaço, os dados completos do Cosmic-Ray estão disponíveis no repositório[2].

Tabela 7 – Detecção de falhas de operadores de mutação do MutPy

Operador	**Mortos**	**Vivos**	**Score**
AOD	815	118	0.87
AOR	17255	3412	0.83
ASR	4490	933	0.83
BCR	638	375	0.63
CDI	13511	2765	0.83
COD	6545	1040	0.86
COI	27731	4162	0.87
EHD	748	68	0.92
EXS	51	765	0.06
LCR	2308	2010	0.53
LOR	1174	79	0.94
OIL	9896	2450	0.80
RIL	5114	2035	0.72
ROR	20798	7885	0.73
SDI	15109	1167	0.93
SDL	72931	16978	0.81
SIR	1321	19	0.99
SVD	39052	4261	0.90
ZIL	11116	1220	0.90

Considerando os modelos de falhas do MutPy e do Cosmic-Ray, representados por seus operadores de mutação, nosso objetivo é identificar os operadores de mutação que geram mutantes mais fáceis de serem eliminados pelos casos de teste gerados pelo LLM, bem como aqueles mais difíceis de serem detectados pelo LLM. O score de mutação representa a eficácia geral do conjunto de testes na detecção de falhas. De maneira geral, como pode ser observado nas Tabelas 7 e 8, os conjuntos de teste gerados pelo LLM apresentam pontuações de mutação mais altas no MutPy do que no Cosmic-Ray.

[2] <https://github.com/aurimrv/python_experiments2>

Tabela 8 – Detecção de falhas de operadores de mutação do Cosmic-Ray

Operador	**Mortos**	**Vivos**	**Score**
AddNot	25146	4013	0.86
ExceptionReplacer	748	68	0.92
NumberReplacer	51222	28116	0.65
ReplaceAndWithOr	1029	1651	0.38
ReplaceBinaryOperator_Add_Div	5861	186	0.97
ReplaceBinaryOperator_BitAnd_Add	22	0	1.00
...	...	...	...
ReplaceBinaryOperator_Pow_Sub	41	0	1.00
ReplaceBinaryOperator_Sub_Div	4510	83	0.98
ReplaceBreakWithContinue	62	103	0.38
ReplaceComparisonOperator_Eq_Is	81	4483	0.02
ReplaceComparisonOperator_Is_Eq	1	354	0.00
ReplaceComparisonOperator_IsNot_NotEq	0	760	0.00
ReplaceComparisonOperator_NotEq_IsNot	0	394	0.00
...	...	...	...
ReplaceOrWithAnd	2213	389	0.85
ReplaceTrueWithFalse	345	183	0.65
ZeroIterationForLoop	6687	462	0.94

Isso sugere que o MutPy gera mutantes que são possivelmente mais fáceis de detectar e que os casos de teste gerados pelo LLM estão melhor alinhados com os tipos de mutações introduzidas pelo MutPy. Por outro lado, o Cosmic-Ray introduz mutações mais sutis, que são mais difíceis de serem detectadas por testes automatizados.

Dadas as limitações de tempo, operadores de mutação que exibem altas pontuações de mutação ($\geq$0,90) podem ser evitados, uma vez que são mais facilmente detectáveis pelos conjuntos de teste gerados pelo LLM. Considerando o MutPy, esses operadores são: SIR (Slice Index Remove), LOR (Logical Operator Replacement), SDI (Statement Deletion), SVD (Self Variable Deletion) e ZIL (Zero Iteration Loop). Considerando o Cosmic-Ray, esses operadores são: múltiplas substituições de operadores binários (BitAnd, Pow), ReplaceBinaryOperator_Add_Div, ExceptionReplacer e ZeroIterationForLoop.

Com base na natureza desses operadores de mutação, pode-se sugerir que os testes gerados pelo LLM são particularmente eficazes na detecção de:

- ❏ Problemas de limites em laços (ZIL/ZeroIterationForLoop);
- ❏ Problemas de tratamento de exceções (EHD/ExceptionReplacer);
- ❏ Exclusão de declarações ou variáveis (SDI/SVD);
- ❏ Certas substituições de operadores aritméticos, especialmente relacionadas à divisão.

Por outro lado, os conjuntos de teste gerados pelo LLM apresentaram dificuldade na detecção de outros tipos de falhas. Especialmente no MutPy, falhas modeladas por EXS (Exception Swallowing), LCR (Logical Connector Replacement) e BCR (Break/Continue Replacement) apresentaram pontuações de mutação muito baixas. O mesmo ocorre

para o Cosmic-Ray, considerando os operadores de mutação ReplaceComparisonOperator_Is_Eq, ReplaceComparisonOperator_Eq_Is, ReplaceComparisonOperator_IsNot_NotEq, ReplaceComparisonOperator_NotEq_IsNot, ReplaceAndWithOr, ReplaceBreakWithContinue, muitos dos quais apresentam pontuações de 0,0, indicando que requerem casos de teste muito específicos para serem detectados.

Uma análise detalhada dos dados revela padrões interessantes:

1. Substituições assimétricas de operadores: Substituir o operador A por B frequentemente apresenta uma taxa de detecção diferente de substituir B por A. Por exemplo:

   - Substituir `==` por `is` apresenta score de 0,02;
   - Substituir `is` por `==` apresenta score de 0,00.

2. Sensibilidade de operadores de comparação: Os testes do LLM parecem particularmente fracos na detecção de mudanças sutis em comparações, especialmente aquelas envolvendo verificações de identidade;

3. Operadores aritméticos versus lógicos: Os testes são geralmente mais eficazes na detecção de mutações em operadores aritméticos (AOR: 0,83) do que em conectores lógicos (LCR: 0,53);

4. Sensibilidade ao fluxo de controle: Mutações que afetam sutilmente o fluxo do programa (como substituir `break` por `continue` ou vice-versa) são mais difíceis de detectar do que aquelas que provocam falhas mais evidentes.

Uma observação final é que, entre os mutantes vivos, alguns podem ser equivalentes. Trabalhos futuros envolverão a identificação manual desses mutantes para aprimorar a precisão da análise.

Respondendo à RQ3, os testes gerados pelo LLM podem se concentrar no "caminho feliz" e em erros comuns, deixando de detectar diferenças semânticas sutis que não causam falhas evidentes imediatamente. Os testes parecem menos eficazes na detecção de mutações que alteram o comportamento do programa de maneiras que exigem uma compreensão profunda da semântica do Python, especialmente em relação à identidade de objetos, fluxo de controle e tratamento de exceções. Esses resultados sugerem que, para uma testagem mais robusta, os testes gerados pelo LLM devem ser complementados com testes manualmente elaborados que abordem especificamente as áreas fracas identificadas. Alternativamente, pode-se desenvolver uma estratégia incremental e criar prompts específicos, considerando esses tipos de falhas potenciais ao gerar os testes.

4.3 Ameaças à Validade

Diversas ameaças à validade devem ser consideradas ao avaliar a capacidade dos LLMs de gerar testes de forma autônoma.

Como mencionado anteriormente, os 40 programas selecionados para análise representam apenas um pequeno subconjunto das inúmeras configurações comumente encontradas em aplicações do mundo real.

Essa amostra pode não capturar toda a diversidade e complexidade das aplicações em Python, limitando a generalização de nossos resultados. Modelos generativos mais avançados que o ChatGPT-3.5-turbo poderiam melhorar a geração automatizada de testes, mas o custo e o escopo orientaram a escolha deste modelo, visto que este estudo constitui uma exploração inicial do teste generativo totalmente automatizado para Python.

Além disso, os testes pré-existentes utilizados como baseline não têm garantia de terem sido criados manualmente ou por ferramentas de geração de testes; eles servem apenas como referência, pois estavam disponíveis no mesmo conjunto de dados do qual os programas foram extraídos.

Por fim, refinamentos adicionais e experimentação com prompts variados poderiam aprimorar os resultados, uma vez que este estudo utilizou um único prompt para geração e outro para correção. Ademais, utilizar apenas um modelo LLM também representa uma limitação, pois múltiplos modelos poderiam revelar variações de desempenho. Ainda assim, o ChatGPT-3.5-turbo apresentou desempenho impressionante, considerando seu custo, especialmente ao combinar casos de teste gerados em diferentes temperaturas para aumentar a diversidade.

Capítulo 5

Conclusão

Os resultados deste estudo destacam a eficácia mista do ChatGPT-3.5-turbo na geração de casos de teste válidos para Python. Para códigos mais simples, ele produziu consistentemente testes que passaram na validação, às vezes atingindo 100% em determinadas temperaturas. No entanto, quando se consideram todos os programas, a taxa de sucesso dos testes gerados caiu abaixo de 28%, principalmente devido a inúmeros casos em que o modelo não conseguiu produzir testes válidos. Essa inconsistência indica que, embora os LLMs demonstrem potencial para geração autônoma de testes, eles apresentam dificuldades com aplicações mais complexas ou menos determinísticas.

O impacto da temperatura no sucesso da geração de testes foi notável. Configurações menos determinísticas (0,5 ou superiores) geralmente resultaram em mais testes funcionais, com as maiores taxas de sucesso observadas nas temperaturas 0,5 e 1,0, superando a média em 13,94% e 15,6%, respectivamente. Além disso, nas avaliações de cobertura, a tendência se manteve, com temperaturas mais altas produzindo casos de teste mais eficazes em termos de cobertura de decisão. No entanto, o elevado desvio padrão entre as configurações de temperatura e os programas indica variabilidade significativa, enfatizando que o desempenho atual do modelo na geração de testes pode variar amplamente dependendo da estrutura do código e do nível de aleatoriedade.

Quanto às pontuações de mutação, os casos de teste gerados pelo LLM alcançaram resultados próximos à baseline, especialmente com o MutPy. Embora os testes de referência do Pynguin e os testes pré-existentes ainda tenham superado ligeiramente os gerados pelo LLM, a diferença foi de apenas cerca de 2,03%. Isso sugere que, apesar dos desafios em gerar testes válidos de forma consistente, os testes gerados pelo LLM que foram executados com sucesso mostraram-se eficazes na identificação de mutantes.

Trabalhos futuros poderiam avaliar modelos generativos mais avançados, como versões

mais recentes do ChatGPT ou outros LLMs, visando aprimorar a geração automatizada de testes. Os scripts deste estudo podem ser facilmente adaptados para outros modelos. Além disso, explorar a capacidade dos LLMs em detectar mutantes específicos ou equivalentes permanece uma questão em aberto, já que as ferramentas de mutação atuais não realizam a identificação de mutantes equivalentes. Ademais, considerando que o experimento foi conduzido com um número limitado de programas, estudos futuros poderão explorar um conjunto maior de sujeitos.

Uma direção promissora envolve testar prompts variados, adaptados a características específicas do código, potencialmente melhorando a precisão e a cobertura dos testes, visto que este estudo utilizou um único prompt para todos os casos, tanto para geração quanto para correção, além de aplicar a minimização de conjuntos de teste para eficiência de custo. A combinação de diferentes modelos de LLM, iniciando com modelos básicos e posteriormente utilizando modelos mais avançados, também oferece oportunidades para atender melhor aos requisitos de teste complexos e aprimorar os conjuntos de testes gerados.

Referências

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: **XXVII International Conference on Software Engineering – ICSE'05**. St. Louis, MO, USA: ACM Press, 2005. p. 402–411. ISBN 1-59593-963-2.

ARAUJO, F. S.; VINCENZI, A. M. R. How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level? In: **Anais do Simpósio Brasileiro de Qualidade de Software (SBQS)**. São Luiz, MA: SBC, 2020. p. 151–159. ISBN 978-1-4503-8923-5. Disponível em: <https://sol.sbc.org.br/index.php/sbqs/article/view/14211>.

BINGHAM, A. Cosmic Ray: mutation testing for Python. 2021. Disponível em: <https://github.com/sixty-north/cosmic-ray>.

GUERINO, L. R. et al. Static and Dynamic Comparison of Mutation Testing Tools for Python. In: **Proceedings of the XXIII Brazilian Symposium on Software Quality**. New York, NY, USA: Association for Computing Machinery, 2024. (SBQS '24), p. 199–209. ISBN 9798400717772. Disponível em: <https://doi.org/10.1145/3701625.3701659>.

GUERINO, L. R.; VINCENZI, A. M. R. An Experimental Study Evaluating Cost, Adequacy Effectiveness of Pynguin's Test Sets. In: **8th Brazilian Symposium on Systematic and Automated Software Testing – SAST'2023**. Campo Grande, MS: ACM Press, 2023. p. 5–14.

HOSSNER, P. et al. MutPy: a mutation testing tool for Python 3.x source code. 2021. Disponível em: <https://github.com/mutpy/mutpy>.

KONUK, M.; BAGLUM, C.; YAYAN, U. Evaluation of Large Language Models for Unit Test Generation. In: **2024 Innovations in Intelligent Systems and Applications Conference (ASYU)**. [S.l.: s.n.], 2024. p. 1–6.

LUKASCZYK, S.; KROIß, F.; FRASER, G. An empirical study of automated unit test generation for Python. **Empirical Software Engineering**, v. 28, n. 2, p. 36, jan. 2023. ISSN 1573-7616.

PETROVIć, G. et al. Does Mutation Testing Improve Testing Practices? In: **Proceedings of the 43rd International Conference on Software Engineering**.

Madrid, Spain: IEEE Press, 2021. (ICSE '21), p. 910–921. ISBN 978-1-4503-9085-9. Place: Madrid, Spain. Disponível em: <https://doi.org/10.1109/ICSE43902.2021.00087>.

SCHMIDT, D. C. Software Testing in the Generative AI Era: A Practitioner's Playbook. **Computer**, v. 58, n. 7, p. 147–152, 2025.

SCHäFER, M. et al. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. **IEEE Transactions on Software Engineering**, v. 50, n. 1, p. 85–105, 2024.

TIOBE Software BV. **TIOBE Index**. 2024. Disponível em: <https://www.tiobe.com/tiobe-index/>.

VINCENZI, A. M. R. et al. The Complementary Aspect of Automatically and Manually Generated Test Case Sets. In: **Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation**. Seattle, USA: ACM, 2016. (A-TEST 2016, v. 1), p. 23–30. ISBN 978-1-4503-4401-2. Event-place: Seattle, WA, USA.

VOGL, S. et al. EvoSuite at the SBST 2021 Tool Competition. In: **2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)**. Madrid, Spain: IEEE, 2021. p. 28–29.

WANG, J. et al. Software Testing With Large Language Models: Survey, Landscape, and Vision. **IEEE Trans. Softw. Eng.**, v. 50, n. 4, p. 911–936, fev. 2024. ISSN 0098-5589. Publisher: IEEE Press. Disponível em: <https://doi.org/10.1109/TSE.2024.3368208>.

ZHANG, J. et al. Less is more: On the importance of data quality for unit test generation. **Proc. ACM Softw. Eng.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. FSE, jun. 2025. Disponível em: <https://doi.org/10.1145/3715778>.