

Geração de casos de teste com IA Generativa para plataformas de juiz online: um estudo comparativo com o Pyguin

Júlio César da S. Dantas¹, Francisco Genivan Silva^{1,2}, Jadson Lucas Gomes Souza¹,
Eduardo Henrique da Silva Aranha¹

¹Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

²Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte

dantas.j.c.s@gmail.com, genivan.silva@ifrn.edu.br, jadson.souza@ufrn.br,
eduardoaranha@dimap.ufrn.br

Abstract. *Online judges are widely used in programming education, but their effectiveness depends on the availability of comprehensive problem sets and test suites to accurately evaluate solutions. This work compares automatic test generation using GPT-5.4 and Pyguin across 120 LeetCode problems, employing code coverage and mutation score as evaluation metrics. Using a self-refining strategy, the AI-based approach achieved a median code coverage of 95% and outperformed Pyguin in 84% of the problems in terms of mutation score. The results suggest that generative artificial intelligence models constitute a promising alternative for automated test generation and outperform another search-based automated approach.*

Resumo. *Juízes online são amplamente utilizados no ensino de programação, mas dependem de conjuntos de questões e testes abrangentes para avaliar soluções corretamente. Este trabalho compara a geração automática de testes com GPT-5.4 e Pyguin em 120 problemas do LeetCode, utilizando cobertura de código e mutation score como métricas. Com uma estratégia de self-refining, a abordagem baseada em IA alcançou 95% de cobertura em média e superou o Pyguin em 84% dos problemas em termos do mutation score. Os resultados sugerem que modelos de inteligência artificial generativa constituem uma alternativa promissora para a geração automática de testes, e que supera outro método de geração automática de testes.*

1. Introdução

Um Juiz Online, também conhecido como sistema de avaliação automática, é uma plataforma que disponibiliza problemas de programação para serem resolvidos por meio da submissão de códigos-fonte em uma ou mais linguagens de programação. Após a submissão, as soluções são avaliadas automaticamente com base em um conjunto de casos de teste previamente definidos, compostos por entradas e suas respectivas saídas esperadas. O processo de correção consiste na execução do programa submetido sobre cada caso de teste disponível, comparando-se os resultados produzidos com as saídas esperadas. Uma solução é considerada correta quando atende aos requisitos estabelecidos por todos os testes aplicados. Dessa forma, os Juízes Online permitem a avaliação

rápida, padronizada e escalável de programas, reduzindo a necessidade de correção manual [Francisco et al. 2016, Carvalho et al. 2016].

Os Juízes Online oferecem vários benefícios para o ensino de programação, destacando-se a automatização da correção de exercícios e o fornecimento de *feedback* imediato aos estudantes. Essas características reduzem a carga de trabalho dos professores, permitem que os alunos aprendam no próprio ritmo e incentivam a prática contínua e a autoaprendizagem. Além disso, esses sistemas podem disponibilizar uma ampla variedade de problemas e auxiliar o acompanhamento do desempenho dos estudantes ao longo do tempo. Também podem promover maior motivação, engajamento e assiduidade dos alunos, ao mesmo tempo em que garantem transparência e padronização no processo de avaliação [Francisco et al. 2016].

Embora os Juízes Online tenham se consolidado como ferramentas importantes para o ensino de programação, sua efetividade depende diretamente da qualidade dos problemas e dos casos de teste utilizados na avaliação. A elaboração de boas questões exige não apenas conhecimento do conteúdo abordado, mas também a definição cuidadosa de enunciados claros, níveis adequados de dificuldade e critérios de avaliação compatíveis com os objetivos de aprendizagem [Dantas and Aranha 2025]. Da mesma forma, a construção de conjuntos de testes abrangentes é uma tarefa complexa, pois é necessário contemplar casos comuns, situações de borda e possíveis estratégias incorretas adotadas pelos estudantes.

Nos últimos anos, o avanço dos modelos de Inteligência Artificial (IA) Generativa tem despertado interesse na automatização desse processo. Diversos trabalhos têm investigado o uso dessas tecnologias para auxiliar professores na criação de questões, geração de enunciados, produção de soluções de referência e adaptação de exercícios para diferentes níveis de dificuldade, como nos trabalhos de [Santana et al. 2025a, Santana et al. 2025b, Dantas and Aranha 2025]. Esses estudos indicam que a IA possui potencial para reduzir o esforço necessário na criação de conteúdo educacional, tornando mais viável a manutenção e expansão de plataformas de avaliação automática.

Entretanto, embora o uso de Inteligência Artificial para apoiar a criação de questões tenha recebido atenção crescente da comunidade científica, sua aplicação na construção e validação de casos de teste ainda permanece relativamente pouco explorada [Cao et al. 2026]. Essa limitação é especialmente relevante no contexto dos Juízes Online, nos quais a qualidade dos casos de teste influencia diretamente a confiabilidade da avaliação. Conjuntos de testes insuficientes podem permitir a aprovação de soluções incorretas ou reprovar implementações válidas, comprometendo a eficácia da correção automática e a qualidade do *feedback* fornecido aos estudantes.

A preocupação com a geração automática de testes não é recente. Ao longo dos anos, diferentes abordagens foram propostas, incluindo técnicas aleatórias e baseadas em busca, resultando em ferramentas consolidadas como Randoop e EvoSuite [Silva et al. 2025]. Essas soluções foram desenvolvidas principalmente para a linguagem Java, o que limita sua aplicação em ambientes educacionais que utilizam outras linguagens amplamente adotadas, como Python.

Nesse cenário, [Lukasczyk and Fraser 2022] apresentam o Pynguin, uma ferramenta para geração automática de testes em Python baseada em técnicas de

busca (DynaMOSA, MIO, MOSA, random, Whole Suite, ou Whole Suite with archive) [Lukasczyk and Fraser 2022]. Diferentemente de outras abordagens que exigem informações fornecidas manualmente pelos desenvolvedores, o Pynguin busca produzir testes com elevada cobertura de código de forma automatizada. Essas características tornam a ferramenta particularmente interessante para o contexto dos Juízes Online, uma vez que ela pode auxiliar professores e mantenedores dessas plataformas na criação e ampliação de conjuntos de testes, reduzindo o esforço manual necessário para construir avaliações mais robustas e abrangentes.

Paralelamente, os grandes modelos de linguagem (*Large Language Models* – LLMs) têm demonstrado capacidades cada vez mais avançadas na geração de código e na resolução de problemas computacionais complexos. Apesar desses avanços, [Cao et al. 2026] destacam que o uso desses modelos em atividades de garantia de qualidade de *software*, especialmente na criação estratégica de casos de teste para verificação e depuração de programas, ainda carece de investigações mais aprofundadas.

Diante desse cenário, este trabalho justifica-se pela necessidade de investigar o potencial de modelos de Inteligência Artificial Generativa na construção automática de casos de teste para problemas utilizados em Juízes Online. Embora ferramentas tradicionais de geração automática de testes, como o Pynguin, apresentem resultados promissores na obtenção de elevada cobertura de código, ainda é necessário compreender se abordagens baseadas em grandes modelos de linguagem são capazes de produzir testes com qualidade equivalente ou superior. Para isso, serão analisadas métricas comumente utilizadas na avaliação de suítes de teste, como cobertura de código e *mutation score*, permitindo comparar o desempenho dos testes gerados por IA com aqueles produzidos pelo Pynguin.

Para alcançar esse objetivo, a pesquisa foi conduzida em duas etapas complementares. Na primeira etapa, diferentes técnicas de engenharia de *prompts* serão avaliadas utilizando um conjunto reduzido de trechos de código, com o propósito de identificar a estratégia mais eficaz para a geração automática de testes. Essa avaliação será realizada por meio de métricas de cobertura de código e *mutation score*. Na segunda etapa, a técnica eleita será aplicada a um conjunto mais amplo de problemas, permitindo uma análise mais abrangente de seu desempenho. Os testes gerados por essa abordagem serão então comparados aos produzidos pelo Pynguin, utilizando as mesmas métricas de avaliação, com o objetivo de investigar o potencial dos modelos de Inteligência Artificial Generativa como alternativa ou complemento às técnicas tradicionais de geração automática de testes.

2. Trabalhos relacionados

O estudo de [Yuan et al. 2024] analisa a capacidade do ChatGPT de gerar testes unitários para métodos em Java, avaliando os artefatos produzidos quanto à corretude, suficiência, legibilidade e usabilidade. Os autores também propõem um *framework* baseado em múltiplas interações com o modelo e o comparam tanto com uma abordagem de chamada única quanto com técnicas tradicionais de geração automática de testes. Os resultados mostram que o ChatGPT apresenta desempenho superior a diversas abordagens baseadas em aprendizado de máquina em termos de compilação e execução dos testes gerados, embora ainda produza uma quantidade significativa de testes com erros de compilação ou execução, o que limita sua utilização direta em alguns cenários.

Em relação à cobertura e suficiência, os testes gerados pelo ChatGPT alcançaram

resultados comparáveis aos produzidos manualmente por desenvolvedores, apresentando altos índices de cobertura e características semelhantes às de testes escritos por humanos. Além disso, muitos participantes consideraram que os testes poderiam ser utilizados com poucas ou nenhuma modificação. Um dos principais achados do estudo foi a eficácia do *framework* iterativo proposto, que realiza ciclos sucessivos de geração e refinamento dos testes, praticamente dobrando as taxas de sucesso de compilação e execução em comparação à estratégia de chamada única. Esses resultados indicam que abordagens iterativas podem reduzir limitações dos modelos de linguagem e aumentar sua utilidade na geração automática de testes.

De forma semelhante, [Tang et al. 2024] comparam o desempenho do ChatGPT com o EvoSuite na geração automática de testes unitários para aplicações Java. Os autores avaliam aspectos como corretude, legibilidade, cobertura de código e capacidade de detecção de defeitos. O estudo emprega uma abordagem de *one-shot prompting*, na qual o modelo recebe apenas uma instrução inicial para gerar os testes, sem etapas posteriores de refinamento.

Os resultados obtidos são consistentes com aqueles observados por [Yuan et al. 2024], indicando que aproximadamente 70% dos testes produzidos pelo ChatGPT compilam e executam corretamente. Além das métricas tradicionais de qualidade, o estudo também analisa a conformidade dos testes gerados com padrões de estilo de código amplamente utilizados na indústria. A análise revela que as violações mais frequentes estão relacionadas à indentação quando avaliadas segundo o Google Style Guide e à ausência de comentários Javadoc ou ao uso de números mágicos quando considerada a convenção SUN Style.

Apesar dessas limitações, os autores observam que a maior parte dos testes gerados apresenta baixa complexidade cognitiva e ciclomática, tornando-os relativamente fáceis de compreender, manter e evoluir. No entanto, quando comparado ao EvoSuite, o ChatGPT apresenta desempenho inferior em algumas métricas técnicas. Em cobertura de código, a diferença pode chegar a aproximadamente 20 pontos percentuais em determinados cenários experimentais. De forma semelhante, o EvoSuite demonstrou maior capacidade de detecção de defeitos, superando o ChatGPT em aproximadamente cinco pontos nessa métrica. Esses resultados sugerem que, embora os modelos de linguagem apresentem vantagens relacionadas à legibilidade e à proximidade com testes produzidos por humanos, ferramentas especializadas ainda mantêm vantagens importantes em aspectos tradicionalmente associados à efetividade dos testes.

[Guilherme and Vincenzi 2023] também investigam o uso do ChatGPT para geração de testes unitários em Java por meio de uma estratégia baseada em *one-shot prompting*. Contudo, diferentemente dos estudos anteriores, o foco principal da pesquisa está na análise do impacto do parâmetro de temperatura sobre a qualidade dos testes gerados. A temperatura controla o nível de aleatoriedade das respostas produzidas pelo modelo e, consequentemente, pode influenciar diretamente a qualidade do código gerado.

Os experimentos realizados demonstram que diferentes configurações de temperatura produzem resultados significativamente distintos. Entre os valores avaliados, a temperatura de 0,6 apresentou o melhor desempenho geral, resultando em menores taxas de erro e maiores índices de sucesso na compilação e execução dos testes. Os autores

concluem que nem todos os valores de temperatura são igualmente adequados para tarefas de geração automática de testes, sendo necessária uma configuração apropriada para maximizar a qualidade dos artefatos produzidos.

Além disso, os resultados indicam que, sob determinadas configurações, o desempenho do ChatGPT pode se aproximar daquele obtido pelo EvoSuite em métricas como cobertura de código e sucesso na execução dos testes. Essa constatação reforça o potencial dos modelos de linguagem como ferramentas de apoio à engenharia de software, especialmente quando combinados com estratégias adequadas de parametrização e interação. O estudo destaca ainda a necessidade de pesquisas futuras voltadas à identificação das configurações mais adequadas para diferentes cenários de geração automática de testes, uma vez que pequenas alterações nos parâmetros do modelo podem produzir impactos significativos na qualidade dos resultados obtidos.

Diferente de outros trabalhos na literatura, e de maneira bastante semelhante com o que se pretende aqui, [Bhatia et al. 2024] também comparam o desempenho de testes gerados por IA com o Pynguin, no contexto de Python. Entre os resultados, os autores apontam que o ChatGPT e o Pynguin apresentaram uma cobertura de código bastante semelhante e comparável, sem diferenças estatisticamente significativas tanto para amostras pequenas quanto grandes de código; também que ao alimentar o ChatGPT em várias rodadas com os índices das linhas que haviam ficado sem cobertura na tentativa anterior, os pesquisadores conseguiram aumentar consideravelmente a cobertura média de código e que esse ganho de desempenho atingiu um ponto de estabilização após 4 iterações e, finalmente, que apesar da boa cobertura alcançada pelo ChatGPT, uma parcela significativa das asserções geradas por ele estava incorreta (cerca de 39% em códigos baseados em funções e 28% em códigos baseados em classes). Isso indica uma tendência do modelo em priorizar a cobertura de código em detrimento da precisão lógica do teste.

Este trabalho se distancia de outros da literatura, em especial deste último, quando propõe o uso do *mutation score* na avaliação dos casos de teste e quando foca no contexto de Juizes Online. Embora métricas de cobertura de código forneçam uma visão sobre a extensão do código exercitada pelas ferramentas, elas são insuficientes para atestar a capacidade real de detecção de defeitos das suítes de testes geradas por LLMs [Silva et al. 2025]. Além disso, o trabalho usa modelos mais recentes de IA generativa, explorando avanços significativos obtidos nos últimos anos e investigando seu potencial em um cenário ainda pouco estudado pela literatura.

2.1. Pynguin

O Pynguin é uma ferramenta de código aberto escrita em Python usada na geração de testes unitários automaticamente. O funcionamento da ferramenta é dividido em etapas modulares e independentes que começam com a análise do módulo Python fornecido como entrada. A partir dessa análise, o Pynguin extrai informações sobre classes, funções, métodos e parâmetros para construir um mapa completo chamado cluster de teste, além de inspecionar módulos vizinhos para identificar os tipos de dados disponíveis no contexto [Bhatia et al. 2024].

Na fase de construção dos casos de teste, o sistema escolhe uma função alvo e analisa os parâmetros necessários de trás para frente. Se a função exigir tipos simples como números inteiros, a ferramenta gera atribuições de variáveis aleatórias antes da

chamada da função, enquanto para objetos mais complexos ela tenta construí-los de forma recursiva. Para guiar a geração desses dados de entrada, o usuário pode escolher entre diversos algoritmos de busca consagrados, como DynaMOSA, MIO e Whole Suite. O Pynquin então modifica o *bytecode* do Python em tempo real para rastrear e medir a cobertura de linhas ou ramificações alcançada pelos testes gerados, repetindo esse ciclo até atingir um critério de parada configurado, como um limite de tempo ou a cobertura total do código [Lukasczyk and Fraser 2022].

Após definir as entradas dos testes, a ferramenta utiliza testes de mutação por meio de uma versão customizada do MutPy para criar as asserções de regressão de forma inteligente. Esse processo gera versões intencionalmente modificadas do código original e compara o comportamento de ambas para descobrir quais valores mudam, criando verificações focadas em capturar essas diferenças e minimizando a quantidade de asserções desnecessárias. Por fim, o Pynquin converte sua estrutura interna de testes em código Python real e exporta o resultado formatado no estilo do *framework* PyTest, embora sua arquitetura modular permita a adaptação fácil para outros formatos como o *unittest* [Lukasczyk and Fraser 2022].

3. Metodologia

Nesta seção, são discutidos os contornos do experimento realizado, como os passos seguidos, as decisões metodológicas tomadas e as bibliotecas usadas.

3.1. Dataset, prompts e definições

O conjunto de dados, disponibilizado por [Chen 2026], contém aproximadamente três mil trechos de código em linguagem Python extraídos do LeetCode, uma plataforma de desafios de programação. Os códigos selecionados apresentam uma complexidade ciclomática que varia de 1 a 38, calculada a partir da Árvore de Sintaxe Abstrata (AST) das funções. Os testes unitários automatizados foram executados com o *framework* Pytest (versão 9.0.3), a cobertura foi calculada com *coverage* (versão 5.5), enquanto o *mutation score* foi obtido por meio da ferramenta Mute (versão 3.1.0). *Prompts* usados, implementação, código de referência das questões e relatório do experimentos, com as métricas brutas de cobertura e *mutation score* pode ser acessadas no link¹.

A construção e a calibração dos *prompts* baseiam-se no método proposto por [Shah 2024], cujo objetivo é garantir a generalização, a explicabilidade e a reprodutibilidade dos experimentos. Fundamentado na validação humana consensual das respostas e na consistência das saídas do modelo, o processo estrutura-se de forma reversa na linha de desenvolvimento, dividindo-se em quatro fases principais.

A primeira fase consiste no estabelecimento de um fluxo inicial simplificado, utilizando instruções diretas e básicas para gerar as primeiras saídas do modelo. Na segunda fase, definem-se os critérios de avaliação dessas respostas antes que qualquer refinamento no *prompt* ocorra, assegurando o foco estrito na tarefa. Para tanto, os avaliadores analisam as saídas de forma independente e mensuram as métricas de interesse do problema — neste estudo, a cobertura de código e o *mutation score*.

Com os critérios estabelecidos e os avaliadores alinhados, a terceira fase concentra-se no desenvolvimento iterativo do *prompt*. Novas entradas são processadas

¹https://osf.io/9t7ke/overview?view_only=1edcfe53646e41bab9da98f8668ce0f4

e submetidas à validação para verificar o atendimento aos requisitos estipulados. Caso o modelo falhe ou ocorram divergências de interpretação, o próprio *prompt* é revisado e expandido. Esse ciclo repete-se até que a instrução atinja um limiar de sucesso e consenso satisfatório.

Por fim, a quarta fase realiza a validação final de todo o fluxo de trabalho utilizando um conjunto de dados inédito — composto, neste experimento, por 120 trechos de código. Se essa avaliação independente demonstrar um alto índice de qualidade e conformidade nas respostas, o processo é considerado validado; do contrário, retorna-se às etapas anteriores para novos ajustes.

As técnicas de *prompt* que competiram nessa primeira etapa do experimento foram *Few-Shot*, que consiste na inclusão de um pequeno número de exemplos de demonstração no corpo do *prompt*, *Chain-of-Thought*, que instrui o modelo a externalizar seu processo lógico passo a passo antes de apresentar a resposta final, *Prompt chaining*, que decompõem uma tarefa complexa em uma sequência de subetapas interdependentes, onde a saída (*output*) de um *prompt* serve como a entrada (*input*) para o próximo e *self-refining*, que consiste em um processo iterativo no qual o próprio modelo avalia e aprimora suas saídas iniciais; em vez de aceitar a primeira resposta gerada, a técnica consiste em alimentar o modelo com o seu próprio código de teste e instruí-lo a realizar correções [Oshin and Campos 2025]. A técnica eleita é esmiuçada adiante.

3.2. Desenho experimental e decisões metodológicas

Esta pesquisa divide-se em duas etapas. A primeira etapa consiste na experimentação de *prompts* para a seleção da técnica que será aplicada ao conjunto de dados completo. Essa fase preliminar utilizou 10 observações extraídas aleatoriamente. Com base nas métricas de cobertura e *mutation score* obtidas nesse subconjunto, elegeu-se a abordagem de *prompt* ideal para ser comparada aos testes gerados pelo Pynguin.

A segunda etapa compreende a geração completa de testes, utilizando tanto a ferramenta Pynguin quanto a técnica de *prompt* selecionada. Para a composição desse cenário de testes, foram selecionados apenas os grupos de complexidade que continham, no mínimo, 10 observações. Ao total, 12 grupos atenderam a esse critério, resultando nos 120 trechos de código analisados nas seções seguintes, com complexidade entre 2 e 12.

Ambos os processos de geração (por Inteligência Artificial e por Pynguin) foram submetidos a um limite de tempo (*timeout*) de 180 segundos. Esse mesmo teto temporal foi aplicado aos cálculos de cobertura e *mutation score*. A imposição desse limite justifica-se pela necessidade de viabilizar a execução do experimento em tempo hábil, evitando travamentos causados por arquivos com complexidade atípica. Mutatest foi chamado *flag* na sua definição padrão, o que fez ele testar todos os operadores disponíveis e falhar se algum teste do arquivo não funcionar.

O modelo de Inteligência Artificial selecionado para a geração dos testes foi o GPT-5.4. Embora essa versão fique ligeiramente atrás do GPT-5.5 em termos de capacidade geral em contextos de programação, sua escolha justificou-se por dois fatores principais. Primeiro, pelo aspecto econômico, dado que o GPT-5.4 apresenta metade do custo do modelo mais recente. Segundo, e de maior relevância prática, os testes preliminares da primeira etapa revelaram que o GPT-5.5 não foi capaz de concluir a geração dentro do limite estipulado de três minutos, a despeito das variações configuradas para o parâmetro

de *reasoning*, inviabilizando sua aplicação no restante do estudo. Para a execução do GPT-5.4, o nível de *reasoning* foi mantido no valor padrão (*medium*) e a temperatura foi configurada em 1.0.

4. Resultados

Nesta seção, são discutidos os resultados da pesquisa: primeiro, da eleição da técnica de *prompt* com melhor desempenho de cobertura e *mutation score* para um subconjunto de trechos de código e, depois, da comparação da técnica eleita com os testes gerados pelo Pynguin.

4.1. Definição da técnica de *prompt*

Visto que a literatura não é uníssona quanto às melhores técnicas de *prompt* aplicadas a esse problema — com abordagens que variam desde o *one-shot* em [Guilherme and Vincenzi 2023] e [Tang et al. 2024], passando por um formato específico de *self-refining* em [Yuan et al. 2024], até o *prompt chaining* em [Dantas and Aranha 2025] —, este trabalho dedicou uma etapa preliminar para avaliar o desempenho de diferentes estratégias antes da execução do experimento com o conjunto completo de trechos.

À exceção do *self-refining*, nenhuma das técnicas de *prompt* avaliadas foi capaz de gerar, para as 10 amostras aleatórias iniciais, testes aptos a serem submetidos ao cálculo do *mutation score*. Esse comportamento ocorreu porque as suítes de testes geradas pelas demais abordagens continham ao menos uma falha de execução — ou seja, apresentavam asserções incorretas frente ao código original —, o que inviabilizava a análise de mutantes. Esses resultados preliminares corroboram outros trabalhos, que já reportaram dificuldades na geração de suítes prontos para uso na primeira iteração [Cao et al. 2026, Yuan et al. 2024] - mesmo com iterações de refino dos *prompts* e com o uso de modelos mais capazes.

Desta maneira, a técnica de *prompt* escolhida para rivalizar com o Pynguin foi a de *self-refining*, que funciona como a seguir: na primeira fase, coordenada pelo Agente Gerador, o *script* compila um *prompt* detalhado contendo o código-fonte original e diretrizes rígidas de qualidade, com foco em cenários extremos, alta cobertura de código e resiliência a mutações. Essa instrução é enviada à API da OpenAI para a criação de uma suíte de testes em *pytest*.

A segunda fase ativando um fluxo de validação e auto-refinamento limitado a até três tentativas de correção. O *script* executa o *pytest* contra o código original, captura a saída do console para contabilizar o sucesso ou falha dos testes e registra incrementalmente esses dados.

Se a execução for bem-sucedida e livre de erros, o ciclo para aquele módulo é encerrado e o programa avança para o próximo problema do conjunto de dados. Caso contrário, se o console reportar falhas ou erros de compilação, o Agente Corretor é acionado para formular uma requisição de depuração à API. Esse novo chamado fornece ao agente o código original, o teste que falhou e o registro exato de erros do console. A IA processa o contexto, devolve uma versão corrigida da suíte e o *script* reinicia o *loop* de validação. Esse processo de *feedback* repete-se até que os testes passem de forma limpa ou o limite de três tentativas se esgote. Se o limite for atingido sem sucesso, a falha é

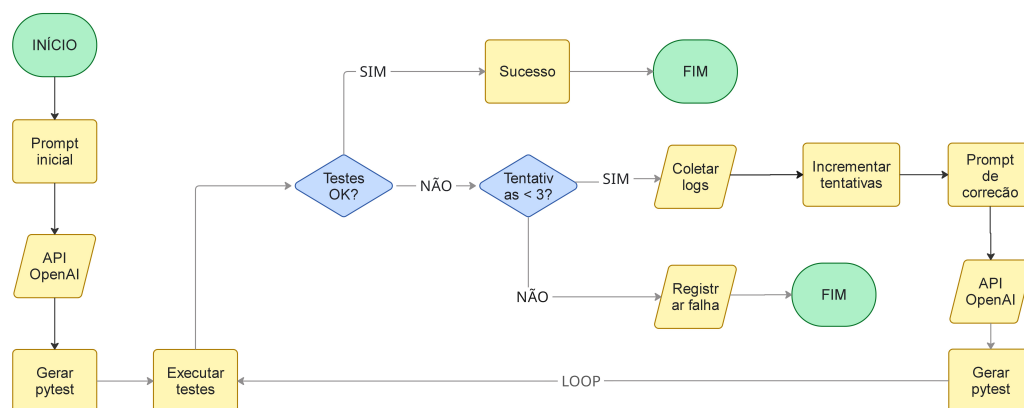


Figura 1. Esquema de geração de testes da estratégia de *self-refining*.

computada, a última versão gerada é mantida em disco e o experimento prossegue para o próximo arquivo; dessa forma, evita-se o travamento do fluxo em um único problema, no entanto, ao mesmo tempo, se preserva um conjunto de testes cujas asserções incorretas fatalmente comprometerão o cálculo subsequente do *mutation score*.

4.2. Geração por IA e Pyguin

Na geração de testes utilizando a abordagem baseada em *self-refining*, apenas 37,8% das suítes de teste foram aprovadas na primeira execução, sem necessidade de refinamento. Após a primeira iteração de correção, outros 46,2% dos casos passaram a executar corretamente, elevando a taxa de sucesso da abordagem. As iterações subsequentes apresentaram ganhos mais modestos: 6,7% das suítes foram aprovadas após a segunda tentativa e 5,0% após a terceira e última iteração de refinamento. Ao final do processo, apenas 4,2% dos arquivos de teste permaneceram com falhas que não puderam ser corrigidas dentro do limite estabelecido de três iterações - estes arquivos são salvos para fins do cálculo de cobertura, mas não terão o seu *mutation score* calculado. Esses resultados indicam que a maior parte dos problemas é resolvida já na primeira etapa de refinamento, enquanto refinamentos adicionais contribuem de forma incremental para o aumento da taxa de aprovação.

Além da correção das falhas que motivaram o processo de refinamento, observou-se que as suítes de testes aprovadas apresentaram um aumento médio de 31,9% no número de casos de teste entre a versão inicial e a versão final aprovada. Esse resultado sugere que o processo de refino não apenas corrigiu erros presentes nos testes gerados originalmente, mas também contribuiu para a expansão das suítes de teste, potencialmente ampliando a cobertura dos cenários avaliados.

Outro resultado relevante observado neste estudo foi que, em média, 16,5% dos casos de teste presentes em cada suíte gerada e que precisou passar por refino falharam na primeira execução. Esse resultado é consistente com evidências reportadas em trabalhos anteriores e sugere que a geração automática de testes por modelos de IA generativa ainda está sujeita à produção de artefatos incorretos ou incompletos. Nesse contexto, os resultados obtidos reforçam a justificativa para a adoção de abordagens baseadas em *self-refining*, capazes de corrigir iterativamente os problemas identificados e aumentar a taxa

de aprovação das suítes geradas.

No que se refere à geração de testes com o Pynguin, aproximadamente 7% dos trechos de código analisados não tiveram suítes de teste produzidas, uma vez que o processo de geração excedeu o limite de tempo estabelecido de três minutos. Entre os casos em que a geração foi concluída com sucesso, cerca de 29% das suítes produzidas apresentaram algum erro de execução, impossibilitando a realização da análise por meio do *mutation score*. Esses resultados evidenciam limitações práticas da ferramenta tanto em termos de desempenho quanto na geração consistente de suítes de teste executáveis.

4.3. Análise da cobertura e do Mutation Score

As análises a seguir contemplam exclusivamente os arquivos para os quais as suítes de testes foram geradas com sucesso. Foram desconsiderados, portanto, os casos em que o Pynguin não gerou testes ou nos quais os testes gerados ocorreram erros de execução. Em relação à cobertura de código, a IA alcançou mediana próxima de 95%, enquanto o Pynguin apresentou mediana ligeiramente inferior, em torno de 92%. Além disso, a dispersão dos resultados da IA é menor, indicando maior consistência entre os diferentes problemas avaliados. Embora ambas as abordagens tenham atingido valores máximos próximos de 100%, o Pynguin apresentou uma quantidade significativamente maior de valores *outliers*, incluindo casos com cobertura inferior a 30%, sugerindo dificuldades em alguns problemas específicos.

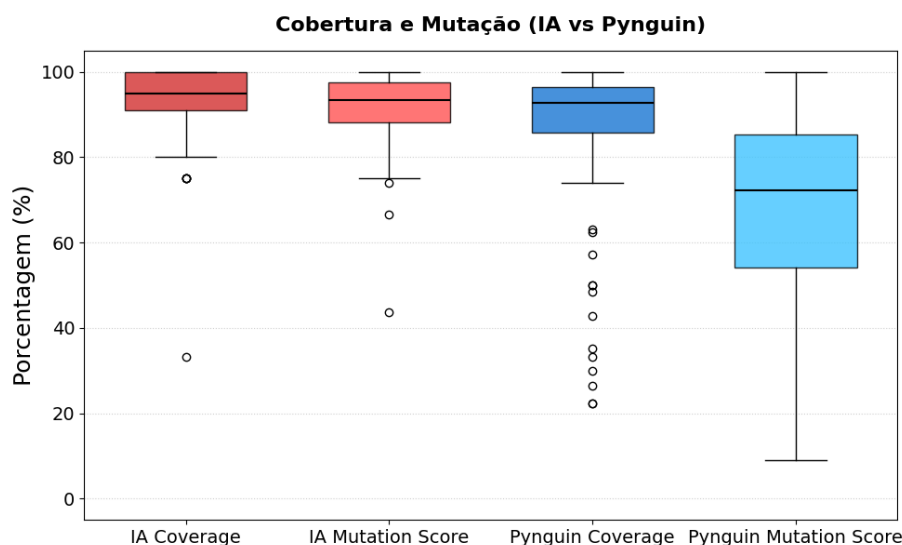


Figura 2. Comparação do desempenho de cobertura e mutation score para testes gerados por IA e por Pynguin.

A diferença torna-se mais evidente na análise do *mutation score*. Os testes gerados pela IA apresentaram mediana superior a 93%, com a maior parte dos resultados concentrada entre aproximadamente 88% e 97%. Isso indica que, além de executar uma grande parte do código, os testes foram eficazes na detecção de mutantes, demonstrando boa capacidade de revelar defeitos. O Pynguin, por sua vez, apresentou mediana próxima de 72% e uma variabilidade muito maior, com resultados distribuídos desde valores inferiores a 10% até valores próximos de 100%. Essa dispersão sugere que, embora o Pynguin

seja capaz de produzir excelentes conjuntos de testes em alguns casos, seu desempenho é menos previsível e mais dependente das características do problema analisado.

Em termos de cobertura de código, os testes gerados por IA apresentaram desempenho comparável ao obtido pelo Pynguin. Aproximadamente 70% dos arquivos analisados alcançaram níveis idênticos de cobertura entre as duas abordagens, enquanto nos 30% restantes os testes produzidos por IA obtiveram cobertura superior. Não foram observados casos em que os testes gerados pelo Pynguin superassem os produzidos por IA nesse critério. Esses resultados indicam que, sob a perspectiva da cobertura de código, a abordagem baseada em IA foi capaz de igualar ou superar sistematicamente o desempenho da ferramenta tradicional de geração automática de testes. Para o *mutation score*, os resultados são mais variados, com os testes gerados por IA superando os do Pynguin em 84% dos casos.

A análise da correlação de Pearson entre a complexidade dos programas e as métricas avaliadas indica que a complexidade não foi um fator determinante para explicar o desempenho das abordagens. Os coeficientes observados foram baixos tanto para a cobertura dos testes gerados por IA (0,27) e pelo Pynguin (-0,07) quanto para o *mutation score* dos testes gerados por IA (0,10) e pelo Pynguin (0,03). Esses valores caracterizam correlações fracas ou praticamente inexistentes, sugerindo que a variação das métricas não está fortemente associada à complexidade dos programas analisados. Dessa forma, outros fatores, como as características da lógica implementada, a estrutura das funções, a presença de casos de borda e a natureza dos caminhos de execução, provavelmente exerceram maior influência sobre os resultados obtidos. Esse achado reforça a interpretação de que a superioridade observada da abordagem baseada em IA em relação ao Pynguin não decorre de um possível viés relacionado à complexidade dos problemas avaliados, mas sim de diferenças inerentes à capacidade de geração dos casos de teste produzidos por cada abordagem - cabe observar, no entanto, o recorte de apenas 120 trechos de código.

Por fim, cabe pontuar que nem a cobertura de código nem o *mutation score* apresentaram distribuição normal, justificando o emprego do teste não paramétrico de Wilcoxon. Para o *mutation score*, obteve-se $p = 4,05 \times 10^{-12}$, confirmando a forte significância estatística e a superioridade da geração por IA, como visto nas análises descritivas acima. A cobertura também alcançou significância ($p = 2,22 \times 10^{-3}$, valor inferior ao limite de $\alpha = 0,05$), reforçando as conclusões favoráveis à abordagem de *self-refine*. A análise da magnitude dessas diferenças via *Cliff's Delta* revelou comportamentos distintos entre as métricas. No que tange ao *mutation score*, observou-se um efeito de magnitude grande ($\delta = 0,7240$), corroborado por um *Cohen's d* expressivo ($d = 1,48$), o que demonstra que a superioridade da IA na detecção de mutantes possui alta relevância prática. Em contrapartida, para a métrica de cobertura, o efeito foi classificado como pequeno ($\delta = 0,1648$, $d = 0,37$). Esses resultados indicam que, embora a IA apresente vantagem sistemática em ambas as frentes, o impacto é substancialmente mais pronunciado na capacidade de revelar falhas do que na simples exploração de caminhos de execução, evidenciando que a IA gera casos de teste qualitativamente mais rigorosos mesmo quando a cobertura quantitativa se aproxima da do Pynguin.

5. Considerações finais

Este trabalho investigou a aplicação de modelos de Inteligência Artificial Generativa na geração automática de suítes de testes para problemas utilizados em plataformas de Juiz Online, com foco na redução do esforço necessário para a construção e manutenção de casos de validação em disciplinas de programação. Para isso, foram comparadas diferentes estratégias de *prompts* e, posteriormente, os testes produzidos pela abordagem baseada em IA foram confrontados com aqueles gerados pelo Pynguin, uma ferramenta amplamente utilizada para geração automática de testes em Python.

Os experimentos mostraram que a utilização de uma única interação com o modelo de linguagem não foi suficiente para garantir a obtenção de suítes de testes executáveis; apenas 37,8% das suítes geradas inicialmente foram aprovadas sem necessidade de refinamento. Entretanto, a adoção de uma estratégia baseada em *self-refining* elevou significativamente essa taxa de sucesso, permitindo que a maior parte das falhas identificadas fosse corrigida nas iterações subsequentes. Além da correção dos problemas encontrados, observou-se um aumento médio de 31,9% no número de casos de teste presentes nas suítes aprovadas quando comparadas às versões iniciais.

Em relação ao Pynguin, verificou-se que parte dos programas analisados não teve suas suítes geradas dentro do limite de tempo estabelecido, enquanto uma parcela significativa dos testes produzidos apresentou falhas de execução que impediram o cálculo do *mutation score*. Essas limitações impactaram diretamente a quantidade de artefatos válidos disponíveis para avaliação e evidenciaram dificuldades da ferramenta em lidar com determinados programas presentes no conjunto analisado.

A comparação entre as duas abordagens indicou resultados favoráveis à estratégia baseada em IA. No critério de cobertura de código, os testes gerados por IA apresentaram desempenho igual ou superior ao do Pynguin em todos os arquivos avaliados, alcançando os mesmos níveis de cobertura em aproximadamente 70% dos casos e superando a ferramenta nos demais. Resultados ainda mais expressivos foram observados na análise de *mutation score*, em que as suítes geradas por IA apresentaram valores substancialmente superiores aos obtidos pelo Pynguin na maior parte dos problemas avaliados. Em conjunto, esses resultados sugerem que a abordagem baseada em modelos de linguagem foi capaz de produzir testes mais efetivos na detecção de comportamentos incorretos e mutações artificiais do código.

Como trabalhos futuros, recomenda-se ampliar o conjunto de programas avaliados para aumentar a generalização dos resultados, investigar abordagens híbridas que combinem geração automática de testes e modelos de linguagem, e realizar estudos em ambientes educacionais reais para avaliar o impacto dessas técnicas na qualidade do *feedback* e na confiabilidade dos sistemas de correção automática. Além disso, futuras pesquisas podem analisar o equilíbrio entre custo computacional, consumo de *tokens* e qualidade dos testes gerados por modelos de linguagem.

Por fim, cabe dizer que este estudo possui algumas limitações que devem ser consideradas. Os resultados podem ter sido influenciados pela configuração do modelo GPT-5.4, pela variabilidade natural das respostas geradas por modelos de linguagem e pelo limite de execução adotado, que pode ter afetado de forma diferente as ferramentas avaliadas. Além disso, métricas como cobertura de código e *mutation score* podem não cap-

turar completamente a qualidade dos testes gerados, especialmente devido à presença de asserções fracas e às limitações da ferramenta de mutação utilizada. O conjunto de dados analisado também foi relativamente pequeno, com 120 soluções, e composto exclusivamente por problemas algorítmicos em Python extraídos do LeetCode, o que restringe a generalização dos resultados para aplicações reais, sistemas mais complexos e outras linguagens de programação.

Referências

- Bhatia, S., Gandhi, T., Kumar, D., and Jalote, P. (2024). Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code, LLM4Code '24*, page 54–61, New York, NY, USA. Association for Computing Machinery.
- Cao, Y., Chen, Z., Quan, K., Zhang, Z., Wang, Y., Dong, X., Feng, Y., He, G., Huang, J., Li, J., Tan, Y., Tang, J., Tang, Y., Wu, J., Xiao, Q., Zheng, C., Zhou, S., Zhu, Y., Huang, Y., and He, T. (2026). Can llms generate reliable test case generators? a study on competition-level programming problems.
- Carvalho, L., Oliveira, D., and Gadelha, B. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In *Anais do XXVII Simpósio Brasileiro de Informática na Educação*, pages 140–149, Porto Alegre, RS, Brasil. SBC.
- Chen, P.-Y. (2026). Leetcode solutions. <https://github.com/walkccc/LeetCode>. Repositório GitHub. Acesso em: 31 maio 2026.
- Dantas, J. and Aranha, E. (2025). Sistema multiagente para a geração automática de questões de programação. In *Anais Estendidos do XIV Congresso Brasileiro de Informática na Educação*, pages 346–352, Porto Alegre, RS, Brasil. SBC.
- Francisco, R., Júnior, C. P., and Ambrósio, A. P. (2016). Juiz online no ensino de programação introdutória - uma revisão sistemática da literatura. In *Anais do XXVII Simpósio Brasileiro de Informática na Educação*, pages 11–20, Porto Alegre, RS, Brasil. SBC.
- Guilherme, V. and Vincenzi, A. (2023). An initial investigation of chatgpt unit test generation capability. In *Anais do VIII Simpósio Brasileiro de Testes de Software Sistemático e Automatizado*, page 15–24, Porto Alegre, RS, Brasil. SBC.
- Lukasczyk, S. and Fraser, G. (2022). Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, ICSE '22*, page 168–172, New York, NY, USA. Association for Computing Machinery.
- Oshin, M. and Campos, N. (2025). *Learning LangChain: Building AI and LLM Applications with LangChain and LangGraph*. O'Reilly Media, Inc., Sebastopol, CA, USA. Copyright © 2025 Olumayowa “Mayo” Olufemi Oshin. All rights reserved. Printed in the United States of America.
- Santana, A., Silva, F. G., Dantas, J., Souza, J., and Aranha, E. (2025a). Geração automática de questões de programação usando LLM: Um relato de experiência. In *Anais*

do XXXIII Workshop sobre Educação em Computação (WEI), pages 1415–1425, Maciό, AL, Brasil. Sociedade Brasileira de Computação.

Santana, A., Silva, F. G., Souza, J. L. G., da Silva Dantas, J. C., and Aranha, E. (2025b). Geração de questões de programação baseada em templates e ia generativa. In *Workshop de Informática na Escola, CBIE*.

Shah, C. (2024). From prompt engineering to prompt science with human in the loop. In *Proceedings of ACM Conference on Human in the Loop*, Seattle, WA, USA. arXiv preprint arXiv:2401.04122, May 10, 2024.

Silva, E., Coelho, R., and Silva, L. (2025). Llms as test generators: A comparative benchmarking study. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*, pages 25–36, Porto Alegre, RS, Brasil. SBC.

Tang, Y., Liu, Z., Zhou, Z., and Luo, X. (2024). Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*.

Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., and Lou, Y. (2024). Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE).