REVISTA TÓPICOS
ISSN 2965-6672 | Qualis A2

# AVALIAÇÃO DA CORREÇÃO FUNCIONAL E DA CAPACIDADE DE DETECÇÃO DE TESTES EM CÓDIGO GERADO POR INTELIGÊNCIA ARTIFICIAL GENERATIVA

## EVALUATION OF FUNCTIONAL CORRECTNESS AND TEST DETECTION CAPABILITY IN CODE GENERATED BY GENERATIVE ARTIFICIAL INTELLIGENCE

Ciências Exatas e da Terra • 09/11/2026

REGISTRO DOI: 10.70773/revistatopicos/794136472

Regina Maria Duarte de Oliveira[1]

**RESUMO**

A inteligência artificial generativa amplia as possibilidades de desenvolvimento de software ao permitir a produção de código-fonte a partir de instruções em linguagem natural. Nesse cenário, a avaliação da correção funcional e da eficácia dos testes automatizados torna-se essencial para verificar a confiabilidade das implementações. Este estudo tem como objetivo avaliar a conformidade funcional de códigos Java gerados por inteligência artificial e investigar a capacidade de detecção das suítes de testes utilizadas. A pesquisa apresenta caráter empírico e exploratório, envolvendo três tarefas de programação: cálculo de descontos, controle de estoque e gerenciamento de reservas. Foram produzidas quinze implementações independentes com o modelo GPT-6, avaliadas por meio de testes automatizados com JUnit 5. Como etapa complementar, utilizou-se a ferramenta PIT Mutation Testing para analisar a primeira implementação de cada tarefa. Todas as soluções foram aprovadas nas 270 execuções de casos de teste. Na auditoria de mutação, 62 dos 73 mutantes gerados foram inicialmente detectados. Após a inclusão de testes complementares direcionados a condições de fronteira e à validação de uma entrada nula, esse número aumentou para 66, permanecendo sete mutantes sobreviventes. Os resultados evidenciam a importância de associar testes funcionais à análise de mutação, proporcionando uma avaliação mais criteriosa da capacidade de detecção dos testes aplicados ao código gerado por inteligência artificial.

**Palavras-chave:** Inteligência artificial generativa; Geração de código; Correção funcional; Testes automatizados; Testes de mutação.

**ABSTRACT**

Generative artificial intelligence expands software development possibilities by enabling source code generation from natural

language instructions. In this context, assessing functional correctness and the effectiveness of automated testing is essential for evaluating the reliability of software implementations. This study aims to assess the functional conformance of Java code generated by artificial intelligence and investigate the fault-detection capability of the test suites employed. The research follows an empirical and exploratory approach involving three programming tasks: discount calculation, inventory control, and room reservation management. Fifteen independent implementations were generated using the GPT-6 model and evaluated through automated tests with JUnit 5. As a complementary step, PIT Mutation Testing was applied to the first implementation of each task. All implementations passed the 270 test-case executions. During mutation testing, 62 of the 73 generated mutants were initially detected. After additional tests targeting boundary conditions and null-input validation were introduced, the number of detected mutants increased to 66, leaving seven surviving mutants. These findings highlight the importance of combining functional testing with mutation analysis to provide a more rigorous assessment of the fault-detection capability of automated tests applied to AI-generated code.
**Keywords:** Generative artificial intelligence; Code generation; Functional correctness; Automated testing; Mutation testing.

## 1. INTRODUÇÃO

A Inteligência Artificial Generativa vem ganhando espaço na Engenharia de Software, sobretudo no apoio à escrita, à análise e à manutenção de código-fonte. Com os avanços dos modelos de linguagem de grande porte, tornou-se possível obter implementações a partir de descrições em linguagem natural, o que vem modificando a maneira como determinadas atividades de

desenvolvimento são realizadas. Essa facilidade, porém, levanta uma questão importante: até que ponto o código produzido por esses modelos atende corretamente aos requisitos solicitados? A geração de uma solução aparentemente funcional não significa, necessariamente, que todas as regras de negócio tenham sido contempladas. Nesse sentido, Tosi (2024) e Idrisov e Schlippe (2024) discutem a qualidade e a correção de programas gerados por inteligência artificial, mostrando a importância de avaliar essas implementações para além de sua execução inicial.

Um dos principais desafios dessa avaliação está em verificar se o programa responde adequadamente às diferentes situações previstas nos requisitos. Os testes automatizados são amplamente utilizados para essa finalidade, pois permitem confrontar os resultados obtidos com os comportamentos esperados. Ainda assim, uma implementação pode passar por todos os testes disponíveis e apresentar falhas em situações que não foram contempladas pela suíte. Essa limitação foi explorada por Liu et al. (2023), que identificaram problemas na capacidade de conjuntos de testes empregados em benchmarks de programação de distinguir soluções corretas de implementações com comportamentos inadequados. O estudo evidencia que a quantidade e a abrangência dos cenários avaliados influenciam diretamente a confiança que se pode atribuir aos resultados.

Outro aspecto a ser considerado é que executar diferentes partes do código não significa, necessariamente, verificar seu comportamento de maneira eficaz. Uma suíte pode alcançar elevada cobertura e, mesmo assim, deixar passar alterações que comprometam determinadas regras funcionais. Inozemtseva e Holmes (2014) investigaram essa relação e observaram que a cobertura,

considerada isoladamente, apresenta limitações como indicador da eficácia dos testes. Uma alternativa para aprofundar essa análise são os testes de mutação. Nessa técnica, pequenas modificações são introduzidas artificialmente no código para verificar se os casos de teste conseguem identificar os comportamentos alterados. A ferramenta PIT, apresentada por Coles et al. (2016), permite aplicar essa abordagem a programas Java e obter indicadores sobre a capacidade de detecção das suítes.

A análise de mutação, entretanto, também apresenta limitações. Nem toda modificação introduzida no código provoca uma diferença perceptível em seu funcionamento. Algumas alterações preservam o comportamento original e dão origem aos chamados mutantes equivalentes, cuja identificação exige uma análise cuidadosa (Schuler; Zeller, 2013). Ao mesmo tempo, o uso dessa técnica em pesquisas envolvendo inteligência artificial tem recebido atenção. Cabral et al. (2025), por exemplo, avaliaram testes unitários gerados por modelos de linguagem para programas Java, utilizando testes de mutação como um dos critérios de análise. Embora esse tipo de investigação se concentre na qualidade dos testes produzidos por IA, seus resultados ajudam a compreender a importância dos mecanismos utilizados para avaliar programas.

A partir dessas discussões, surge o problema que orienta este estudo: em que medida implementações Java geradas por inteligência artificial generativa atendem aos requisitos funcionais estabelecidos e qual é a capacidade dos testes automatizados de identificar alterações potencialmente defeituosas nesses programas? A questão parte de uma preocupação prática da Engenharia de Software: a aprovação em uma suíte de testes representa um resultado favorável, mas sua interpretação depende

também da capacidade dessa suíte de revelar comportamentos incorretos. Por esse motivo, avaliar somente a quantidade de testes aprovados pode oferecer uma visão incompleta da confiabilidade das implementações.

A pesquisa se justifica pela necessidade de compreender melhor os critérios empregados na validação de código gerado por inteligência artificial, especialmente em atividades de desenvolvimento e garantia da qualidade de software. Para isso, foram selecionadas três tarefas de programação em Java, envolvendo cálculo de descontos, controle de estoque e gerenciamento de reservas. Cada tarefa foi submetida a cinco gerações independentes com o modelo GPT-6, resultando em quinze implementações. A avaliação foi realizada com testes funcionais definidos previamente, seguida de uma análise das diferenças estruturais entre as soluções e de uma auditoria complementar das suítes por meio da ferramenta PIT. Também foram examinados os efeitos da inclusão de testes complementares voltados a condições de fronteira e ao tratamento de entradas inválidas, elaborados a partir da inspeção dos mutantes sobreviventes.

O objetivo deste trabalho é avaliar a correção funcional observada em implementações Java geradas por inteligência artificial generativa e examinar a capacidade das suítes de testes automatizados utilizadas para verificar seu comportamento. A análise considera os resultados dos testes funcionais, as diferenças encontradas entre implementações de uma mesma tarefa e a capacidade de detecção de mutações artificiais, antes e depois da inclusão de casos de teste complementares. Com essa abordagem, pretende-se contribuir para a discussão sobre os limites dos testes automatizados na avaliação de código gerado por IA oferecendo

evidências experimentais que auxiliem na interpretação dos resultados e na adoção de práticas de validação mais criteriosas.

## 2. FUNDAMENTAÇÃO TEÓRICA

### 2.1. Inteligência Artificial Generativa na Engenharia de Software

Os modelos de linguagem de grande porte passaram a ocupar um espaço cada vez mais significativo no desenvolvimento de software. Uma de suas aplicações mais conhecidas é a geração de código-fonte a partir de instruções escritas em linguagem natural, permitindo que desenvolvedores obtenham implementações iniciais sem precisar escrever cada trecho manualmente. Embora essa possibilidade represente uma mudança importante na prática de programação, o código produzido ainda precisa ser avaliado. Uma solução pode apresentar sintaxe válida, compilar corretamente e, mesmo assim, não contemplar todas as regras previstas para seu funcionamento.

Ao investigar esse problema, Tosi (2024) avaliou códigos produzidos por diferentes ferramentas de inteligência artificial para três tarefas de programação em Java. O autor utilizou testes funcionais e métricas de qualidade para examinar as implementações, identificando tanto a capacidade dos modelos de resolver os problemas propostos quanto a necessidade de supervisão técnica. O estudo chama a atenção para um aspecto importante: a geração automática não elimina as etapas de verificação que fazem parte do desenvolvimento de software.

A qualidade do código gerado também foi abordada por Idrisov e Schlippe (2024), que analisaram programas desenvolvidos em Java, Python e C++. Os pesquisadores consideraram critérios como

correção, eficiência e manutenibilidade, utilizando, entre outros indicadores, a complexidade ciclomática e as métricas de Halstead. Essa abordagem permite observar que a avaliação de um programa não se limita ao resultado de sua execução. Implementações diferentes podem atender aos mesmos requisitos, mas apresentar diferenças na organização do código, na quantidade de decisões lógicas e na facilidade de compreensão.

Essas características ganham importância quando o software precisa ser modificado. Borg et al. (2026) investigaram os efeitos do uso de assistentes de IA sobre a manutenção posterior do código, em um experimento com 151 participantes. Na etapa dedicada à evolução das implementações, os pesquisadores não encontraram diferenças estatisticamente significativas no tempo necessário para concluir as atividades nem na qualidade avaliada. Os resultados não permitem afirmar que o código desenvolvido com auxílio de IA seja necessariamente mais fácil ou mais difícil de manter. Indicam, porém, que os possíveis efeitos dessa tecnologia precisam ser examinados a partir de critérios específicos, considerando as condições em que o software for desenvolvido e utilizado.

### 2.2. Correção Funcional e Eficácia dos Testes Automatizados

Verificar a correção funcional de um programa significa examinar se ele responde conforme o esperado às situações descritas em seus requisitos. Essa avaliação envolve tanto o processamento de entradas válidas quanto o tratamento de condições excepcionais, como valores fora dos limites permitidos, operações inconsistentes e identificadores inválidos. Em aplicações que trabalham com regras de negócio, também é necessário verificar se operações malsucedidas preservam a integridade dos dados. Os testes

automatizados ajudam nesse processo ao permitir que os mesmos cenários sejam executados repetidamente, com resultados que podem ser comparados aos comportamentos esperados.

O problema é que a aprovação em uma suíte de testes não significa, por si só, que o programa esteja livre de falhas. Liu et al. (2023) investigaram essa questão ao desenvolver o EvalPlus, uma estrutura voltada à avaliação mais rigorosa de programas gerados por modelos de linguagem. Ao ampliar os casos de teste de benchmarks utilizados na avaliação desses modelos, os autores encontraram implementações que haviam sido aprovadas anteriormente, mas falharam diante de novas entradas. O trabalho evidencia uma limitação importante: a confiança atribuída à correção de uma solução depende também da abrangência dos testes utilizados para avaliá-la.

Outro cuidado envolve a interpretação das métricas de cobertura. Uma linha de código pode ser executada durante um teste sem que o resultado dessa execução seja verificado de maneira suficiente. Inozemtseva e Holmes (2014) analisaram a relação entre cobertura e eficácia dos testes, encontrando uma associação baixa a moderada quando considerado o efeito da quantidade de casos de teste. Isso significa que alcançar uma cobertura elevada não assegura, necessariamente, que a suíte consiga identificar alterações responsáveis por comportamentos incorretos.

No caso de programas gerados por inteligência artificial, essas limitações merecem atenção especial. Os testes funcionais permitem verificar a conformidade das implementações com os cenários definidos, mas não oferecem evidências sobre todas as situações que poderiam ocorrer durante sua utilização. Uma

avaliação mais consistente pode, portanto, combinar a execução dos casos previstos com técnicas que examinem a capacidade dos testes de perceber modificações no comportamento do programa.

### 2.3. Testes de Mutação e Limitações da Avaliação

Os testes de mutação oferecem uma forma de investigar a eficácia de uma suíte a partir de alterações controladas no código. A técnica consiste em modificar pequenos elementos de um programa, como operadores relacionais, expressões aritméticas, condições e chamadas de métodos. Cada versão modificada, denominada mutante, é submetida aos testes existentes. Quando um teste falha em razão da alteração, considera-se que o mutante foi detectado. Caso os testes continuem sendo aprovados, o mutante permanece sobrevivente. A relação entre o número de mutantes detectados e o total considerado na análise permite calcular a pontuação de mutação.

Para aplicações Java, uma das ferramentas disponíveis é o PIT, apresentada por Coles et al. (2016). A ferramenta aplica mutações ao código compilado e utiliza mecanismos de otimização para reduzir o custo de execução da análise. Sua integração com o Maven facilita a utilização em projetos que já possuem testes automatizados, enquanto os relatórios gerados permitem localizar as alterações realizadas e verificar quais foram identificadas pela suíte.

Apesar dessas possibilidades, a pontuação de mutação precisa ser interpretada com cautela. Schuler e Zeller (2013) estudaram os chamados mutantes equivalentes, que surgem quando uma alteração não produz diferenças no comportamento observável do programa. Nesses casos, mesmo uma suíte suficientemente

abrangente pode não detectar a modificação, pois o resultado funcional permanece o mesmo. A identificação de mutantes equivalentes não é simples e constitui uma das dificuldades da técnica. Por essa razão, encontrar um mutante sobrevivente não significa, automaticamente, que exista uma falha na implementação ou uma deficiência nos testes.

Há ainda uma distinção necessária entre o código originalmente desenvolvido e as alterações introduzidas durante a análise. Os mutantes são produzidos artificialmente para avaliar a capacidade de detecção dos testes. Eles não devem ser confundidos com defeitos que já estavam presentes no programa antes da aplicação da ferramenta. Essa diferença é particularmente relevante em pesquisas sobre código gerado por inteligência artificial, pois evita atribuir ao modelo erros que foram inseridos exclusivamente como parte do procedimento experimental.

### 2.4. Testes de Mutação na Avaliação de Sistemas Apoiados por Inteligência Artificial

O uso de testes de mutação em pesquisas envolvendo modelos de linguagem tem contribuído para investigar a qualidade das suítes automatizadas e desenvolver estratégias para aprimorá-las. Cabral et al. (2025), por exemplo, avaliaram testes unitários gerados pelo ChatGPT e pelo DeepSeek para classes Java provenientes do conjunto Defects4J. A pesquisa utilizou indicadores de mutação para examinar a capacidade dos testes produzidos pelos modelos, considerando também dificuldades relacionadas à geração e à execução dos casos. A comparação mostrou diferenças entre as ferramentas avaliadas e destacou a importância de verificar se os

testes conseguem identificar comportamentos incorretos, em vez de considerar apenas sua execução bem-sucedida.

Outra contribuição nessa área foi apresentada por Moradi Dakhel et al. (2024), que desenvolveram o MuTAP. A proposta utiliza os resultados da análise de mutação como orientação para a geração de novos testes por modelos de linguagem. Quando determinados mutantes sobrevivem, essas informações são aproveitadas para direcionar a criação de casos capazes de examinar comportamentos ainda pouco explorados. Os resultados indicaram melhorias na capacidade de detecção de programas defeituosos em relação às abordagens utilizadas como comparação. A pesquisa mostra, assim, que os mutantes sobreviventes podem servir não apenas como indicadores de limitações, mas também como ponto de partida para aprimorar uma suíte.

Lu et al. (2026) também investigaram a ampliação automatizada de suítes de testes com apoio de modelos de linguagem, utilizando mutantes sobreviventes para orientar o processo. O estudo examinou métodos com diferentes características e avaliou os efeitos dos testes adicionais sobre a pontuação de mutação e a identificação de implementações incorretas. Os pesquisadores observaram melhorias na capacidade de detecção após a ampliação das suítes, indicando que a avaliação inicial de um programa pode deixar de identificar comportamentos inadequados quando os testes disponíveis não contemplam determinadas situações.

Embora esses trabalhos utilizem técnicas semelhantes, existem diferenças importantes entre seus objetivos. Cabral et al. (2025), Moradi Dakhel et al. (2024) e Lu et al. (2026) investigam, sob diferentes perspectivas, a geração e o aprimoramento de testes com

auxílio de modelos de linguagem. Nesta pesquisa, o objeto principal de avaliação é o código gerado pela inteligência artificial. As implementações são submetidas a suítes previamente definidas, enquanto os testes de mutação são utilizados em uma etapa complementar para examinar a capacidade dessas suítes de detectar alterações no programa. A inclusão posterior de casos específicos também permite observar se determinadas limitações identificadas na auditoria podem ser reduzidas.

Os estudos apresentados mostram que a avaliação de código gerado por inteligência artificial envolve duas questões relacionadas, mas distintas: verificar se a implementação atende aos requisitos e determinar até que ponto os testes utilizados são capazes de revelar comportamentos incorretos. A primeira questão diz respeito aos resultados funcionais do programa; a segunda, à qualidade dos mecanismos empregados para avaliá-lo. É nessa relação que se concentra o presente trabalho, ao analisar implementações Java geradas por inteligência artificial generativa e utilizar testes de mutação para complementar a interpretação dos resultados funcionais. A proposta não é demonstrar a ausência absoluta de defeitos, mas examinar o alcance das evidências obtidas e suas limitações para a garantia da qualidade de software.

## 3. METODOLOGIA

### 3.1. Caracterização da Pesquisa

Esta pesquisa caracteriza-se como um estudo empírico exploratório, de abordagem predominantemente quantitativa, com análise descritiva de implementações de software geradas por inteligência artificial. O trabalho foi desenvolvido em ambiente computacional,

sem a participação de voluntários, tendo como unidades de análise os códigos-fonte produzidos por um modelo de linguagem e os resultados obtidos durante sua avaliação.

O procedimento experimental foi dividido em duas etapas. A primeira concentrou-se na geração de implementações Java e na verificação de seu comportamento por meio de testes funcionais previamente definidos. A segunda consistiu em uma auditoria complementar das suítes de testes, utilizando a técnica de análise de mutação para investigar sua capacidade de detectar alterações introduzidas artificialmente nos programas.

A amostra foi selecionada de forma intencional, considerando tarefas com diferentes quantidades de operações, regras de validação e condições de execução. Não se pretendeu representar estatisticamente toda a variedade de aplicações desenvolvidas com inteligência artificial, mas examinar, em um conjunto delimitado de problemas, a relação entre os resultados funcionais das implementações e os instrumentos empregados em sua avaliação.

### 3.2. Definição das Tarefas e Composição da Amostra

Foram elaboradas três tarefas de programação em Java, cada uma acompanhada de requisitos funcionais e de uma interface de implementação previamente estabelecida. A seleção procurou contemplar situações distintas de desenvolvimento, desde uma operação de cálculo até funcionalidades que exigem controle de estado e verificação de conflitos.

A primeira tarefa, identificada como T01, consistiu na implementação de uma calculadora de descontos. O programa deveria receber valores monetários e percentuais utilizando a classe

BigDecimal, rejeitar entradas nulas ou fora dos limites permitidos e retornar o resultado com duas casas decimais, empregando arredondamento HALF_UP.

A segunda tarefa, T02, envolveu um sistema de controle de estoque em memória. Os requisitos contemplaram cadastro e consulta de produtos, entrada e saída de quantidades e transferência de saldos entre identificadores distintos. Também foram estabelecidas regras para impedir quantidades inválidas, saldos negativos, operações que ultrapassassem o limite do tipo inteiro e alterações indevidas do estado após operações rejeitadas.

A terceira tarefa, T03, correspondeu a um sistema de reservas de salas. Cada reserva deveria possuir identificador único, sala e intervalo de horário, representado por valores inteiros entre 0 e 1440 minutos. O programa deveria impedir reservas sobrepostas na mesma sala, permitir horários adjacentes, realizar cancelamentos e preservar o estado diante de operações inválidas. Para a verificação de conflitos, adotou-se a interpretação de intervalos semiabertos, nos quais o instante inicial pertence à reserva e o instante final não pertence.

Para cada tarefa foram solicitadas cinco implementações independentes, identificadas de G01 a G05, totalizando quinze códigos-fonte. As tarefas foram tratadas como problemas distintos, enquanto as cinco gerações de cada uma permitiram observar possíveis variações entre soluções produzidas a partir dos mesmos requisitos. A classificação das tarefas por complexidade foi utilizada apenas como referência descritiva de seu escopo, sem constituir uma medida formal de complexidade computacional.

### 3.3. Ambiente de Desenvolvimento e Geração dos Códigos

O experimento utilizou Java 21 (Oracle, s.d.), Maven para gerenciamento e execução dos projetos e JUnit 5 (JUnit, s.d.) para a implementação dos testes automatizados. As classes foram organizadas em pacotes específicos para cada tarefa, preservando os nomes dos métodos e suas assinaturas, conforme os contratos funcionais estabelecidos.

A geração dos códigos foi realizada por meio da interface do ChatGPT, com seleção do modelo GPT-6, identificado conforme a documentação técnica da OpenAI (s.d.). Cada implementação foi solicitada em uma nova conversa temporária, sem utilização de histórico personalizado. Para uma mesma tarefa, foi empregado um prompt padronizado contendo o contexto do projeto, a classe a ser implementada e os requisitos funcionais correspondentes. O objetivo desse procedimento foi reduzir diferenças decorrentes da formulação das instruções e evitar que as respostas anteriores fossem utilizadas como contexto de novas gerações.

Após cada resposta, o código produzido foi identificado de acordo com sua tarefa e geração e inserido no respectivo projeto Java. As implementações foram mantidas sem correções manuais antes da execução dos testes, de modo que os resultados refletissem o comportamento das soluções originalmente fornecidas pelo modelo. Foram preservados os arquivos de código e os registros das execuções para posterior análise.

Antes da coleta definitiva, foram realizados procedimentos preliminares para verificar a estrutura dos projetos e o funcionamento do ambiente de testes. Esses registros não

integraram a amostra oficial. A avaliação principal considerou exclusivamente as quinze implementações obtidas após a definição do protocolo experimental.

### 3.4. Avaliação Funcional das Implementações

A verificação funcional foi realizada com suítes de testes JUnit 5, elaboradas a partir dos requisitos de cada tarefa antes da coleta principal. Cada suíte reuniu casos públicos e reservados, destinados a examinar operações válidas, rejeição de entradas inválidas, condições de fronteira e preservação do estado dos programas, quando aplicável.

A tarefa T01 contou com quatorze casos de teste por implementação, sendo quatro públicos e dez reservados. Na T02 foram utilizados dezoito casos, distribuídos entre seis públicos e doze reservados. Para a T03, a suíte foi composta por vinte e dois casos, dos quais oito eram públicos e quatorze reservados.

Cada uma das cinco implementações de uma tarefa foi submetida à mesma suíte funcional. A execução ocorreu por meio do comando mvn clean test, utilizando o Maven para compilar o projeto e executar os testes. Os resultados foram registrados com base na quantidade de testes executados, aprovados e reprovados, além de eventuais erros de execução.

Ao todo, o planejamento compreendeu 270 execuções de casos de teste, correspondentes às quinze implementações avaliadas. Esse total representa aplicações repetidas dos casos às diferentes gerações, não 270 programas independentes. A aprovação de um caso foi interpretada como evidência de conformidade com o

comportamento especificamente verificado, sem pressupor a ausência de defeitos em situações não contempladas pela suíte.

A Tabela 1 apresenta uma síntese das tarefas utilizadas no experimento, da quantidade de implementações geradas e da composição das suítes de testes. Também identifica as implementações selecionadas para a auditoria de mutação e a quantidade de casos acrescentados na etapa complementar.

**Tabela 1.** Caracterização das tarefas e do procedimento experimental.

| Característica | T01 | T02 | T03 |
|---|---|---|---|
| **Funcionalidade** | Cálculo de descontos | Controle de estoque | Gerenciamento de reservas |
| **Linguagem** | Java 21 | Java 21 | Java 21 |
| **Implementações geradas** | 5 | 5 | 5 |
| **Testes públicos** | 4 | 6 | 8 |
| **Testes reservados** | 10 | 12 | 14 |
| **Testes por implementação** | 14 | 18 | 22 |
| **Implementação selecionada para auditoria** | G01 | G01 | G01 |
| **Testes complementares** | 0 | 4 | 2 |

**Fonte:** Autoria própria (2026).

Os testes complementares foram aplicados exclusivamente às implementações G01 durante a auditoria de mutação e não integram as suítes utilizadas na avaliação funcional das quinze implementações.

### 3.5. Auditoria das Suítes por Meio de Testes de Mutação

Após a avaliação funcional, realizou-se uma auditoria complementar com a ferramenta PIT Mutation Testing, versão 1.19.4, integrada ao Maven e ao JUnit 5, conforme os procedimentos de utilização documentados pelo PITEST (s.d.). Diferentemente da etapa anterior, que envolveu quinze implementações, essa auditoria utilizou somente a primeira geração de cada tarefa, identificada como G01. A escolha permitiu realizar uma investigação inicial da capacidade de detecção das suítes, sem atribuir seus resultados automaticamente às demais gerações.

A análise foi conduzida em cópias separadas dos projetos, preservando os arquivos originais. O PIT introduziu mutações em elementos do código compilado, incluindo alterações de operadores relacionais e aritméticos, inversões de condições, substituições de valores retornados e remoções de chamadas de métodos. Os programas modificados foram submetidos aos testes existentes, permitindo classificar os mutantes conforme os resultados da execução.

Foram considerados detectados os mutantes que provocaram a reprovação de pelo menos um teste. Os mutantes que não ocasionaram falhas permaneceram classificados como sobreviventes, sujeitos a uma avaliação posterior de seu significado. Essa análise é necessária porque algumas modificações podem

preservar o comportamento funcional do programa, caracterizando possíveis mutantes equivalentes. Assim, os sobreviventes não foram automaticamente tratados como defeitos existentes nas implementações originais.

A pontuação de mutação foi calculada pela razão entre o número de mutantes detectados e o total de mutantes gerados pelo PIT, expressa em percentual. O indicador foi empregado para comparar a capacidade de detecção das suítes nas diferentes etapas da auditoria. Os possíveis mutantes equivalentes foram mantidos no denominador, uma vez que não houve classificação formal que justificasse sua exclusão.

Após a primeira execução do PIT, os relatórios foram examinados para identificar os mutantes sobreviventes e verificar quais alterações poderiam revelar limitações das suítes. Essa análise deu origem a uma etapa exploratória de complementação dos testes. Inicialmente, foram acrescentados três casos na T02, voltados a operações realizadas nos limites permitidos para os saldos e as transferências, e dois na T03, destinados à verificação dos horários extremos aceitos pelo sistema de reservas.

Na T02, a análise posterior dos mutantes ainda sobreviventes identificou uma situação que exigia investigação adicional. A remoção da validação do identificador de origem no método de transferência poderia permitir que um valor nulo provocasse uma NullPointerException, em vez da IllegalArgumentException exigida pelos requisitos. Para verificar esse comportamento, foi elaborado um quarto teste complementar, responsável por avaliar a exceção lançada em uma transferência com origem nula e a preservação do saldo após a operação rejeitada.

As auditorias foram repetidas após a inclusão dos testes complementares, sem alterações nas implementações originais ou na configuração do PIT. Na T02, o procedimento compreendeu três execuções: a primeira com a suíte original, a segunda após a inclusão dos três testes de fronteira e a terceira após o teste específico para origem nula. Na T03, foram realizadas duas execuções, antes e depois da inclusão dos testes adicionais. Essa sequência permitiu acompanhar as alterações na quantidade de mutantes detectados e sobreviventes. Como os novos casos foram elaborados a partir da inspeção dos relatórios iniciais, os resultados foram tratados como parte de uma investigação exploratória, permanecendo separados da avaliação funcional originalmente planejada.

### 3.6. Organização e Análise dos Dados

Os dados foram organizados por tarefa, geração e etapa de avaliação. Na análise funcional, foram registrados os resultados das suítes de testes para cada implementação, permitindo calcular as proporções de aprovação e identificar eventuais falhas associadas aos requisitos avaliados.

Também foi realizada uma inspeção descritiva dos códigos-fonte, com atenção às estruturas de dados utilizadas, à organização dos métodos auxiliares e às estratégias de validação. Essa comparação teve como finalidade identificar diferenças de implementação entre respostas produzidas para uma mesma tarefa. Não foi adotado um índice quantitativo único de diversidade estrutural, de modo que as observações foram tratadas como características descritivas dos códigos examinados.

Na auditoria com PIT, foram registrados os mutantes gerados, detectados e sobreviventes, além da pontuação de mutação correspondente. Para as tarefas T02 e T03, os indicadores foram comparados entre as execuções realizadas antes e depois da inclusão dos testes complementares. Na T02, também foi considerada a terceira execução, destinada a verificar especificamente a remoção da validação do identificador de origem. Os resultados dessa etapa permaneceram separados da avaliação funcional das implementações originais, uma vez que as alterações produzidas pelo PIT não constituem defeitos naturalmente gerados pelo modelo de inteligência artificial.

A análise foi predominantemente descritiva, por meio de tabelas, proporções e comparação entre os resultados observados. Não foram realizados testes de hipóteses ou inferências estatísticas sobre populações mais amplas, considerando o número reduzido de tarefas e o caráter intencional da amostra.

Por fim, a interpretação dos resultados considerou limitações relacionadas à abrangência dos testes, à possibilidade de mutantes equivalentes, à utilização de apenas uma implementação por tarefa na auditoria e às condições de geração dos códigos pela interface do modelo. Esses aspectos foram considerados para delimitar as conclusões e evitar generalizações sobre a correção de programas produzidos por inteligência artificial generativa.

## 4. RESULTADOS E DISCUSSÕES

### 4.1. Resultados da Avaliação Funcional

A avaliação funcional reuniu quinze implementações Java produzidas pelo modelo GPT-6 para três tarefas de programação. Os

códigos foram submetidos às respectivas suítes de testes sem alterações manuais na lógica originalmente gerada. Todas as implementações passaram nos testes previstos, não sendo registradas falhas durante as execuções oficiais.

A Tabela 2 reúne os resultados obtidos em cada tarefa, considerando a quantidade de implementações, os casos de teste executados e as respectivas taxas de aprovação.

**Tabela 2.** Resultados da avaliação funcional das implementações.

| Tarefa | Implementações | Testes por implementação | Testes aprovados | Taxa de aprovação |
|---|---|---|---|---|
| **T01 — Calculadora de descontos** | 5 | 14 | 70/70 | 100% |
| **T02 — Controle de estoque** | 5 | 18 | 90/90 | 100% |
| **T03 — Sistema de reservas** | 5 | 22 | 110/110 | 100% |
| **Total** | **15** | — | **270/270** | **100%** |

**Fonte:** Autoria própria (2026).

Na T01, os testes contemplaram o cálculo dos descontos, a validação dos parâmetros e o arredondamento dos valores. Todas as cinco implementações apresentaram os resultados esperados. O mesmo ocorreu na T02, cujas verificações envolveram cadastro de produtos, movimentações de estoque, transferências e rejeição de operações

inválidas. Na T03, as implementações também foram aprovadas nos cenários de criação e cancelamento de reservas, validação de horários e identificação de conflitos entre intervalos.

Esses resultados indicam que, para os requisitos e casos de teste considerados, o modelo produziu soluções funcionalmente adequadas. É necessário, porém, observar o alcance dessa constatação. As 270 aprovações correspondem à execução das suítes sobre quinze implementações, e não à avaliação de 270 programas independentes. Além disso, não encontrar falhas nos cenários previstos não significa que o código esteja livre de defeitos.

Liu et al. (2023) discutem essa limitação ao demonstrar que a ampliação de conjuntos de testes pode revelar problemas em implementações anteriormente aprovadas. A observação é pertinente à presente pesquisa: embora as suítes tenham sido elaboradas a partir dos requisitos funcionais, elas não abrangem necessariamente todas as situações possíveis de execução. A aprovação integral deve, portanto, ser compreendida como uma evidência de conformidade nos cenários examinados, e não como comprovação de correção absoluta.

### 4.2. Diferenças Estruturais Entre as Implementações

O fato de todas as implementações terem alcançado os mesmos resultados funcionais não significa que os códigos apresentem a mesma organização. A inspeção das soluções revelou diferenças nas estratégias de cálculo, na representação dos dados e na distribuição das responsabilidades entre os métodos.

Na T01, foram observadas variações na forma de calcular o desconto e de organizar as expressões aritméticas. Algumas gerações,

entretanto, produziram códigos textualmente idênticos. Isso mostra que a realização de solicitações independentes, mesmo em conversas distintas, não resulta necessariamente em soluções diferentes.

As implementações da T02 apresentaram maior proximidade estrutural. Todas utilizaram estruturas baseadas em Map para armazenar os saldos dos produtos e adotaram verificações de identificadores, quantidades e limites numéricos antes de modificar os dados. As diferenças apareceram principalmente na organização dos métodos auxiliares, na nomenclatura e nas mensagens utilizadas para informar operações inválidas.

Na T03, as escolhas de implementação foram mais perceptíveis. Três gerações utilizaram record para representar as reservas, enquanto duas recorreram a classes internas estáticas. A organização da verificação de conflitos também variou. A G02 apresentou um método auxiliar denominado existeConflito(), ao passo que as outras quatro implementações realizaram essa verificação diretamente no método disponivel().

**Tabela 3.** Características estruturais observadas nas implementações da T03.

| Característica | G01 | G02 | G03 | G04 | G05 |
|---|---|---|---|---|---|
| **Representação da reserva** | record | Classe interna estática | record | record | Classe interna estática |
| **Armazenamento** | HashMap | HashMap | HashMap | HashMap | HashM |

| | | | | | |
|---|---|---|---|---|---|
| **Método** | Não | Sim | Não | Não | Não |

⚠ Esta tabela possui muitas colunas e foi cortada para impressão. Para visualizá-la completa, acesse o artigo original em: https://revistatopicos.com.br/artigos/avaliacao-da-correcao-funcional-e-da-capacidade-de-deteccao-de-testes-em-codigo-gerado-por-inteligencia-artificial-generativa?noblockage

**Fonte:** Autoria própria (2026).

Apesar dessas diferenças, nenhuma das estratégias apresentou resultados funcionais distintos nos testes executados. As implementações adotaram caminhos diferentes para atender às mesmas regras, mantendo o comportamento esperado dentro dos cenários avaliados.

Idrisov e Schlippe (2024) também consideram características internas do código ao avaliar soluções geradas por inteligência artificial, mostrando que a análise de qualidade envolve aspectos que vão além da correção funcional. Neste estudo, entretanto, a comparação estrutural foi realizada de maneira descritiva, sem o emprego de métricas formais de complexidade ou manutenibilidade. Por esse motivo, não é possível afirmar que determinada organização seja superior às demais. O que se observa é que soluções com resultados funcionais equivalentes podem apresentar decisões de implementação distintas.

### 4.3. Resultados da Auditoria com Testes de Mutação

A aprovação de todas as implementações nos testes funcionais levou à realização de uma auditoria complementar das suítes. O objetivo foi investigar se os casos disponíveis conseguiriam identificar alterações introduzidas artificialmente nos programas.

Para isso, utilizou-se o PIT Mutation Testing sobre a primeira implementação de cada tarefa, identificada como G01.

Na execução inicial, o PIT gerou 73 mutantes nas três implementações selecionadas. Destes, 62 foram detectados pelos testes, enquanto onze permaneceram sobreviventes. A suíte da T01 identificou todas as alterações produzidas pela ferramenta. Já nas tarefas T02 e T03, algumas modificações não provocaram falhas nos testes.

A inspeção dos relatórios iniciais motivou a elaboração de testes complementares para as tarefas T02 e T03. Na primeira, foram acrescentados três casos direcionados a condições de fronteira; na segunda, dois testes voltados aos limites permitidos dos intervalos de reserva. Após uma nova execução do PIT, a análise dos sobreviventes da T02 levou à inclusão de um quarto teste, destinado a verificar o tratamento de uma transferência com identificador de origem nulo. Essa última verificação deu origem a uma terceira execução da auditoria da T02.

A Tabela 4 apresenta a evolução da detecção de mutantes, distinguindo a avaliação inicial, a complementação por testes de fronteira e o resultado após a investigação adicional.

**Tabela 4.** Evolução da detecção de mutantes nas auditorias.

| Tarefa | Mutantes gerados | Detecção inicial | Após 1ª complementação | Detecção final | Sobreviventes fi |
|---|---|---|---|---|---|
| T01-G01 | 9 | 9 | 9 | 9 | 0 |
| T02-G01 | 32 | 26 | 29 | 30 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| T03-G01 | 32 | 27 | 27 | 27 | 5 |
| Total | 73 | 62 | 65 | 66 | 7 |

⚠ Esta tabela possui muitas colunas e foi cortada para impressão. Para visualizá-la completa, acesse o artigo original em: https://revistatopicos.com.br/artigos/avaliacao-da-correcao-funcional-e-da-capacidade-de-deteccao-de-testes-em-codigo-gerado-por-inteligencia-artificial-generativa?noblockage

**Fonte:** Autoria própria (2026).

A T01 não recebeu testes complementares, mantendo nove mutantes detectados em todas as etapas. Na T02, a detecção final inclui o quarto teste, direcionado à validação de origem nula. A pontuação agregada final corresponde a 66 mutantes detectados entre 73 gerados (90,41%).

Na T01, todos os nove mutantes foram detectados pela suíte original. As alterações envolveram principalmente condições de validação e substituição de valores retornados. O resultado indica que os testes foram capazes de identificar todas as modificações produzidas pelo PIT nessa implementação. Isso não significa, entretanto, que a suíte seja capaz de detectar qualquer defeito possível, pois a análise depende dos operadores de mutação utilizados e das características do código examinado.

Na T02, a inclusão dos testes complementares produziu uma melhoria progressiva na capacidade de detecção da suíte. Na primeira execução, 26 dos 32 mutantes foram detectados, correspondendo a uma pontuação de 81,25%. Após a inclusão de três casos voltados a condições de fronteira, esse número passou para 29, elevando a pontuação para 90,63%. As novas verificações

contemplaram uma entrada de estoque que alcançava o limite máximo do tipo inteiro, uma transferência que consumia todo o saldo disponível na origem e outra que elevava o saldo do destino ao valor máximo permitido.

A inspeção dos mutantes que permaneceram sobreviventes revelou outra situação relevante. No método de transferência, a remoção da validação do identificador de origem poderia permitir que um valor nulo chegasse à expressão origem.equals(destino), provocando uma NullPointerException em lugar da IllegalArgumentException exigida pelo contrato funcional. Para verificar essa possibilidade, foi acrescentado um quarto teste, destinado a confirmar o tipo de exceção lançado e a preservação do saldo após a tentativa de transferência inválida.

Na terceira execução do PIT, a mutação correspondente à remoção dessa validação passou de sobrevivente para detectada. Com isso, a quantidade de mutantes identificados na T02 aumentou para 30 dos 32 gerados, alcançando uma pontuação de 93,75%. O resultado confirma que a suíte anterior deixava de distinguir uma alteração capaz de violar um requisito funcional, mesmo com a aprovação dos testes existentes. A implementação original, entretanto, já realizava corretamente essa validação, de modo que a falha identificada pertence ao comportamento do mutante artificial, e não ao código produzido pelo GPT-6.

A evolução observada na T02 mostra a utilidade de examinar individualmente os mutantes sobreviventes. Os testes de fronteira ampliaram a verificação de operações válidas nos limites permitidos, enquanto o quarto caso acrescentou uma situação de entrada inválida cujo tratamento exigia uma exceção específica. Essa

estratégia se aproxima das propostas de Moradi Dakhel et al. (2024) e Lu et al. (2026), que utilizam informações dos mutantes sobreviventes para orientar o aprimoramento das suítes. Neste estudo, a complementação ocorreu manualmente e em escala reduzida, mas permitiu identificar uma limitação concreta da avaliação inicial.

Na T03, o comportamento foi diferente. Os dois testes acrescentados verificaram reservas iniciadas no minuto zero e encerradas no minuto 1440, ambos permitidos pelos requisitos. Mesmo após essa inclusão, a quantidade de mutantes detectados permaneceu em 27 dos 32 gerados, mantendo a pontuação de 84,38%.

A diferença entre os resultados da T02 e da T03 mostra que o aumento da quantidade de testes não produz, necessariamente, uma melhoria na pontuação de mutação. Para que isso ocorra, os novos casos precisam exercitar situações capazes de distinguir o comportamento original daquele introduzido pelos mutantes. A quantidade de testes, portanto, não deve ser analisada separadamente de sua finalidade e de sua capacidade efetiva de detecção.

### 4.4. Análise dos Mutantes Sobreviventes

Ao final das auditorias complementares, sete mutantes permaneceram sobreviventes, sendo dois na T02 e cinco na T03. A inspeção dos relatórios mostrou que essas ocorrências estavam relacionadas à remoção de chamadas de validação e a alterações em condições de comparação. Embora os mutantes tenham permanecido sem detecção, isso não significa que todos representem modificações capazes de produzir falhas funcionais.

Na T02, os dois sobreviventes restantes estão associados à remoção de validações de identificadores nos métodos obterSaldo() e transferir(). A análise do código indica que essas verificações podem ser realizadas indiretamente por outras operações do programa. No primeiro caso, a ausência da validação em obterSaldo() ainda permite que identificadores inexistentes sejam rejeitados pela consulta ao mapa. No segundo, a retirada da validação do destino em transferir() não elimina a verificação posteriormente realizada por obterSaldo(destino). Considerando os requisitos estabelecidos, essas alterações apresentam indícios de equivalência funcional, embora não tenham sido formalmente classificadas como mutantes equivalentes.

A situação foi diferente na mutação que removia a validação do identificador de origem da transferência. Inicialmente, essa alteração permanecia sobrevivente. A inspeção do código mostrou que uma origem nula poderia alcançar a chamada origem.equals(destino) e provocar uma exceção incompatível com o contrato funcional. A inclusão de um teste específico confirmou a capacidade de detectar essa diferença: na execução seguinte, o PIT classificou o mutante como KILLED. O resultado evidencia uma limitação da suíte anterior e demonstra como a análise direcionada dos sobreviventes pode contribuir para a elaboração de verificações mais adequadas.

Na T03, cinco mutantes continuaram sobreviventes após a inclusão dos testes complementares. Três deles estavam relacionados à remoção de chamadas de validação de identificadores ou horários, algumas das quais também eram realizadas em outros pontos do fluxo. Os dois restantes envolviam alterações em condições de fronteira. Como essas modificações não foram analisadas individualmente por meio de testes específicos, não foi possível

determinar se todas preservam o comportamento funcional previsto.

A possibilidade de equivalência funcional corresponde a uma das limitações dos testes de mutação discutidas por Schuler e Zeller (2013). Uma alteração pode sobreviver porque os testes são insuficientes, mas também porque não produz diferenças observáveis no funcionamento do programa. Por isso, a interpretação da pontuação de mutação exige atenção às características de cada alteração, principalmente quando o relatório apresenta validações redundantes ou condições que podem ser logicamente equivalentes.

Cabe ressaltar que os mutantes foram introduzidos artificialmente pelo PIT e não pertenciam às implementações originalmente geradas pelo GPT-6. Assim, os sete sobreviventes registrados ao final das auditorias representam alterações que não foram detectadas pelas suítes utilizadas, sem corresponder necessariamente a defeitos funcionais presentes nos códigos originais.

**4.5. Discussão dos Resultados e Limitações**

Os resultados do experimento mostram a importância de distinguir a conformidade funcional de um programa da capacidade dos testes utilizados para avaliá-lo. Todas as quinze implementações geradas pelo GPT-6 passaram integralmente pelas suítes funcionais. Ainda assim, a auditoria com PIT identificou alterações artificiais que os testes não conseguiram detectar.

A diferença entre a aprovação funcional e a capacidade de detecção dos testes tornou-se mais evidente na T02. A inclusão de três casos de fronteira permitiu identificar três mutantes adicionais, enquanto

uma verificação específica para origem nula detectou uma alteração que violava o tratamento de exceções previsto nos requisitos. Na T03, por outro lado, os dois novos testes não modificaram a quantidade de mutantes detectados. Essa comparação mostra que a melhoria de uma suíte depende da capacidade dos casos acrescentados de distinguir comportamentos corretos e incorretos, e não apenas do aumento na quantidade de testes executados.

Os achados se aproximam das discussões de Inozemtseva e Holmes (2014), que apontam limitações no uso de métricas de cobertura como indicadores isolados da eficácia dos testes. Também dialogam com Liu et al. (2023), cujo estudo demonstra que programas aprovados em determinados conjuntos de testes podem apresentar falhas quando submetidos a avaliações mais abrangentes. Embora o experimento aqui apresentado seja menor e utilize procedimentos diferentes, seus resultados reforçam a necessidade de interpretar a aprovação funcional à luz das características das suítes empregadas.

Algumas limitações, contudo, restringem o alcance das conclusões. A amostra foi composta por três tarefas de programação e cinco implementações para cada uma, todas produzidas por um único modelo de linguagem. Não houve comparação com outros modelos nem com códigos desenvolvidos manualmente. As tarefas também apresentavam características próprias, impedindo atribuir eventuais diferenças observadas exclusivamente ao seu nível de complexidade.

A auditoria de mutação ficou restrita à primeira geração de cada tarefa. Portanto, as pontuações obtidas não podem ser estendidas automaticamente às outras doze implementações. Além disso, os testes complementares foram definidos após a análise dos relatórios

iniciais, o que caracteriza essa etapa como exploratória. As melhorias observadas na T02 demonstram o efeito daqueles casos específicos sobre os mutantes produzidos, mas não permitem estimar o desempenho da suíte diante de qualquer alteração possível.

A comparação estrutural também apresentou limitações. Embora tenham sido identificadas diferenças na organização dos códigos, não foram aplicadas métricas padronizadas que permitissem comparar quantitativamente sua complexidade ou manutenibilidade. Por essa razão, as observações não sustentam conclusões sobre superioridade técnica entre as soluções. Da mesma forma, não foi possível avaliar o esforço necessário para corrigir defeitos, uma vez que nenhuma das implementações originais apresentou falhas nos casos de teste executados e não houve procedimentos de reparo.

Mesmo com essas restrições, a pesquisa oferece uma contribuição para a discussão sobre a avaliação de código gerado por inteligência artificial. Os resultados mostram que a aprovação em testes funcionais precisa ser interpretada considerando o alcance dos cenários avaliados e a capacidade de detecção das suítes. A análise de mutação acrescenta uma perspectiva complementar, permitindo identificar alterações que merecem investigação e orientar a elaboração de testes mais específicos. Assim, mais do que atribuir uma medida definitiva de qualidade às implementações, o estudo evidencia a importância de avaliar também os instrumentos utilizados para verificar seu comportamento.

## 5. CONSIDERAÇÕES FINAIS

Os resultados deste estudo mostram que a aprovação de um programa em testes automatizados não é suficiente para assegurar sua correção em todas as situações. Embora as implementações geradas por inteligência artificial atendam aos requisitos contemplados pelas suítes utilizadas, a confiabilidade dessa avaliação depende também da capacidade dos testes de identificar comportamentos incorretos. Essa distinção é importante, especialmente diante da crescente utilização de modelos de linguagem no desenvolvimento de software.

A análise de mutação permite aprofundar essa avaliação ao identificar alterações no código que passam despercebidas pelos testes existentes. Na tarefa de controle de estoque, os casos adicionais voltados a condições de fronteira ampliam a quantidade de mutantes detectados. A investigação de um dos sobreviventes também revela uma lacuna relacionada ao tratamento de identificadores nulos, confirmada por um teste específico. No sistema de reservas, por outro lado, os testes complementares não modificam a pontuação obtida. Essa diferença mostra que a melhoria de uma suíte depende dos comportamentos efetivamente examinados e da capacidade dos casos de teste de distinguir desvios em relação aos requisitos.

A comparação entre as implementações também revela que soluções produzidas a partir das mesmas instruções podem apresentar diferenças na organização do código, mesmo quando alcançam resultados funcionais equivalentes. Essas variações reforçam a importância de observar não apenas o comportamento dos programas, mas também as decisões adotadas em sua construção. Dessa maneira, os objetivos da pesquisa são alcançados dentro das condições estabelecidas, tanto na avaliação funcional

quanto na análise das diferenças estruturais e da capacidade de detecção das suítes.

A contribuição do trabalho está em demonstrar a importância de utilizar testes funcionais e testes de mutação de forma complementar na avaliação de código gerado por inteligência artificial. Enquanto os primeiros verificam o atendimento aos requisitos nos cenários definidos, os segundos ajudam a identificar limitações dos testes empregados. A pesquisa também chama a atenção para a necessidade de examinar os mutantes sobreviventes antes de associá-los a possíveis falhas, já que algumas alterações podem não produzir diferenças no comportamento do programa. Essas observações são relevantes para profissionais de desenvolvimento e qualidade de software que utilizam ferramentas de IA e precisam avaliar a confiabilidade das soluções produzidas.

O alcance das conclusões, contudo, está restrito às tarefas, ao modelo e aos procedimentos utilizados. A quantidade limitada de problemas analisados e a aplicação dos testes de mutação a apenas uma implementação de cada tarefa não permitem generalizar os resultados para outros contextos. Pesquisas futuras podem ampliar o conjunto de tarefas, comparar diferentes modelos de linguagem e aplicar a análise de mutação a todas as implementações geradas. Também é pertinente investigar outras estratégias de elaboração de testes e sua contribuição para a identificação de falhas. O aprofundamento dessas questões pode favorecer práticas de avaliação mais consistentes e um uso mais criterioso da inteligência artificial no desenvolvimento de software.

## REFERÊNCIAS BIBLIOGRÁFICAS

**BORG, M., Hewett, D., Hagatulah, N. et al.** Echoes of AI: Investigating the downstream effects of AI assistants on software maintainability. Empir Software Eng 31, 161 (2026). https://doi.org/10.1007/s10664-026-10889-1.

**CABRAL, Pedro Fernando Marinho et al. (2025).** Evaluating LLM-Generated Unit Tests with Mutation Testing: ChatGPT vs DeepSeek. Anais do XXIV Simpósio Brasileiro de Qualidade de Software (SBQS), p. 120–130. https://doi.org/10.5753/sbqs.2025.13792.

**COLES, Henry et al. (2016).** PIT: A Practical Mutation Testing Tool for Java (Demo). Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA), p. 449–452. https://doi.org/10.1145/2931037.2948707.

**IDRISOV, Baskhad; SCHLIPPE, Tim (2024).** Program Code Generation with Generative AIs. Algorithms, v. 17, n. 2, artigo 62. https://doi.org/10.3390/a17020062.

**INOZEMTSEVA, Laura; HOLMES, Reid (2014).** Coverage Is Not Strongly Correlated with Test Suite Effectiveness. Proceedings of the 36th International Conference on Software Engineering (ICSE). https://doi.org/10.1145/2568225.2568271.

JUNIT. **JUnit 5 User Guide.** Disponível em: https://docs.junit.org/5.12.0/user-guide/. Acesso em: 02 out. 2026.

**LIU, Jiawei et al. (2023).** Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. Advances in Neural Information Processing Systems, v. 36 (NeurIPS). https://doi.org/10.52202/075280-0943.

**LU, Zeyu et al. (2026).** Beyond Coverage: Automatic Test Suite Augmentation for Enhanced Effectiveness using Large Language Models. Proceedings of the ACM on Programming Languages, v. 10, OOPSLA1, artigo 143, p. 1403–1431. https://doi.org/10.1145/3798251.

**MORADI DAKHEL, Arghavan et al. (2024).** Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. Information and Software Technology, v. 171, artigo 107468. https://doi.org/10.1016/j.infsof.2024.107468.

OPENAI. **GPT-6 e outros modelos no ChatGPT.** Disponível em: https://help.openai.com/pt-br/articles/20001354-gpt-6-and-other-models-in-chatgpt. Acesso em: 02 out. 2026.

**ORACLE. JDK 21 Documentation.** Disponível em: https://docs.oracle.com/en/java/javase/21/. Acesso em: 02 out. 2026.

PITEST. **Quickstart for Maven Users.** Disponível em: https://pitest.org/quickstart/maven/. Acesso em: 02 out. 2026.

**Schuler, D. and Zeller, A. (2013),** Covering and Uncovering Equivalent Mutants. Softw. Test. Verif. Reliab., 23: 353-374. https://doi.org/10.1002/stvr.1473.

**TOSI, Davide (2024).** Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation. Future Internet, v. 16, n. 6, artigo 188. https://doi.org/10.3390/fi16060188.

---

[1] Bacharela em Engenharia de Software pelo Centro Universitário Internacional (UNINTER). Pós-graduada em Inteligência Artificial e

Ciência de Dados pelo Centro Universitário UniFatecie. E-mail:
acesse o artigo original para visualizar o e-mail