Análise da Qualidade e Eficácia do Código Gerado por LLMs: Um Estudo com Problemas da Plataforma LeetCode

Bernardo Aquino Capello Coelho[1]

[1] Departamento de Engenharia de Software – Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Belo Horizonte – Brasil

baccoelho@sga.pucminas.br

Abstract. *Large Language Models (LLMs) are revolutionizing software engineering, particularly through the automation of code generation tasks. Despite increasing developers' productivity, significant doubts remain regarding the quality and accuracy of the generated code. This study investigates the effectiveness and quality of code generated by four LLMs (GPT-4, Gemini, Claude 3 Haiku, and Llama 3.1) across 616 Python programming problems from LeetCode using, resulting in a total of 2,464 responses. The tools demonstrated accuracy rates exceeding 50%. Additionally, it was observed that rewriting the problem statements also influenced the accuracy rate. In total, approximately 2.8K maintainability issues and 13.9K technical debt points were identified, with a low average occurrence per response.*

Resumo. *Large Language Models (LLMs) estão revolucionando a engenharia de software, particularmente por meio da automação de tarefas de geração de código. Apesar de aumentarem a produtividade dos desenvolvedores, ainda existem dúvidas significativas sobre a qualidade e assertividade do código gerado. Neste trabalho, investiga-se a eficácia e qualidade do código gerado por quatro LLMs (GPT-4, Gemini, Claude 3 Haiku e Llama 3.1) em 616 problemas de programação do LeetCode escritos em Python, o que resultou em um total de 2.464 respostas. As ferramentas apresentaram taxas de assertividade superiores a 50%. Além disso, observou-se que a reescrita dos enunciados dos problemas também influenciou na taxa de assertividade. Ao todo, foram identificadas aproximadamente 2.8K problemas de manutenibilidade e 13.9K pontos de dívida técnica, com baixa ocorrência média por resposta.*

Bacharelado em Engenharia de *Software* - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): Danilo Boechat Seufitelli - daniloboechat@pucminas.br
Orientador de conteúdo (TCC I): João Pedro Oliveira Batisteli - joao.batisteli@sga.pucminas.br
Orientador acadêmico (TCC I): Cleiton Silva Tavares - cleitontavares@pucminas.br
Orientadora do TCC II: Aline Norberta de Brito - alinebrito@pucminas.br

Belo Horizonte, 24 de Novembro de 2024.

1. Introdução

Apesar dos avanços significativos ao longo das décadas, ainda há um longo caminho a se percorrer para alcançar a geração de código completamente automática e prática, especialmente quando se trata de gerar código com qualidade [Moradi Dakhel et al. 2023]. Nos últimos anos, contudo, ferramentas de geração de código baseadas em Inteligência Artificial (IA) e o surgimento de *Large Language Models* (LLM) ganharam um considerável destaque [Wang and Chen 2023]. Um exemplo notável é o `GitHub Copilot`,[1] impulsionado pelo modelo Codex[2] da OpenAI[3], que demonstra a habilidade de gerar código preciso para diversas tarefas de programação [Moradi Dakhel et al. 2023, Vaithilingam et al. 2022].

Existem estudos que focam na correção e avaliação dos códigos gerados pelo `GitHub Copilot` em diferentes tipos de contexto. Por exemplo, Rubio et al. (2023) concentraram-se na avaliação da eficácia das soluções geradas por LLMs em atividades universitárias envolvendo problemas de programação de disciplinas de um curso superior de ciência da computação. Já Finnie-Ansley et al. (2022) realizaram uma análise comparativa sobre a assertividade entre alunos e o *Davinci* (versão *beta* do modelo *Codex*) em questões de um curso de programação. Adicionalmente, alguns estudos exploram a interação humana e a eficiência com que os desenvolvedores utilizam LLMs. Porém, estes estudos ainda se concentram exclusivamente na ferramenta `Copilot` e não envolvem uma análise do código [Moradi Dakhel et al. 2023]. Além disso, não houve investigação sobre as soluções com defeitos e possíveis sugestões de melhorias utilizando a linguagem natural e diferentes entradas. Portanto, é possível encontrar problemas de qualidade de código gerado por estas ferramentas, podendo afetar negativamente a compreensão do código, introduzir *bugs* ou criar vulnerabilidades de segurança [Liu et al. 2024]. **Nesse contexto, o problema que este trabalho aborda é uma carência de estudos que comparem e analisem códigos-fonte gerados por LLMs no que tange aspectos de eficácia e qualidade.**

Adicionalmente, devido ao novo ambiente criado pelas IAs, surgem preocupações sobre plágio e dependência excessiva. Estudantes podem sentir-se tentados a utilizar ferramentas de geração de código para concluir tarefas de programação sem compreender os conceitos subjacentes, o que pode comprometer o desenvolvimento de habilidades essenciais, como o pensamento crítico e a capacidade de resolução de problemas [Reeves et al. 2023]. Usuários sem qualquer experiência em programação podem utilizar LLMs para gerar trechos de código a partir de requisitos em linguagem natural [Becker et al. 2023]. Além disso, a rápida e ampla disponibilidade destas ferramentas tem levado à especulações sobre o futuro da educação em computação e programação no geral [Welsh 2022]. Embora essas ferramentas possam auxiliar e agilizar o processo de geração de código e desenvolvimento, persistem dúvidas quanto à sua confiabilidade e assertividade, especialmente quanto à qualidade do código produzido e seu possível impacto na segurança e na inteligibilidade humana das respostas, uma vez que elas têm sido cada vez mais utilizadas em diversos ambientes. Trabalhos como o de Liu et al. (2024) investigaram mais a fundo o funcionamento e sintetização mais aprofundado de

[1] `https://github.com/features/copilot`
[2] `https://openai.com/blog/openai-codex/`
[3] `https://openai.com/about`

diversos LLM, mas eles não abordaram aspectos relacionados à qualidade do código e compreensão humana das respostas.

Desta forma, o objetivo geral deste trabalho é **realizar uma análise comparativa entre quatro LLMs difundidos no mercado: `GPT-4`[4], `Gemini`[5], `Claude 3 Haiku`[6] e `Llama 3.1`[7]**. Os objetivos específicos são: (I) analisar a qualidade do código gerado pelos LLMs; (II) avaliar a assertividade das respostas das LLMs; e (III) investigar se variações na formulação das perguntas feitas às LLMs impactam o código gerado.

Os resultados desta pesquisa indicaram que os modelos `Claude 3 Haiku` e `GPT-4` alcançaram alta assertividade (superior a 80%) em problemas de programação, demonstrando bom desempenho tanto de modelos especializados quanto robustos. Apesar da menor taxa de assertividade (51%), o modelo `Llama 3` destacou-se pelos melhores indicadores de qualidade estrutural. Além disso, variações nas perguntas influenciaram significativamente as soluções, resultando em uma oscilação de cerca de 52% na assertividade em problemas anteriormente incorretos ou com erros de execução. Com base nesses resultados, foram geradas implicações relevantes para a comunidade de *software*.

As próximas seções do artigo estão divididas da seguinte forma: A Seção 2 apresenta a fundamentação teórica e a Seção 3 abrange os trabalhos relacionados. A Seção 4 descreve a metodologia proposta para a realização do trabalho. As seções 5 e 6 apresentam e discutem os resultados obtidos na pesquisa, respectivamente. A Seção 7 contempla as ameaças a validade e suas mitigações. Por fim, a Seção 8 apresenta a conclusão e trabalhos futuros.

2. Fundamentação Teórica

Nesta seção, são apresentados os conceitos fundamentais para compreender o trabalho. Especificamente, na Seção 2.1 é discutida a definição de *Large Language Models* (LLM). Por fim, na Seção 2.2 são abordado os conceitos relacionados a Qualidade de Software.

2.1. Large Language Models (LLM)

Large Language Models (LLM) são ferramentas de inteligência artificial baseadas em redes neurais multidimensionais, treinadas com vasta quantidades de dados, geralmente vindos da internet [Alberts et al. 2023]. LLMs assim como outros modelos de linguagem, aprendem padrões e associações entre palavras nos dados de treinamento, para prever a próxima palavra em uma sequência, permitindo gerar saídas com base na probabilidade [Bhayana 2024]. Embora modelos de linguagem probabilísticos existam há décadas, os LLMs revolucionaram o processamento de linguagem natural (PLN) nos últimos anos [Kasneci et al. 2023]. Isso é causado pela adoção da arquitetura de transformadores [Devlin et al. 2018, Tay et al. 2022] e o uso do mecanismo de atenção [Vaswani et al. 2017], os quais têm melhorado significativamente a capacidade dos modelos de linguagem para lidar com dependências de grande alcance em textos feitos em linguagem natural. Especificamente, a arquitetura de transformadores por Vaswani et al. (2017) emprega o mecanismo de autoatenção para avaliar a importância de diferentes

[4]`https://chatgpt.com/`
[5]`https://gemini.google.com/`
[6]`https://www.anthropic.com/claude`
[7]`https://www.llama.com/`

partes da entrada ao gerar previsões. Isso permite que o modelo compreenda melhor as relações entre as palavras em uma sentença, independentemente de sua posição.

De forma geral, LLMs são treinados para prever a provável palavra a seguir, dada uma sugestão textual [Gmeiner and Yildirim 2023]. No entanto, essa descrição de alto nível não captura a ampla gama de coisas que os LLM podem fazer. Novamente como exemplo, o `ChatGPT` que foi treinado utilizando um imenso conjunto de dados, foi a primeira inteligência artificial baseada em LLM que demonstrou amplamente a flexibilidade desses modelos, apresentando bom desempenho em uma ampla gama de tarefas de linguagem natural, como escrever *e-mails*, artigos, fazer resumo de documentos e até a escrita de código para Software [Taecharungroj 2023].

2.2. Qualidade de Software

Qualidade de *software* é um conceito crucial que abrange a capacidade de um produto atender às necessidades explícitas e implícitas dos usuários [Nistala et al. 2019]. A compreensão dessa qualidade é multidimensional e interdependente, exigindo uma avaliação cuidadosa em várias camadas [Gillies 2011]. Por isso, a normatização dos aspectos de desenvolvimento de *software*, como proposto pela ISO/IEC 9126, distingue a qualidade em diferentes tipos, incluindo interna, externa e em uso. Essas normas estabelecem métricas e características que devem ser especificadas, medidas e avaliadas para garantir um produto final de alta qualidade [ISO/IEC 25010:2011 2017].

A qualidade de *software* é um elemento essencial para garantir que o produto final atenda às expectativas dos usuários e seja resiliente a falhas. Métricas específicas de qualidade estrutural, são definidas para garantir a confiabilidade, segurança e manutenibilidade do código-fonte [Curtis et al. 2022].

3. Trabalhos Relacionados

Nesta seção, são discutidos os trabalhos de outros autores que ajudam a explicar o contexto em que este artigo aborda, em conjunto com estudos que analisam o mesmo processo ou utilizam de metodologias semelhantes a que propomos. De maneira específica, os trabalhos relacionados discutidos envolvem a análise da qualidade do código e assertividade das respostas fornecidas pelas LLMs.

Nguyen e Nadi (2022) conduziram uma pesquisa empírica para avaliar a corretude e compreensibilidade das sugestões de código providas pelo `GitHub Copilot`. Para isso, foi utilizada uma abordagem que envolveu quatro linguagens de programação a partir de 33 questões do LeetCode. Além disso, a compreensibilidade das soluções foi analisada usando métricas de complexidade ciclomática e complexidade cognitiva do SonarQube. Os resultados revelaram que as sugestões do `Copilot` em Java apresentaram a maior taxa de corretude (57%), enquanto JavaScript obteve o menor desempenho (27%). Ademais, as sugestões do `Copilot` demonstraram baixa complexidade, sem diferenças significativas entre as linguagens de programação. O trabalho tem uma grande relação com o estudo proposto neste artigo, pois, avalia a qualidade do código gerado por IAs utilizando questões de programação e utiliza de métricas semelhantes para avaliar a compreensibilidade humana.

Rubio et al. (2023) tem como foco a exploração experimental, análise e avaliação das sugestões feitas pela ferramenta `GitHub Copilot` em tópicos de programação re-

lacionados ao curso de Ciências da Computação na Universidade de Bio-Bio. Os artefatos em questão, envolvem sentenças escritas em linguagem natural, obtidas dos problemas de programação das disciplinas de: Introdução à Programação, Programação Orientada a Objetos, Estrutura de Dados, Análise e Desenho de Algoritmos, Fundamentos de Ciências da Computação, Administração e Programação de Bases de Dados e Inteligência Artificial. O estudo observou que, em certos casos, as sugestões do `Copilot` careciam de funcionalidade adequada, sugerindo comentários ou código não pertinentes à tarefa. Adicionalmente, ao utilizar o SonarQube para avaliar a qualidade do código, foram identificados *bugs* e problemas de manutenibilidade nas respostas analisadas. Esse estudo é relevante para a construção deste trabalho, uma vez que utiliza métodos semelhantes para realizar a correção de sugestões de código gerado pela LLM, além de fornecer uma possibilidade de ferramenta a ser utilizado neste trabalho.

De outra perspectiva, Denny et al. (2023) propõem uma análise do desempenho atual do `GitHub Copilot` na resolução de problemas de programação relacionados ao tópico de introdução à ciência da computação em um repositório público. O estudo investigou sobre a interação humana com ferramentas de geração de código, por meio da Engenharia de *Prompts*. As saídas dos modelos de LLMs são muito sensíveis às suas entradas, dessa forma foi explorado o desempenho atual do `GitHub Copilot` e a eficácia da Engenharia de *Prompts* por meio de uma metodologia que envolveu a cópia das descrições dos problemas do CodeCheck para um editor Visual Studio Code com a extensão `Copilot` ativada. Como resultado, identificaram falhas comuns do `Copilot`, categorizando 15 problemas como "Conceituais", 4 como "Prompts Fracos", 11 como "Prompts Verbosos"e 4 como "Ambíguos". Assim, esse estudo é relevante no contexto deste trabalho, pois parte da metodologia utilizada para correção e categorização dos *prompts*, foram usadas neste trabalho de maneira automatizada.

Mastropaolo et al. (2023) realizam uma análise sobre como mudanças na descrição do problema em linguagem natural afetam as sugestões de código geradas pelo `GitHub Copilot`. Investigaram se diferentes formas de descrever a mesma questão levam a recomendações de código distintas, o que poderia questionar a consistência e confiabilidade do `Copilot`. A pesquisa envolveu a geração automática de 892 métodos Java a partir de suas descrições originais no Javadoc, e através das ferramentas PEGASUS e *Translation Pivoting* (TP). Os resultados da análise mostram que, em cerca de 46% dos casos, alterar a descrição resulta em sugestões de código diferentes, com algumas diferenças até mesmo afetando a correção do código gerado em aproximadamente 28%. Esse estudo é relevante para a construção deste trabalho, uma vez que propõe metodologias para gerar descrições semanticamente equivalentes por meio de ferramentas automatizadas e que serão usadas no presente estudo para aperfeiçoar as repostas geradas pelas LLMs.

Por fim, Su et al. (2023) propõem um estudo sobre um método de avaliação relacionado a capacidade de geração de código de LLMs. O método proposto neste estudo avalia a capacidade de geração de código em seis dimensões: validade do código, correção do código, complexidade do código, confiabilidade do código, segurança do código e legibilidade do código. Para alcançar esse objetivo, é proposto um conjunto de dados para testar as capacidades de código, que compreende 45 perguntas de codificação. Neste estudo foram abordados as ferramentas `ChatGPT, Claude, Spark` e `Bing`

AI. Através da avaliação experimental e análise de dados, os pesquisadores descobrem que os LLMs baseados em busca, como o Bing AI, apresentam capacidades de geração de código mais robustas do que os modelos pré-treinados, como ChatGPT, Claude e Spark. Além disso, observaram que os LLMs atuais possuem fortes habilidades de compreensão de linguagem natural, e que os erros nas sugestões de código são mais propensos a serem causados por problemas no código em si do que por problemas de compreensão. Assim, tal estudo comparativo é de grande relevância na construção deste trabalho, uma vez que propõem metodologias para avaliação de desempenho utilizadas neste estudo, em um método similar baseado em testes e ferramentas de geração de métricas com o SonarQube.

4. Materiais e Métodos

O estudo descrito neste trabalho é de natureza quantitativa com objetivo exploratório [Wohlin et al. 2012], pois envolve a análise de aspectos de assertividade e qualidade do código gerado por quatro modelos de LLMs disponíveis no mercado. A Figura 1 mostra uma visão geral da metodologia empregada neste estudo, que envolve seis grandes etapas: coleta de perguntas provenientes da plataforma LeetCode, geração de respostas por meio de LLMs, análise da assertividade das respostas, avaliação da assertividade após parafraseamento, métricas de qualidade, e geração de visualizações. Nas subseções a seguir, cada uma delas é detalhada.

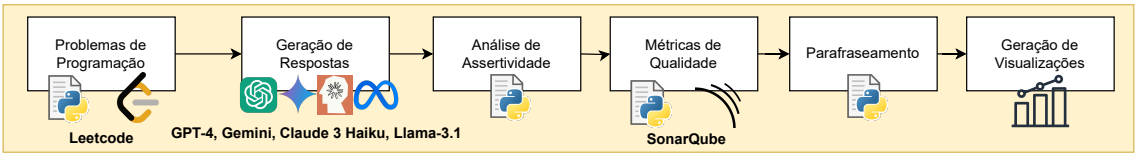


Figura 1. Visão geral da metodologia

4.1. Seleção de Problemas de Programação na Plataforma LeetCode

O conjunto de dados inicial deste trabalho consiste em problemas de programação provenientes do LeetCode,[8] uma plataforma que têm sido amplamente utilizada pela comunidade de *software* para treinamento, capacitação e por empresas em processos seletivos [Cui et al. 2024, Leetcode 2024]. Utilizando a *Application Interface Programming* (API) provida pela mesma, criou-se um conjunto de *scripts* para selecionar os problemas que possuem acesso gratuito e que estão na categoria de algoritmos. Os metadados incluem, por exemplo, informações sobre número de resoluções, nível de dificuldade, número de *likes* e *dislikes*. Em seguida, desta amostra, foram selecionados os top-25%, ordenados pelo número de submissões, por se tratar de uma métrica que sugere problemas de interesse da comunidade.

A Figura 2 mostra um exemplo de problema de programação proposto para a plataforma LeetCode denominado *Two Sum*.[9] Como pode-se observar, a tarefa consiste em propor um trecho de código para solucionar um problema de busca de valores em um *array*. O objetivo é que o desenvolvedor retorne as duas posições no *array* que correspondem a soma do número alvo (*target*). Este problema possui aproximadamente 28M submissões, com uma taxa de assertividade de aproximadamente 54% (Novembro, 2024).

[8] https://leetcode.com
[9] https://leetcode.com/problems/two-sum

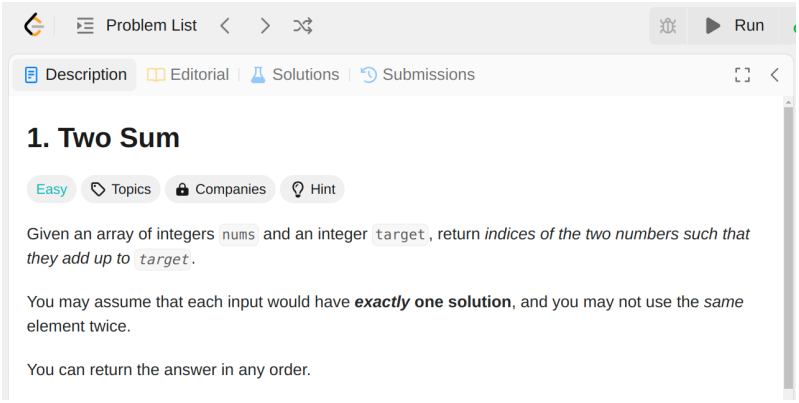


Figura 2. Exemplo de problema de programação da plataforma LeetCode (Two Sum problem)

4.2. Geração de Respostas para os Problemas de Programação

Este trabalho considera respostas geradas por quatro LLMs populares `GPT-4`, `Gemini`, `Claude 3 Haiku` e `Llama 3.1`, desenvolvidas e mantidas por grandes companhias de *software* [OpenAI 2024, Google 2024, Claude 2024, Meta 2024]. Para gerar as respostas aos problemas selecionados, adotou-se a biblioteca G4F.[10] Esta biblioteca é capaz de se comunicar com as LLMs foco deste estudo, permitindo a automação da geração das respostas. As perguntas foram estruturadas em arquivos JSON, e em seguida, submetidas para resolução por cada LLM.

O texto a seguir mostra um exemplo da mensagem enviada para os modelos através da biblioteca G4F, que inclui o enunciado do problema, especificações detalhadas sobre o formato esperado pelo LeetCode e instruções para o uso da assinatura requerida pelo interpretador da plataforma. O código foi gerado em Python, devido à popularidade e simplicidade da linguagem [Lopes and Hora 2022, Buscemi 2023], além de estar entre as linguagens mais populares no ranking de LLMs.[11]

> *Generate only Python code for the following problem:*
>
> ***[code problem from LeetCode platform]***
>
> *Ensure that the function maintains the same name and the same number of parameters as in this example:*
>
> ***[class and method definition for Python code]***
>
> *Only use the "class Solution". Do not create another class and do not use any libraries. I want only the Python code without additional text or comments.*

A Figura 3 mostra um exemplo da assinatura requerida para o problema *Two Sum*, mencionado anteriormente. Como pode-se observar, o trecho de código esperado requer uma classe denominada *Solution* e um método $twoSum$ que recebe três parâmetros.

[10] `https://g4f.ai`
[11] `https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`

Figura 3. Exemplo de assinatura requerida para submissão de problema na plataforma LeetCode (Two Sum problem)

4.3. Análise da Assertividade e Qualidade das Respostas

Após a geração dos problemas via LLMs, as respostas foram submetidas na plataforma LeetCode para verificar a assertividade. Para tanto, elaborou-se um conjunto de *scripts*, que gerencia *tokens* de sessão e *Cross Site Request Forgery* (CSRF) para autenticação, garantindo a validade dos mesmos através de funções de rotação e validação.

Os *scripts* executam requisições GraphQL para obter detalhes dos problemas e configurações, utilizando essas informações para interpretar e submeter soluções de código geradas pelas LLMs. Em seguida, verifica-se o status das submissões para determinar a assertividade das soluções, confirmando se elas passam no interpretador e se atingem todos os casos de teste necessários para serem consideradas corretas. Por fim, os dados coletados nesta etapa são armazenados em um arquivo JSON.[12]

Para avaliação da assertividade considera-se o conceito de eficácia definido em trabalhos anteriores [Denny et al. 2023]. Dessa forma, as respostas são classificadas nas seguintes categorias baseadas em seu desempenho:

- **Correta:** resposta que passou pelo interpretador e por todos os casos de teste propostos pela plataforma LeetCode;
- **Incorreta:** resposta que passou pelo interpretador, mas não obteve sucesso em todos os casos de teste propostos pela plataforma LeetCode;
- **Errada:** resposta que apresentou erro de sintaxe e que não passou pelo interpretador.

Finalmente, para as respostas corretas, analisa-se também a qualidade dos trechos de código, através da ferramenta SonaQube. A escolha deve-se a popularidade da mesma no contexto de análise estática de código [Vassallo et al. 2018, Marcilio et al. 2019, Rocha et al. 2024]. Especificamente, consideram-se as seguintes métricas detectadas pela ferramenta [SonarSource 2024]:

- **Manutenibilidade:** problemas no código que dificultam sua leitura, compreensão e modificação, incluindo, por exemplo, "*code smells*";
- **Dívida Técnica:** medida do esforço necessário para corrigir problemas no código, representando o custo acumulado de manutenção futura.

[12]Também foi realizada uma inspeção manual dos resultados gerados por meio de *scripts*, com o objetivo de identificar falhas e possíveis melhorias. Por exemplo, uma amostra das respostas geradas foi submetida manualmente à plataforma LeetCode. Algumas das respostas produzidas pelas LLMs que falharam na submissão foram inspecionadas manualmente.

4.4. Parafraseamento dos Problemas de Programação

Como última etapa deste trabalho, verifica-se a assertividade das respostas em caso de parafraseamento (isto é, incorretas ou erradas). Em outras palavras, é verificado se a reescrita do enunciado do problema, mantendo o significado original, pode impactar na taxa de assertividade das LLMs. Nesta etapa são analisadas todas as respostas que não foram aprovadas pelo interpretador do LeetCode ou pelos casos de testes.

Utilizou-se o modelo *Pegasus* para o parafraseamento do enunciado dos problemas de programação [Mastropaolo et al. 2023].[13] Após o parafraseamento dos enunciados, o fluxo é executado novamente, para geração de novas respostas e avaliação do impacto na assertividade das soluções.

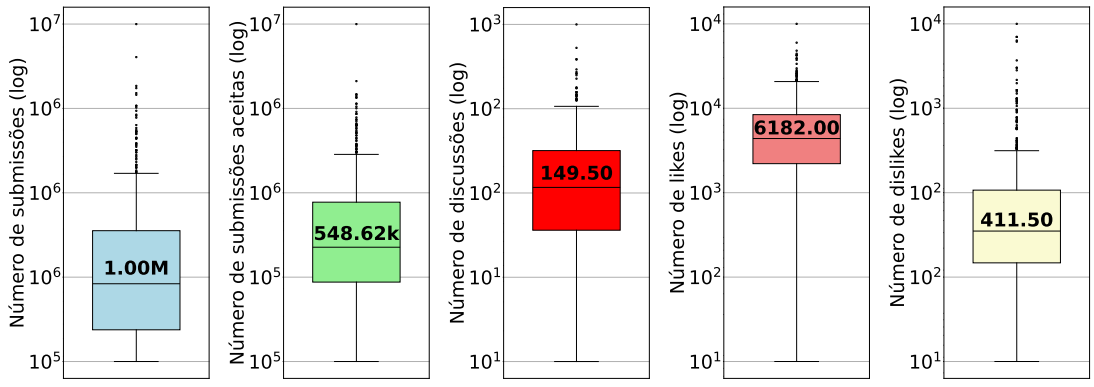
5. Resultados

Nesta seção, apresentam-se os resultados obtidos neste trabalho, bem como a caracterização do conjunto de dados utilizado.

5.1. Caracterização do Conjunto de Dados

Inicialmente, foram selecionados 2.894 problemas de programação da plataforma LeetCode. Em seguida, os problemas foram filtrados para incluir apenas os disponíveis gratuitamente. Após isso, eles foram ordenados pelo número de submissões, e o primeiro quartil foi selecionado para análise. Dessa forma, este trabalho inclui a análise da respostas de 616 problemas de algoritmos mais populares disponíveis na plataforma LeetCode. Para cada problema foram geradas quatro respostas produzidas por LLMs distintas, resultando na análise de 2.464 trechos de código.

A caracterização dos dados coletados é realizada por meio de gráficos *boxplot* na escala logarítmica, conforme apresentado na Figura 4. Especificamente, apresenta-se métricas de engajamento dos problemas, considerando a quantidade de submissões, aceitações, discussões, *likes* e *dislikes*. Como pode-se observar, os dados indicam uma alta variabilidade no engajamento dos problemas.



(a) Submissões (b) Aceitações (c) Discussões (d) *Likes* (e) *Dislikes*

Figura 4. Caracterização dos problemas LeetCode

O número de submissões por problema variam entre 373.977 e 26.747.970, sendo a mediana igual a 1.000.526 submissões, com 75% dos problemas apresentando até 2

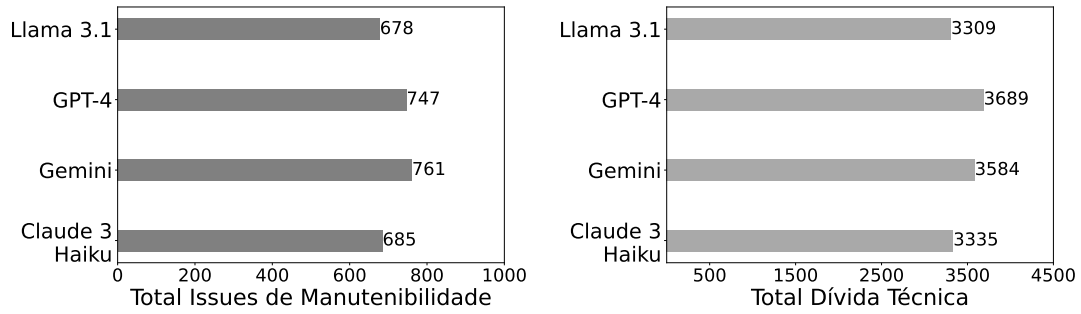
[13] https://huggingface.co/tuner007/pegasus_paraphrase

milhões de submissões (Figura 4(a)). A Figura 4(b) mostra que as aceitações variam entre 103.157 e 14.327.700, com uma mediana de 548.619,5 e 75% dos problemas com aproximadamente 1 milhão de aceitações. Na Figura 4(c) o gráfico de discussões indica que a maioria dos problemas possui até 250 discussões, com uma mediana de 149,5. Isso sugere que a maioria dos problemas não gera um número elevado de debates. Em termos de popularidade, os número de *likes* na figura 4(d) variam entre 80 e 57.813, com uma mediana de 6.182, demonstrando que alguns problemas são significativamente mais apreciados que outros. Por fim, o número de *dislikes*, por sua vez, varia entre 37 e 18.882, com uma mediana de 411,5 (Figura 4(e)).

Em relação a distribuição dos problemas por nível de dificuldade, 217 problemas são classificados como fáceis pela plataforma LeetCode, 334 como dificuldade média e 65 como difíceis.

5.2. Qualidade do Código Gerado por LLMs

Inicialmente, analisam-se a qualidade dos 2.464 trechos de código gerados pelas LLMs, para os problemas da plataforma LeetCode. Detectou-se 2.871 *issues* relacionadas à problemas de manutenibilidade e 13.917 pontos de dívida técnica, conforme apresentado na Figura 5. Nos próximos parágrafos, os resultados são discutidos por categoria.



(a) *Manutenibilidade* (b) *Dívida técnica*

Figura 5. Quantidade de *issues* de manutenibilidade e dívida técnica em códigos gerados por LLMs

Manutenibilidade. A Figura 5(a) mostra o total de *issues* relacionadas à manutenibilidade identificadas por cada LLM. Como pode-se observar, detectou-se 2.871 *issues*. O modelo `Gemini` apresenta a maior quantidade de *issues* (761 ocorrências, 26.5%), enquanto o modelo `Llama 3.1` possui a menor frequência (678 ocorrências, 23.6%). Considerando a distribuição de *issues* por resposta, os valores são significativamente baixos. Por exemplo, a mediana é de aproximadamente uma *issue* por resposta. O modelo `Gemini` varia de 0 e 10 *issues* por resposta gerada, `Claude 3 Haiku` entre 0 e 8, `GPT-4` entre 0 e 9 e `Llama 3.1` variando entre 0 e 9.

Para avaliar os valores das métricas calculadas para as respostas geradas utiliza-se o teste de *Kruskal-Wallis* [Kruskal and Wallis 1952], um procedimento estatístico não paramétrico que avalia as medianas das amostras. Essa metodologia permite inferir a presença de diferenças estatísticas significativas entre três ou mais amostras de dados. Caso o *p-value* deste teste for menor ou igual a 0,05 indica que as distribuições são estatisticamente diferentes. Neste contexto, obteve-se *p-value* $= 0,019$ para a métrica de *issues* de manutenibilidade, sugerindo que as distribuições são estatisticamente diferentes.

O *Listing* 1 mostra um exemplo de *issue* de manutenibilidade gerada pelo modelo `Llama 3.1` para o problema *Second Minimum Node In a Binary Tree*,[14] que consiste em encontrar o segundo menor valor em uma árvore binária, onde cada nó tem dois ou nenhum filho. Como pode-se notar, existe um *code smell* na Linha 8, que indica condicionais que podem ser simplificadas (*"Nested Control Flow"*).

```
1 class Solution(object):
2     def findSecondMinimumValue(self, root):
3         if not root: return -1
4         ar = set()
5         def dfs(root):
6             ar.add(root.val)
7             if root.left:
8                 if root.left.val != root.val:)
9                     dfs(root.left)
```

Listing 1. Exemplo de issue em código gerado por LLM (`Llama 3.1`)

Dívida técnica. A Figura 5(b) mostra os resultados relacionados a dívida técnica total por LLM, que compreendem 13.917 ocorrências. O modelo `GPT-4` possui a maior taxa de dívida técnica (3.689 ocorrências, 26,5%), enquanto o modelo `Llama 3.1` possui a menor frequência (3.309 ocorrências, 23,8%). Considerando a distribuição de dívida técnica por código gerado, os valores variam entre aproximadamente 0 e 44 *issues* (`Gemini` 0–50; `Claude 3 Haiku` 0–35; `GPT-4` 0–51; `Llama 3.1` 0–41). A mediana gira em torno de cinco pontos de dívida técnica por resposta gerada. O teste de Kruskal-Wallis resultou em um *p-value* $= 0,03$ entre os grupos, sugerindo que as distribuições são estatisticamente diferentes.

5.3. Assertividade do Código Gerado por LLMs

A Tabela 1 mostra um visão geral das aproximadamente 2.5 mil respostas geradas pelas LLMs, em relação a assertividade. Obteve-se 1.777 respostas corretas e 111 incorretas. Existem também erros de sintaxe, impedindo a interpretação pela plataforma LeetCode (573 respostas erradas).

Tabela 1. Frequência da assertividade por LLM

Modelo	Soluções	Corretas	Incorretas	Erradas	Assertividade (%)
Llama 3.1	616	314	24	278	50.97
GPT-4	616	500	22	92	81.17
Gemini	616	420	29	166	68.18
Claude 3 Haiku	616	543	36	37	88.15
Total	2,464	1,777	111	573	-
Média	616.00	444.25	27.75	143.25	**72.12**

Observa-se que a maioria dos modelos apresentam uma alta taxa de assertividade, indicando uma boa precisão na geração de código.

Por exemplo, o modelo `GPT-4` teve uma taxa de assertividade de 81,17%, enquanto o modelo `Claude 3 Haiku` apresentou uma taxa de 88,15%. Em contraste,

[14] `https://leetcode.com/problems/second-minimum-node-in-a-binary-tree/`

o modelo `Gemini` teve uma taxa de assertividade de 68,18%. Por fim, o modelo `Llama 3.1`, teve uma taxa de assertividade de 50,97%, mostrando um desempenho intermediário.

Como consequência, o número de respostas incorretas e erradas é relativamente menor. O modelo `GPT-4` teve 92 casos de erro e 29 casos incorretos, enquanto o modelo `Claude 3 Haiku` apresentou 37 casos de erro e 36 casos incorretos. As maiores taxas de falhas foram encontradas nos modelos `Gemini` e `Llama 3.1`. Especificamente, o modelo `Gemini`, teve 166 casos de erro e 29 casos incorretos, enquanto o modelo `Llama 3.1` teve 24 casos de erro e 278 casos incorretos.

5.4. Análise da Assertividade dos Problemas após Parafraseamento

Para as 684 respostas incorretas ou erradas, realizou-se o parafraseamento do enunciado do problema correspondente. O trecho a seguir mostra um exemplo de enunciado após a reescrita do mesmo através do modelo *Pegasus* (ver mais detalhes na Seção 4.4). Como pode-se observar, a mesma instrução é apresentada de formas distintas, solicitando que dois números sejam somados e apresentados em formato de uma lista.[15]

> ***Enunciado original:*** *"You are given two non-empty linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading zero, except the number 0 itself."*

> ***Enunciado parafraseado:*** *"Two non-empty linked lists that represent two non-negative numbers are provided to you. Each of their nodes has a single digit, with the most important digit appearing first. Return the total as a linked list after adding the two numbers. You might suppose that, except from the number 0 itself, neither of the two numbers contain a leading zero."*

Este processo foi realizado para todas as respostas incorretas, resultando em uma taxa de assertividade de aproximadamente 52%. A Tabela 2 mostra os resultados por LLM. Como pode-se observar, o modelo Llama 3.1 possui os maiores valores de assertividade após o parafraseamento, enquanto apenas 11 respostas foram assinaladas como corretas no modelo Claude 3 Haiku após a reescrita do enunciado. A melhoria de assertividade nesta etapa, como a do Llama 3.1, não indica necessariamente um desempenho superior, uma vez que, ele falhou em problemas simples na primeira etapa de análise da assertividade, onde os outros modelos se saíram melhor.

O Listing 2 mostra um exemplo de código que foi gerado pelo modelo GPT-4. Esta resposta corresponde a um problema que solicita a maior *substring* ímpar presente em uma *string* numérica (*Largest Odd Number in String*[16]). Entretanto, o modelo forneceu um código incorreto, isto é, que não atende aos casos de testes propostos. O Listing 3 exibe um código distinto, gerado para o mesmo problema. Desta vez, o modelo recebeu o enunciado parafraseado, o que resultou em uma resposta correta.

[15] `https://leetcode.com/problems/add-two-numbers-ii`

[16] `https://leetcode.com/problems/largest-odd-number-in-string`

Tabela 2. Frequência da assertividade dos modelos após parafraseamento

Modelo	Soluções	Corretos	Incorretos	Erros	Assertividade (%)
Llama 3.1	302	213	21	68	70.37
Gemini	114	65	9	40	56.67
GPT-4	195	127	14	54	65.22
Claude 3 Haiku	73	11	4	58	15.22
Total	684	416	48	220	-
Média	171	104	12	55	**51.87**

```
class Solution:
    def largestOddNumber(self, num: str) -> str:
        for char in reversed(num):
            if int(char) % 2 != 0:
                return num[:len(num) - num.index(char
    )]
        return ""
```

Listing 2. Código gerado pelo GPT-4 (resposta incorreta)

```
class Solution:
    def largestOddNumber(self, num: str) -> str:
        for i in range(len(num) - 1, -1, -1):
            if int(num[i]) % 2 != 0:
                return num[:i+1]
        return ""
```

Listing 3. Código gerado pelo GPT-4 (resposta correta)

6. Discussão de Resultados

A alta taxa de assertividade das LLMs revela o seu potencial para aplicações práticas. O modelo `Claude 3 Haiku` apresentou uma alta taxa de assertividade para a geração de respostas para problemas de programação (88,15%), o que o torna promissor para aplicações onde precisão é crítica [Nguyen and Nadi 2022]. Os demais modelos também apresentaram uma taxa significativa de assertividade, reforçando a possibilidade de uso de LLMs em contextos profissionais e educacionais que possuem preocupações com a confiabilidade dos resultados.

A análise da qualidade do código pode complementar decisões relacionadas à escolha de LLMs. Embora o `Llama 3.1` tenha apresentado baixa assertividade (50,97%), seus bons resultados em manutenibilidade sugerem que a qualidade estrutural do código não garante, necessariamente, a resolução correta de problemas [Coignion et al. 2024]. Isso sugere que, ao selecionar um modelo para desenvolvimento de soluções de *software*, é importante considerar tanto a assertividade quanto a qualidade, equilibrando a precisão necessária com a facilidade de manutenção do código gerado.

O parafraseamento de instruções revela oportunidades para melhorias na interação com LLMs. A análise mostra que a assertividade geral dos modelos aumentou após o parafraseamento dos problemas (Tabela 2). Esse aumento destaca a importância de uma formulação cuidadosa das instruções, visto que este tipo de técnica têm sido amplamente investigada para aumento da produtividade no desenvolvimento *software*, como por

exemplo, para detecção de *code smells* [Silva et al. 2024], migração de APIs de forma automática [Almeida et al. 2024], e melhorias na escrita de testes [Alshahwan et al. 2024].

A facilidade de uso e assertividade das LLMs levanta questões sobre o seu uso em processos seletivos. Assim como o LeetCode, existem outras plataformas populares que disponibilizam diversos problemas e desafios de programação, com HackerRank[17] e CodeChef.[18] Grandes empresas de *software* têm utilizado estes sistemas em processos seletivos. Dessa forma, os resultados deste trabalho mostram uma possibilidade de fraudes nestas etapas, visto que foi possível resolver um conjunto significativo de problemas de forma fácil utilizando LLMs. Em outras palavras, candidatos podem fazer uso modelos LLMs para atingir bons resultados com esforço mínimo, um desafio que têm sido discutido na literatura recente [Canagasuriam and Lukacik 2024]. Isso ressalta a importância de desenvolver métodos de avaliação mais robustos e complementares, que não dependam apenas do desempenho em plataformas de algoritmos. É fundamental que recrutadores levem em consideração essas limitações e combinem avaliações com entrevistas técnicas práticas e supervisionadas para obter uma visão mais precisa das habilidades reais dos candidatos.

7. Ameaças à Validade

Nesta seção, são apresentadas as ameaças à validade deste estudo, assim como as estratégias adotadas para mitigá-las [Wohlin et al. 2012].

Primeiramente, em relação à validade externa, os resultados obtidos podem não ser generalizáveis para outros contextos, como problemas de algoritmos fora da plataforma LeetCode ou em linguagens de programação não abordadas. Além disso, as LLMs utilizadas neste estudo são modelos específicos, e suas performances podem não refletir o comportamento de outros modelos disponíveis no mercado. A escolha dos problemas mais populares e gratuitos da plataforma também pode introduzir viés, pois há uma predominância de questões de nível fácil, que refletem principalmente o contexto acadêmico, limitando a representatividade em cenários mais complexos. No entanto, este estudo inclui uma amostra significativa, composta por 616 problemas de algoritmos mais populares, o que, junto às quatro LLMs, proporciona uma base de 2.464 respostas para as conclusões.

Adicionalmente, pode-se identificar uma ameaça à validade de construção, uma vez que as métricas de qualidade adotadas, manutenibilidade e dívida técnica, podem não cobrir todos os aspectos qualitativos na avaliação das respostas geradas pelas LLMs. Para minimizar essa limitação, foram definidas métricas predefinidas e documentadas [SonarSource 2024], e todas as respostas foram revisadas sob as mesmas condições. Além disso, a análise da assertividade foi realizada em larga escala através da automação com *scripts*, sendo passível de falhas. Entretanto, utilizou-se a API oficial disponibilizada pela plataforma LeetCode. Ademais, o processo descrito na metodologia também foi executado de forma manual para uma amostra de problemas, visando validar e comparar as respostas obtidas.

Por fim, uma ameaça importante à replicabilidade do estudo está relacionada à volatilidade das LLMs e à variabilidade de suas respostas ao longo do tempo. Como essas

[17] `https://www.hackerrank.com`
[18] `https://www.codechef.com`

ferramentas passam por atualizações contínuas, tanto nos modelos quanto em seus dados subjacentes, o mesmo problema submetido em momentos diferentes pode resultar em respostas distintas. Essa característica das LLMs pode dificultar a reprodução exata dos resultados obtidos neste estudo, sendo recomendável que futuras pesquisas considerem um ambiente controlado ou versões fixas dos modelos, se possível, para garantir maior estabilidade e consistência nos resultados.

8. Conclusão

Este estudo teve como objetivo investigar a qualidade e assertividade do código gerado por diferentes *Large Language Models* ao resolver problemas de algoritmos. Para tanto, utilizou-se 616 problemas de programação populares na plataforma LeetCode. Para cada problema, uma solução foi gerada utilizando quatro LLMs: `GPT-4`, `Gemini`, `Claude 3 Haiku` e `Llama 3.1`. A análise concentrou-se na assertividade das respostas, bem como na avaliação da qualidade do código gerado. Adicionalmente, os enunciados dos problemas foram parafraseados, visando verificar o impacto na assertividade das respostas. Apresentam-se a seguir os principais resultados:

- O modelo Claude 3 Haiku apresentou uma maior taxa de assertividade, enquanto o Llama 3.1 teve a menor taxa de acertos, sugerindo que a estrutura interna e o treinamento dos modelos impactam diretamente sua capacidade de resolver problemas de programação.
- A análise do parafraseamento mostrou que, a reformulação dos enunciados de problemas pode melhorar a taxa de assertividade das LLMs. Por exemplo, obteve-se um aumento de 51,87% da taxa de sucesso em relação as respostas incorretas e com erros, isso destaca a ambiguidade dos enunciados como uma barreira para a assertividade dos modelos.
- A maioria dos problemas de código gerados pelos modelos não foi grave, com um impacto menor em termos de manutenibilidade e confiabilidade, o que sugere que os modelos podem gerar códigos suficientemente bons para uma grande variedade de problemas.

Como trabalhos futuros, recomenda-se investigar a qualidade e assertividade em problemas e contextos de maior complexidade, com ênfase em áreas críticas, como sistemas financeiros e de segurança. Sugere-se, ainda, a realização de um estudo qualitativo para aprofundar a análise dos problemas de código identificados, incluindo a exploração da perspectiva de desenvolvedores sobre as características e a complexidade do código gerado pelos modelos. Por fim, é recomendada a utilização de outras ferramentas de análise da qualidade de código, de modo a avaliar categorias distintas de problemas além das questões relacionadas à manutenibilidade e dívida técnica.

9. Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em:

`https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2024-1-tcci-0393100-pes-bernardo-aquino`

Referências

Alberts, I. L., Mercolli, L., Pyka, T., Prenosil, G., Shi, K., Rominger, A., and Afshar-Oromieh, A. (2023). Large language models (llm) and chatgpt: what will the impact on nuclear medicine be? *European journal of nuclear medicine and molecular imaging*, 50(6):1549–1552.

Almeida, A., Xavier, L., and Valente, M. T. (2024). Automatic library migration using large language models: First results. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–7.

Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., Marginean, A., Sengupta, S., and Wang, E. (2024). Automated unit test improvement using large language models at meta. In *32nd ACM International Conference on the Foundations of Software Engineering (FSE)*, page 185–196.

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., and Santos, E. A. (2023). Programming is hard - or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 500–506, New York, NY, USA. Association for Computing Machinery.

Bhayana, R. (2024). Chatbots and large language models in radiology: A practical primer for clinical and research applications. *Radiology*, 310(1):e232756.

Buscemi, A. (2023). A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *arXiv preprint arXiv:2308.04477*.

Canagasuriam, D. and Lukacik, E.-R. (2024). Chatgpt, can you take my job interview? examining artificial intelligence cheating in the asynchronous video interview. *International Journal of Selection and Assessment*.

Claude (2024). Claude 3 haiku. Disponível em: `Claude.ai`. Acesso em Novembro de 2024.

Coignion, T., Quinton, C., and Rouvoy, R. (2024). A performance study of llm-generated code on leetcode. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, EASE '24, page 79–89, New York, NY, USA. Association for Computing Machinery.

Cui, J., Zhang, R., Zhou, F., Li, R., Song, Y., and Gehringer, E. (2024). How much effort do you need to expend on a technical interview? a study of leetcode problem solving statistics. In *26th International Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10.

Curtis, B., Martin, R. A., and Douziech, P.-E. (2022). Measuring the structural quality of software systems. *Computer*, 55(3):87–90.

Denny, P., Kumar, V., and Giacaman, N. (2023). Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 1136–1142, New York, NY, USA. Association for Computing Machinery.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., and Prather, J. (2022). The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA. Association for Computing Machinery.

Gillies, A. (2011). *Software quality: theory and management*. Lulu. com.

Gmeiner, F. and Yildirim, N. (2023). Dimensions for designing llm-based writing support. In *In2Writing Workshop at CHI*.

Google (2024). Gemini - chat to supercharge your ideas. Disponível em: `https://gemini.google.com`. Acesso em Novembro de 2024.

ISO/IEC 25010:2011 (2017). Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. Standard, International Organization for Standardization.

ISO/IEC 91260:2003 (2003). Engenharia de software - qualidade de produto. Standard, Associação Brasileira de Normas Técnicas.

Kasneci, E., Seßler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., Günnemann, S., Hüllermeier, E., et al. (2023). Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274.

Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621.

Leetcode (2024). Leetcode - the world's leading online programming learning platform. Disponível em: `https://leetcode.com`. Acesso em Novembro de 2024.

Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., and Lo, D. (2024). Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Lopes, M. and Hora, A. (2022). How and why we end up with complex methods: A multi-language study. *Empirical Software Engineering*, 27:1–42.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219.

Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., and Bavota, G. (2023). On the robustness of code generation techniques: An empirical study on github copilot. In *45th International Conference on Software Engineering (ICSE)*, pages 2149–2160.

Meta (2024). Llama - he open-source ai model you can fine-tune, distill and deploy anywhere is now available in more versions. Disponível em: `https://www.llama.com`. Acesso em Novembro de 2024.

Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., and Jiang, Z. M. J. (2023). Github copilot ai pair programmer: Asset or liability? *J. Syst. Softw.*, 203(C).

Nguyen, N. and Nadi, S. (2022). An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 1–5, New York, NY, USA. Association for Computing Machinery.

Nistala, P., Nori, K. V., and Reddy, R. (2019). Software quality models: A systematic mapping study. In *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 125–134.

OpenAI (2024). Gpt-4. Disponível em: `https://openai.com/index/gpt-4/`. Acesso em Novembro de 2024.

Reeves, B., Sarsa, S., Prather, J., Denny, P., Becker, B. A., Hellas, A., Kimmel, B., Powell, G., and Leinonen, J. (2023). Evaluating the performance of code generation models for solving parsons problems with small prompt variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 299–305, New York, NY, USA. Association for Computing Machinery.

Rocha, O. V., Brito, A., Cleiton Tavares, L. X., and Assis, S. (2024). Analisando a qualidade do código em plataformas de cursos online abertos e massivos. In *12th Workshop on Software Visualization, Maintenance and Evolution (VEM). XV Brazilian Conference on Software: Theory and Practice (CBSoft)*, pages 1–12.

Rubio, C., Mella, F., Martínez, C., Segura, A., and Vidal, C. (2023). Exploring copilot github to automatically solve programming problems in computer science courses. In *2023 42nd IEEE International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–8.

Silva, L. L., Silva, J. R. d., Montandon, J. E., Andrade, M., and Valente, M. T. (2024). Detecting code smells using chatgpt: Initial insights. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 400–406.

SonarSource (2024). metric definitions — docs.sonarsource.com. `https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/`. [Acesso em: 25-05-2024].

Su, H., Ai, J., Yu, D., and Zhang, H. (2023). An evaluation method for large language models' code generation capability. In *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*, pages 831–838.

Taecharungroj, V. (2023). "what can chatgpt do?" analyzing early reactions to the innovative ai chatbot on twitter. *Big Data and Cognitive Computing*, 7(1):35.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. (2022). Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6).

Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs experience: Evaluating the usability of code generation tools powered by large language models.

In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA. Association for Computing Machinery.

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. C. (2018). Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, J. and Chen, Y. (2023). A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289.

Welsh, M. (2022). The end of programming. *Commun. ACM*, 66(1):34–35.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.