Analisando a Qualidade e Eficácia de Códigos Gerados por LLMs: Um Estudo com Problemas da Plataforma LeetCode

Bernardo Aquino,[1] **Aline Brito,**[2] **Cleiton Tavares,**[1]
Danilo Boechat,[3] **João Pedro Batisteli**[1]

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

[2]Instituto de Ciências Exatas e Biológicas, Departamento de Computação
Universidade Federal de Ouro Preto (UFOP)
Ouro Preto, MG – Brazil

[3]Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte, MG – Brazil

bernardoaquinoc@gmail.com,aline.brito@ufop.edu.br,

{cleitontavares,joao.batisteli}@pucminas.br,danilobs@cefetmg.br

Abstract. *Large Language Models are revolutionizing software engineering by automating code generation. Despite this increase in developer productivity, concerns remain about the quality and accuracy of the generated code. This study evaluates the effectiveness and quality of 2,464 code solutions generated by GPT-4, Gemini, Claude 3 Haiku, and Llama for 616 LeetCode problems in Python, achieving accuracy rates above 50%. Our analysis reveals that model performance is sensitive to the problem's formulation. Furthermore, while the code accumulates significant technical debt in aggregate (including 2,8K maintainability issues), the average incidence per individual solution remains low.*

Resumo. *Large Language Models (LLMs) estão revolucionando a engenharia de software ao automatizar tarefas de geração de código. Apesar do aumento na produtividade dos desenvolvedores, ainda há dúvidas sobre a qualidade e taxa de acerto do código gerado. Este trabalho investiga a eficácia e qualidade de 2.464 respostas de código gerado via GPT-4, Gemini, Claude 3 Haiku e Llama 3.1 em 616 problemas do LeetCode em Python. Os resultados mostram taxas de acertos superiores a 50%. A análise também revela que o resultado é sensível à formulação do problema e que, embora o código gerado acumule um débito técnico expressivo (2.8K problemas de manutenibilidade e 13.9K de dívida técnica), a incidência média por resposta individual permanece baixa.*

1. Introdução

Gerar código de forma automática e com qualidade continua a ser um desafio, mesmo com os progressos da área de engenharia de *software* [Dakhel et al. 2023]. Nos últimos anos, as ferramentas de geração de código baseadas em Inteligência Artificial (IA) e o

surgimento de *Large Language Models* (LLM) ganharam um considerável destaque, visando aumentar a produtividade dos desenvolvedores [Wang and Chen 2023]. Um exemplo notável é o `GitHub Copilot`,impulsionado pelo modelo Codex da OpenAI, que demonstra a habilidade de gerar código preciso para diversas tarefas de programação [Dakhel et al. 2023, Vaithilingam et al. 2022].

Diversos trabalhos focam na correção e avaliação de códigos gerados pelo `GitHub Copilot` em diferentes contextos. Por exemplo, Rubio et al. (2023) avaliam a 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) comparam os acertos entre alunos e o *Davinci* (versão *beta* do modelo *Codex*) em questões de um curso de programação. Outros estudos exploram a interação humana e a eficiência com que os desenvolvedores utilizam LLMs. Porém, tais estudos focam exclusivamente na ferramenta `Copilot` e não envolvem uma análise do código [Dakhel et al. 2023]. Além disso, diferentes estudos utilizaram problemas do LeetCode como forma de avaliar ferramentas de LLM [Billah et al. 2024, Guimaraes et al. 2025, Oertel et al. 2025]. Porém, não há 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].

Além disso, o cenário das IAs gera preocupações com plágio e dependência excessiva. Afinal, iniciantes em programação podem 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]. Além disso, 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].

Somados a este contexto, a rápida e ampla disponibilidade de ferramentas generativas tem levado a especulações sobre o futuro da educação em computação e desenvolvimento de *software* no geral [Welsh 2022]. Embora essas ferramentas possam auxiliar e agilizar o processo de geração de código e desenvolvimento, há dúvidas quanto à confiabilidade, taxa de acertos e a qualidade do código produzido. Ademais, seu uso crescente levanta preocupações sobre o impacto na segurança e na inteligibilidade das respostas. Trabalhos como o de Liu et al. (2024) investigaram mais a fundo o funcionamento e sintetização mais aprofundado de 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 para a resolução de problemas da plataforma LeetCode[1] utilizando quatro LLMs difundidos no mercado: `GPT-4`, `Gemini`, `Claude 3 Haiku` e `Llama 3.1`. Os objetivos específicos são: (I) analisar a qualidade do código gerado pelas LLMs; (II) avaliar a taxa de acertos das respostas das LLMs; e (III) investigar se variações na formulação das perguntas feitas às LLMs impactam o código gerado.

Como resultado, os modelos `Claude 3 Haiku` e `GPT-4` apresentam acertos

[1] `https://leetcode.com`

superior a 80% em problemas de programação, demonstrando bom desempenho tanto de modelos especializados quanto robustos. Apesar da menor taxa de acertos (51%), o modelo `Llama 3` destacou-se pelos melhores indicadores de qualidade estrutural. Observou-se ainda que variações nos enunciados influenciaram significativamente as respostas dos modelos, resultando em um aumento de aproximadamente 52% nos acertos em problemas anteriormente incorretos ou com erros de execução. Com base nesses resultados, foram geradas implicações relevantes para a comunidade de *software*.

O restante do artigo se divide da seguinte forma: A Seção 2 apresenta o referencial teórico e discute os trabalhos relacionados.A Seção 3 descreve a metodologia proposta para a realização do trabalho. As seções 4 e 5 apresentam e discutem os resultados obtidos na pesquisa. A Seção 6 contempla as ameaças a validade e suas mitigações. Por fim, a Seção 7 apresenta a conclusão e trabalhos futuros.

2. Trabalhos Relacionados

LLMs são redes neurais baseadas na arquitetura de transformadores e mecanismos de atenção, treinadas com vastos *corpus* de dados para prever a próxima palavra em uma sequência [Alberts et al. 2023]. Essa capacidade permite a execução de várias tarefas de linguagem natural, incluindo a geração de código [Taecharungroj 2023]. A literatura recente foca em avaliar a qualidade desse código. Estudos como os de Nguyen e Nadi (2022) e Rubio et al. (2023) investigam a corretude e a qualidade das sugestões do GitHub Copilot em desafios de programação (LeetCode) e métricas de ferramentas de análise estática como o SonarQube. Segundo os autores, o código gerado apresenta falhas de funcionalidade, *bugs* e problemas de manutenibilidade apesar de sua simplicidade.

Outra vertente de pesquisa concentra-se na sensibilidade dos LLMs às entradas do usuário (*prompts*). Denny et al. (2023) exploram como a engenharia de *prompts* afeta o desempenho na resolução de problemas, categorizando falhas comuns em *conceituais*, *ambíguas* ou decorrentes de *prompts fracos*. De forma complementar, Mastropaolo et al. (2023) demonstram que pequenas alterações semânticas na descrição de um problema podem levar o Copilot a gerar códigos funcionalmente diferentes e incorretos. Juntos, esses estudos evidenciam que a confiabilidade e a consistência das respostas de um LLM são altamente dependentes da forma como a requisição é formulada.

Em uma avaliação mais holística, Su et al. (2023) propõem uma metodologia para analisar a capacidade de geração de código de diferentes LLMs através de múltiplas dimensões como validade, correção, complexidade, segurança e legibilidade. Ao comparar modelos como ChatGPT e Bing AI, eles observam que LLMs que integram busca em tempo real podem oferecer maior robustez. Tais trabalhos são fundamentais por estabelecerem métodos de avaliação de desempenho, comparação entre ferramentas e análise de qualidade que informam a metodologia adotada neste artigo.

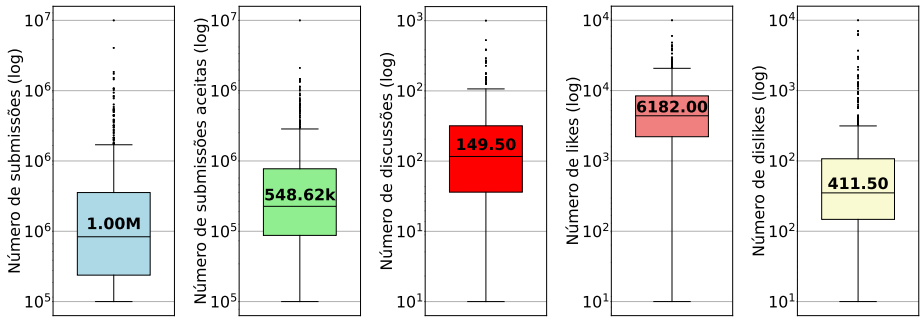
3. Metodologia

Esta seção apresenta a metodologia que inclui desde a seleção de problemas de programação na plataforma LeetCode e a subsequente geração de respostas, até uma análise de sua taxa de acertos e qualidade.

3.1. Seleção de Problemas de Programação na Plataforma LeetCode

Um conjunto de *scripts* foi elaborado 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, foram selecionados os top-25%, ordenados pelo número de submissões, por se tratar de uma métrica que sugere problemas de interesse da comunidade. Especificamente, 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 resposta 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 1. 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 1. Caracterização dos problemas LeetCode

A Figura 1(a) mostra que o número de submissões por problema varia entre 373.977 e 26.747.970 (mediana de 1.000.526 submissões) com 75% dos problemas apresentando até 2 milhões de submissões. A Figura 1(b) mostra que as aceitações variam entre 103.157 e 14.327.700 (mediana de 548.619,5) e 75% dos problemas com cerca de 1 milhão de aceitações. Na Figura 1(c) observa-se que a maioria dos problemas possui até 250 discussões (mediana de 149,5). Isso sugere que a maioria dos problemas não gera um número elevado de debates. Em termos de popularidade, o número de *likes* (Figura 1(d)) varia entre 80 e 57.813 (mediana de 6.182) e demonstra que alguns problemas são significativamente mais apreciados que outros. Por fim, o número de *dislikes* varia entre 37 e 18.882, com uma mediana de 411,5 (Figura 1(e)). Em relação à distribuição dos problemas por nível de dificuldade, 217 problemas são classificados como fáceis pela plataforma LeetCode, 334 como dificuldade média e 65 como difíceis.

3.2. Geração de Respostas para os Problemas de Programação

Este trabalho considera respostas geradas por quatro LLMs populares `GPT-4`, `Gemini 2.5 flash`, `Claude 3 Haiku` e `Llama 3.1`, desenvolvidas e mantidas por gran-

des companhias de *software*. Adotou-se a biblioteca G4F para gerar as respostas aos problemas selecionados de forma automatizada. 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,[2] 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], além de estar entre as linguagens mais populares no ranking de LLMs.[3]

> *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 2 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.

```python
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
```

Figura 2. Exemplo de assinatura requerida para submissão de problema na plataforma LeetCode (Two Sum problem)

3.3. Análise da Taxa de Acertos e Qualidade das Respostas

Após a geração das soluções dos problemas via LLMs, as respostas foram submetidas na plataforma LeetCode para verificar a taxa de acertos. Para tanto, elaborou-se um conjunto de *scripts* que interpretam e submetem soluções de código geradas pelas LLMs.

Para avaliação das respostas considera-se o conceito de eficácia definido em trabalhos anteriores [Mastropaolo et al. 2023a]: (i) *correta:* passou pelo interpretador e por todos os casos de teste propostos pela plataforma LeetCode; (ii) *incorreta:* passou pelo interpretador, mas não obteve sucesso em todos os casos de teste propostos pela plataforma LeetCode; (iii) *errada:* apresentou erro de sintaxe e que não passou pelo interpretador.

Por fim, analisa-se a qualidade dos trechos de código para respostas corretas utilizando SonarQube. A escolha deve-se a popularidade da mesma no contexto de análise estática de código [Rocha et al. 2024]. Especificamente, consideram-se as métricas de manutenabilidade e dívida técnica detectadas pela ferramenta.

[2] `github.com/xtekky/gpt4free`

[3] `https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`

3.4. Parafraseamento dos Problemas de Programação

Como última etapa deste trabalho, verifica-se a taxa de acerto 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 acertos das LLMs. Nesta etapa são analisadas todas as respostas que não foram aprovadas pelo interpretador do LeetCode ou pelos casos de testes.

O modelo *Pegasus* foi utilizado para o parafraseamento do enunciado dos problemas de programação.[4] Em seguida, o fluxo é executado novamente para gerar novas respostas e avaliar o impacto das soluções. O trecho a seguir mostra um exemplo de enunciado após a reescrita do mesmo. 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.[5]

> ***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."*

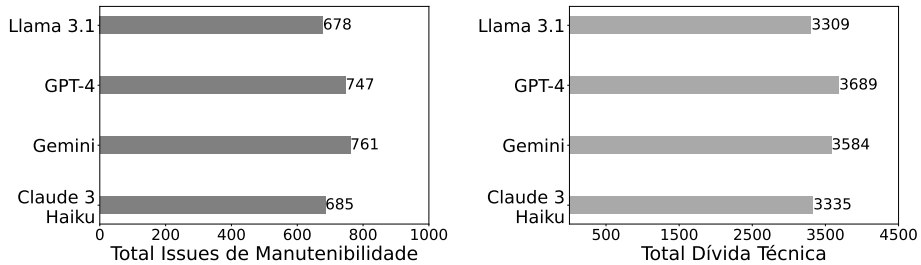
> ***Enunciado parafraseado:*** *"Two non-empty linked lists that represent two non-negative numbers are provided to you. Each of their nodes has a single digit, with the most important digit appearing first. Return the total as a linked list after adding the two numbers. You might suppose that, except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

4. Resultados

Nesta seção, apresentam-se os resultados obtidos neste trabalho.

4.1. Qualidade do Código Gerado por LLMs

Inicialmente, analisam-se a qualidade dos 2.464 trechos de código gerados para os 616 problemas selecionados, através do SonarQube, . Detectou-se 2.871 *issues* relacionadas à problemas de manutenibilidade e 13.917 pontos de dívida técnica, conforme apresentado na Figura 3. Nos próximos parágrafos, os resultados são discutidos por categoria.



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

Figura 3. Quantidade de *issues* de manutenibilidade e dívida técnica em códigos gerados por LLMs

[4]https://huggingface.co/tuner007/pegasus_paraphrase
[5]https://leetcode.com/problems/add-two-numbers-ii

Manutenibilidade. A Figura 3(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%). 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 utilizase 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*[6], 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 3(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.

4.2. Taxa de Acertos do Código Gerado por LLMs

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

Observa-se que a maioria dos modelos apresenta uma alta taxa de acertos, indicando uma boa precisão na geração de código. Por exemplo, o modelo `GPT-4` teve uma

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

Tabela 1. Frequência respostas corretas, incorretas e erradas por LLM

Modelo	Soluções	Corretas	Incorretas	Erradas	Acertos (%)
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**

taxa de acertos de 81,17%, enquanto o modelo `Claude 3 Haiku` apresentou uma taxa de 88,15%. Em contraste, o modelo `Gemini` teve uma taxa de acertos de 68,18%. Por fim, o modelo `Llama 3.1`, teve uma taxa de acertos 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.

4.3. Análise da Taxa de Acertos dos Problemas após Parafraseamento

Para as 684 respostas incorretas ou erradas, realizou-se o parafraseamento do enunciado do problema correspondente.

Este processo foi realizado para todas as respostas incorretas, resultando em uma taxa de acertos 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 acertos após o parafraseamento, enquanto apenas 11 respostas foram assinaladas como corretas no modelo Claude 3 Haiku após a reescrita do enunciado. A melhoria na taxa de acertos 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 acertos, onde os outros modelos se saíram melhor.

Tabela 2. Frequência de acertos dos modelos após parafraseamento

Modelo	Soluções	Corretos	Incorretos	Erros	Acertos (%)
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**

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*[7]). Entretanto, o modelo forneceu

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

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.

```
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)

5. Discussão de Resultados

A análise da qualidade do código pode complementar decisões relacionadas à escolha de LLMs. Embora o `Llama 3.1` tenha apresentado baixa de acertos (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 taxa de acertos 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 taxa de acertos 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 taxa de acertos 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[8] e CodeChef[9]. 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.

[8] `https://www.hackerrank.com`
[9] `https://www.codechef.com`

6. Ameaças à Validade

Esta seção descreve as principais ameaças à validade deste estudo, bem como as estratégias adotadas para mitigá-las. No que diz respeito à validade externa, os resultados obtidos podem não ser generalizáveis para outros contextos, como problemas fora da plataforma LeetCode, outras linguagens de programação não abordadas ou modelos e versões de LLMs distintos dos utilizados. Além disso, a seleção de problemas gratuitos e populares tende a favorecer questões mais simples, de domínios distintos e com viés acadêmico. No entanto, o uso de uma amostra expressiva, 616 problemas resolvidos por quatro LLMs, totalizando 2.464 respostas, contribui para a robustez das análises.

Referente à validade de construção, reconhece-se que as métricas de manutenibilidade e dívida técnica não capturam todos os aspectos qualitativos das respostas. Para mitigar essa limitação, foram adotadas métricas predefinidas e documentadas,[10] aplicadas de forma uniforme em todas as respostas analisadas. Além disso, apenas o conteúdo textual dos problemas da plataforma LeetCode foram considerados. A avaliação da taxa de acertos foi conduzida de maneira automatizada via *scripts* integrados à API oficial da LeetCode, somados à validação manual em uma amostra dos dados. Por fim, destaca-se como ameaça à replicabilidade a natureza volátil das LLMs, cujas respostas podem variar ao longo do tempo. Recomenda-se, portanto, que estudos futuros utilizem ambientes controlados ou versões fixas dos modelos para garantir maior consistência nos resultados.

7. Conclusão

Este estudo teve como objetivo avaliar a taxa de acertos e a qualidade do código gerado por quatro Large Language Models: `GPT-4`, `Gemini`, `Claude 3 Haiku` e `Llama 3.1`, na resolução de 616 problemas populares da plataforma LeetCode. As análises consideraram tanto a taxa de acerto das respostas quanto aspectos relacionados à manutenibilidade e à dívida técnica do código produzido. Além disso, os enunciados dos problemas foram parafraseados com o intuito de investigar o impacto da reformulação textual no desempenho dos modelos. Os resultados indicam que o `Claude 3 Haiku` apresentou a maior taxa de acertos, enquanto o `Llama 3.1` obteve o pior desempenho, evidenciando o papel decisivo das arquiteturas e dos dados de treinamento na eficácia das soluções geradas. Observou-se, ainda, que o parafraseamento contribuiu significativamente para a melhoria da taxa de acertos, com aumento de até 51,87% na taxa de acertos em respostas inicialmente incorretas ou com erros, o que destaca a ambiguidade textual dos enunciados como uma barreira relevante para o desempenho dos modelos.

Como direções futuras, sugere-se ampliar a análise para outros domínios, como finanças e segurança. Também é recomendada a realização de estudos qualitativos considerando a percepção dos desenvolvedores.

Disponibilidade de Artefato. Os *scripts* e *dataset* encontram-se disponíveis em: `github.com/alinebrito/vem2025-replication-package-qualidade-llm`

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

[10] `https://docs.sonarsource.com/sonarqube-server/latest/user-guide/code-metrics/metrics-definition/`

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 *54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506.

Billah, M. M., Roy, P. R., Codabux, Z., and Roy, B. (2024). Are large language models a threat to programming platforms? an exploratory study. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 292–301.

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*.

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.

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., and Jiang, Z. M. J. (2023). Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734.

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*, pages 10–19.

Guimaraes, E., Nascimento, N., Nelapati, A., and Shivalingaiah, C. (2025). Analyzing prominent llms: An empirical study of performance and complexity in solving leetcode problems. In *29th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, page 7–16.

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.

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. on Software Engineering and Methodology*, 33(5):1–26.

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.

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

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

Nguyen, N. and Nadi, S. (2022). An empirical evaluation of github copilot's code suggestions. In *The 2022 Mining Software Repositories Conference: MSR 2022: 18-20 May 2022, Virtual23-24 May 2022, Pittsburgh, Pennsylvania: Proceedings*, pages 1–5. Association for Computing Machinery, ACM.

Oertel, J., Klünder, J., and Hebig, R. (2025). Don't settle for the first! how many github copilot solutions should you check? *Information and Software Technology*, 183:107737.

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 *Procs of Conf. on Innovation and Technology in Computer Science Education*, pages 299–305.

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 International Symposium on Empirical Software Engineering and Measurement*, page 400–406.

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.

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 *Chi conference on human factors in computing systems*, pages 1–7.

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.