Avaliação de Qualidade de Código Java gerado por *Large Language Models*

Marco Tullio Oliveira[1], **Pedro Márcio Oliveira Silveira**[1], **Michelle Hanne S. de Andrade**[2]

[1]Pontifícia Universidade Católica de Minas Gerais
(PUC Minas), Belo Horizonte – MG – Brasil

[2]Instituto de Ciência Exatas e Informática (ICEI)
Departamento de Engenharia de Software - PUC Minas

marco.oliveira.1278372@sga.pucminas.br, pedro.marcio@sga.pucminas.br,

michelleandrade@pucminas.br

Abstract. *The quality of software systems has become increasingly relevant due to their widespread use in various fields. Large Language Models (LLMs) have emerged as a promising tool for improving software quality, but there are still gaps in understanding how LLMs affect this quality. This study addressed this gap by investigating the impact of LLMs on the quality of generated software. To achieve this objective, analyses were conducted to measure the quality of code generated on Java language by LLMs, using a dataset of 204 programming problems. This study aimed to contribute to closing the existing gap in the literature on the topic by analyzing quality metrics related to the effective application of LLMs in software development.*

Resumo. *A qualidade dos sistemas de software tem se tornado cada vez mais relevante devido ao seu amplo uso em diversas áreas. Os modelos de linguagem de grande porte (LLMs, do inglês Large Language Models) têm surgido como uma ferramenta promissora para aprimorar a qualidade do software, mas ainda há lacunas no entendimento de como os LLMs afetam essa qualidade. Este trabalho abordou essa lacuna, propondo investigar o impacto dos LLMs na qualidade do software gerado. Para alcançar esse objetivo, realizou-se análises para mensurar a qualidade do código gerado na linguagem Java por LLMs, utilizando 204 problemas de programação. Este estudo buscou contribuir para a redução da lacuna existente na literatura sobre o tema, ao analisar métricas de qualidade relacionadas à aplicação eficiente de LLMs no desenvolvimento de software.*

1. Introdução

Um dos desafios da Engenharia de Software é a qualidade nas etapas de desenvolvimento. Bibiano (2022) revela que, com frequência, os desenvolvedores aplicam refatoração composta com o objetivo de remover completamente *Code Smells*. Nos últimos anos, a Inteligência Artificial (IA) tem feito um progresso significativo em áreas como reconhecimento de imagem e reconhecimento de voz, entre outras [LUO and XIE 2018]. Nesse contexto, verifica-se que a IA pode ser uma aliada valiosa na busca pela qualidade de software, visto que as tendências apontam para o uso cada vez mais presente da IA [LUO and XIE 2018], [IMAI 2022].

A literatura explora a qualidade de software sob diferentes perspectivas. Por exemplo, Lu (2022) propõem um modelo de avaliação para testes de software críticos à segurança, enquanto Tang (2023) foca na criação de uma ferramenta para avaliar a qualidade da documentação. Outros estudos abordam o tema *Code Smells*, investigando desde os desafios de sua remoção [BIBIANO 2022], até a calibração de modelos de detecção automatizada com *feedback* humano para lidar com sua natureza subjetiva [NANADANI et al. 2023]. Este estudo busca contribuir para a avaliação de diferentes LLMs na qualidade de código Java.

A IA tem a capacidade de analisar dados complexos automaticamente fazendo o uso de Redes Neurais e algoritmos de Processamento de Linguagem Natural avançado, conforme demonstrado por Hourani et al.(2019). Empresas como Meta e Google exploram possibilidades dos LLMs desenvolvendo aplicações, como uma ferramenta para melhoria de testes unitários desenvolvidos originalmente por humanos [ALSHAHWAN et al. 2024]. Hourani, t al. (2019) preveem que, nos próximos 4 a 8 anos, a IA substituirá engenheiros de QA (*Quality Assurance*). Zhao (2021) revela que a qualidade é essencial para o software, pois software de baixa qualidade pode causar consequências variadas e graves. Portanto, nesse contexto, é importante entender como as IAs afetam a qualidade de software, não apenas em termos de testes, mas também em outros aspectos de qualidade como a manutenibilidade.

O objetivo geral deste trabalho é avaliar a qualidade de código Java gerada por LLMs. Destacam-se os seguintes objetivos específicos: (i) avaliar métricas de qualidade como Complexidade Ciclomática, Complexidade Cognitiva e *Code Smells* na geração de código por LLMs; (ii) realizar comparação entre diferentes LLMs, avaliando critérios como assertividade dos LLMs e tempo de execução dos algoritmos, e (iii) examinar o desempenho dos LLMs em tarefas de codificação de diferentes níveis de dificuldade.

Como resultado deste estudo, espera-se obter a comparação de como as IAs afetam a qualidade do software. Dessa forma, o estudo pretende fornecer uma base para futuras pesquisas e aplicações práticas, auxiliando desenvolvedores e organizações a tomarem decisões conscientes sobre o uso de IA em seus projetos de software.

Este artigo é organizado da seguinte forma: a Seção 2 apresenta os Trabalhos Relacionados; a Seção 3 evidencia os Materiais e Métodos utilizados para este estudo; a Seção 4 apresenta a Caracterização do Conjunto de Dados; a Seção 5 aborda a Discussão dos Resultados, e por fim, a Seção 6 apresenta a Conclusão e Trabalhos Futuros.

2. Trabalhos Relacionados

Nesta seção, são apresentados os trabalhos relacionados que abordam o uso de LLMs na geração e na qualidade de código.

Coignion et al. (2024) realizaram um estudo aprofundado sobre a eficiência dos LLMs na geração de código. O *dataset* foi composto de 204 problemas extraídos do LeetCode[1], distribuídos em três níveis de dificuldade: 56 fáceis, 104 médios e 44 difíceis. O estudo comparou 18 LLMs para a resolução dos problemas. Os resultados indicaram que, em média, os LLMs foram capazes de gerar soluções com desempenho comparável às soluções humanas, sendo que alguns modelos produziram código mais eficiente em

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

termos de tempo de execução. O presente estudo tem semelhanças com Coignion et al., como a origem do *dataset*, mas este trabalho utiliza outros LLMs e métodos de medição de qualidade diferentes, como SonarQube.

Merkel e Dörpinghaus (2025) realizaram um estudo quantitativo para avaliar a qualidade do código gerado por LLM em comparação com soluções desenvolvidas por humanos na plataforma LeetCode. Para isso, utilizaram o modelo *GPT-4o* e problemas extraídas do próprio LeetCode. O SonarQube e a correção do LeetCode foram escolhidos para obtenção de métricas. Os resultados indicaram que as soluções geradas pelo *GPT-4o* apresentaram uma menor incidência de *Code Smells* e uma menor Complexidade Cognitiva em comparação com o código humano. Ambos os estudos têm semelhanças, como o uso do SonarQube[2], mas o presente estudo não utiliza soluções desenvolvidas por humanos.

Mayer et al. (2024) realizaram um estudo comparativo para avaliar a performance de diferentes LLMs na geração de código a partir de texto. Foram analisados cinco modelos: *ChatGPT, BingChat, Bard, Llama2 e Code Llama*, considerando três métricas principais: corretude, tempo de execução e uso de memória. Os experimentos indicaram que o *ChatGPT* superou os demais modelos, resolvendo corretamente mais de 50% das tarefas, enquanto modelos como *Llama2 e Code Llama* tiveram um desempenho inferior, com menos de 10% de acertos. Ambos os estudos têm como foco LLMs, entretanto o presente estudo utiliza métricas diferentes, como Complexidade Cognitiva, e avalia também o *DeepSeek*.

Niu et al. (2024) avaliaram a eficiência do código gerado por LLMs, investigando sua execução em diferentes *benchmarks*. Para medir a eficiência dos códigos gerados, os autores utilizaram um ambiente de execução controlado, analisando métricas como tempo de execução e taxa de aceitação. O estudo destacou que a engenharia de *prompts* pode melhorar a eficiência do código gerado, especialmente ao adotar abordagens baseadas em *chain-of-thought*. Diferentemente do presente estudo, que utiliza ferramentas de métricas como SonarQube, a pesquisa de Niu et al. foca na utilização de *benchmarks*.

3. Materiais e Métodos

Nesta pesquisa, foi adotada uma abordagem quantitativa, voltada à análise do impacto de modelos de LLMs na qualidade do software Java. Dado o escopo do estudo e com base nos objetivos específicos, foram elaboradas três questões de pesquisa (QP), cada uma acompanhada de suas respectivas hipóteses: a hipótese nula (H0), que considera a inexistência de diferença estatisticamente significativa entre os atributos avaliados dos códigos produzidos pelas LLMs, e a hipótese alternativa (H1), que pressupõe a existência de tal diferença para pelo menos um dos modelos analisados.

- (QP1) Existe diferença significativa na assertividade entre os códigos gerados pelas distintas LLMs?
- (QP2) Existe diferença significativa na Complexidade Ciclomática dos códigos gerados pelas distintas LLMs?
- (QP3) Existe diferença significativa na Complexidade Cognitiva dos códigos gerados pelas distintas LLMs?

[2]*https://www.sonarsource.com/*

Na QP1, avalia-se a taxa de soluções corretas, considerando o percentual de respostas marcadas como *Accepted* pela plataforma LeetCode. Para validar essa métrica, é necessário que o código-fonte sugerido seja aprovado em todos os casos de teste disponibilizados.

Na QP2, investiga-se a qualidade estrutural dos códigos por meio da Complexidade Ciclomática. Por último, na QP3, analisa-se a legibilidade do código-fonte gerado pelas LLMs, utilizando a métrica de Complexidade Cognitiva.

A Complexidade Ciclomática é uma métrica que quantifica o número de caminhos independentes em um programa, onde valores menores sugerem código mais simples e menos custoso de testar e manter [NGUYEN and NADI 2022]. Já a Complexidade Cognitiva avalia a dificuldade de compreensão do código-fonte, considerando a quantidade de conceitos e estruturas que um programador precisa assimilar, onde valores mais baixos indicam maior legibilidade e facilidade de manutenção [NGUYEN and NADI 2022].

Para uma validação estatística consolidada emprega-se o teste de normalidade de Shapiro–Wilk e o teste não paramétrico de Mann–Whitney U [BARBETTA et al. 2010].

3.1. Arranjo Experimental

Para tanto, selecionou-se um conjunto de 204 problemas de código extraídos da plataforma LeetCode. O LeetCode é um ambiente *online* que possui uma vasta gama de problemas abordando codificação de algoritmos. Foram selecionados aleatoriamente: 56 problemas classificados como fáceis, 104 como médios e 44 como difíceis. Essa abordagem permitiu isolar variáveis específicas, como a dificuldade dos problemas e a performance dos LLMs, para determinar com maior precisão sua influência na qualidade do software.

Para a extração dos problemas do LeetCode, foi desenvolvido um *script* em Python versão 3.13.2, assim como todos os *scripts* utilizados para este estudo. Com isso, automatizou-se a captura dos seguintes atributos relacionados a cada problema: ID, Título, Nível de dificuldade, Link, Descrição, Código base em Java[3]. Esse *script* utilizou GraphQL[4] para a captura de forma aleatória dos problemas e geração de arquivo *CSV*, contendo as informações estruturadas dos problemas.

Posteriormente, foram criados *scripts* para a leitura do arquivo *CSV* e submissão dos problemas nas LLMs. A linguagem de programação adotada foi *Java*. Como ferramentas, foram utilizados os LLMs *ChatGPT*[5], *DeepSeek*[6] e *Gemini*[7]. Os *scripts* foram similares, apresentando algumas diferenças como o nome do modelo usado e configurações específicas de cada empresa. Em síntese, a descrição e o código base em Java eram inseridos dinamicamente no *prompt*. Foi instruído a retornar apenas o código com a resolução do problema. O código fonte gerado pela LLM então foi inserido em um novo *CSV* contendo os seguintes campos: ID, Título, Dificuldade, Link, Descrição, Código Resolvido por *LLM* em Java, nome da LLM. Todo esse processo foi repetido para cada um dos 204 problemas. A ferramenta de avaliação de qualidade escolhida, além do LeetCode, foi o SonarQube.

[3]Código previamente fornecido pelo LeetCode com nomes, tipos de funções e parâmetros já definidos
[4]*https://graphql.org/*
[5]*https://openai.com/chatgpt/overview/*
[6]*https://www.deepseek.com/*
[7]*https://gemini.google.com/*

A plataforma utilizada para o uso dos LLMs foi o *GitHub Models*[8], porém alguns modelos utilizados apresentavam uma limitação diária de uso na plataforma. Para lidar com isso, nos *scripts* que utilizam esta plataforma, foi criado um arquivo para o controle da quantidade de execuções realizadas e restantes para a totalidade dos problemas. Para cada um dos 204 problemas, foi gerada uma única solução por LLM, utilizando a primeira resposta obtida, sem qualquer tipo de realimentação para refinar o resultado.

Após a geração das soluções, cada código foi submetido à plataforma LeetCode para verificação automática dos resultados, a fim de confirmar se a implementação dos LLMs atingiu a funcionalidade esperada para cada problema. Para realização das submissões, foi implementado um *script* em Python e Selenium para realizar o *login* no LeetCode, efetuar a submissão dos problemas resolvidos por *LLMs* e extração dos resultados.

Em seguida, os códigos foram analisados por meio do SonarQube, ferramenta responsável por coletar métricas de qualidade, tais como Complexidade Ciclomática, Complexidade Cognitiva e *Code Smells*[9]

Dessa forma, os métodos adotados neste estudo possibilitaram análises entre as soluções geradas pelos diferentes LLMs, permitindo identificar padrões e diferenças na qualidade do código produzido. A utilização de um conjunto padronizado de problemas e de métricas quantitativas contribui para uma análise imparcial do impacto dos LLMs no desenvolvimento de software.

Tratando dos ambientes em que os códigos foram executados, foi escolhido o PyCharm 2024.3.2 para Windows. Os códigos foram executados em duas máquinas distintas: (1) Windows 11 *Home* 24H2, *RAM* de 16 GB, Armazenamento de 512 GB, Processador 11th Gen Intel i7-11390H 3.40GHz 2.92 GHz, Placa de Vídeo NVIDIA GeForce MX450 1GB; (2) Windows 10, *RAM* de 16 GB, 1 terabyte NVME, Processador Ryzen 7 5800 x 8 - Core, Placa de Vídeo RTX 3060.

Os artefatos deste estudo se encontram em https://zenodo.org/records/15742458.

4. Caracterização dos Dados

O presente estudo utilizou-se de 204 problemas do LeetCode, entre eles 56 fáceis, 104 médios e 44 difíceis. Esses problemas foram resolvidos na linguagem Java pelos seguintes modelos de linguagem: DeepSeek V3, DeepSeek V3 0325, ChatGPT 4o, ChatGPT 4o-mini, Gemini 2.0 Flash e Gemini 2.5 Pro. A *temperature* adotada para os modelos DeepSeek e Gemini foi a padrão para gerar os códigos. No modelo ChatGPT foi testado a *temperature* padrão e a *temperature = 0.1*, sendo a última escolhida para geração dos códigos. O GPT-4o, DeepSeek V3 e DeepSeek V3 0325 levaram 5 dias para gerar todos os códigos, já o GPT-4o-mini levou 2 dias para gerar os códigos. O Gemini 2.5 levou 6 dias para gerar todos os códigos, já o Gemini 2.0 Flash levou 1 dia para gerar todos os códigos.

A Figura 1 apresenta os histogramas que descrevem a distribuição do percentual de acerto obtido pelos 204 problemas. Para todos os casos adotou-se largura de classe de 10 p.p. (0–10%, 10–20%, ..., 90–100%). Convém frisar que o último intervalo do

[8] *https://github.com/marketplace?type=models*

[9] *Code Smells* são um sinal de alerta em um código-fonte que indica um problema mais profundo, embora não seja um erro que impeça a execução do programa para a avaliação da qualidade do código gerado.

histograma (90 – 100 p.p.) agrupa todas as submissões cujo percentual de acertos se encontra nessa faixa, isto é, contabiliza simultaneamente os problemas resolvidos de forma completa (100%) e aqueles com taxas de sucesso ligeiramente inferiores.

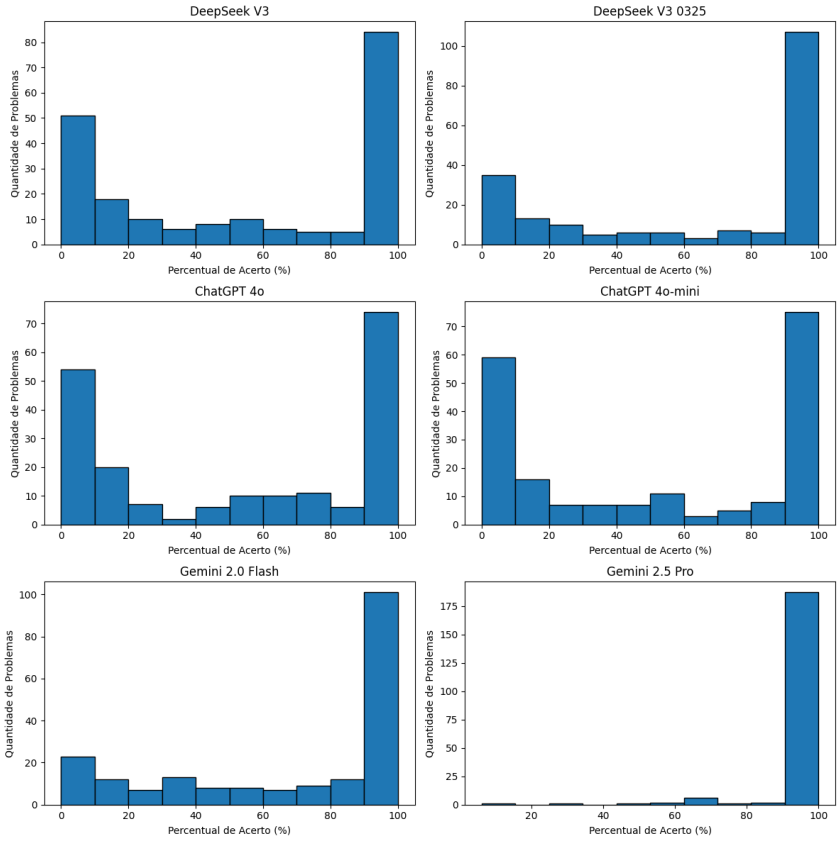

Figura 1. Histograma de percentual de acertos por modelo.

DeepSeek V3 0325 solucionou integralmente 77 problemas ($37{,}75\,\%$ do conjunto), configurando o segundo maior índice de acertos completos; o DeepSeek V3 obteve 61 *Accepted* ($29{,}90\,\%$) com perfil semelhante ao Gemini 2.0 Flash, com a mesma quantidade e percentual de acertos; o ChatGPT 4o registrou 53 aprovações ($25{,}98\,\%$), ao passo que sua variante GPT 4o-mini obteve os mesmos valores de assertividade; por fim, o Gemini 2.5 Pro destacou-se com 168 *Accepted* ($82{,}35\,\%$). Quando a análise se restringe apenas às submissões *Accepted*, observa-se uma redução geral dos valores extremos em comparação ao conjunto completo, embora a hierarquia relativa entre os modelos se mantenha.

Se tratando de Complexidade Ciclomática, a Figura 2 mostra que o Gemini 2.5 Pro preserva a mediana mais elevada (≈ 8) e o maior intervalo interquartil ($\approx 6,3$), alcançando máxima de 26. Isso indica que, mesmo nas soluções corretas, o modelo introduz fluxos de controle mais densos. O Gemini 2.0 Flash permanece em segundo lugar (mediana ≈ 7, máximo = 21), seguido pelo DeepSeek V3 0325 (mediana ≈ 6, máximo = 23). Já os modelos ChatGPT 4o e 4o-mini concentram suas medianas em torno de 4 pontos, com valores máximos não ultrapassando 10 pontos, sugerindo implementações mais concisas e homogêneas. O DeepSeek V3 mantém a mediana em cinco e limite superior

de 14 pontos, reforçando o caráter estruturalmente simples das suas respostas aprovadas. Os modelos que não apresentaram *outliers* foram os modelos ChatGPT 4o e 4o-mini.

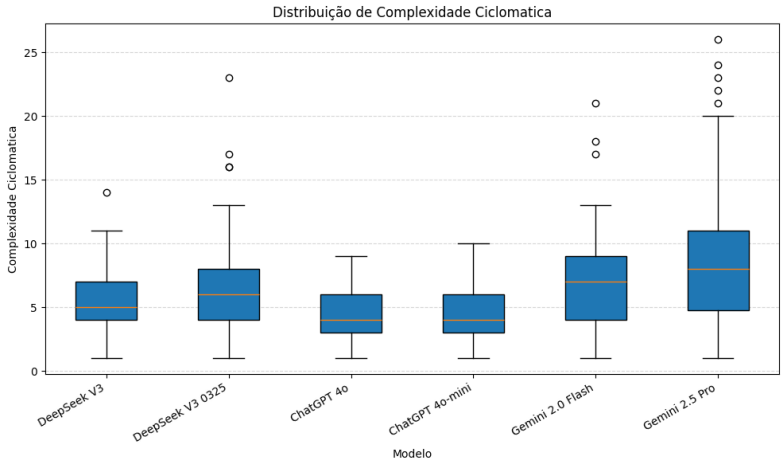

Figura 2. Boxplot da Complexidade Ciclomática por LLMs

O padrão também se replica na métrica de Complexidade Cognitiva. É evidenciado na Figura 3 que o Gemini 2.5 Pro, apresentando maior quantidade de amostras aceitas no LeetCode, exibe a maior mediana (≈ 10) e atinge valores máximos de 90 pontos, revelando aninhamentos e encadeamentos que podem dificultar a leitura. O Gemini 2.0 Flash segue-lhe (mediana ≈ 8, máximo = 26), enquanto o DeepSeek V3 0325 oscila na faixa entre 7 pontos de mediana e 90 pontos de máxima. Por outro lado, ChatGPT 4o e 4o-mini mantêm medianas de 5 a 6 e limites superiores de 20 pontos, registrando a menor carga cognitiva. O DeepSeek V3 situa-se no meio-termo (mediana ≈ 6, máximo = 25). O único modelo que não apresentou *outliers* foi o Gemini 2.0 Flash.

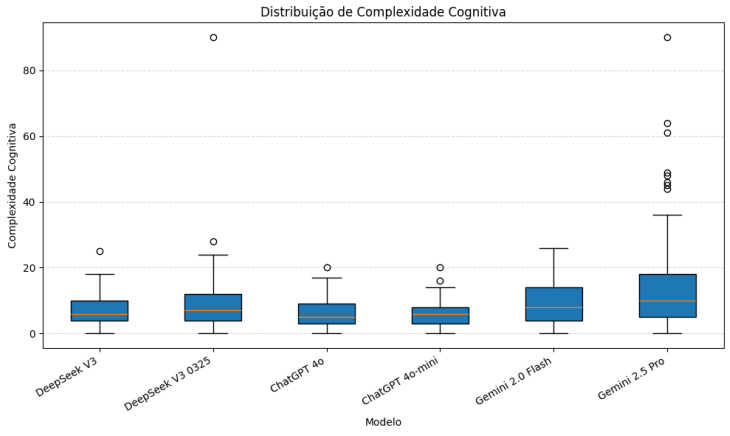

Figura 3. Boxplot da Complexidade Cognitiva por LLMs

Para os *Code Smells*, o Gemini 2.5 Pro mantém a maior mediana (≈ 5) e a cauda superior mais extensa (até 28 *smells*). Gemini 2.0 Flash e DeepSeek V3 0325 alinham-se em mediana 4, sendo suas máximas de 17 e 14 pontos (incluindo seus *outliers*), respectivamente. Já o DeepSeek V3 reduz a mediana para 4 e registra o limite superior de 9

pontos. As variantes ChatGPT 4o e 4o-mini confirmam o melhor perfil qualitativo, com mediana de 3 *smells*, sendo suas máximas valores de 8 e 9 pontos (incluindo seus *outliers*), respectivamente. Todos os modelos apresentaram *outliers*.

Em síntese, os modelos com maior taxa de acertos, notadamente o Gemini 2.5 Pro, tendem a gerar soluções mais complexas, enquanto ChatGPT 4o/4o-mini se destacam pela concisão estrutural e limpeza de código, ainda que resolvam menos problemas. Os *outliers* evidenciam que alguns problemas específicos são particularmente desafiadores para as LLMs, expondo limites em termos de clareza ou estilo nas soluções geradas.

5. Discussão dos Resultados

Esta seção discute criticamente os resultados obtidos, relacionando-os às três questões de pesquisa e às hipóteses respectivas.

Para atender à QP1, verificou-se a existência de diferenças nas taxas de *Accepted* entre as seis LLMs ao transformar a quantidade de acertos em percentuais. Cada modelo submeteu 204 problemas ao LeetCode, obtendo as seguintes taxas de acerto: ChatGPT 4o (25,98 %), ChatGPT 4o-mini (25,98%), DeepSeek V3 (29,90 %), DeepSeek V3 0325 (37,75 %), Gemini 2.0 Flash (29,90 %) e Gemini 2.5 Pro (82,35 %). Com base nesses valores, procedeu-se inicialmente ao teste de normalidade de Shapiro–Wilk sobre os vetores binários de aceitação (1 = *Accepted*; 0 = *Not Accepted*) referentes a cada LLM. Os resultados indicaram p-valores inferiores a 0,05 para todas as seis LLMs, indicando clara violação do pressuposto de normalidade. Em face desse resultado, adotou-se o teste não paramétrico de Mann–Whitney U em comparações pareadas, com alternativa unilateral ("Gemini 2.5 Pro $>$ outro modelo"), a fim de verificar se o modelo Gemini 2.5 Pro apresentava proporção de *Accepted* estatisticamente superior às demais. Observou-se diferença significativa em todas as comparações envolvendo Gemini 2.5 Pro (por exemplo: Gemini 2.5 Pro vs. ChatGPT 4o: $p = 1,82 \times 10^{-30}$; Gemini 2.5 Pro vs. DeepSeek V3 0325: $p = 2,03 \times 10^{-20}$; Gemini 2.5 Pro vs. Gemini 2.0 Flash: $p = 7,71 \times 10^{-27}$). Em virtude de todos os *p-values* serem inferiores ao nível de significância de 0,05, rejeita-se a hipótese nula H_0 de igualdade de percentual de assertividade e acolhe-se H_1, de que existe pelo menos uma diferença estatisticamente significativa na assertividade das soluções produzidas pelas LLMs analisadas. Conclui-se, portanto, que Gemini 2.5 Pro exibe superioridade estatisticamente significativa em termos de assertividade em relação aos demais modelos.

Com base nos dados das distribuições da Complexidade Ciclomática foi realizado o teste de normalidade de Shapiro-Wilk apontou *p*-valor inferior a 0,05 para todas as seis LLMs, indicando violação do pressuposto de normalidade das distribuições de Complexidade Ciclomática. Dessa forma, procedeu-se ao teste não paramétrico de Mann-Whitney U em comparações pareadas entre os seis grupos. Verificou-se diferença estatisticamente significativa (por exemplo, DeepSeek V3 vs. DeepSeek V3 0325: $p = 0{,}0486$; ChatGPT 4o vs. Gemini 2.0 Flash: $p = 0{,}00099$; Gemini 2.0 Flash vs. Gemini 2.5 Pro: $p = 0{,}0368$), enquanto apenas quatro pares não apresentaram diferença significativas (por ex., DeepSeek V3 vs. ChatGPT 4o: $p = 0{,}189$). Em face destas evidências, rejeita-se a hipótese nula H_0 de igualdade de Complexidade Ciclomática entre as LLMs, acolhendo-se H_1 de que existe pelo menos uma diferença estatisticamente significativa na Complexidade Ciclomática dos códigos produzidos pelas LLMs analisadas.

Para a questão de pesquisa QP3, referente à existência de diferença estatisticamente significativa na Complexidade Cognitiva dos códigos gerados pelas LLMs, aplicou-se inicialmente o teste de normalidade de Shapiro-Wilk aos valores de Complexidade Cognitiva das submissões *Accepted* de cada modelo. Em todos os seis casos verificou-se rejeição do pressuposto de normalidade, o que motivou o uso de testes não paramétricos de Mann-Whitney U em comparações pareadas. Nove comparações apresentaram p-valor inferior a $\alpha = 0{,}05$, destacando-se DeepSeek V3 vs. Gemini 2.5 Pro ($p = 1{,}59 \times 10^{-5}$), DeepSeek V3 vs. Gemini 2.0 Flash ($p = 0{,}0253$) e DeepSeek V3 0325 vs. ChatGPT 4o ($p = 0{,}0250$), ao passo que pares como DeepSeek V3 vs. DeepSeek V3 0325 ($p = 0{,}185$) e ChatGPT 4o vs. ChatGPT 4o-mini ($p = 0{,}734$) não evidenciaram diferença significativa. Diante desses resultados, rejeita-se a hipótese nula H_0 e acolhe-se a hipótese alternativa H_1 de que existe pelo menos uma diferença estatisticamente significativa na Complexidade Cognitiva dos códigos produzidos pelas LLMs analisadas.

6. Conclusão

Este trabalho investigou o impacto de seis LLMs na qualidade do software gerado, por meio de três questões de pesquisa: assertividade (QP1), Complexidade Ciclomática (QP2) e Complexidade Cognitiva (QP3). Os testes estatísticos aplicados (Shapiro–Wilk e Mann–Whitney U) levaram a rejeição de todas as H_0 e ao acolhimento das H_1.

Em termos de assertividade, o modelo Gemini 2.5 Pro superou os demais, alcançando 82,35 % de soluções *Accepted*, seguido pelo DeepSeek V3 0325, que obteve a segunda melhor taxa de acertos entre as LLMs avaliadas 38%. Contudo, o Gemini 2.5 Pro apresentou a maior Complexidade Ciclomática e Cognitiva, bem como o maior número de *Code Smells*. Por outro lado, ChatGPT 4o e sua variante mini mostraram-se os mais concisos, com valores medianos de complexidade significativamente menores, ainda que com taxas de acerto mais modestas. Com isso, os objetivos do trabalho foram respondidos.

Estes resultados são importantes para os desenvolvedores que usam ou pretendem utilizar essas LLMs, pois fornecem dados sobre qual modelo é mais adequado para diferentes objetivos. Com base nos resultados deste estudo, recomenda-se que os desenvolvedores considerem a utilização do Gemini 2.5 Pro quando a assertividade for um ponto prioritário. Por outro lado, caso a principal preocupação esteja relacionada à Complexidade Ciclomática e/ou Cognitiva, sugere-se a utilização do ChatGPT 4o ou de sua variante mini que apresentaram resultados medianos de complexidade menores aos demais modelos avaliados.

Para estudos futuros, é recomendado expandir a análise, incluindo um maior número de LLMs, como o Claude ou a LLaMA, bem como diferentes linguagens de programação, como Python e C#. Por fim, seria interessante investigar o impacto dessas LLMs em projetos e cenários reais, além de avaliar a qualidade do código produzido a longo prazo.

Referências

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 *Companion Pro-*

ceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pages 185–196.

BARBETTA, P. A., REIS, M. M., and BORNIA, A. C. (2010). *Estatística: para cursos de engenharia e informática*. Editora da UFSC.

BIBIANO, A. C. (2022). Completeness of composite refactorings for smell removal. In *Companhia Proceedings da 44ª Conferência Internacional IEEE/ACM sobre Engenharia de Software (ICSE-Companion)*, pages 264–268, Pittsburgh, PA, EUA.

COIGNION, T., QUINTON, C., and Rouvoy, R. (2024). A performance study of llm-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 79–89.

HOURANI, H., HAMMAD, A., and LAFI, M. (2019). The impact of artificial intelligence on software testing. pages 565–570.

IMAI, S. (2022). Is github copilot a substitute for human pair-programming? an empirical study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 319–321.

LU, Y., LI, C., WANG, S., LIU, Y., and DAI, J. (2022). A quality evaluation method for software testing about safety-critical software. pages 35–42.

LUO, X. and XIE, L. (2018). Research on artificial intelligence-based sharing education in the era of internet+. In *2018 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*, pages 335–338.

MAYER, L., HEUMANN, C., and Aßenmacher, M. (2024). Can opensource beat chatgpt?–a comparative study of large language models for text-to-code generation. *arXiv preprint arXiv:2409.04164*.

MERKEL, M. and Dörpinghaus, J. (2025). A case study on the transformative potential of ai in software engineering on leetcode and chatgpt. *arXiv preprint arXiv:2501.03639*.

NANADANI, H., SAAD, M., and SHARMA, T. (2023). Calibrating deep learning-based code smell detection using human feedback. pages 37–48.

NGUYEN, N. and NADI, S. (2022). An empirical evaluation of github copilot's code suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 1–5.

NIU, C., ZHANG, T., LI, C., LUO, B., and NG, V. (2024). On evaluating the efficiency of source code generated by llms. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 103–107.

TANG, H. and NADI, S. (2023). Evaluating software documentation quality. pages 67–78.

ZHAO, Y., HU, Y., and GONG, J. (2021). Research on international standardization of software quality and software testing. pages 56–62.