Análise da Qualidade e Assertividade Do Código Gerado por LLMs em C#: Um estudo com problemas de Algoritmos

Diego Machado Cordeiro[1]

[1]Instituto de Ciências Exatas e Informática, Pontifícia Universidade Católica de Minas Gerais (PUC Minas). Av. Brasil, 2023 – 30140-002 – Belo Horizonte – MG – Brasil

diego.cordeirosga.pucminas.br

Abstract. *This work presents a comparative analysis of the quality and accuracy of C# code generated by large language models (LLMs) such as GPT-5, DeepSeek, and Gemini. The evaluation considered code quality indicators such as bugs, memory usage, and execution time. Using algorithmic problems from leetcode, the generated code underwent assertion tests and static analysis. Quantitatively, GPT-5 achieved a 99.4% success rate, outperforming DeepSeek (96.1%) and Gemini (91.3%) by a margin of over 3 percentage points in accuracy. These results highlight GPT-5's superior reliability and efficiency in code generation, while DeepSeek and Gemini demonstrated solid but less consistent performance. This study reinforces the growing potential of LLMs for automatic code generation and their varying effectiveness depending on model design and complexity.*

Resumo. *Este trabalho apresenta uma análise comparativa da qualidade e assertividade do código em C# gerado por grandes modelos de linguagem (LLMs), sendo eles GPT-5, DeepSeek e Gemini. Foram avaliados indicadores como presença de bugs, uso de memória e tempo de execução. Utilizando problemas algorítmicos da plataforma leetcode, os códigos foram submetidos a testes de assertividade e análise estática. De forma quantitativa, o GPT-5 obteve uma taxa de acerto de 99,4%, superando o DeepSeek (96,1%) e o Gemini (91,3%) por mais de 3 pontos percentuais. Os resultados confirmam o desempenho superior do GPT-5 em precisão e eficiência, enquanto o DeepSeek e o Gemini apresentaram consistência, mas menor estabilidade. Esses achados reforçam o potencial das LLMs na geração automática de código e evidenciam diferenças relevantes entre os modelos analisados.*

Bacharelado em Engenharia de *Software* - PUC Minas
Trabalho de Conclusão de Curso (TCC)
Orientador de conteúdo (TCC I): João Oliveira - joãobatisteli@pucminas.br
Orientador de conteúdo (TCC I): Joana Souza - joanasouza@pucminas.br
Orientador de conteúdo (TCC I): Leonardo Vilela - leonardocardoso@pucminas.br
Orientador acadêmico (TCC I): Cleiton Tavares - cleitontavares@pucminas.br
Orientador do TCC II: João Oliveira - joãobatisteli@pucminas.br

Belo Horizonte, 10 de 12 de 2025.

1. Introdução

A inteligência artificial (IA), como uma tecnologia disruptiva emergente, tem promovido mudanças profundas em diversas áreas do conhecimento, como a economia, a educação e

a saúde [Li et al. 2021]. Sua capacidade de processar grandes volumes de informação de forma automatizada e executar tarefas intelectualmente complexas destaca seu potencial de aplicação em praticamente todas as esferas da atividade humana [Samuel et al. 2022], dentre essas atividades está a geração de código, que vem auxiliando programadores a desenvolverem soluções mais rapidamente. Apesar dos avanços recentes, a qualidade do código gerado automaticamente por *Large Language Models* (LLMs) ainda desperta questionamentos. Estudos apontam que, embora os códigos gerados por modelos como o *Generative Pre-trained Transformer* (GPT-4) [Achiam et al. 2023] sejam, em muitos casos, mais legíveis e aderentes a padrões de estilo, eles ainda apresentam falhas que exigem validação humana para garantir a precisão e o funcionamento correto [Poldrack et al. 2023].

A crescente adoção de LLMs na geração de código tem despertado grande interesse, sobretudo por seu potencial de facilitar o acesso à programação e aumentar a produtividade [Kazemitabaar et al. 2023]. Apesar disso, a qualidade e a assertividade do que é produzido levantam preocupações. Estudos apontam que a ausência de análise crítica no código gerado por LLMs pode resultar em falhas, vulnerabilidades e na propagação de *bugs*, comprometendo a confiabilidade [Negri-Ribalta et al. 2024]. A relevância dessas preocupações é corroborada por dados significativos. A dívida técnica, por exemplo, consome até 42% do tempo dos desenvolvedores, enquanto códigos de baixa qualidade podem apresentar até 15 vezes mais defeitos e exigir 124% mais tempo para correção [Tornhill and Borg 2022]. No que diz respeito à eficiência, estudos indicam variações expressivas no desempenho de códigos gerados automaticamente, especialmente quanto ao tempo de execução. Niu et al. (2024) avaliaram a eficiência de códigos produzidos por LLMs em benchmarks como HumanEval, MBPP e LeetCode, constatando diferenças significativas no tempo de execução entre modelos, com o GPT-4 apresentando, em média, o melhor desempenho e menor tempo de execução. Esses resultados ressaltam como diferenças sutis na implementação podem impactar diretamente a performance e a viabilidade prática de soluções produzidas por modelos de linguagem. Diante desse cenário, **o problema central deste trabalho reside na incerteza quanto à capacidade dos LLMs de gerarem código em C# que seja, simultaneamente, correto, eficiente e de alta qualidade em problemas algorítmicos, considerando aspectos como presença de *bugs*, complexidade ciclomática, segurança, uso de memória e tempo de execução**. A escolha pela linguagem C# para a condução dos experimentos fundamenta-se em sua ampla adoção na indústria de software e em sua representatividade dentro do paradigma orientado a objetos [Nanz and Furia 2015].

A qualidade do código é um fator essencial no desenvolvimento de software, sendo frequentemente avaliada por atributos como legibilidade, manutenibilidade e eficiência [Fawareh et al. 2024]. Com a evolução da inteligência artificial, a geração automática de código tem se tornado um tema de grande interesse, tanto na área acadêmica quanto na indústria [Eltabakh et al. 2024]. Diante disso, é fundamental investigar e avaliar a qualidade do código gerado por LLMs, como realizado por Clark et al. (2024) , que analisaram mais de 600 trechos de código Python gerados pelo ChatGPT e demonstraram que, embora o modelo produza soluções de boa qualidade e consistentes, ainda há ocorrência de erros ocasionais que exigem revisão antes da integração. À medida que empresas e desenvolvedores adotam essas tecnologias para acelerar a criação de software, cresce a necessidade de compreender seus impactos na qualidade do código. Ana-

lisar esses efeitos pode ajudar a identificar possíveis limitações e riscos, além de fornecer subsídios para melhorar o uso de LLMs no desenvolvimento de sistemas mais confiáveis e seguros.

Com base nas lacunas identificadas na literatura e na necessidade de compreender melhor o comportamento das LLMs na geração de código, **o objetivo geral deste trabalho é realizar uma análise comparativa da qualidade e da assertividade do código em C# produzido por modelos de linguagem voltados à programação, sendo eles o GPT-5, o DeepSeek e o Gemini, considerando aspectos de correção e desempenho**. Para alcançar esse propósito, os objetivos específicos são: (i) analisar a qualidade estática dos códigos C# gerados por LLMs, observando a presença de *bugs* e *code smells*, a aderência a padrões de codificação e a complexidade ciclomática do código; (ii) coletar e avaliar métricas de desempenho em tempo de execução, como uso de memória e tempo de processamento; (iii) verificar a assertividade das respostas geradas, avaliando se as soluções atendem corretamente aos requisitos dos problemas propostos; (iv) comparar as versões Lite e completas dos modelos de linguagem, observando diferenças de desempenho, assertividade e qualidade do código gerado.

Os resultados desta pesquisa demonstraram diferenças relevantes entre os modelos analisados na geração de código em C#. O ChatGPT-5 apresentou o melhor desempenho geral, com maior taxa de aceitação (99,4%), menor tempo de execução (33,53 ms) e baixo número de erros. O DeepSeek V3.2 obteve resultados próximos, mostrando alta consistência e estabilidade. O Gemini 2.5 Flash teve desempenho intermediário (91,3%), enquanto a versão Lite apresentou desempenho inferior (82,4%) e maior número de falhas de compilação. A análise estática indicou baixos índices de vulnerabilidades em todos os modelos, mas o ChatGPT-5 e o DeepSeek geraram códigos mais limpos e estáveis, com menor incidência de *code smells*. Já os modelos Gemini produziram soluções mais extensas e complexas. Esses achados evidenciam que modelos de maior capacidade geram código mais funcional e aderente a boas práticas de engenharia de software.

O restante deste trabalho está organizado da seguinte forma: a Seção 2 apresenta os conceitos abordados neste estudo. A Seção 3 discute os principais trabalhos relacionados. A Seção 4 detalha os materiais e métodos utilizados neste estudo. A Seção 5 apresenta e discute os resultados obtidos. As Seções 6 e 7 apresentam as ameaças à validade, as estratégias de mitigação adotadas, a conclusão e os trabalhos futuros.

2. Fundamentação Teórica

Esta seção tem como objetivo apresentar os principais conceitos teóricos que fundamentam o desenvolvimento deste trabalho. Inicialmente, é discutido o conceito de LLMs, conforme detalhado na Seção 2.1. Em seguida, aborda-se o conceito de qualidade de código, conforme apresentado na Seção 2.2. Na sequência, a Seção 2.3 explora os fundamentos relacionados ao desempenho de software, com o intuito de contextualizar sua relevância no escopo da presente pesquisa.

2.1. LLMs

Modelos de Linguagem de Grande Escala (*LLMs*), fundamentados na arquitetura de aprendizado profundo *Transformer* [Vaswani et al. 2017], têm sido amplamente aplicados a diferentes domínios, incluindo a geração automática de código. Nesse contexto, os *Code*

LLMs destacam-se como ferramentas relevantes no desenvolvimento de software. Treinados com vastos volumes de dados, que combinam código-fonte, documentação técnica e linguagem natural, esses modelos são capazes de interpretar descrições textuais e gerar código em diversas linguagens, como Python, Java e C#.

Ferramentas como ChatGPT, Gemini e DeepSeek exemplificam esse avanço, oferecendo funcionalidades que vão da geração e completamento de código à tradução entre linguagens, explicação de trechos e correção automática de erros [Chen et al. 2021, Roziere et al. 2023]. Essas capacidades otimizam o tempo de desenvolvimento, democratizam o acesso à programação, e tornam o processo mais acessível, eficiente e produtivo, especialmente para iniciantes e em contextos de prototipação rápida [Jiang et al. 2024].

Além disso, as diferenças entre modelos completos e versões *lite* impactam diretamente tarefas complexas, como a geração de código. Modelos completos oferecem arquiteturas maiores, janelas de contexto ampliadas e melhor otimização, enquanto versões *lite* priorizam eficiência e menor custo computacional, apresentando limitações em cenários mais exigentes. Essa distinção é evidenciada por [Shean et al. 2025], que mostram que o DeepSeek-R1-Lite, embora competitivo em questões simples, tem desempenho inferior aos modelos completos em tarefas que requerem maior raciocínio, especialmente em comparação ao OpenAI o1 Pro.

2.2. Qualidade do Código

A qualidade do código-fonte é essencial para garantir confiabilidade, manutenibilidade, testabilidade e legibilidade. Ela envolve a capacidade de ler, manter e testar o código de forma a atender às necessidades explícitas e implícitas do projeto, abrangendo tanto aspectos técnicos quanto organizacionais [Shyamal et al. 2023].

Sua avaliação considera atributos como complexidade ciclomática, presença de *bugs*, vulnerabilidades e *code smells*, que afetam diretamente a manutenção e a robustez do software. Modelos de avaliação buscam alinhar esses indicadores a métricas e referências definidas por especialistas, frequentemente validadas por ferramentas de análise estática como o SonarQube [Shyamal et al. 2023].

Além das normas ISO/IEC 9126-1:2001 e ISO/IEC 25010:2011, que definem atributos como manutenibilidade, eficiência e confiabilidade [ISO 2001, ISO 2011], o SWEBOK complementa essa base ao reunir práticas e métricas amplamente reconhecidas na engenharia de software, fortalecendo a fundamentação teórica do estudo [Bourque et al. 2024].

2.3. Desempenho de Software

O desempenho de software refere-se à eficiência com que um sistema utiliza recursos como CPU, memória, entrada e saída e rede, considerando também a arquitetura em que é executado [Hennessy and Patterson 2011]. Suas principais métricas incluem a complexidade de tempo, que indica o crescimento das operações conforme o tamanho da entrada, e a complexidade de espaço, que avalia a memória utilizada [Cormen et al. 2022]. A primeira é crucial para escalabilidade e velocidade, enquanto a segunda garante uso eficiente da memória e evita lentidão ou falhas [Jalaman and Teleron 2024]. Por isso, a otimização de tempo e espaço é fundamental, especialmente em ambientes com recursos limitados ou grandes volumes de dados.

A melhoria conjunta desses fatores contribui diretamente para a qualidade do código. Algoritmos rápidos e com uso parcimonioso de memória favorecem escalabilidade e confiabilidade, e a norma ISO/IEC 25010:2011 destaca o desempenho, incluindo tempo de resposta, uso de recursos e adaptação a diferentes cargas, como atributo essencial. Wahlgren et al. (2022) reforçam que estratégias de desagregação e provisionamento dinâmico de memória são importantes para lidar com diferentes padrões de acesso. Dessa forma, ao comparar códigos gerados por LLMs, medir tempo de execução e uso de memória torna-se fundamental para avaliar a adequação das soluções propostas.

3. Trabalhos Relacionados

Nesta seção, são discutidos os estudos que ajudam a explicar o contexto deste trabalho, ao lado de estudos que usaram metodologias semelhantes para avaliar a qualidade ou a eficiência do código gerado por LLMs. Especificamente, os trabalhos relacionados discutidos nesta seção envolvem a avaliação da qualidade do código gerado por LLMs, a comparação entre diferentes modelos de LLMs e a análise de métricas de desempenho em código gerado automaticamente.

O estudo de Niu et al. (2024) analisou a eficiência do código gerado por LLMs como GPT-4 e Code Llama em benchmarks como HumanEval e LeetCode, focando em tempo de execução e complexidade algorítmica. A metodologia incluiu a execução de soluções geradas pelos modelos, comparando-as com códigos otimizados escritos por humanos, além de investigar padrões de ineficiência e limitações dos modelos, como o uso inadequado de estruturas de dados e algoritmos subótimos. Os resultados mostraram que a eficiência não está diretamente relacionada à taxa de acerto dos modelos e que técnicas como *chain-of-thought prompting* podem melhorar o desempenho, especialmente em problemas complexos. Embora relacionado ao presente trabalho na avaliação de LLMs, o estudo diferiu por incluir uma análise mais abrangente da qualidade do código, contemplando métricas estáticas como bugs e code smells, por focar especificamente na linguagem C#, que não fora abordada na pesquisa de referência, e por empregar um conjunto mais diversificado de modelos, ampliando as possibilidades de comparação. Essa abordagem multidimensional complementou investigações anteriores sobre eficiência e forneceu novas perspectivas sobre a qualidade do código gerado.

De forma complementar, o estudo de Patel et al. (2024) comparou códigos gerados por inteligência artificial (IA) e códigos escritos por humanos, com foco em métricas de software e bugs. Utilizando 90 problemas do LeetCode, os autores coletaram soluções humanas e geraram equivalentes com o GitHub Copilot. As métricas e bugs foram extraídos com SpotBugs e PMD, revelando correlações entre características do código e ocorrência de bugs específicos, além de padrões que impactam a qualidade. A metodologia analisou métricas de qualidade e padrões de falhas, oferecendo uma visão detalhada da relação entre estrutura do código e problemas detectados. O estudo se relaciona ao presente trabalho pelo uso do LeetCode e de ferramentas de análise estática, mas difere quanto ao foco: Patel et al. comparam código humano e gerado por IA, enquanto este trabalho compara diferentes LLMs. Também há diferenças na linguagem, pois o estudo citado utiliza Java, enquanto este utiliza C#, e na abrangência dos modelos, já que Patel analisa apenas o GitHub Copilot, enquanto aqui são avaliados vários LLMs.

O artigo de Nikolaidis et al. (2024) avaliou a eficácia do ChatGPT e do GitHub

Copilot na geração de soluções em Python para problemas do LeetCode. Foram selecionados 60 problemas, divididos igualmente em níveis de dificuldade (fácil, médio e difícil), utilizando a ferramenta de seleção aleatória da própria plataforma. Cada problema foi resolvido pelos modelos sem intervenção humana inicial, sendo posteriormente realizada a análise de performance e de tipos de erros. Além disso, o estudo investigou a capacidade do ChatGPT de incorporar feedback de erros para melhorar suas soluções ao longo de múltiplas iterações. Foram avaliadas métricas como complexidade ciclomática, número de linhas de código e tempo e espaço de execução. Os resultados mostram que o ChatGPT teve um desempenho ligeiramente superior ao Copilot, com 15% mais soluções corretas, e que ambos apresentaram maior incidência de erros lógicos do que de tempo de execução ou limite de tempo. O estudo se relaciona com o presente trabalho ao avaliar a capacidade de LLMs em resolver problemas algorítmicos utilizando plataformas de codificação. No entanto, difere por abordar apenas dois modelos e focar na linguagem Python, enquanto o presente trabalho avalia quatro LLMs aplicados à linguagem C#.

O estudo de Nguyen e Nadi (2022) avaliou o desempenho do GitHub Copilot na geração de código para 33 problemas do LeetCode em quatro linguagens de programação distintas: Java, Python, C e JavaScript. Para isso, foram construídos contextos de consulta baseados na descrição natural dos problemas, que alimentaram o Copilot para gerar a primeira sugestão de solução. A avaliação considerou a corretude das respostas, utilizando os próprios testes automáticos do LeetCode, e a compreensibilidade do código por meio da análise de complexidade ciclomática e cognitiva realizada com a ferramenta SonarQube. Os resultados mostraram que o Copilot obteve a maior taxa de corretude para Java (57%) e o pior desempenho para JavaScript (27%). Em termos de qualidade estrutural, o código gerado apresentou, em geral, baixa complexidade, embora tenham sido encontrados problemas, como dependência de funções auxiliares não implementadas. Este estudo se relaciona com o presente trabalho por avaliar a capacidade de LLMs em resolver problemas algorítmicos e coletar métricas de qualidade de código. Contudo, difere por focar exclusivamente no GitHub Copilot e não considerar métricas de desempenho de execução, enquanto o presente trabalho analisa quatro LLMs e inclui métricas como uso de memória e tempo de processamento.

O artigo de Coelho (2024) analisou a qualidade e a assertividade do código produzido por quatro LLMs (GPT-4, Gemini, Claude 3 Haiku e Llama 3.1) em 616 problemas algorítmicos do LeetCode, utilizando a biblioteca GPT-4 Free (G4F) para automação das requisições. A G4F, um projeto de código aberto e prova de conceito (*proof of concept*), permite chamadas a múltiplos provedores de LLMs com suporte a balanceamento de carga, controle de fluxo e gerenciamento de *timeouts*. Cada problema foi resolvido em Python seguindo um formato padronizado e, posteriormente, as respostas foram submetidas ao LeetCode para classificação em corretas, incorretas ou erradas. O estudo também utilizou o SonarQube para extrair métricas de qualidade do código, como manutenibilidade e dívida técnica, e testou o impacto do parafraseamento dos enunciados na assertividade das soluções. Os resultados mostraram que o GPT-4 e o Claude 3 Haiku obtiveram as maiores taxas de acerto, enquanto o Llama 3.1 teve o pior desempenho. Observou-se ainda que o parafraseamento contribuiu para melhorar a taxa de sucesso em casos inicialmente falhos. O estudo se relaciona com o presente trabalho ao avaliar código gerado por LLMs para problemas algorítmicos e coletar métricas de qualidade, mas difere por adotar apenas Python e não considerar métricas de execução, como tempo de processamento e

uso de memória, abordadas neste trabalho.

4. Materiais e Métodos

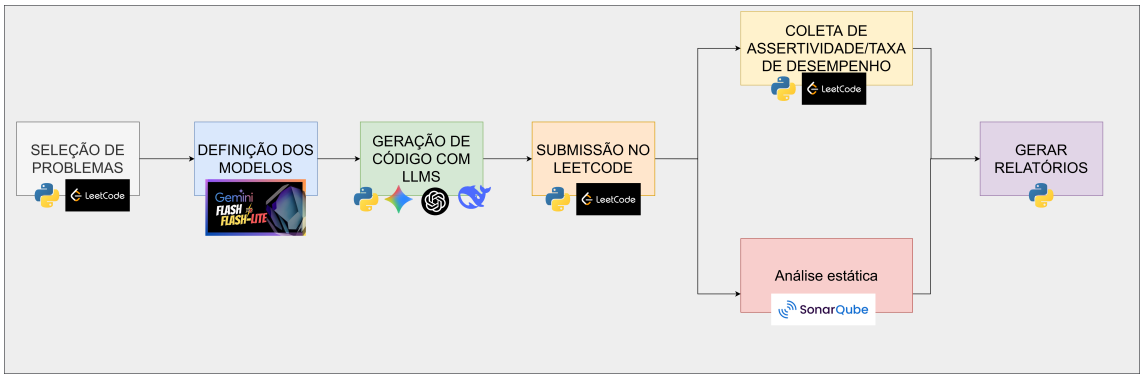

Figura 1. Visão geral da metodologia

Nesta seção, são descritos os materiais e métodos utilizados na condução deste estudo. Este trabalho é classificado como uma pesquisa quantitativa com o objetivo exploratório [Wohlin et al. 2012], pois busca analisar a qualidade e assertividade do código gerado por diferentes LLMs. Dessa forma, adotou-se uma abordagem quantitativa, permitindo uma análise ampla dos dados coletados e contribuindo para uma compreensão mais aprofundada do tema abordado. Além disso, detalha-se o processo de seleção das questões utilizadas nos experimentos e o método empregado para a análise dos resultados, incluindo os critérios de avaliação e as métricas consideradas para aferir a qualidade do código gerado. A Figura 1 apresenta um resumo das etapas da metodologia, que envolvem a seleção de problemas na plataforma LeetCode para coleta de metadados, a definição e comparação entre as versões *Lite* e completas dos modelos de linguagem, a geração de código por diferentes LLMs, a submissão e avaliação das soluções na plataforma LeetCode para verificação de assertividade e desempenho, a análise estática dos códigos com a ferramenta SonarQube e, por fim, a geração de relatórios e visualizações dos dados.

4.1. Seleção de problemas

A seleção de problemas para esta pesquisa utilizou a plataforma LeetCode, fundada em 2015 com a missão de auxiliar engenheiros de software no aprimoramento de suas habilidades e no avanço de suas carreiras [Cui et al. 2024]. Para isso, foram desenvolvidos os *scripts* de coleta de problemas[1] responsáveis por filtrar automaticamente os problemas gratuitos categorizados como algoritmos. Os metadados coletados para cada problema incluíram informações como nível de dificuldade, número de submissões, quantidade de *likes* e *dislikes* e o trecho de código disponibilizado pela plataforma.

A partir dessa base, foram escolhidos 1.000 problemas, distribuídos em 500 de nível médio, 250 fáceis e 250 difíceis. Essa distribuição foi definida de modo a privilegiar uma maior proporção de problemas de dificuldade intermediária e quantidades equivalentes nos níveis fácil e difícil, seguindo a metodologia proposta em estudos anteriores [Coignion et al. 2024, Wang et al. 2024]. Tal abordagem contribui para uma melhor representatividade do conjunto de dados, contemplando diferentes perfis de problemas e

[1]Repositório no GitHub – Script de Coleta

ampliando a diversidade dos cenários analisados. O conjunto final de problemas utilizados encontra-se disponível no *Repositório de Problemas*[2], permitindo a verificação e replicação dos experimentos.

Finalmente, os problemas selecionados foram armazenados em um arquivo no formato *JavaScript Object Notation* (JSON). A escolha deste formato se deu pela sua simplicidade, amplo suporte e facilidade na manipulação posterior dos dados para análises ou outras aplicações. Cada registro no arquivo contém informações relevantes de cada problema, como título, nível de dificuldade, descrição do problema e o código fornecido inicialmente pela plataforma.

4.2. Etapa preliminar: definição dos modelos a serem utilizados

Antes da execução principal dos experimentos, foi conduzida uma etapa preliminar com o objetivo de definir quais versões das LLMs seriam empregadas. Nessa fase, compararam-se os modelos Gemini 2.5 Flash-Lite e Gemini 2.5 Flash. Essa comparação inicial seguiu o mesmo procedimento metodológico descrito nas seções seguintes, incluindo a geração de código, submissão automática das soluções no LeetCode, análise de assertividade e desempenho, e avaliação estática do código com o SonarQube. O propósito dessa etapa não foi realizar uma análise comparativa de desempenho entre diferentes LLMs, mas sim verificar se a versão *Lite*, de menor custo, poderia substituir a versão completa nas demais fases do estudo.

4.3. Geração de código por LLMs

Na etapa de geração de código, cada enunciado de problema foi submetido a três modelos de linguagem de grande porte: Gemini 2.5 Flash, ChatGPT-5 e DeepSeek-V3.2, desenvolvidos e mantidos pelas empresas Google, OpenAI e High-Flyer AI, respectivamente. A comunicação com esses modelos e a obtenção das respostas ocorreram por meio de suas APIs oficiais, que possibilitam a interação programática e padronizada com diferentes LLMs, facilitando o processo de automação e controle dos experimentos.

Para assegurar a consistência e a comparabilidade entre os resultados, todas as solicitações foram feitas de forma explícita na linguagem C# por sua ampla aplicação prática e pela carência de estudos envolvendo essa linguagem em pesquisas com modelos de linguagem. Essa padronização visa eliminar variáveis externas decorrentes de diferenças sintáticas ou semânticas entre linguagens, permitindo que a análise se concentrasse apenas nas características dos problemas e nas estratégias de resolução adotadas pelos modelos.

Os códigos gerados pelas LLMs foram armazenados localmente para garantir sua integridade antes da análise e submissão na plataforma LeetCode, responsável por avaliar a assertividade, o desempenho e a qualidade das soluções produzidas. Para padronizar as respostas e assegurar a comparabilidade entre os resultados, utilizou-se um *prompt* padronizado (Apêndice A) com instruções rígidas sobre linguagem, estrutura e formato de saída.

4.4. Submissão no LeetCode

A etapa de submissão no LeetCode das respostas geradas foi realizada de forma automática por meio de um *script* desenvolvido em Python. Este *script* é responsável pela

[2]Repositório no GitHub – Conjunto de Problemas

autenticação na plataforma, gerenciando de forma segura o token de sessão necessário para o acesso à API do LeetCode. Para cada solução gerada, o *script* realiza o envio do código, associando-o ao problema correspondente e aguardando a resposta da plataforma quanto à validação dos testes. O processo de submissão inclui o controle automático de estados de autenticação, como a renovação de tokens expirados, além do tratamento de possíveis erros de comunicação ou falhas na submissão. Com isso, garante-se que todas as soluções sejam enviadas de maneira confiável e eficiente. O *script* de submissão de soluções[3] também permite a coleta estruturada dos resultados fornecidos pela plataforma, incluindo relatório de testes, tempo de execução, uso de memória e percentual de acertos, assegurando a consistência dos dados para a etapa de avaliação subsequente.

4.5. Análise estática do código gerado

A análise da qualidade do código é essencial para compreender o impacto das soluções geradas por LLMs em um contexto de engenharia de software. Ferramentas de análise estática, como o SonarQube, são amplamente utilizadas para avaliar atributos como manutenibilidade, segurança e confiabilidade, identificando automaticamente bugs, vulnerabilidades e code smells. Estudos mostram que esse tipo de prática contribui para reduzir custos de manutenção e mitigar riscos em sistemas de grande escala [Tornhill and Borg 2022]. Além disso, o SonarQube oferece suporte nativo e altamente consolidado para C#, com regras específicas para o ecossistema .NET, o que o torna particularmente adequado para avaliar o código gerado neste estudo.

A avaliação qualitativa foi conduzida por meio da análise estática do código utilizando o SonarQube, complementando as métricas de assertividade e desempenho. A análise concentrou-se na identificação de vulnerabilidades, code smells, bugs, complexidade ciclomática e aderência a padrões de codificação. Essa etapa foi conduzida em paralelo com a coleta dos dados de assertividade e desempenho, proporcionando uma visão integrada da qualidade do código gerado pelas LLMs.

4.6. Geração de relatórios e coleta de métricas

Para a etapa de geração dos relatórios, todos os dados obtidos incluindo as taxas de aceitação, o desempenho das submissões e os resultados das análises realizadas foram consolidados em relatórios automatizados. A linguagem Python foi empregada como ferramenta principal, devido à sua ampla adoção em atividades de análise e visualização de dados, bem como pela disponibilidade de bibliotecas robustas para manipulação e apresentação de resultados.

Entre as bibliotecas utilizadas, destaca-se a *pandas*[4], responsável pela organização, manipulação e estruturação dos dados em formato tabular. Para a geração das visualizações, empregou-se a *matplotlib*[5], escolhida por sua versatilidade na criação de gráficos estáticos. Além disso, a biblioteca *NumPy*[6] foi utilizada para o processamento numérico. Os relatórios produzidos incluíram tabelas e gráficos comparativos entre os diferentes modelos de linguagem analisados, possibilitando uma avaliação detalhada e objetiva do desempenho geral dos sistemas na geração de código em C#.

[3]Repositório no GitHub – Script de Submissão
[4]`https://pandas.pydata.org/`
[5]`https://matplotlib.org/`
[6]`https://numpy.org/`

5. Resultados

Nesta seção, são apresentados os principais resultados obtidos a partir das análises quantitativas realizadas sobre os problemas coletados da plataforma LeetCode. A subseção 5.1 aborda a caracterização do conjunto de dados utilizado neste estudo, enquanto as subseções 5.2, 5.3 e 5.4 apresentam, respectivamente, a comparação entre as versões do modelo Gemini, a análise de desempenho geral dos modelos avaliados e a avaliação da qualidade estrutural do código por meio de análise estática.

5.1. Caracterização do conjunto de dados

A base de dados é composta por um total de 1.000 problemas extraídos da plataforma LeetCode, dos quais 500 são classificados como de dificuldade "fácil", 250 como "média" e 250 como "difícil". Essa distribuição visa garantir uma representatividade equilibrada entre os diferentes níveis de complexidade, possibilitando uma análise comparativa mais robusta.

Apesar da variação no número de submissões, a taxa média de aceitação manteve-se superior a 50% em todos os níveis de dificuldade. Essa taxa representa a proporção de submissões que foram aprovadas nos testes oficiais do LeetCode, indicando quantas soluções enviadas pelos usuários passaram em todos os casos de teste. Contudo, os problemas classificados como "difíceis" apresentaram a menor taxa de aceitação entre as três categorias, resultado que pode ser associado à maior complexidade algorítmica e à necessidade de soluções mais elaboradas para sua resolução. A Figura 2 apresenta a taxa média de aceitação por dificuldade, enquanto a Figura 3 complementa a análise ao mostrar a mediana de submissões associada a cada categoria.

A Figura 4 mostra a média do número de submissões por nível de dificuldade da questão. A figura demonstra uma tendência decrescente no número médio de submissões à medida que a dificuldade dos problemas aumenta. Esse comportamento sugere que os usuários tendem a submeter mais soluções para problemas fáceis, possivelmente devido à maior acessibilidade desses desafios, tanto para iniciantes quanto para usuários que buscam prática rápida.

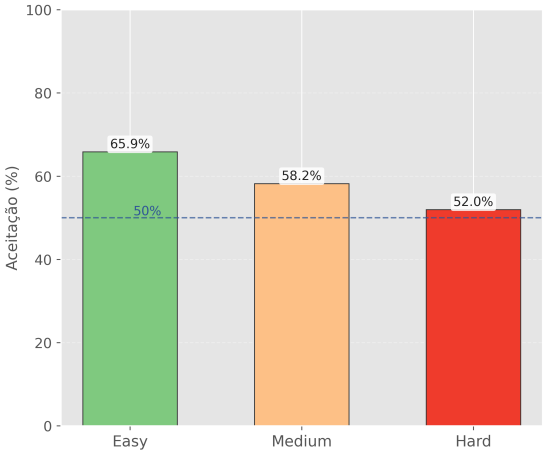

Figura 2. Taxa média de aceitação por dificuldade

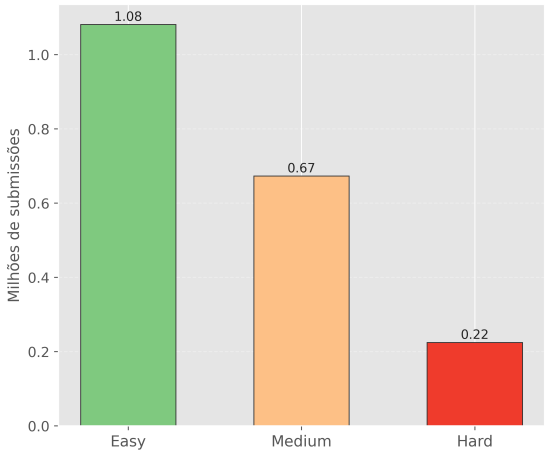

Figura 3. Mediana de submissões por dificuldade

Esses dados reforçam a importância de uma distribuição equilibrada dos problemas por nível de dificuldade ao construir conjuntos de dados para avaliação de modelos de linguagem. Caso contrário, a predominância de problemas fáceis poderia enviesar os resultados, levando a uma superestimação da capacidade dos modelos. Esse cuidado é essencial para garantir uma avaliação justa e representativa da real performance das LLMs em tarefas de resolução algorítmica.

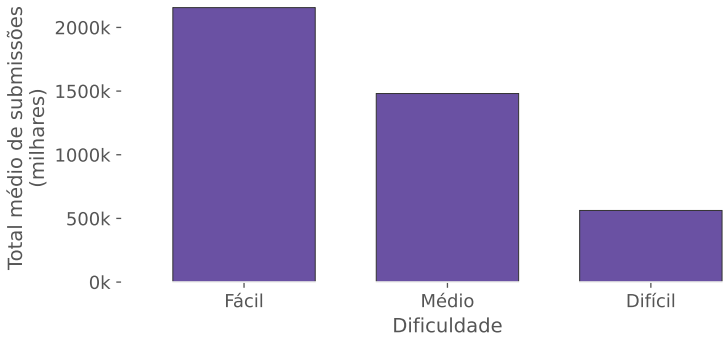

Figura 4. Taxa de submissões vs dificuldade

5.2. Comparação entre os modelos Gemini-2.5-Flash e Gemini 2.5 Flash-Lite

Para a definição sobre o uso das versões mini/lite ou convencionais dos modelos de linguagem, foi conduzido um experimento comparativo entre o Gemini-2.5-Flash e o Gemini-2.5-Flash-Lite. O experimento seguiu integralmente a metodologia previamente descrita, sendo utilizado como referência para a escolha das demais LLMs empregadas neste estudo.

Os resultados indicaram que a versão convencional do Gemini alcançou uma taxa de aceitação de 91,3%, enquanto a versão Lite apresentou 82,4%, revelando uma diferença de 10,8% em favor do modelo completo. Além da diferença na taxa de aceitação, o Gemini-2.5-Flash demonstrou desempenho superior em estabilidade e eficiência. O tempo médio de execução dos códigos avaliados pela plataforma LeetCode foi de 46,90 ms para o Gemini-2.5-Flash e de 55,27 ms para o Gemini-2.5-Flash-Lite. O uso médio de memória também se manteve próximo entre os modelos, registrando 54,20 MB para o Gemini e 52,46 MB para o Gemini-Lite. Quanto à estrutura das soluções geradas, o Gemini apresentou complexidade média de 7,01 e 27,42 linhas de código, enquanto o Lite obteve médias de 6,61 e 26,94 linhas.

Em relação à qualidade do código, ambos os modelos apresentaram índices baixos de falhas. O Gemini registrou 11 bugs e 979 code smells, enquanto o Gemini-Lite apresentou 11 bugs e 861 code smells. Apesar da proximidade desses valores, o número mais elevado de erros de compilação no modelo Lite (79, em comparação a 13 no modelo completo) comprometeu parte das análises, reduzindo a consistência dos resultados obtidos pelo SonarQube. Considerando a diferença relativa de 10,8% em favor do Gemini-2.5-Flash, somada ao seu desempenho mais estável em termos de tempo de execução, compilação e qualidade estrutural, optou-se pela utilização das versões completas (não Lite/Mini) dos demais modelos, de modo a assegurar maior robustez e consistência aos resultados experimentais.

5.3. Comparação geral

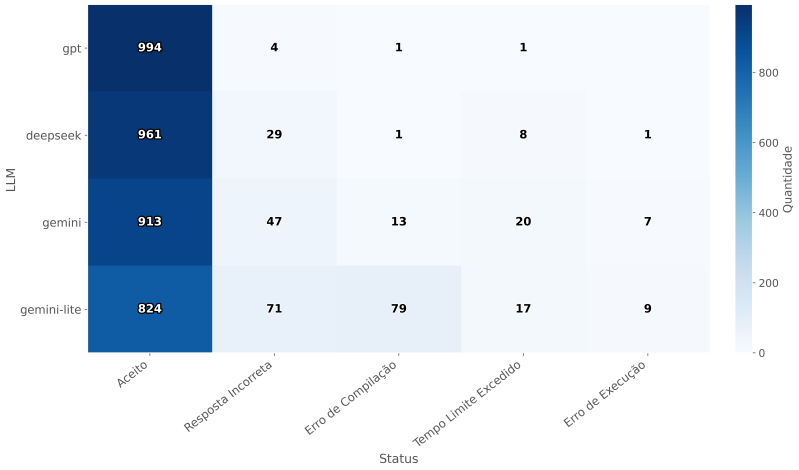

Figura 5. Heatmap de status por submissão

Diante dos resultados obtidos na comparação inicial entre as versões do Gemini, ampliou-se o escopo dos experimentos para incluir outros modelos de linguagem. Esta subseção apresenta o desempenho consolidado dos quatro modelos avaliado, Gemini-2.5-Flash, Gemini-2.5-Flash-Lite, ChatGPT-5 e DeepSeek-V3.2, todos submetidos aos mesmos 1000 enunciados de problemas. A análise abrange aspectos de assertividade, eficiência, estabilidade e qualidade do código gerado. A distribuição completa dos resultados pode ser visualizada na Figura 5.

Em relação à taxa de aceitação, observou-se que o ChatGPT-5 obteve o melhor resultado, com 994 soluções aceitas (99,4%), seguido pelo DeepSeek com 961 (96,1%), o Gemini-2.5-Flash com 913 (91,3%) e o Gemini-2.5-Flash-Lite com 824 (82,4%). Esses valores indicam uma diferença relativa 20,63% entre o melhor e o pior desempenho, demonstrando que modelos de maior capacidade apresentam resultados substancialmente superiores na geração de código em C#.

A análise dos tipos de erro revelou comportamentos distintos entre os modelos. O Gemini-2.5-Flash-Lite apresentou 79 erros de compilação, 71 respostas incorretas, 17 casos de *Time Limit Exceeded* e 9 erros de execução. O Gemini-2.5-Flash registrou 13 erros de compilação, 47 respostas incorretas, 20 casos de limite de tempo e 7 erros de execução. Já o DeepSeek apresentou apenas 1 erro de compilação, 29 respostas incorretas, 8 limites de tempo e 1 erro de execução, enquanto o ChatGPT-5 manteve o melhor desempenho também neste aspecto, com apenas 1 erro de compilação, 4 respostas incorretas, 1 limite de tempo e nenhuma falha de execução. A grande quantidade de erros de compilação no modelo Lite impactou negativamente sua taxa de sucesso e a completude da análise de qualidade, uma vez que o SonarQube não pôde processar os códigos que não foram compilados com sucesso.

No que se refere ao desempenho, observa-se na Tabela 1 que os tempos médios de execução dos códigos gerados variaram de 33,53 ms a 55,27 ms. O modelo GPT-5 apresentou o melhor desempenho, com o menor tempo médio de execução (33,53 ms), seguido pelo DeepSeek (43,07 ms), Gemini 2.5 Flash (46,90 ms) e Gemini 2.5 Flash-Lite

(55,27 ms). Embora as diferenças sejam relativamente pequenas, nota-se uma tendência de maior eficiência nos modelos completos em relação às versões Lite. O uso médio de memória manteve-se estável entre os modelos, oscilando entre 52,46 MB (Gemini-Lite) e 54,75 MB (GPT-5), o que indica ausência de variações significativas nesse aspecto.

Tabela 1. Desempenho — Tempo e Memória (ordenado por tempo médio de execução)

Modelo	Tempo médio (ms)	Memória média (MB)
GPT-5	**33,53**	54,75
DeepSeek	43,07	54,59
Gemini 2.5 Flash	46,90	**54,20**
Gemini 2.5 Flash-Lite	55,27	52,46

5.4. Análise Estática

A análise estrutural do código mostrou que o ChatGPT-5 produziu soluções mais complexas, com média de complexidade 8,31 e 26,52 linhas de código. O Gemini-2.5-Flash apresentou valores médios de 7,01 de complexidade e 27,42 linhas, seguido pelo DeepSeek com 6,96 e 24,91 linhas, e o Gemini-Lite com 6,61 e 26,94 linhas.

Quanto aos indicadores de qualidade, todos os modelos apresentaram baixos índices de vulnerabilidades e *security hotspots*. O DeepSeek e o ChatGPT-5 registraram quatro bugs cada, o Gemini-2.5-Flash e o Gemini-Lite onze cada. Em relação aos *code smells*, os valores foram 979 para o Gemini, 923 para o ChatGPT-5, 861 para o Gemini-Lite e 798 para o DeepSeek, revelando pequenas variações entre os modelos. Essa diferença pode estar associada à diversidade estrutural das soluções geradas e à taxa de códigos efetivamente analisados pelo SonarQube, já que falhas de compilação podem ter reduzido o número de arquivos avaliados em alguns casos. A Figura 6 apresenta, em escala logarítmica, a distribuição de *bugs* e *code smells* por modelo, evidenciando que o número de alertas reflete mais a completude das análises do que uma diferença significativa de qualidade entre os modelos.

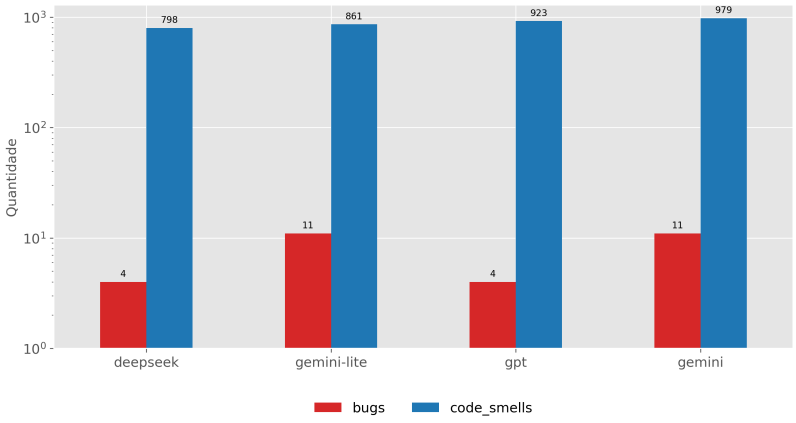

Figura 6. Quantidade de bugs e *code smells* por LLM em escala logarítmica

De forma geral, os resultados indicam que o ChatGPT-5 apresentou o desempenho mais consistente, combinando alta taxa de aceitação, eficiência de execução e estabi-

lidade de compilação. O DeepSeek demonstrou desempenho semelhante, com resultados sólidos e baixa taxa de erros. O Gemini-2.5-Flash obteve desempenho intermediário, com boa taxa de sucesso, porém maior ocorrência de erros de tempo e compilação. Já o Gemini-2.5-Flash-Lite apresentou o desempenho mais limitado, tanto em precisão quanto em estabilidade, sendo fortemente impactado por falhas de compilação e menor robustez das soluções geradas.

6. Ameaças a validade

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

No que diz respeito à validade externa, os resultados obtidos neste estudo podem não ser plenamente generalizáveis para outros contextos, como problemas de algoritmos fora da plataforma LeetCode ou em diferentes linguagens de programação. O uso de um conjunto limitado de questões pode fazer com que as conclusões reflitam mais o comportamento das LLMs em relação ao formato e ao padrão específico de desafios da plataforma do que um desempenho geral em tarefas de programação. Para mitigar essa limitação, foi selecionada uma amostra diversificada e balanceada de problemas, distribuída entre os níveis de dificuldade fácil, médio e difícil, buscando representar uma gama mais ampla de estruturas algorítmicas e minimizar potenciais vieses de seleção.

A escolha de apenas três modelos: ChatGPT-5, Gemini 2.5 Flash e DeepSeek V3.2, também restringe a generalização dos achados, uma vez que outros modelos com arquiteturas, tamanhos e bases de treinamento distintas poderiam apresentar comportamentos diferentes. Essa decisão, no entanto, foi necessária para garantir a viabilidade do estudo e permitir uma análise mais aprofundada de cada modelo. Buscou-se mitigar essa limitação selecionando LLMs de origens e arquiteturas diversas, representando diferentes abordagens de geração de código e assegurando maior diversidade entre os resultados.

Por fim, a validade de construto pode ter sido afetada por possíveis vieses de aferição, especialmente na análise qualitativa dos códigos e na categorização das falhas. Para mitigar essa limitação, todas as avaliações foram conduzidas com base em métricas objetivas e padronizadas, extraídas de ferramentas automatizadas, como o SonarQube e os relatórios do LeetCode, o que assegurou maior consistência e reprodutibilidade dos resultados. No entanto, reconhece-se que o uso de problemas do LeetCode como instrumento de medição não abrange plenamente todos os aspectos da qualidade de software, como manutenibilidade e arquitetura, concentrando-se predominantemente em características algorítmicas.

7. Conclusão

Neste trabalho, foi realizada uma análise comparativa entre códigos gerados por três modelos de LLMs, com o objetivo de avaliar diferenças de desempenho, assertividade e qualidade do código produzido em C#. Os resultados evidenciam variações relevantes entre os modelos analisados. O ChatGPT-5 apresentou o melhor desempenho geral, com maior taxa de aceitação (99,4%), menor tempo médio de execução (33,53 ms) e baixa incidência de erros, demonstrando maior estabilidade e maturidade na geração de código. O DeepSeek V3.2 também obteve resultados consistentes, com alta taxa de sucesso e poucas falhas de compilação, configurando-se como uma alternativa competitiva.

Já o Gemini-2.5-Flash apresentou desempenho intermediário, mantendo bons índices de acerto e eficiência, mas com um número ligeiramente superior de falhas e maior tempo de execução em relação aos modelos de ponta. Por fim, o Gemini-2.5-Flash-Lite apresentou desempenho mais limitado, afetado principalmente por falhas de compilação e menor consistência nas soluções geradas.

A análise estrutural e de qualidade do código indicou que, embora todos os modelos apresentem resultados satisfatórios em termos de ausência de vulnerabilidades e baixa incidência de bugs, há diferenças sutis na complexidade e na manutenibilidade das soluções produzidas. O ChatGPT-5 e o DeepSeek, por exemplo, geraram códigos mais enxutos e estáveis, enquanto o Gemini apresentou soluções mais extensas e com maior quantidade de *code smells*. Esses resultados sugerem que modelos de maior capacidade e treinamento mais abrangente tendem a produzir código não apenas mais funcional, mas também mais próximo de boas práticas de engenharia de software.

De forma geral, o estudo demonstra que o uso de LLMs para resolução de problemas algorítmicos em C# é viável e pode gerar resultados de alta qualidade, especialmente quando se empregam modelos de última geração. No entanto, observou-se que o desempenho varia significativamente conforme o modelo e a complexidade dos problemas, o que reforça a importância de uma avaliação cuidadosa e contextualizada na adoção dessas ferramentas em ambientes de desenvolvimento reais.

Como trabalhos futuros, propõe-se expandir a análise para outras linguagens de programação, como Java e Kotlin, e ampliar o conjunto de problemas para além do LeetCode, de modo a reduzir vieses e aumentar a representatividade dos cenários avaliados. Também se recomenda incorporar métricas mais abrangentes de qualidade de software, incluindo manutenibilidade e aspectos arquiteturais, bem como testar um conjunto maior de LLMs, contemplando modelos adicionais ou emergentes, como Claude e Mistral. Por fim, a replicação do estudo em diferentes plataformas e momentos poderá mitigar as ameaças à validade identificadas, permitindo avaliar o impacto da evolução contínua das LLMs e fortalecendo a robustez das conclusões.

Pacote de Replicação

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

`https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-diego-machado#`

Referências

[Achiam et al. 2023] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

[Bourque et al. 2024] Bourque, P., Fairley, R. E., and Society, I. C. (2024). *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 4.0*. IEEE Computer Society Press, Washington, DC, USA, 4rd edition.

[Chen et al. 2021] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[Clark et al. 2024] Clark, A., Igbokwe, D., Ross, S., and Zibran, M. F. (2024). A quantitative analysis of quality and consistency in ai-generated code. In *2024 7th International Conference on Software and System Engineering (ICoSSE)*, pages 37–41.

[Coelho 2024] Coelho, B. A. C. (2024). Análise da qualidade e eficácia do código gerado por llms: Um estudo com problemas da plataforma leetcode. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) – Pontifícia Universidade Católica de Minas Gerais (PUC Minas).

[Coignion et al. 2024] 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.

[Cormen et al. 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

[Cui et al. 2024] 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 *2024 36th International Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10.

[Eltabakh et al. 2024] Eltabakh, T. M., Nabil Soudi, N., and Shawky, D. (2024). Quality of ai-generated vs. human-generated code. In *2024 34th International Conference on Computer Theory and Applications (ICCTA)*, pages 200–205.

[Fawareh et al. 2024] Fawareh, H., Al-Shdaifat, H. M., M, A.-R., Fawareh, F. A., and Khouj, M. (2024). Investigates the impact of ai-generated code tools on software readability code quality factor. In *2024 25th International Arab Conference on Information Technology (ACIT)*, pages 1–5.

[Hennessy and Patterson 2011] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[ISO 2001] ISO (2001). Iso/iec 9126-1:2001 — software engineering — product quality — part 1: Quality model. ISO/IEC, Geneva, Switzerland.

[ISO 2011] ISO (2011). Iso/iec 25010:2011 — systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. ISO/IEC, Geneva, Switzerland.

[Jalaman and Teleron 2024] Jalaman, J. R. C. and Teleron, J. I. (2024). Optimizing operating system performance through advanced memory management techniques: A comprehensive study and implementation. *Engineering and Technology Journal*, 9(5):4137–4143.

[Jiang et al. 2024] Jiang, J., Wang, F., Shen, J., Kim, S., and Kim, S. (2024). A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

[Kazemitabaar et al. 2023] Kazemitabaar, M., Chow, J., Ma, C. K. T., Ericson, B. J., Weintrop, D., and Grossman, T. (2023). Studying the effect of ai code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA. Association for Computing Machinery.

[Li et al. 2021] Li, X., Ma, Z., Tu, Y., and Du, Y. (2021). Study on the application of artificial intelligence technology in empowering education : Taking "intelligent learning partner"as an example. In *2021 2nd International Conference on Information Science and Education (ICISE-IE)*, pages 1449–1453.

[Nanz and Furia 2015] Nanz, S. and Furia, C. A. (2015). A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788.

[Negri-Ribalta et al. 2024] Negri-Ribalta, C., Geraud-Stewart, R., Sergeeva, A., and Lenzini, G. (2024). A systematic literature review on the impact of ai models on the security of code generation. *Frontiers in Big Data*, 7:1386720.

[Nguyen and Nadi 2022] 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.

[Nikolaidis et al. 2024] Nikolaidis, N., Flamos, K., Gulati, K., Feitosa, D., Ampatzoglou, A., and Chatzigeorgiou, A. (2024). A comparison of the effectiveness of chatgpt and co-pilot for generating quality python code solutions. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pages 93–101.

[Niu et al. 2024] Niu, C., Zhang, T., Li, C., Luo, B., and Ng, V. (2024). On evaluating the efficiency of source code generated by llms. In *2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (Forge) Conference Acronym:*, pages 103–107.

[Patel et al. 2024] Patel, A., Sultana, K. Z., and Samanthula, B. K. (2024). A comparative analysis between ai generated code and human written code: A preliminary study. In *2024 IEEE International Conference on Big Data (BigData)*, pages 7521–7529.

[Poldrack et al. 2023] Poldrack, R. A., Lu, T., and Beguš, G. (2023). Ai-assisted coding: Experiments with gpt-4. *arXiv preprint arXiv:2304.13187*.

[Roziere et al. 2023] Roziere, B., Nguyen, A., Damoc, B., Izacard, G., Rambur, C., Li, S., Tunstall, L., Xu, C., Azhar, F. H., Simonyan, K., et al. (2023). Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

[Samuel et al. 2022] Samuel, J., Kashyap, R., Samuel, Y., and Pelaez, A. (2022). Adaptive cognitive fit: Artificial intelligence augmented management of information facets and representations. *International journal of information management*, 65:102505.

[Shean et al. 2025] Shean, R., Shah, T., Pandiarajan, A., Tang, A., Bolo, K., Nguyen, V., and Xu, B. (2025). A comparative analysis of deepseek r1, deepseek-r1-lite, openai o1 pro, and grok 3 performance on ophthalmology board-style questions. *Scientific Reports*, 15(1):23101.

[Shyamal et al. 2023] Shyamal, D., Asanka, P., and Wickramaarachchi, D. (2023). A comprehensive approach to evaluating software code quality through a flexible quality model. In *2023 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, volume 6, pages 1–8.

[Tornhill and Borg 2022] Tornhill, A. and Borg, M. (2022). Code red: the business impact of code quality - a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*, TechDebt '22, page 11–20, New York, NY, USA. Association for Computing Machinery.

[Vaswani et al. 2017] 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.

[Wahlgren et al. 2022] Wahlgren, J., Gokhale, M., and Peng, I. B. (2022). Evaluating emerging cxl-enabled memory pooling for hpc systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 11–20.

[Wang et al. 2024] Wang, L., Shi, C., Du, S., Tao, Y., Shen, Y., Zheng, H., and Qiu, X. (2024). Performance review on llm for solving leetcode problems. In *2024 4th International Symposium on Artificial Intelligence and Intelligent Manufacturing (AIIM)*, pages 1050–1054. IEEE.

[Wohlin et al. 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., Wesslén, A., et al. (2012). *Experimentation in software engineering*, volume 236. Springer.

Apêndice

A. Prompt Utilizado para Geração das Respostas das LLMs

Prompt 1: Prompt for LLMs

[description problem from LeetCode platform]
STRICT INSTRUCTIONS:
- Use c# only.
- You MUST keep and return the full class 'Solution' and the full method exactly as provided.
- Insert the solution ONLY inside the method body.
- Do not change the class name, method name, or method signature.
- Do not add or remove braces outside the method body.
- Do not add 'using' statements or imports.
- Do not add comments, explanations, or extra text.
- Do not wrap the answer in Markdown or backticks.
- Output must be valid, compilable c# code only.
- Output must be the entire updated Code Snippet exactly as given, not just the body.
Code Snippet: [code snippet]