Análise Comparativa de Código Gerado por IA: ChatGPT, Gemini e Claude em Métricas de Qualidade de Software

Gustavo H. dos S. RIEGERT[1]

[1]Instituto de Ciências exatas e Informática – Pontifícia Universidade Católica de Minas Gerais (PUC MG)
Belo Horizonte – MG – Brazil

ghsriegert@sga.pucminas.br

Abstract. *With the growing adoption of Large Language Models (LLMs) in software development, it becomes essential to evaluate not only functional correctness but also the structural quality of the generated code. This study compares Java solutions produced by the models ChatGPT, Gemini, and Claude, using 527 LeetCode problems and automated analysis via SonarQube. The evaluation considers maintainability (cyclomatic complexity and code smells), assertiveness (accuracy rate, attempts until acceptance, and difficulty levels solved), and operational performance (execution time and memory usage). The results reveal distinct profiles among the models. Gemini exhibited higher assertiveness and better temporal performance. Claude produced simpler and more consistent code. ChatGPT achieved the highest problem coverage and demonstrated better memory efficiency, although it required greater maintenance effort. These findings offer practical guidelines for choosing LLMs as programming assistants, highlighting the trade-offs between maintainability, correctness, and performance.*

Resumo. *Com a crescente adoção de Large Language Models (LLMs) no desenvolvimento de software, torna-se essencial avaliar não apenas a corretude funcional, mas também a qualidade estrutural do código gerado. Este estudo compara soluções Java produzidas pelos modelos ChatGPT, Gemini e Claude, utilizando 527 problemas do LeetCode e análise automatizada via SonarQube. A avaliação considera manutenibilidade (complexidade ciclomática e code smells), assertividade (taxa de acertos, tentativas até aceitação e níveis de dificuldade resolvidos) e desempenho operacional (tempo de execução e uso de memória). Os resultados evidenciam perfis distintos entre os modelos. O Gemini apresentou maior assertividade e melhor desempenho temporal. O Claude produziu códigos mais simples e consistentes. O ChatGPT alcançou a maior cobertura de problemas e demonstrou melhor eficiência de memória, embora exigisse maior esforço de manutenção. Esses achados oferecem diretrizes práticas para a escolha de LLMs como assistentes de programação, destacando os trade-offs entre manutenibilidade, corretude e desempenho.*

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

Orientador de conteúdo (TCC I): Joana Souza - joanasouza@pucminas.br
Orientador de conteúdo (TCC I): Leonardo Vilela - leonardocardoso@pucminas.br
Orientador de conteúdo (TCC I): João Batisteli - joaobatisteli@pucminas.br
Orientador acadêmico (TCC I): Cleiton Tavares - cleitontavares@pucminas.br
Orientador do TCC II: João Batisteli - joaobatisteli@pucminas.br

Belo Horizonte, 20 de Dezembro de 2025.

1. Introdução

A popularização dos *Large Language Models* (LLMs) vem mudando a prática do desenvolvimento de software, sendo usados como assistentes na geração de *snippets* de códigos ou até soluções mais completas [Svyatkovskiy et al. 2020]. Segundo estatísticas do GitHub em 2022, 40% do código em projetos na plataforma foi escrito por Inteligência Artificial (IA), com ganhos de até 55% na produtividade [Rosenbaum 2022]. Contudo, embora modelos como GPT-4 contribuam para o rendimento, estudos indicam que desenvolvedores que o utilizam são mais propensos a introduzir vulnerabilidades de segurança ao código [Perry et al. 2023]. Além disso, a avaliação do código gerado por IA contém métricas como alta complexidade ciclomática e baixa legibilidade, situação que resultou em apenas 26% dos códigos estarem corretos na primeira tentativa, o que sugere um retrabalho frequente [Jain et al. 2024]. Nesse contexto, destaca-se a necessidade de análises comparativas sobre a qualidade de LLMs no mercado para orientar os desenvolvedores na escolha da ferramenta mais adequada a um contexto.

Com base no cenário atual da geração automática de código, observa-se que as avaliações disponíveis tendem a priorizar critérios como correção funcional e segurança, deixando em segundo plano outros indicadores relevantes de qualidade de software [Vaithilingam et al. 2022, Bae et al. 2024]. Além disso, ainda são raras análises que comparem, de forma conjunta, os principais LLMs do mercado na tarefa de gerar código Java, linguagem amplamente utilizada segundo dados do Stack Overflow (2024). Outro ponto importante é que o ganho de eficiência proporcionado por esses modelos varia conforme o nível de conhecimento do desenvolvedor [Osorio et al. 2025]. Diante desse contexto, o problema central deste estudo consiste na **falta de dados quantitativos sobre o esforço de retrabalho, como o processo iterativo de refinamento das respostas geradas pelos LLMs, envolvendo reformulações de *prompts*, correções de código e novas submissões, até a obtenção de uma solução funcional aceita, em problemas Java de diferentes níveis de complexidade.** Assim, torna-se essencial investigar qual modelo oferece o melhor equilíbrio entre produtividade e qualidade em múltiplos contextos de codificação.

A adoção crescente de LLMs no desenvolvimento de software tem sido acompanhada por desafios relacionados à qualidade do código gerado automaticamente [Chen et al. 2021]. Pesquisas anteriores indicam que aspectos como a formulação do *prompt*, a ocorrência de vulnerabilidades e a eficiência algorítmica apresentam variações significativas entre diferentes modelos [Liu et al. 2024, Niu et al. 2024], evidenciando a complexidade envolvida no uso dessas ferramentas em cenários reais de programação.

Além disso, estudos mostram que essas variações não se restringem à corretude funcional, os modelos podem gerar códigos com diferentes níveis de manutenibilidade, estabilidade estrutural e necessidade de retrabalho, incluindo problemas como *code smells*, falhas de compilação e ineficiências que impactam diretamente a produtividade do desenvolvedor [Jesse et al. 2023]. Assim, compreender como diferentes LLMs se comportam diante de múltiplos critérios de qualidade torna-se essencial para orientar seu uso em ambientes práticos de desenvolvimento.

Este estudo tem como objetivo geral **comparar a qualidade do código Java gerado por LLMs como ChatGPT, Gemini e Claude em relação a problemas do LeetCode**, com ênfase em métricas de qualidade de software como desempenho, eficácia e manutenabilidade. Os objetivos específicos deste estudo incluem: (i) a seleção de um conjunto representativo de problemas do LeetCode, abarcando diferentes níveis de dificuldade e tipos de raciocínio algorítmico, a fim de estabelecer um cenário padronizado para a avaliação dos modelos; (ii) a geração de soluções em Java utilizando diferentes LLMs, registrando todo o processo de interação, desde as tentativas iniciais até as reformulações de *prompts* e correções necessárias; e (iii) a avaliação da qualidade e do desempenho das respostas produzidas, considerando a eficiência algorítmica, expressa pelo tempo de execução e uso de memória, a manutenibilidade do código, por meio de métricas como complexidade ciclomática e ocorrência de *code smells*, e a produtividade efetiva dos modelos, medida pelo número de interações e pelo tempo requerido para alcançar uma solução funcional.

Os resultados obtidos permitiram quantificar a relação entre produtividade imediata e os custos ocultos, identificando qual modelo oferece o melhor equilíbrio para diferentes contextos, além de mapear inconsistências que contribuem para os custos como retrabalho e déficit de manutenibilidade, correlacionando-os com métricas de qualidade. Essas conclusões fundamentarão diretrizes práticas para seleção de LLMs, subsidiando decisões informadas e reduzindo a adoção acrítica dessas ferramentas.

O restante do trabalho está organizado da seguinte maneira. A seção 2 apresenta a fundamentação teórica, com conceitos sobre LLMs e métricas de qualidade. Em seguida, a seção 3 discute sobre os trabalhos relacionados, analisando pesquisas que convergem sobre a área. A seção 4 descreve a metodologia, à qual detalha a seleção de problemas do LeetCode, configuração dos LLMs e critérios de avaliação. Na seção 5, os resultados são apresentados, trazendo um comparativo entre o desempenho dos modelos, seguido pela conclusão, que oferece recomendações e perspectivas futuras.

2. Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais que sustentam a proposta deste trabalho, com foco na geração de código por LLMs, nas métricas de avaliação da qualidade de software e nos critérios de desempenho associados.

2.1. Large Language Models (LLMs) na Geração de Código

Os LLMs utilizam a arquitetura de *transformers* [Vaswani et al. 2017], cujo mecanismo de atenção permite ao modelo ponderar cada *token* de entrada em relação a todos os demais, capturando dependências de longo alcance no texto ou no código. No pré-treinamento, esses LLMs processam grandes volumes de código-fonte e, em seguida,

são submetidos ao *fine-tuning* em bases específicas de programação, o que refina seus parâmetros para tarefas como geração de *snippets*, refatoração e correção de *bugs*. Esse processo de pré-treinamento em larga escala, aliado ao ajuste fino supervisionado, impacta diretamente aspectos como legibilidade, eficiência e segurança do código gerado, devido à natureza dos dados de treinamento e às estratégias de geração adotadas pelos modelos [Liu et al. 2024].

Além disso, o comportamento dos LLMs na geração de código está diretamente relacionado aos padrões aprendidos durante o pré-treinamento. Como apontam Chen et al. (2021), modelos treinados em grandes volumes de código tendem a reproduzir tanto boas práticas quanto inconsistências presentes nesses repositórios, influenciando métricas de legibilidade, eficiência e qualidade estrutural. Estudos recentes mostram que características do conjunto de dados de treinamento impactam o desempenho operacional do código gerado, especialmente no tempo de execução e no uso de memória, aspectos ligados à complexidade assintótica [Niu et al. 2024, Cormen et al. 2001]. Assim, a qualidade do código produzido por LLMs depende não apenas da sua arquitetura, mas também da natureza e diversidade dos dados que compõem seu treinamento, afetando atributos clássicos da engenharia de software [Pressman and Maxim 2020].

2.2. Qualidade de Software e Métricas de Avaliação

A qualidade de software diz respeito à capacidade de um sistema atender tanto aos requisitos funcionais quanto aos não funcionais, podendo ser analisada por meio de métricas estáticas que avaliam o código sem a necessidade de execução [Pressman and Maxim 2020]. Essas métricas permitem identificar propriedades estruturais relevantes para a clareza, confiabilidade e evolução do software.

No que se refere ao desempenho, medidas de tempo e memória estão teoricamente associadas à complexidade assintótica dos algoritmos [Cormen et al. 2001]. No entanto, em ambientes práticos como o LeetCode, fatores adicionais, incluindo características do compilador, condições da plataforma e variações nos dados de entrada, podem influenciar significativamente os resultados observados, gerando desvios em relação ao comportamento previsto pela análise formal [Niu et al. 2024].

A manutenibilidade do código, por sua vez, pode ser analisada a partir de métricas estruturais como a complexidade ciclomática, que quantifica o número de possíveis caminhos de execução em um programa. Valores acima de 10 estão associados ao aumento da probabilidade de falhas e à dificuldade de evolução do código [Watson and McCabe 1996]. Além disso, padrões sintáticos indesejáveis, conhecidos como *code smells*, podem indicar a existência de dívidas técnicas que afetam a legibilidade, a clareza e a extensibilidade do software [Liu et al. 2024].

Assim, a combinação entre métricas estáticas, indicadores de eficiência algorítmica e mecanismos de detecção de *code smells* fornece uma base objetiva para analisar limitações estruturais e comportamentais do código. Essa abordagem é aplicável tanto na avaliação tradicional de software quanto na análise de soluções geradas automaticamente por LLMs, permitindo compreender de forma mais abrangente suas qualidades e limitações.

3. Trabalhos Relacionados

Pesquisas recentes têm aprofundado a compreensão dos diferentes fatores de qualidade do código gerado por modelos de LLMs, revelando custos ocultos que vão além da simples correção funcional. O trabalho de Liu et al. (2024) avalia 4066 soluções produzidas pelo ChatGPT (GPT-3.5-turbo) para 2033 problemas do LeetCode em Java e Python, no qual empregava métricas de correção e manutenibilidade. Embora 67% das soluções fossem funcionalmente corretas, 47% apresentavam graves problemas de manutenibilidade, como erros de compilação, saídas incorretas, inconsistências de estilo e ineficiências algorítmicas. Os autores demonstraram que, com *feedback* apropriado, o modelo foi capaz de "auto-reparar" até 60% das falhas identificadas. Este trabalho evidencia os custos associados ao uso de LLMs e inspira a abordagem, na qual amplia-se a análise para incluir três modelos distintos (ChatGPT, Gemini e Claude) e incorpora métricas quantitativas adicionais, como complexidade ciclomática e ocorrência de *code smells*, para mapear de forma abrangente as dimensões de qualidade do código gerado.

De maneira complementar, Jesse et al. (2023) investigaram a frequência de falhas simples, conhecidos como *Simple, Stupid Bugs* (SStuBs), em 16.899 trechos de código do *dataset* ManySStuBs4J. O estudo testa se modelos de LLMs como o Codex reproduziam variantes incorretas ou corretas de trechos de códigos já ajustados em repositórios reais. Os resultados indicaram que os modelos geravam até duas vezes mais *bugs* do que correções, mesmo quando o problema havia sido documentado e resolvido anteriormente. Além disso, a inclusão de comentários explicativos no *prompt* reduziu significativamente a taxa de geração de erros, ressaltando a importância do contexto fornecido ao LLM. Essa investigação complementa este trabalho ao demonstrar que LLMs podem introduzir falhas evitáveis em cenários simples, reforçando a necessidade de avaliar múltiplos fatores de qualidade para compreender sua real confiabilidade na geração de código.

Perry et al. (2023) exploram os encadeamentos de falhas na segurança do desenvolvimento de *software* com o uso de assistentes de IA, por meio de um experimento controlado com 54 desenvolvedores. Os autores dividiram o experimento em dois grupos: (1) auxiliado por IA; (2) sem a assistência de IA, os participantes tiveram suas soluções avaliadas pelo programa SpotBugs, uma ferramenta para detecção de vulnerabilidades em códigos Java. Embora o grupo assistido tenha apresentado 12% soluções mais funcionais, foram identificadas uma taxa de vulnerabilidade de segurança 29% maior, incluindo falhas críticas como *SQL injection*. Esses dados ressaltam que ganhos de produtividade obtidos pelos LLMs podem ocultar riscos que comprometam a segurança das aplicações. Assim, o estudo contribui para este trabalho ao evidenciar que a adoção de LLMs deve considerar não apenas corretude funcional, mas também impactos em segurança e confiabilidade do código gerado.

Em um estudo comparativo, Hamer et al. (2024) discorre sobre a segurança de trechos de código Java gerados pelo ChatGPT com respostas humanas extraídas do StackOverflow. Analisando 108 pares de soluções por meio da ferramenta CodeQL, os autores observaram que o ChatGPT produziu 20% menos vulnerabilidades totais (248 contra 302), porém apenas 25% das falhas coincidiam com aquelas encontradas em soluções do StackOverflow. Além disso, o modelo apresentou um conjunto menor de categorias de vulnerabilidades com 19 *Common Weakness Enumeration* (CWEs) contra 22 no StackOverflow, com predominância de falhas relacionadas à implementação criptográfica. Os

autores concluíram que, independentemente da fonte de dados obtida, o código deve passar por uma validação antes de ser utilizado em produção. Essa comparação reforça a hipótese de embora a adoção de LLMs reduza algumas vulnerabilidades do código, ainda exige uma verificação humana. Diante disso, impacta a eficiência de geração automática, especialmente quando avalia-se três modelos de forma integrada em problemas com variados níveis de complexidade.

Adicionalmente, Niu et al. (2024) investigaram a eficiência algorítmica de códigos *Python* gerados por diferentes LLMs utilizando três *benchmarks*: (1) HumanEval; (2)*Mostly Basic Python Problems* (MBPP) e (3) uma coleção de problemas do LeetCode; Foram comparados modelos como GPT-4, GPT-3.5, DeepSeek Coder, Code Llama e WizardCoder. Os resultados apontaram que a eficiência de execução não se correlaciona diretamente com a taxa de acerto funcional, embora o GPT-4 tenha obtido o melhor desempenho em Pass@10, o GPT-3.5 frequentemente produziu soluções com menor tempo de execução. Além disso, técnicas de *prompting*, como *chain-of-thought*, elevaram a eficiência do código em até 18% para problemas de maior complexidade. Este trabalho alinha-se à proposta, que também investiga eficiência, mas a estende ao incorporar métricas de manutenibilidade e esforço de refinamento para oferecer um panorama mais completo da qualidade do código.

Embora esses estudos tenham contribuído de forma significativa para elucidar aspectos isolados da qualidade de código gerado por IA, o presente trabalho diferencia-se ao propor uma análise integrada de múltiplas dimensões como funcionalidade, manutenibilidade e eficiência aplicada a três modelos de última geração (GPT-4, Gemini 2.5 Flash e Claude-3-5-haiku-20241022) em conjuntos de problemas com diferentes níveis de complexidade. Essa abordagem em conjunto permite não apenas identificar qual modelo entrega maior qualidade em termos absolutos, mas também compreender os *trade-offs* envolvidos e os cenários nos quais cada modelo se sobressai, orientando desenvolvedores e organizações na adoção consciente de LLMs como ferramentas de auxílio à programação.

4. Materiais e Métodos

Figura 1. Metodologia de cinco etapas para o desenvolvimento deste projeto.

Nesta seção são apresentados os materiais e métodos utilizados neste estudo, caracterizado como uma pesquisa quantitativa, de natureza comparativa e experimental. A abordagem quantitativa justifica-se pela coleta de métricas como tempo de processamento, consumo de memória e indicadores de qualidade de software pelos códigos gerados pelos modelos GPT-4, Gemini 2.5 Flash e Claude-3-5-haiku-20241022 ao resolver problemas algorítmicos. O viés experimental fundamentou-se na seleção uniforme dos problemas, a

padronização dos *prompts* utilizados e a aplicação dos mesmos procedimentos de geração e avaliação de código para todos os modelos, permitindo uma comparação equitativa dos LLMs.

A Figura 1 resume o fluxo metodológico adotado neste estudo, composto por cinco etapas principais. A primeira etapa consiste na definição do *dataset*, incluindo filtragem, seleção e balanceamento dos problemas, garantindo representatividade e padronização das entradas. A segunda etapa envolve a geração automática de código via APIs dos modelos, seguindo um protocolo uniforme de *prompts* para assegurar comparabilidade entre as LLMs. Em seguida, na terceira etapa, são aplicadas as métricas de avaliação, combinando análise estática (via SonarQube) e indicadores funcionais extraídos das submissões do LeetCode. A quarta etapa realiza a comparação sistemática dos modelos, considerando desempenho, assertividade e qualidade estrutural. Por fim, a quinta etapa consolida *insights* sobre os pontos fortes e limitações de cada LLM, permitindo interpretar os resultados de forma integrada e orientar recomendações práticas.

4.1. Definição do Dataset

Os problemas utilizados neste estudo foram obtidos a partir de um *dataset* construído com questões da plataforma LeetCode, o que garantiu uma base padronizada e representativa para a análise, conforme descrito no trabalho de Xia et al. (2025). O processo de preparação do conjunto de dados iniciou-se com a filtragem do LeetCodeDataset [Xia et al. 2025]. Foram excluídos os 256 problemas adicionados após julho de 2024, que compõem a base de avaliação do estudo original, bem como as questões pagas, devido às restrições de custo da pesquisa. Após essa filtragem, o dataset foi dividido em cinco partes equivalentes, e apenas uma delas foi selecionada para uso, resultando em 527 problemas a partir da base inicial de 2.681.

Essa seleção preservou a proporção original entre os níveis de dificuldade, garantindo consistência metodológica e mantendo o foco na base principal do conjunto. O subconjunto final manteve uma distribuição equilibrada entre os níveis de dificuldade, composta por 24,2% de problemas fáceis, 52,9% médios e 22,9% difíceis. Esse balanceamento foi essencial para assegurar diversidade na coleta de dados e possibilitar uma análise abrangente do desempenho dos LLMs em diferentes cenários de programação, refletindo uma ampla gama de desafios técnicos.

4.2. Geração de Código

O ambiente experimental utilizou a linguagem Python 3.13.3 devido às bibliotecas para requisições HTTP, manipulação de dados e operações assíncronas. As interações com os modelos foram realizadas através de suas respectivas APIs oficiais (OpenAI, Google AI for Developers e Anthropic), utilizando planos pagos, assegurando uma comparação quanto à qualidade das soluções geradas. A escolha dessas três plataformas foi justificada pela popularidade no mercado e com diferentes abordagens de LLMs, combinando modelos comerciais amplamente utilizados.

Para cada problema, os LLMs tiveram até cinco tentativas para produzir uma solução correta. Para garantir a comparabilidade entre os modelos, adotou-se um protocolo padronizado de elaboração de *prompts*, estruturado em três etapas: (1) contextualização, (2) descrição do problema e (3) instrução específica, conforme apresen-

tado no Apêndice A. Após a geração das soluções, a ferramenta SonarQube foi utilizada para avaliar a qualidade do código produzido.

Além do processo principal de geração das soluções, foi implementado um mecanismo automático de retentativa para corrigir falhas de submissão no LeetCode. Sempre que um código retornava *Wrong Answer*, *Runtime Error*, *Compile Error* ou erro de limite de tempo/memória, uma função identificava o tipo de falha e criava um *prompt* corretivo baseado na mensagem de erro e no código anterior. Cada problema podia ser reenviado até cinco vezes, aumentando a chance de obtenção de uma solução funcional. O Apêndice B apresenta o *prompt* utilizado para a resubmissão da questão na plataforma, com o objetivo de realizar a correção do erro.

4.3. Aplicação das Métricas de Avaliação

Esta subseção apresenta o conjunto de métricas utilizadas para avaliar, de forma abrangente, a qualidade do código gerado pelos modelos. A Tabela 1 organiza os principais itens considerados e descreve a relevância de cada métrica na análise de soluções eficientes, estruturadas e funcionalmente corretas. A partir dessa sistematização, estabelece-se uma base consistente para a escolha das ferramentas de avaliação e para a identificação de potenciais pontos de melhoria no contexto de geração automática de código.

Tabela 1. Tabela detalhada de avaliação com métricas aplicadas ao código gerado por LLMs

Item	Descrição	Critérios de Avaliação
Eficiência Algorítmica	Avalia o desempenho relativo do código em termos de tempo de execução e uso de memória.	• Tempo de execução: análise do tempo médio por modelo. • Uso de memória: análise de uso de memória média por modelo.
Manutenibilidade	Avalia a qualidade estrutural do código segundo análise estática.	• Complexidade ciclomática: média de valores por modelo. • Quantidade de *code smells*: média de valores por modelo.
Produtividade Efetiva	Mede a rapidez na obtenção de código funcional utilizando o mínimo de refinamentos.	• Número médio de tentativas: 1 (excelente), 2–3 (bom), ≥ 4 (aceitável).
Correção Funcional	Avalia a capacidade do modelo de produzir código correto já na primeira interação.	• Taxa de acerto na primeira tentativa: maior percentual indica maior robustez do modelo.
Desempenho por Dificuldade	Analisa a distribuição de acertos conforme os níveis Fácil, Médio e Difícil	• Quantidade de submissões *Accepted* por nível de dificuldade.
Desempenho por *Tags*	Avalia a especialização dos modelos em categorias específicas de problemas.	• Frequência de acertos por tag (ex.: Array, String, Hash Table).

Para a análise estática da qualidade dos trechos de código Java produzidos pelos modelos, empregou-se o SonarQube, ferramenta responsável pela mensuração de métricas como complexidade ciclomática e detecção de *code smells*, que garantiu uma avaliação consistente da qualidade do código.

Os valores de referência empregados para cada critério de avaliação apresentados na Tabela 1 foram definidos com base em estudos da literatura. Para eficiência algorítmica, tempo de execução e uso de memória foram analisados de forma comparativa, considerando o resultado apresentado pelos modelos nas submissões aceitas. No que diz respeito à manutenibilidade estrutural, adotaram-se as métricas reportadas pelo SonarQube, especialmente complexidade ciclomática orientada pelos parâmetros de Watson

e McCabe (1996) e a quantidade total de *code smells* identificados. Além disso, a análise funcional incluiu a distribuição de acertos por dificuldade (Fácil, Médio e Difícil) e a frequência de acertos por *tags* dominantes, permitindo observar a afinidade de cada modelo com diferentes categorias de problemas. Por fim, a produtividade efetiva foi avaliada pelo número médio de tentativas necessárias até a obtenção de uma solução funcional, alinhada às observações de Wu et al. (2024) sobre padrões de refinamento em tarefas assistidas por LLMs.

4.4. Comparação dos Resultados

Após a finalização desse processo e das verificações realizadas, compararam-se as soluções fornecidas por cada LLM, considerando tanto a qualidade do código quanto a quantidade de iterações necessárias para produzi-lo. Essa análise permitiu identificar as capacidades de cada modelo na geração de código funcional e de boa qualidade, além de evidenciar seus pontos fortes e limitações em diferentes tipos de desafios algorítmicos. Os resultados obtidos contribuem para orientar aprimoramentos futuros e apoiar aplicações práticas desses modelos em contextos reais de desenvolvimento.

Os resultados permitiram compreender as potencialidades e limitações de cada LLM em diferentes cenários de desenvolvimento, fundamentando recomendações práticas para sua adoção. Essa análise final agregou uma perspectiva aplicada à geração automática de código e auxiliou na identificação de pontos onde cada modelo se sobressai, contribuindo para aplicações práticas em contextos reais de desenvolvimento.

5. Resultados

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

5.1. Caracterização do Conjunto de Dados

O conjunto de dados utilizado, introduzido por Xia et al. (2025), é composto por 2.641 problemas de programação extraídos da plataforma LeetCode. Trata-se de um *dataset* particularmente robusto, pois cada problema inclui mais de 100 casos de teste, o que reduz significativamente o risco de falsos positivos na avaliação das soluções geradas pelos modelos.

Além disso, o *dataset* apresenta uma distribuição equilibrada entre os diferentes níveis de dificuldade. Como mostra a Tabela 2, a maior parte dos problemas é classificada como de nível médio, o que é adequado para avaliar o desempenho dos modelos em cenários que exigem domínio de algoritmos clássicos e estratégias de solução eficientes. Essa distribuição contribui para assegurar a representatividade do conjunto de problemas e a consistência das análises realizadas.

Além disso, cada problema é rotulado com múltiplas *tags* de tópicos que descrevem os algoritmos e estruturas de dados envolvidos, como "*Array*", "*String*"e "*Dynamic Programming*". O *dataset* oferece ampla cobertura desses tópicos, o que permite uma análise granular do desempenho dos LLMs em diferentes domínios de programação. A Figura 2 ilustra a frequência com que as dez tags mais populares aparecem no conjunto utilizado, evidenciando a diversidade temática presente nos dados.

Tabela 2. Distribuição de problemas por nível de dificuldade no dataset filtrado

Nível de Dificuldade	Contagem	Proporção (%)
Fácil	127	24,06
Médio	279	52,94
Difícil	121	23,00
Total	**527**	**100,00**

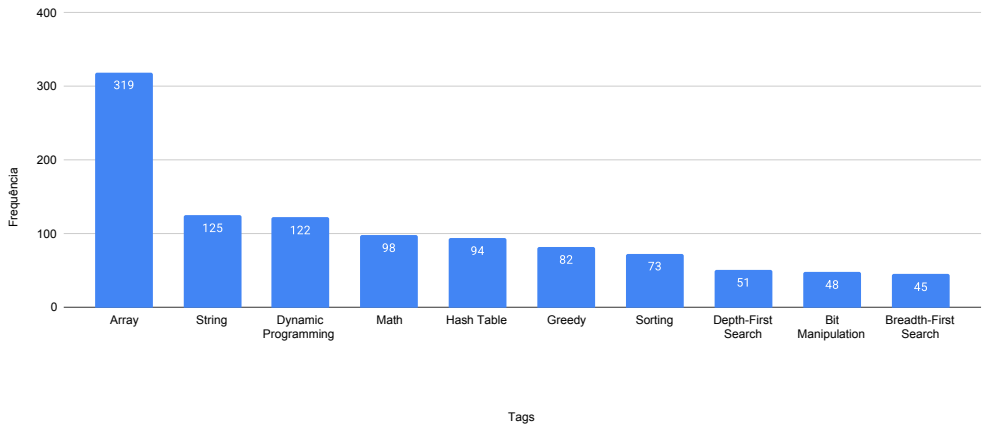

Figura 2. Distribuição das dez maiores tags por frequência.

5.2. Qualidade do código gerado por LLMs

Inicialmente, foram avaliados os indicadores funcionais das submissões, incluindo assertividade e desempenho. Em seguida, analisou-se a qualidade dos trechos de código gerados pelas LLMs para os problemas do LeetCode, considerando métricas de manutenibilidade e complexidade obtidas por meio do SonarQube.

5.2.1. Assertividade das Soluções

A avaliação da assertividade dos modelos considerou três dimensões: (1) o volume de submissões bem-sucedidas, (2) o desempenho na primeira tentativa e (3) o número médio de tentativas necessárias até alcançar uma solução *Accepted*. A Tabela 3 sintetiza o desempenho geral de cada modelo. O Gemini obteve o maior número de acertos, enquanto o Claude apresentou a maior proporção de *Wrong Answers*. Já o ChatGPT mostrou um comportamento intermediário em termos de quantidade absoluta de acertos, ainda que acompanhado de mais falhas que o Gemini.

Além do total de acertos, analisou-se a quantidade de problemas resolvidos já na primeira submissão. O Gemini obteve 285 acertos diretos, seguido de perto pelo Claude com 278. O ChatGPT apresentou apenas 18 acertos na primeira tentativa, um resultado significativamente inferior aos demais modelos. Essa baixa performance, entretanto, não está relacionada à qualidade algorítmica das soluções. Ela ocorre porque o modelo frequentemente gerou códigos com múltiplas classes ou com nomes de classe incorretos. Como o ambiente do LeetCode exige que todo o código esteja contido em

Tabela 3. Distribuição dos status de submissão por modelo

Status	ChatGPT	Claude	Gemini
Accepted	474	450	514
Wrong Answer	47	61	7
Runtime Error	0	1	0
Time Limit Exceeded	6	15	6

uma única classe denominada `Solution`, essas variações resultaram em falhas imediatas de compilação, impedindo a execução dos testes. Assim, o desempenho reduzido do ChatGPT na primeira tentativa reflete problemas de conformidade de formato e não limitações de raciocínio.

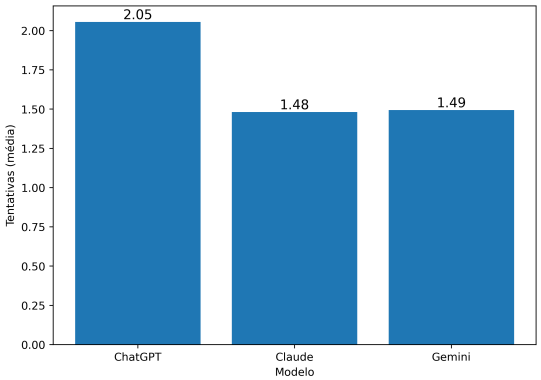

Figura 3. Média de tentativas necessárias para que cada modelo gerasse uma submissão Accepted

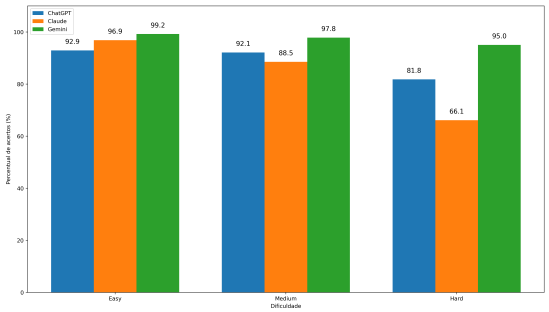

Figura 4. Distribuição da dificuldade entre as submissões Accepted por modelo

A Figura 3 evidencia que o Claude foi o modelo que menos dependeu do mecanismo de retentativa, apresentando a menor média de tentativas até o acerto. ChatGPT e Gemini, por sua vez, necessitaram de mais iterações para alcançar uma solução válida. Já a Figura 4 mostra que todos os modelos concentraram seus acertos em problemas *Easy*, com queda progressiva para *Medium* e *Hard*. Nesse aspecto, o Gemini manteve desempenho relativamente superior na categoria *Hard*, enquanto o Claude mostrou menor assertividade nesse nível de complexidade.

5.2.2. Análise das Principais Tags

A Figura 5 apresenta as dez tags mais frequentes entre as submissões aceitas, distribuídas por modelo. Observa-se que categorias de alta incidência, como *Array*, *String*, *Dynamic Programming* e *Hash Table*, estão presentes nos três modelos, embora com intensidades distintas. O ChatGPT concentra o maior volume absoluto nas principais categorias, enquanto o Gemini demonstra maior afinidade com tópicos de natureza estrutural, como *Dynamic Programming* e *Greedy*. Já o Claude apresenta uma distribuição mais equilibrada entre diferentes classes de problemas, ainda que com um volume total de acertos menor. Esses resultados indicam que cada modelo apresenta tendências específicas em relação aos tipos de problemas que consegue resolver com maior facilidade.

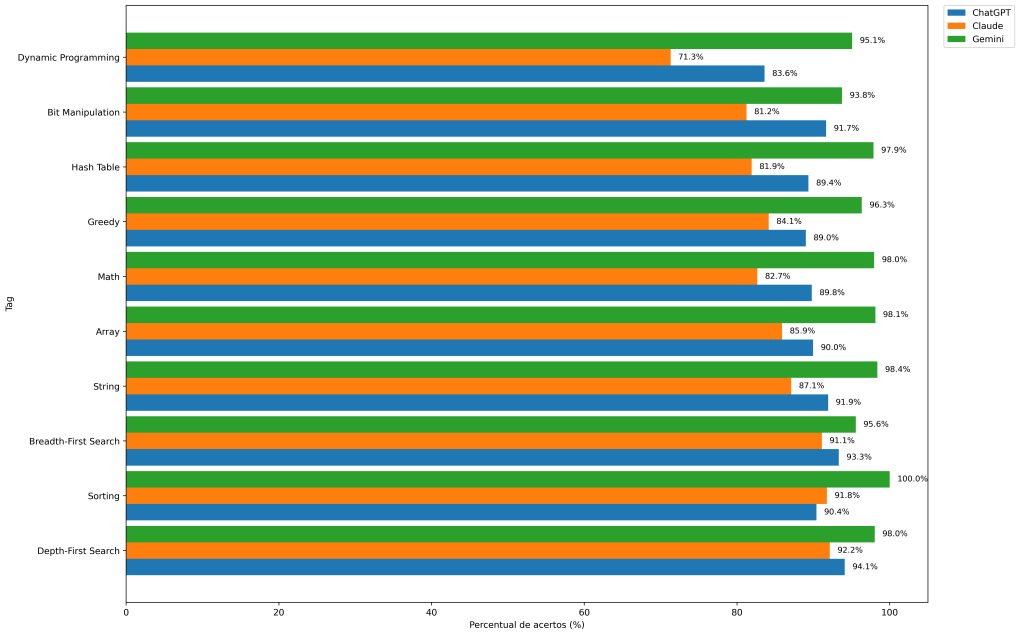

Figura 5. Desempenho dos modelos nas dez principais tags do dataset (percentual de acertos)

Além dessas tendências gerais, a distribuição por tags também revela diferenças importantes no tipo de raciocínio algorítmico favorecido por cada modelo. Categorias como *Dynamic Programming* e *Greedy*, tradicionalmente associadas a problemas que exigem maior capacidade de decomposição e otimização incremental, exibem variações mais acentuadas entre os LLMs, sugerindo que esses modelos respondem de maneira distinta a padrões recorrentes na base de treinamento. Por outro lado, áreas como *Array*, *String* e *Hash Table*, que envolvem manipulações estruturais mais diretas, apresentaram taxas de acertos mais homogêneas, indicando menor sensibilidade ao estilo de geração de cada modelo. Essas diferenças reforçam a hipótese de que o desempenho não é apenas uma função da dificuldade da categoria, mas também da forma como cada LLM internaliza e generaliza padrões algorítmicos ao longo de seu pré-treinamento.

5.3. Desempenho Operacional: Tempo de Execução e Uso de Memória

Foram analisados o tempo médio de execução (ms) e o uso médio de memória (KB) das submissões aceitas, conforme apresentado nas Figuras 6 e 7. Os resultados mostram que os modelos apresentam perfis distintos de desempenho. O Gemini obteve os menores tempos médios de execução, indicando melhor eficiência temporal. O ChatGPT, por sua vez, registrou o menor consumo de memória, sugerindo maior eficiência espacial. Já o Claude apresentou desempenho intermediário em ambas as métricas, coerente com o seu padrão de produzir código mais estável, porém menos otimizado.

5.3.1. Complexidade Ciclomática e *Code Smells*

A Figura 8 apresenta a média da complexidade ciclomática dos códigos gerados por cada modelo. Observa-se que o ChatGPT registrou o maior valor (8,45), seguido pelo

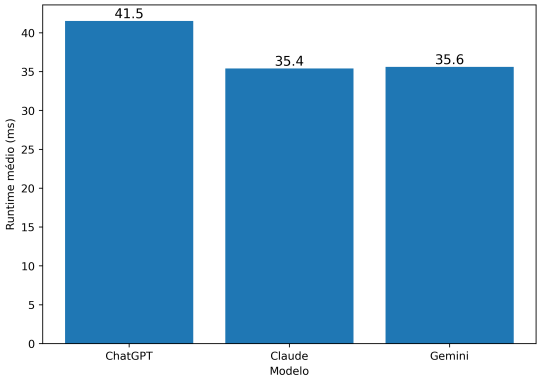

Figura 6. Tempo médio de execução (em milissegundos) das submissões Accepted para cada modelo analisado

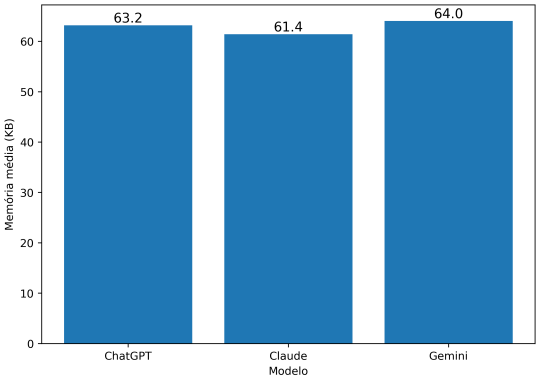

Figura 7. Uso médio de memória (em kilobytes) das submissões Accepted para cada modelo analisado

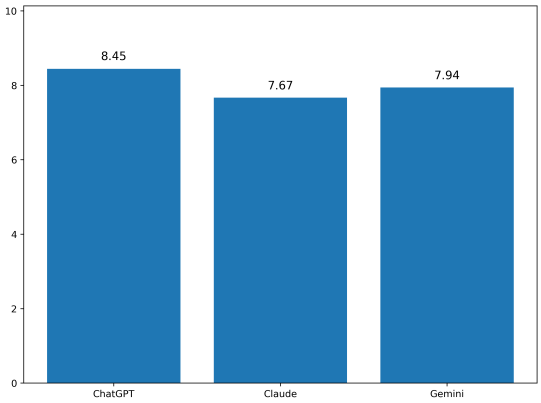

Figura 8. Comparação da média de complexidade ciclomática entre os modelos analisados

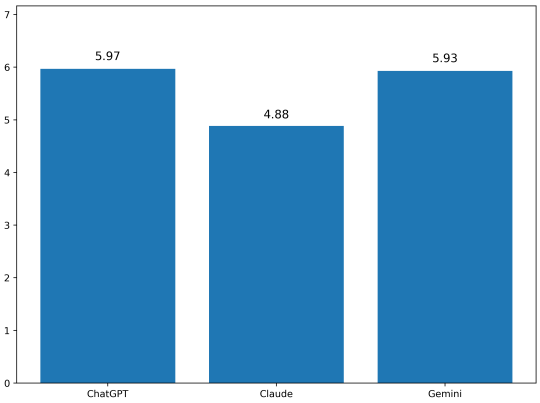

Figura 9. Distribuição da média de *code smells* entre os modelos analisados

Gemini (7,94), enquanto o Claude apresentou a menor complexidade (7,67), indicando uma organização lógica ligeiramente mais simples em comparação aos demais.

A Figura 9 apresenta a média de *code smells* identificados nos códigos produzidos por cada LLM. Os resultados mostram que o ChatGPT obteve o maior número médio de ocorrências (5,97), seguido de perto pelo Gemini (5,93). O Claude apresentou a menor média (4,88). Em termos relativos, o ChatGPT gerou aproximadamente 22,4% mais code smells do que o Claude, enquanto o Gemini apresentou cerca de 21,5% a mais. Já a diferença entre ChatGPT e Gemini foi mínima, cerca de 0,7%, indicando um comportamento bastante similar entre esses dois modelos no que diz respeito a problemas estruturais detectados.

A análise conjunta da complexidade ciclomática e da ocorrência de *code smells* indica que ChatGPT e Gemini tendem a produzir códigos mais ramificados e suscetíveis a padrões associados a maior esforço de manutenção. Ambos apresentam níveis semelhantes de problemas estruturais, refletindo maior variabilidade na organização lógica das soluções. O Claude, por sua vez, apresenta as menores médias em ambas as métricas, re-

sultando em códigos mais lineares, estáveis e com menor incidência de indícios de dívida técnica. Esses resultados sugerem que, embora todos os modelos alcancem a corretude funcional, o Claude se destaca por gerar soluções estruturalmente mais limpas e consistentes, enquanto ChatGPT e Gemini exigem maior atenção em cenários que demandam manutenibilidade e clareza arquitetural.

5.4. Síntese Comparativa dos Resultados

A análise integrada dos resultados evidencia que os três modelos apresentam perfis complementares de desempenho. O Gemini demonstrou a maior assertividade global, alcançando o maior número de submissões *Accepted* e o melhor desempenho temporal, sobretudo em problemas de maior complexidade. O Claude destacou-se pela qualidade estrutural do código, apresentando as menores médias de complexidade ciclomática e de *code smells*, além de exigir menos tentativas para produzir uma solução correta. Já o ChatGPT exibiu a melhor eficiência de memória e a maior cobertura de problemas resolvidos, embora com maior variabilidade estrutural e necessidade de retrabalho.

Esses resultados mostram que não existe um modelo universalmente superior, mas sim diferentes pontos fortes que atendem a necessidades específicas. Em cenários que exigem rapidez e maior taxa de acertos, o Gemini tende a ser mais vantajoso. Quando a manutenibilidade e a simplicidade estrutural são prioritárias, o Claude se sobressai. Já em contextos que valorizam diversidade de soluções e baixo consumo de recursos, o ChatGPT apresenta desempenho competitivo. Essa complementaridade reforça que a escolha do LLM deve considerar o contexto técnico de uso e os critérios de qualidade mais relevantes para o projeto.

6. Ameaças à Validade

Esta seção apresenta as principais ameaças à validade deste estudo, bem como as estratégias adotadas para mitigá-las, conforme recomendações de Wohlin et al. (2012).

Em relação à validade externa, os resultados obtidos podem não ser diretamente generalizáveis para outros contextos além da plataforma LeetCode ou para linguagens distintas de Java. Além disso, os três modelos avaliados, ChatGPT, Claude e Gemini — representam apenas parte do ecossistema de LLMs, e suas performances podem não refletir o comportamento de outros modelos ou de futuras versões. Apesar disso, o conjunto analisado inclui 527 problemas com diversidade de dificuldades e tópicos, o que contribui para ampliar a representatividade dentro do escopo investigado.

Quanto à validade de construção, as métricas utilizadas para avaliar a qualidade do código, tais como complexidade ciclomática, *code smells*, *runtime* e uso de memória não capturam todos os aspectos qualitativos das soluções geradas. Para mitigar essas limitações, foram adotadas métricas documentadas pelo SonarSource, aplicadas de forma uniforme, e amostras foram verificadas manualmente para garantir consistência.

Por fim, a validade de conclusão e replicabilidade é impactada pela natureza não determinística das LLMs e pela sua constante atualização. Como os modelos podem alterar seu comportamento ao longo do tempo, a reprodução exata dos resultados pode ser comprometida. Para reduzir essa ameaça, toda a coleta foi realizada em um único período, utilizando os mesmos *prompts* e parâmetros, além do registro completo das planilhas e *scripts* empregados no processo.

7. Conclusão

Este trabalho analisou comparativamente a qualidade do código gerado pelos LLMs ChatGPT, Gemini e Claude na resolução de problemas do LeetCode em Java, considerando métricas estruturais extraídas pelo SonarQube, indicadores funcionais de assertividade e desempenho operacional. Os resultados evidenciaram comportamentos complementares entre os modelos: o ChatGPT apresentou maior complexidade e volume de *code smells*, o Gemini obteve a maior taxa de acertos e o melhor tempo de execução, enquanto o Claude produziu código estruturalmente mais simples e consistente, requerendo menos tentativas para alcançar uma solução correta.

Esses achados confirmam que não há um modelo superior em todos os critérios avaliados, mas sim diferentes perfis de desempenho. O Gemini se destacou pela eficiência e assertividade, o Claude pela qualidade estrutural e estabilidade das respostas, e o ChatGPT pela combinação entre diversidade de problemas resolvidos, bom uso de memória e desempenho funcional, embora com maior custo de manutenção. Assim, a geração automática de código revela um conjunto de *trade-offs* inerentes entre manutenibilidade, desempenho algorítmico e precisão das respostas.

Por fim, conclui-se que a escolha da LLM mais adequada deve ser guiada pelos requisitos específicos de cada projeto. Como trabalhos futuros, recomenda-se ampliar a análise para outras linguagens, modelos especializados em código e técnicas iterativas de refinamento, permitindo aprofundar a compreensão sobre o papel das LLMs na engenharia de software e aprimorar sua integração em *pipelines* reais de desenvolvimento.

8. 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-gustavo-riegert`

Referências

[Bae et al. 2024] Bae, J., Kwon, S., and Myeong, S. (2024). Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques. *Electronics*, 13(13).

[Chen et al. 2021] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code.

[Cormen et al. 2001] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction To Algorithms*, volume 42. McGraw-Hill Education.

[Hamer et al. 2024] Hamer, S., d'Amorim, M., and Williams, L. (2024). Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 87–94.

[Jain et al. 2024] Jain, A., William, P., Ayasrah, F. T., Lakshmi, G. P., and Diwan, T. D. (2024). Analyzing automatic code generation for learning models in generative ai. In *2024 11th International Conference on Software Defined Systems (SDS)*, pages 50–58.

[Jesse et al. 2023] Jesse, K., Ahmed, T., Devanbu, P. T., and Morgan, E. (2023). Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575.

[Liu et al. 2024] Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., and Lo, D. (2024). Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*, 33(5).

[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 *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, page 103–107, New York, NY, USA. Association for Computing Machinery.

[Osorio et al. 2025] Osorio, L., Neto, P. S., Avelino, G., and Lira, W. (2025). An evaluation of the impact of code generation tools on software development. In *Anais do XXI Simpósio Brasileiro de Sistemas de Informação*, pages 625–634, Porto Alegre, RS, Brasil. SBC.

[Overflow 2024] Overflow, S. (2024). Stack overflow developer survey 2024. `https://survey.stackoverflow.co/2024/technology`. Acessado em: 16 de maio de 2025.

[Perry et al. 2023] Perry, N., Srivastava, M., Kumar, D., and Boneh, D. (2023). Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2785–2799, New York, NY, USA. Association for Computing Machinery.

[Pressman and Maxim 2020] Pressman, R. and Maxim, B. (2020). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

[Rosenbaum 2022] Rosenbaum, E. (2022). Microsoft's github copilot ai is making rapid progress. here's how its human leader thinks about it. `https://www.cnbc.com/2022/10/14/microsoft-ai-leaps-ahead-heres-what-its-human-leader-thinks-about-it.html`.

[Svyatkovskiy et al. 2020] Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. (2020). Intellicode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA. Association for Computing Machinery.

[Vaithilingam et al. 2022] Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered

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

[Vaswani et al. 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

[Watson and McCabe 1996] Watson, A. H. and McCabe, T. J. (1996). Core: Resolving code quality issues using llmsa. Publicação especial nist 500-235, National Institute of Standards and Technology (NIST). Recuperado em 14 de maio de 2011.

[Wohlin et al. 2012] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[Wu et al. 2024] Wu, L., Zhao, Y., Hou, X., Liu, T., and Wang, H. (2024). Chatgpt chats decoded: Uncovering prompt patterns for superior solutions in software development lifecycle. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 142–146.

[Xia et al. 2025] Xia, Y., Shen, W., Wang, Y., Liu, J. K., Sun, H., Wu, S., Hu, J., and Xu, X. (2025). Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms.

Apêndice

A. *Prompt* de Primeira Geração de Código

> *Atue como um desenvolvedor Java sênior e analista de sistemas. Seu objetivo é criar uma solução de alta qualidade para um problema de algoritmo. Garanta que o código seja robusto e eficiente. Aqui está a descrição completa do problema bem como entradas e saídas esperadas:*
>
> *<Nome do problema>*
> *<Descrição completa do problema, extraída diretamente do LeetCode>*
> *<Exemplos de entrada e saída>*
>
> *Escreva uma solução em Java que seja funcional e eficiente, garantindo a correção dos resultados e a adoção de algoritmos otimizados, de modo a alcançar a melhor complexidade de tempo e de espaço possível. Além disso, o código deve ser manutenível e legível, seguindo as convenções de estilo da linguagem Java e incorporando as práticas recomendadas de Clean Code, com nomes claros e consistentes para variáveis e métodos. Retorne apenas a classe Java completa, dentro de um bloco "'java ... "' sem explicações adicionais e comentários no código.*

B. *Prompt* de Correção e Retentativa

> *Você é um programador experiente em Java. Foi enviado um código para o problema "<Nome do Problema>"do LeetCode, mas ele gerou um <Mensagem de erro retornada pelo LeetCode> na submissão.*
>
> *Aqui está o código que foi submetido:*

> *<Código Java submetido anteriormente>*
>
> *Corrija o código para que ele compile e funcione corretamente para o problema informado. Mantenha a lógica original tanto quanto possível. Garanta que o código trate os casos de borda e compile sem erros. Retorne apenas a classe Java completa, dentro de um bloco "'java ... "' sem explicações adicionais e comentários no código.*