Impacto do Código Gerado por Inteligência Artificial na Manutenibilidade do Software

Vitor Lany Freitas Ferreira[1]

[1] Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUCMG)
Belo Horizonte, MG – Brasil

vitor.ferreira.1386402@sga.pucminas.br

Abstract. *AI tools such as GitHub Copilot increase productivity but raise questions about the quality of the code produced. This study investigated the architectural impact of implementing 115 features in 23 open-source Java repositories using systematic reproduction of issues. The results indicate a mixed scenario: AI preserves cohesion and inheritance, but increases coupling and class complexity. In addition, new code smells required 38 minutes of refactoring per task. It is concluded that, without rigorous review, technical debt can offset speed gains.*

Resumo. *Ferramentas de IA como o GitHub Copilot ampliam a produtividade, mas levantam dúvidas sobre a qualidade do código produzido. Este estudo investigou o impacto arquitetural ao implementar 115 funcionalidades em 23 repositórios Java open source, utilizando a reprodução sistemática de issues. Os resultados indicam um cenário misto: a IA preserva a coesão e a herança, mas aumenta o acoplamento e a complexidade das classes. Além disso, novos code smells exigiram 38 minutos de refatoração por tarefa. Conclui-se que, sem revisão rigorosa, a dívida técnica pode neutralizar os ganhos de velocidade.*

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): João Pedro Batisteli - joaobatisteli@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 Pedro Batisteli - joaobatisteli@pucminas.br

Belo Horizonte, 23 de novembro de 2025.

1. Introdução

Com o aumento da relevância da Inteligência Artificial (IA), especialmente dos Grandes Modelos de Linguagem (LLMs, do inglês *Large Language Models*) [Guo 2025], o setor tem recebido investimentos expressivos de empresas como Amazon, Google, Microsoft e Meta, que planejam direcionar mais de US$ 300 bilhões à infraestrutura de IA ainda em 2025 [Times 2025]. Esse avanço tecnológico também tem transformado práticas de desenvolvimento de software por meio de ferramentas como o GitHub Copilot, que utiliza um modelo de IA treinado em repositórios *open-source* para converter instruções em

linguagem natural em código. Atuando como um *pair programmer* virtual, o Copilot fornece trechos de código e sugestões contextuais, auxiliando diretamente o trabalho do desenvolvedor [Nguyen and Nadi 2022].

Embora o uso de ferramentas de IA para a geracão de código tenha promovido avanços expressivos, é necessário **investigar de maneira sistemática não apenas os ganhos imediatos de produtividade oferecidos por assistentes de programação baseados em IA, mas também seus efeitos em médio e longo prazo, especialmente no que diz respeito à qualidade estrutural e à manutenibilidade dos sistemas produzidos**. Em um estudo conduzido por Russo (2024), desenvolvedores foram entrevistados sobre suas percepções a respeito dessas ferramentas, e 13% deles relataram preocupações específicas relacionadas à manutenibilidade do código gerado. Esse dado evidencia uma preocupação crescente entre profissionais da área quanto aos riscos associados à adoção dessas tecnologias. A manutenibilidade, definida como a facilidade de modificar um *software* [IEEE 1990], constitui um dos principais fatores de custo ao longo do ciclo de vida de um sistema [Riaz et al. 2009].

Características como legibilidade, modularidade e aderência a boas práticas de engenharia de *software* são fundamentais para garantir a manutenção eficiente de um código-fonte e, portanto, precisam ser cuidadosamente consideradas ao avaliar a produção de código por ferramentas de IA. Estudos recentes indicam que essas ferramentas nem sempre geram soluções otimizadas nesses aspectos, introduzindo problemas como *code smells* e redundâncias que podem comprometer a escalabilidade dos sistemas e a colaboração entre equipes [Siddiq et al. 2022, Nguyen and Nadi 2022]. A literatura aborda diversas questões relacionadas à geração de código por IA, como eficiência entre linguagens [Nguyen and Nadi 2022], incidência de *bugs* [Clark et al. 2024], elaboração de *prompts* eficazes [Niu et al. 2024], produção de *code smells* [Siddiq et al. 2022] e métricas de previsibilidade de manutenção [Devi et al. 2023], o que reforça a relevância de investigar como esses fatores podem se relacionar com aspectos arquiteturais do software gerado por IA.

Neste contexto, esse trabalho tem como objetivo **avaliar os efeitos da utilização de ferramentas baseadas em inteligência artificial na geração de código sobre a manutenibilidade de *software***. Para isso, são implementadas modificiações por inteiro utilizando a ferramenta GitHub Copilot, a partir de descrições de *issues* extraídas de repositórios *open source* escritos em Java. Os objetivos específicos deste estudo são: (i) selecionar e validar os repositórios e *issues* a serem utilizados nas análises; (ii) mensurar a variação das métricas de manutenibilidade e complexidade fornecidas pela ferramenta CK, comparando os valores antes e depois da geração de código; e (iii) identificar, classificar e ranquear a criticidade dos *code smells* introduzidos após o uso do Copilot, utilizando a ferramenta SonarQube para coleta e avaliação dos dados.

Com este estudo, foi possível analisar de forma abrangente os impactos reais do uso de ferramentas de IA no ciclo de vida de um *software*. Os resultados indicam que, embora a geração automática de código preserve métricas como a coesão das classes, ela tende a aumentar o acoplamento e a complexidade, além de introduzir *code smells*. Dessa forma, este artigo apresenta evidências empíricas que podem auxiliar na tomada de decisão sobre a adoção dessas ferramentas, permitindo ponderar os ganhos de produtividade em relação aos custos de manutenção a longo prazo.

Este artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica, abordando os conceitos centrais sobre manutenibilidade de *software*, métricas de qualidade e o funcionamento das ferramentas de geração automática de código por IA. Na Seção 3, são discutidos os trabalhos relacionados, com destaque para estudos prévios sobre o tema. A Seção 4 descreve os materiais e métodos utilizados na condução do experimento. Em seguida, a Seção 5 apresenta e analisa os resultados obtidos. A Seção 6 discute as possíveis ameaças à validade do estudo. Por fim, a Seção 7 traz a conclusão, sintetizando os principais achados da pesquisa e sugerindo direções para trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta os principais conceitos teóricos que sustentam esta pesquisa. São abordados a manutenibilidade de *software*, as métricas de qualidade utilizadas para análise estrutural do código e o conceito de *code smells*.

2.1. Ferramentas de Geração Automática de Código

A evolução das ferramentas de apoio ao desenvolvimento culminou no surgimento de assistentes de codificação baseados em Inteligência Artificial Generativa, que representam um salto em relação aos recursos tradicionais de autocompletar. Ferramentas como o GitHub Copilot fundamentam-se em LLMs, arquiteturas de redes neurais treinadas em grande volumes da dados textuais, incluindo repositórios de código públicos. Essa base de treinamento permite que os modelos identifiquem padrões sintáticos e semânticos complexos, capacitando-os a prever trechos de código inteiros e a transformar instruções em linguagem natural (*prompts*) em implementações executáveis [Nguyen and Nadi 2022]. A capacidade dessas ferramentas de processar o contexto do arquivo atual e de outros arquivos do projeto permite sugestões mais assertivas e adaptadas ao estilo de codificação existente.

Na prática, essas ferramentas atuam como um *pair programmer*, auxiliando tanto na escrita de código repetitivo quanto na prototipagem de funções complexas. No entanto, a natureza probabilística dos LLMs implica que o código gerado nem sempre é a solução mais eficiente, segura ou correta para o problema apresentado. Como observado por Clark et al. (2024), a produção da IA demanda validação rigorosa e, frequentemente, refatoração por parte dos desenvolvedores para adequar-se aos requisitos não funcionais do sistema. Apesar do potencial para acelerar o desenvolvimento, a facilidade de gerar código em larga escala levanta preocupações crescentes sobre a inserção silenciosa de dívida técnica e os impactos negativos na manutenibilidade do *software* a longo prazo [Russo 2024].

2.2. Manutenibilidade e Métricas de Software

A manutenibilidade de *software* refere-se à facilidade com que um sistema pode ser modificado para corrigir falhas, aprimorar seu desempenho ou adaptar-se a novos requisitos ou ambientes [IEEE 1990]. Trata-se de uma característica central da qualidade, especialmente porque atividades de manutenção representam a maior parte dos custos ao longo do ciclo de vida de um sistema. Quando a arquitetura do software apresenta alto acoplamento, baixa coesão ou estruturas excessivamente complexas, as alterações tornam-se mais difíceis, arriscadas e onerosas, podendo comprometer sua evolução ao longo do tempo [Sommerville 2019].

A avaliação da manutenibilidade frequentemente se apoia em métricas estruturais de código amplamente utilizadas na engenharia de *software* orientada a objetos, como as disponibilizadas pela ferramenta CK [Aniche 2016] e discutidas na literatura [Fenton and Bieman 2014]. Entre essas métricas, destacam-se medidas de profundidade de herança, que indicam o grau de complexidade introduzido por hierarquias extensas; de resposta de uma classe, que refletem a quantidade de métodos potencialmente executados e o tamanho de sua interface; de complexidade interna dos métodos, que influenciam diretamente o esforço de compreensão e modificação; de coesão, que avaliam o alinhamento dos métodos em torno de responsabilidades comuns; e de acoplamento entre classes, que expressam o nível de dependência estrutural entre componentes do sistema.

Essas métricas, consideradas em conjunto, oferecem uma visão estruturada sobre o quão organizado, modular e sustentável é um código-fonte, servindo como base para análises relacionadas à qualidade e à capacidade de evolução de sistemas orientados a objetos.

2.3. Code Smells

O termo *code smell* refere-se a indícios superficiais que sugerem a presença de problemas mais profundos na estrutura de um sistema. Embora não representem necessariamente erros de execução, esses sinais apontam fragilidades no design do código que podem comprometer sua legibilidade, extensibilidade e manutenibilidade ao longo do tempo [Martin 2009]. *Code smells* não indicam que o código está errado, mas revelam trechos que merecem atenção por potencialmente dificultarem futuras modificações ou introduzirem erros sutis com a evolução do software [Martin 2009]. Entre seus exemplos clássicos estão métodos excessivamente longos, classes que acumulam múltiplas responsabilidades, nomes pouco descritivos, trechos de código duplicados e alto acoplamento entre componentes.

Tais sintomas violam princípios fundamentais da engenharia de software, como o Princípio da Responsabilidade Única e o Princípio Aberto–Fechado, prejudicando a modularidade e a capacidade de evolução do sistema [Martin 2009]. Nesse sentido, identificar code smells é um passo essencial para orientar refatorações e manter a qualidade estrutural do código.

3. Trabalhos Relacionados

Nesta seção, são discutidos os estudos que ajudam a explicar o contexto em que este artigo está inserido, juntamente com estudos que analisam o mesmo processo ou utilizam metodologias semelhantes ao que é proposto. Especificamente, os trabalhos relacionados discutidos nesta seção envolvem o uso de ferramentas de geração de código por IA e manutenibilidade de código.

Nguyen e Nadi (2022) propõem uma análise da eficácia do GitHub Copilot na geração automática de código com base em *prompts* derivados de 33 questões da plataforma LeetCode. O estudo avalia um total de 132 sugestões geradas em quatro linguagens de programação distintas, Python, Java, JavaScript e C, com o objetivo de examinar a assertividade e a qualidade dos códigos produzidos. Para isso, utilizaram os testes fornecidos pelo próprio LeetCode como métrica de sucesso das tarefas, enquanto a qualidade estrutural do código foi analisada por meio da ferramenta SonarQube. Os resultados obtidos indicam que os códigos gerados apresentam, em geral, baixa complexidade e boa

compreensibilidade. No entanto, observou-se também a ocorrência de problemas recorrentes, como trechos redundantes ou dependência de funções auxiliares não previamente definidas, evidenciando que o Copilot ainda não é completamente autossuficiente e requer revisão humana. Dentre as linguagens analisadas, Java apresentou a maior taxa de correção (57%), enquanto JavaScript obteve a menor (27%). Assim, o trabalho evidencia o potencial do Copilot para resolução de problemas isolados, mas não explora os impactos arquiteturais da utilização desse código em contextos de desenvolvimento de sistemas reais, lacuna que é abordada na presente pesquisa por meio da implementação de *issues* completas em repositórios Java existentes.

Goel e Kaur (2025) conduziram um estudo empírico voltado à previsão da manutenibilidade de software, no qual coletaram e analisaram métricas de orientação a objetos extraídas de projetos de código aberto, como versões do Android e do Apache Kafka, com o objetivo de identificar atributos capazes de predizer alterações no código entre versões. Utilizando técnicas como a correlação bisserial pontual, os autores demonstraram que métricas clássicas como DIT (profundidade da árvore de herança), NOC (número de filhos), RFC (resposta para uma classe), WMC (métodos ponderados por classe), LCOM (falta de coesão em métodos) e CBO (acoplamento entre objetos) apresentam forte relação com a propensão a mudanças em classes, configurando-se como preditores eficazes de manutenibilidade. Essas evidências empíricas são reforçadas pelos achados de Devi et al. (2023), que, por meio de uma revisão sistemática da literatura entre 1999 e 2022, também apontaram essas mesmas métricas como as mais recorrentes e relevantes em modelos de previsão baseados em técnicas de *soft computing*. A convergência entre a fundamentação teórica e os resultados práticos fornece base sólida para a adoção desses atributos na presente pesquisa, que os utiliza para avaliar a degradação ou preservação da manutenibilidade em códigos gerados por assistentes de inteligência artificial.

Siddiq et al. (2022) questionam a qualidade dos *datasets* utilizados no treinamento de modelos de geração de código automático e a possível propagação de *code smells* para o código gerado. Os autores analisaram três *datasets* públicos para Python (CodeXGlue, APPS e Code Clippy) para verificar a presença de problemas de código e sua replicação em código gerado por modelos como GPT-Code-Clippy (*open-source*) e GitHub Copilot (*closed-source*). O GPT-Code-Clippy foi treinado com os três *datasets* e avaliado com Pylint e Bandit em 508.707 *snippets*. Adicionalmente, 164 *prompts* do *dataset* HumanEval geraram 16.400 saídas, analisadas quanto à presença de *code smells*. O GitHub Copilot foi testado com os mesmos *prompts* no Visual Studio Code. Os resultados demonstram que, embora auxiliem na economia de tempo, os geradores produzem código com *code smells* (e.g., variáveis indefinidas e funções perigosas), evidenciando a necessidade de revisão, mesmo em código funcional. Diferentemente do trabalho anterior, que se concentrou em Python e *datasets* sintéticos, esta pesquisa analisou a linguagem Java em cenários de manutenção real, investigando a persistência de *code smells* e seu impacto na dívida técnica.

Clark et al. (2024) analisam a qualidade e a complexidade do código gerado pelo ChatGPT, bem como a consistência entre diferentes versões da ferramenta para a mesma tarefa. Para isso, a pesquisa usa a base pública DevGPT, que contém mais de 12 mil trechos de código de conversas entre desenvolvedores e o ChatGPT, para criar uma análise com base em 625 *snippets* de código Python retirados da seção GitHub Issues. Para avaliar

a qualidade e complexidade, foram usadas as métricas de Halstead, que analisam aspectos como volume, esforço, tempo de programação e número estimado de *bugs*, comparando os resultados entre versões do modelo. Os dados indicam que o código gerado é, em média, curto e apresenta uma estimativa de cerca de 20,49 *bugs* por *snippet*. Adicionalmente, os valores relacionados a volume e dificuldade evidenciam alta consistência entre as versões analisadas. Diferentemente de Clark et al. (2024), que focaram em métricas de volume e estimativa de *bugs* em trechos isolados, esta pesquisa avalia o impacto estrutural em projetos Java completos, oferecendo uma visão voltada especificamente à arquitetura e manutenibilidade, embora ambas as pesquisas utilizem *issues* do GitHub como base para as tarefas.

Niu et al. (2024) observaram uma lacuna na avaliação da eficiência do código gerado por LLMs, no qual se priorizava a correção funcional até então. Embora existam trabalhos que avaliam a correção, a eficiência é frequentemente negligenciada, apesar de sua importância. A avaliação de LLMs em *benchmarks* como HumanEval, MBPP e LeetCode demonstrou que a eficiência do código gerado independe da taxa de correção e do tamanho dos modelos. Estratégias de construção de *prompt*, como *"chain-of-thought"*, podem melhorar a eficiência, especialmente em problemas complexos. O trabalho demonstra o impacto positivo de *prompts* eficientes na qualidade do código gerado, aspecto que será explorado nesta pesquisa para a obtenção de código mais eficiente.

Diante dos estudos analisados, nota-se uma preocupação comum com a qualidade do código gerado por ferramentas de IA, abordando aspectos como correção, manutenibilidade, eficiência e presença de *code smells*.

4. Materiais e Métodos

O trabalho proposto adota uma abordagem de pesquisa quantitativa, de natureza aplicada. Caracteriza-se como quantitativa por analisar dados numéricos obtidos a partir de métricas de *software*, e como aplicada por tratar de um problema prático enfrentado por desenvolvedores, com o intuito de oferecer informações úteis para profissionais que utilizam ferramentas de geração automática de código em seu cotidiano.

Para a execução deste estudo, considera-se o uso do agente de código GitHub Copilot, baseado no modelo de linguagem GPT-5 mini, para a geração de código em linguagem Java. A escolha do modelo GPT-5 mini foi feita por ser o modelo padrão da ferramenta no momento atual da pesquisa e não consumir requisições *premium* adicionais. A configuração da máquina local não é um fator determinante no experimento, uma vez que a geração do código é processada nos servidores do agente de IA.

Os procedimentos adotados neste estudo foram organizados em três etapas principais, conforme ilustrado no diagrama da Figura 1. Inicalmente foi realizado a seleção dos repositórios e *issues* que compõem o conjunto de análise, etapa na qual são definidos os critérios de escolha e a separação de repositórios e *issues* a serem estudados. Em seguida, ocorre a fase de implementação e coleta de dados, que envolve a preparação do ambiente, a geração de código pelo Github Copilot e a mensuração das métricas necessárias. Por fim, os dados coletados são sintetizados por meio da criação de gráficos e de análises comparativas, permitindo a interpretação dos resultados. Cada uma dessas etapas é detalhada nas subseções seguintes.

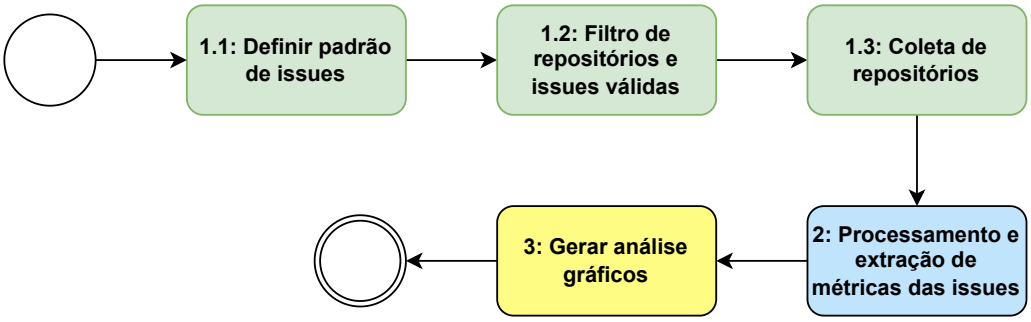

Figura 1. Diagrama do processo metodológico

4.1. Coleta de *issues*

A primeira etapa do processo consistiu na coleta dos repositórios e das *issues* que serviriam de base para o estudo. Essa fase teve início com uma análise exploratória voltada à identificação de *labels* relacionadas à implementação de novas funcionalidades. Para isso, foi reunido um conjunto de 1.000 repositórios *open source* escritos em Java, selecionados entre os mais populares da plataforma GitHub com base no número de estrelas. A inspeção das *labels* presentes nesses repositórios permitiu a elaboração de uma expressão regular capaz de automatizar a detecção das *issues* relevantes: `(enhancement|feature|suggestion|new[-_]?feature|funcionalidade|improvement)`. Após a aplicação dessa expressão, foi realizada uma triagem manual para remover *issues* cujo escopo se restringia à melhoria de documentação, garantindo que o conjunto final contemplasse apenas solicitações de implementação funcional.

Definido o critério de identificação das *labels*, a seleção de repositórios para o experimento partiu de um conjunto de 50 dos projetos Java mais populares. Para garantir que estes eram projetos de *software* executável, foi verificada a presença de arquivos de configuração de gerenciadores de projeto, como Maven (`pom.xml`) ou Gradle (`build.gradle`). Em seguida, foram selecionados os repositórios que possuiam *issues* em aberto que usavam uma *label* que passava pela expressão regular definida anteriormente. Para definir o tamanho mínimo das descrições das *issues* a serem analisadas, foram coletados os tamanhos das descrições de cada repositório e calculou-se a média de 95% da distribuição dos dados, descartando os 5% de valores extremos caracterizados como *outliers*, para assim obter um tamanho de uma descrição suficientemente detalhada a ser analisada.

Em seguida, foram selecionadas as *issues* que comporiam o conjunto analisado. Com o objetivo central de restringir a análise a solicitações de novas funcionalidades ainda não atendidas, foram coletadas, para cada repositório, cinco *issues* recentes que atendessem simultaneamente a três critérios: estarem com o estado ‘aberto’, indicando que a demanda ainda não foi solucionada; possuírem descrição suficientemente detalhada; e não terem um *pull request* já mesclado. Essa filtragem garantiu a consistência do conjunto de dados, excluindo automaticamente repositórios e itens que não representassem demandas pendentes de implementação.

4.2. Processamento das Issues Selecionadas e Extração de Métricas

A segunda etapa, ilustrada na Figura 1 (item 2), abrangeu o processo de preparação do ambiente local, implementação das funcionalidades e coleta dos dados necessários à análise. Inicialmente, cada repositório selecionado teve seu estado atual persistido localmente. Em seguida, o controle de versão original foi desvinculado e um novo repositório Git foi iniciado, utilizando a *branch* `main` como ponto de partida para todas as operações. Ainda nessa fase, foi conduzida uma pré-análise em que foram criadas *branches* específicas, cada uma representando uma *issue*, juntamente com um arquivo de controle destinado a registrar o estado de cada funcionalidade. Esse arquivo permitiu verificar se a *issue* estava pronta para receber código, se já havia sido implementada ou se já havia passado por alguma análise prévia, garantindo rastreabilidade e organização ao longo do processo.

A etapa de implementação foi conduzida manualmente. Para cada repositório preparado localmente, o projeto foi aberto no editor Visual Studio Code, configurado com a integração do GitHub Copilot, e instruções foram fornecidas à ferramenta com base na descrição da *issue* correspondente. Durante essa fase, a saída gerada pela IA foi continuamente avaliada até que fosse obtido código em linguagem Java. Quando a ferramenta produzia respostas incompletas ou em outra linguagem, novas interações eram realizadas, seja abrindo abas com contexto limpo, seja reforçando explicitamente no *prompt* que a geração deveria ocorrer exclusivamente em Java.

Após a conclusão de cada implementação, um *script* desenvolvido em Python, iniciava uma rotina responsável por identificar os repositórios prontos para análise. Para cada repositório selecionado, realizava-se inicialmente uma avaliação no SonarQube, a fim de estabelecer uma *baseline* do projeto. Em seguida, cada *branch* correspondente a uma issue era analisada individualmente, permitindo a coleta dos dados provenientes do SonarQube, das métricas CK e o registro dos arquivos modificados durante a implementação da nova funcionalidade, informações essenciais para análises posteriores.

Todas as etapas tiveram seus resultados armazenados localmente, assegurando rastreabilidade e organização ao longo do processo. Além disso, scripts auxiliares foram utilizados para automatizar partes do fluxo que não influenciavam o resultado final, como o gerenciamento de arquivos e a preparação dos ambientes de análise, reduzindo o esforço manual e garantindo maior consistência operacional.

4.3. Métricas

Para a análise da qualidade do código-fonte gerado pela IA, foi selecionado um conjunto de métricas de *software*. A escolha destas métricas visa avaliar diferentes dimensões da qualidade interna do código, como coesão, acoplamento, complexidade e herança. As definições das métricas selecionadas, detalhadas a seguir, foram baseadas e coletadas utilizando as ferramentas CK e SonarQube.

A avaliação da coesão das classes foi realizada por meio da métrica LCOM, uma versão modificada e normalizada que mensura a falta de coesão entre os métodos de uma classe em uma escala de 0 a 1. Valores próximos de 1 indicam baixa coesão, refletindo classes cujos métodos compartilham poucos atributos ou responsabilidades, já valores próximos de 0 sugerem classes mais coesas e, portanto, mais alinhadas aos princípios de bom design orientado a objetos. Complementando essa análise, o acoplamento entre

classes foi obtido pela métrica CBO, que quantifica o número de dependências externas presentes em uma classe. Essa contagem considera todos os tipos utilizados, como declarações de campos, tipos de retorno de métodos e variáveis locais, excluindo dependências de pacotes nativos da linguagem Java.

A estrutura hierárquica e a complexidade interna das classes são examinadas por meio de três métricas complementares. A profundidade da hierarquia de herança obtida pelo DIT, que mede quantos níveis de *superclasses* antecedem uma classe na árvore de herança, indicando o grau de reutilização e a possível complexidade introduzida por estruturas hierárquicas profundas. A complexidade interna é mensurada pelo WMC, que corresponde à soma das complexidades individuais dos métodos, frequentemente associada à complexidade ciclomática de McCabe, e reflete o esforço necessário para compreender, testar e modificar a classe. Para complementar essa avaliação, empregou-se o RFC, que estima o conjunto de respostas possíveis de uma classe ao contabilizar os métodos que podem ser acionados a partir de suas operações, fornecendo uma indicação do potencial comportamento e da interface efetiva da classe.

Adicionalmente a estas métricas estruturais, foram coletadas métricas específicas do SonarQube focadas na manutenibilidade do código novo. A primeira é a novas questões de manutenibilidade (`new_maintainability_issues`), que contabiliza o número total de novos *code smells* e os classifica por criticidade. A segunda métrica, diretamente ligada à anterior, é o esforço de remediação da manutenibilidade (`new_software_quality_maintainability_remediation_effort`), que estima o tempo (em minutos) necessário para corrigir todas as novas questões de manutenibilidade identificadas.

4.4. Análise dos dados

Tabela 1. Descrição das Métricas Calculadas

Métrica Calculada	Descrição
`lcom_star_modified_avg_delta`	Em *issues* que modificaram arquivos existentes, calcula a média da diferença do LCOM* de cada classe antes (`lcom_star_avg_before`) e depois (`lcom_star_avg_after`) da implementação.
`[MÉTRICA]_avg_delta`	Diferença entre a média das respectivas métricas (CBO, DIT, RFC, WMC e LCOM*) no repositório antes e depois da implementação da *issue*, medindo o impacto real de cada uma no *software* final.
`new_maintainability_issues_[CRITICIDADE], _total`	Armazena a quantidade de *code smells* introduzidos por cada *issue*, apresentando o total e a divisão por nível de criticidade *(Blocker, High, Medium, Low, Info)*.

`new_software_quality_maintainability_remediation_effort`	Estimativa de esforço (em minutos), calculada pelo próprio SonarQube, para a correção de todos os *code smells* introduzidos pela *issue* implementada.

A Tabela 1 apresenta as principais métricas computadas na etapa final da metodologia, na qual foi realizada a análise dos dados coletados ao longo de todo o processo. Um conjunto de programas em Python centralizou as análises nos níveis de repositório e de issue, consolidando as informações em dois artefatos principais. O primeiro reuniu as métricas extraídas do estado original de cada repositório, permitindo estabelecer um panorama inicial dos projetos antes da introdução de novas funcionalidades. O segundo agregou os resultados das análises conduzidas pela ferramenta CK, em nível de classes, os dados coletados pelo SonarQube e um conjunto adicional de métricas calculadas para determinar o impacto associado a cada *issue*.

5. Resultados

Esta seção apresenta os resultados obtidos a partir da análise experimental conduzida em projetos *open source* Java. Os dados foram coletados seguindo a metodologia descrita na Seção 4, visando responder como a geração de código por Inteligência Artificial impacta as métricas de manutenibilidade e a introdução de dívida técnica.

5.1. Visão Geral da Amostra

A análise final consolidou uma amostra de 23 repositórios, após todos os filtros descritas na seção **??** e descartar aqueles que tivessem menos de 5 *issues* válidas. Estes repositórios acumulam 1.045.251 estrelas (média de 45.445) e somavam 25.801 *issues* abertas no momento da coleta. Nesse cenário, o GitHub Copilot foi utilizado para implementar 115 *issues* qualificadas, divididas em cinco por projeto, resultando na avaliação de 281 classes impactadas.

5.2. Análise da Coesão e Estrutura

O primeiro aspecto analisado diz respeito à coesão das classes modificadas pela IA. Vale ressaltar que, devido a limitações técnicas na execução da ferramenta CK em determinados ambientes de *build*, não foi possível extrair as métricas estruturais de 3 dos 23 repositórios da amostra (13%). Assim, as análises estruturais a seguir consideram os dados dos 20 projetos restantes.

Conforme ilustrado na Figura 2, o impacto na métrica LCOM* foi predominantemente neutro. Em 55,7% dos casos, a alteração realizada pela IA não modificou o índice de coesão da classe. Em 27,9% dos casos observou-se uma piora, refletida no aumento do LCOM*, enquanto apenas 16,4% das classes apresentaram melhora, com redução desse valor.

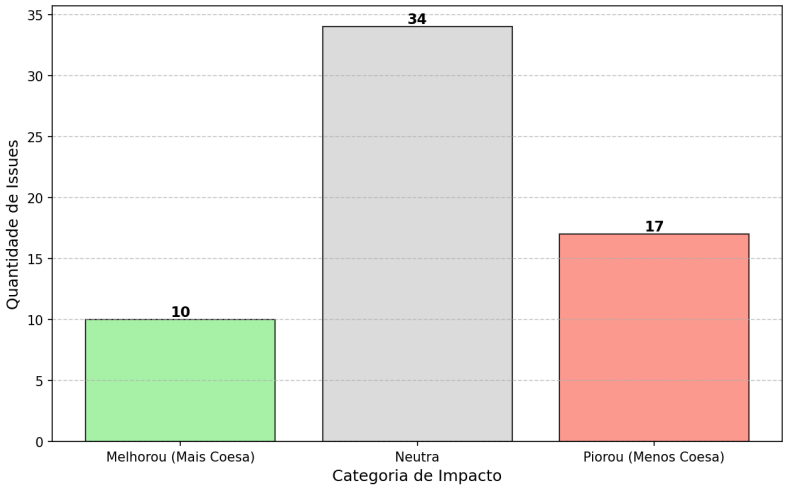

Figura 2. Impacto na coesão (LCOM*) das classes modificadas

O fato de a maioria das alterações serem neutra sugere que a IA, em geral, consegue adicionar funcionalidades sem desestruturar a responsabilidade principal da classe. Contudo, os 27,9% de casos em que a coesão piorou representam um ponto de atenção. Uma classe com baixa coesão tende a violar o rincípio da esponsabilidade única, acumulando responsabilidades distintas [Martin 2009]. Em um projeto real, isso se traduz em código mais difícil de entender, testar e reutilizar, aumentando a complexidade cognitiva para os desenvolvedores que precisarão dar manutenção futura a esse código.

Ao ampliar a análise para as demais métricas estruturais, comparando a variação média (delta) antes e depois de cada implementação, observam-se tendências distintas entre elas. Como a maior parte dessas variações é muito sutil e próxima de zero, uma visualização em escala linear tornaria os padrões praticamente imperceptíveis. Por esse motivo, a Figura 3 utiliza uma escala logarítmica simétrica (symlog), adequada para realçar pequenas diferenças e permitir uma inspeção mais precisa das tendências centrais. Essa abordagem, aliada à remoção de outliers, evidencia o comportamento da mediana e da dispersão de cada métrica, revelando de forma mais clara o impacto real das modificações introduzidas.

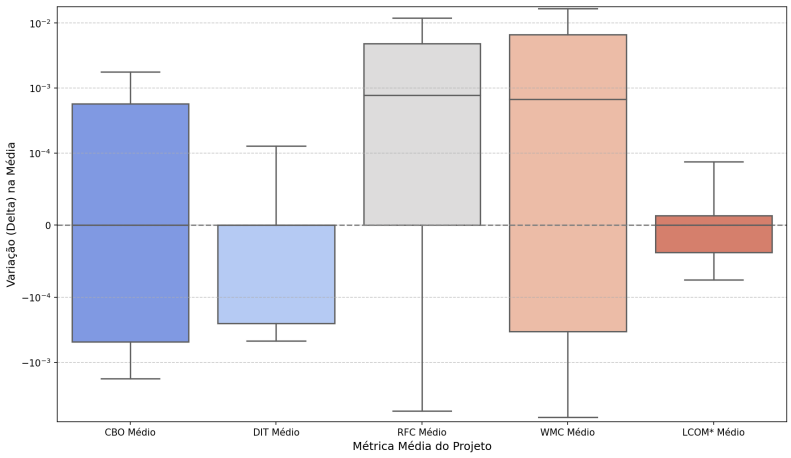

Figura 3. Variação média das métricas CK após implementação (sem outliers)

Observa-se que as métricas DIT e LCOM* tenderam a uma leve melhora ou estabilidade, indicando que o código gerado tende a respeitar a hierarquia de herança existente. No entanto, as métricas de complexidade e acoplamento CBO, RFC e WMC apresentaram uma tendência de piora e uma variação maior. Esse aumento no acoplamento (CBO e RFC) é uma tendência preocupante, pois em um projeto real, classes mais acopladas geram um "*ripple effect*", onde uma alteração em um componente pode exigir modificações em muitos outros, elevando o risco e o custo da manutenção [Riaz et al. 2009]. Similarmente, o aumento da complexidade (WMC) torna as classes mais difíceis de testar e depurar, impactando diretamente o esforço necessário para futuras evoluções do *software* [Sommerville 2019]. Em um projeto real, esse fenômeno se traduz em um aumento progressivo da dificuldade de manutenção, onde futuras alterações se tornam mais arriscadas e onerosas, validando as preocupações de desenvolvedores citadas por Russo. (2024).

5.3. Análise de Code Smells e Dívida Técnica

Além das métricas estruturais, a qualidade do código gerado também foi avaliada por meio da identificação de novos *code smells* detectados pelo SonarQube. Entre as 115 *issues* implementadas, 57 resultaram na introdução de pelo menos um *code smell*. A distribuição da severidade desses problemas, apresentada na Figura 4, revela uma concentração significativa em níveis de maior impacto. A maior parte deles (52,3%) foi classificada como de criticidade *Medium*, enquanto 27,1% se enquadrou na categoria *High*. *Code smells* nesses níveis indicam fragilidades de design que, embora não impeçam a execução do código, comprometem de forma relevante sua manutenibilidade e robustez a longo prazo, aumentando a probabilidade de erros e dificultando futuras modificações [Martin 2009].

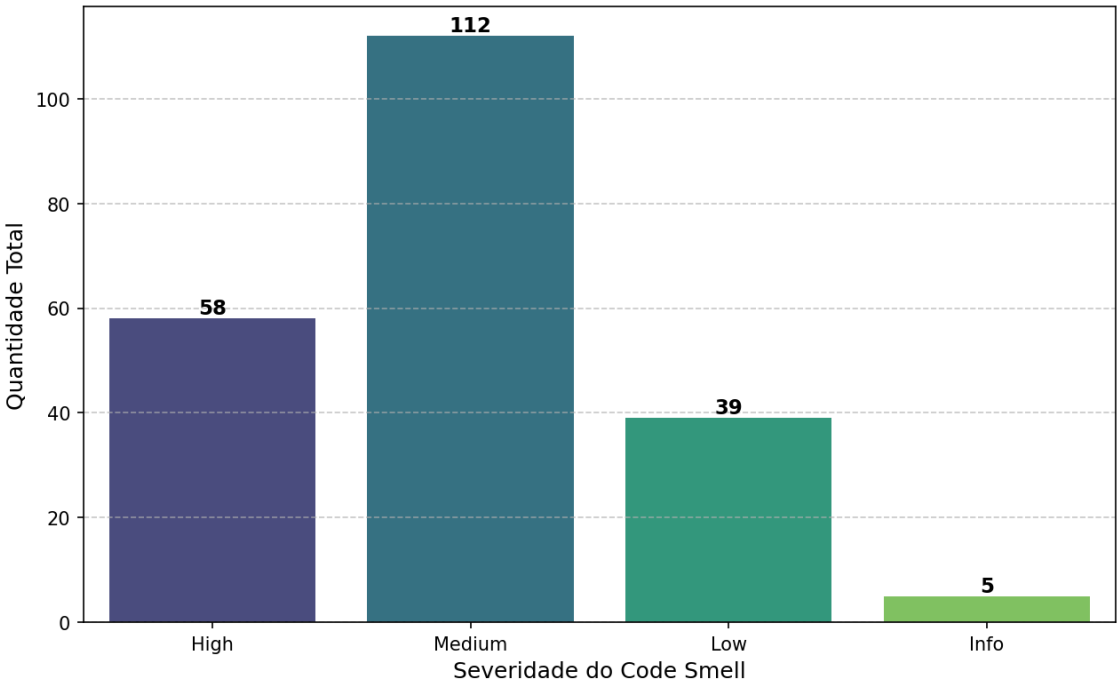

Figura 4. Distribuição da criticidade dos novos Code Smells introduzidos

Por fim, mensurou-se o impacto desses problemas em termos de esforço de correção, uma medida quantificável da dívida técnica introduzida. A Figura 5 ilustra o tempo estimado para remediar os problemas de cada *issue*. A análise indica que, para as *issues* que introduziram débitos, o esforço médio de refatoração é de 38,1 minutos. Embora esse valor unitário possa parecer pequeno, ele funciona como uma espécie de taxa de refatoração associada a cada nova funcionalidade gerada pela IA que introduz *code smells*. Em projetos de grande porte, nos quais centenas de tarefas podem ser aceleradas com o auxílio de IA, o acúmulo desse esforço adicional pode resultar em semanas de trabalho dedicadas exclusivamente à correção de dívida técnica, potencialmente reduzindo de forma significativa os ganhos iniciais de produtividade.

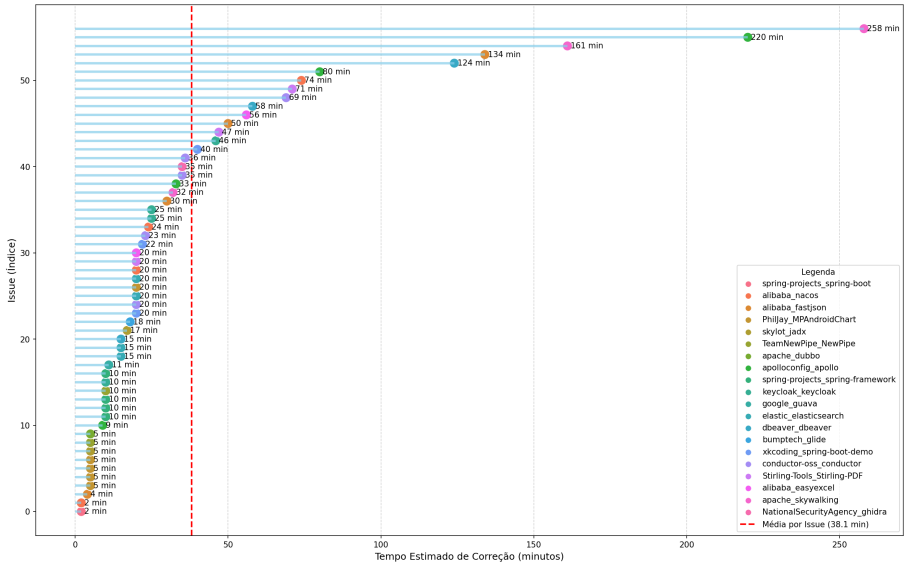

Figura 5. Esforço estimado de remediação por issue (em minutos)

5.4. Discussão dos Resultados

Em síntese, a análise conjunta das métricas revela uma dualidade no comportamento do GitHub Copilot. Enquanto a ferramenta demonstra competência em preservar o escopo local das classes, mantendo a coesão (LCOM*) e a herança (DIT) estáveis, a degradação consistente no acoplamento (CBO, RFC) e na complexidade (WMC) indica que a IA tende a aumentar a interdependência entre componentes. Esse padrão sugere que, no seu estado atual, essas ferramentas atuam com maior eficácia como assistentes de implementação de baixo nível, focadas na resolução imediata de problemas, mas ainda carecem da visão arquitetural necessária para otimizar o design do sistema em uma escala macro, resultando em soluções funcionalmente corretas, porém estruturalmente intrincadas.

Consequentemente, essa limitação arquitetural gera um custo tangível. A alta incidência de *code smells* e o esforço médio de 38 minutos para sua correção configuram um paradoxo de produtividade, pois, embora a geração automática acelere o desenvolvimento, ela também introduz uma dívida técnica oculta que pode transformar o ganho inicial em um passivo de refatoração futura. Nesse cenário, a adoção dessas ferramentas deve vir acompanhada de uma estratégia de governança de código que inclua revisões

humanas rigorosas e monitoramento contínuo, de modo a mitigar a inserção de dívida técnica e preservar a sustentabilidade do *software* a longo prazo.

6. Ameaças à Validade

A condução de um estudo empírico, como o apresentado neste trabalho, envolve decisões metodológicas que podem influenciar a interpretação dos resultados. Esta seção visa discutir, de forma transparente, as potenciais ameaças à validade desta pesquisa, bem como as medidas de mitigação adotadas. As ameaças são categorizadas em validade de construção, interna, externa e de conclusão.

Em relação à validade de construção, os resultados obtidos podem não refletir plenamente o impacto real da IA em um cenário completo de desenvolvimento de software. Isso ocorre porque a análise considerou apenas aspectos estruturais do código, como coesão, acoplamento, complexidade e *code smells*, sem verificar se as implementações estavam corretas ou completas do ponto de vista funcional. Assim, um código estruturalmente adequado pode ainda assim falhar em atender aos requisitos descritos nas *issues*. Além disso, o estudo restringiu-se exclusivamente ao código-fonte Java gerado, desconsiderando artefatos frequentemente envolvidos na implementação de funcionalidades em projetos reais, como ajustes em arquivos de *build*, configurações de ambiente ou integração com testes. Essa delimitação simplifica o processo de implementação analisado e pode limitar a representação fiel das atividades de desenvolvimento como um todo. Ainda assim, o estudo abrangeu 115 implementações distribuídas em 23 repositórios distintos, proporcionando uma base consistente para observar padrões estruturais recorrentes no código produzido pela IA.

Em relação à validade interna, os resultados podem ser influenciados pelas interações necessárias para garantir que a IA produzisse código adequado ao contexto do estudo. Durante a fase de implementação, observou-se que a ferramenta ocasionalmente gerava código em linguagens incorretas, especialmente em repositórios que combinavam Java com Kotlin, YAML ou XML, ou delegava decisões de implementação ao usuário. Para assegurar a consistência do procedimento, intervenções pontuais foram realizadas, orientando a ferramenta a gerar exclusivamente código Java e a assumir as decisões necessárias para completar cada funcionalidade. Em situações em que o comportamento inadequado persistia, uma nova conversa com contexto limpo era iniciada. Embora essas intervenções possam parecer um fator de interferência, elas refletem um cenário realista de uso, no qual o desenvolvedor atua como supervisor ativo do processo. Assim, o desenho experimental buscou reproduzir de forma fiel a dinâmica prática de interação com ferramentas de IA, mantendo o foco e a relevância das implementações geradas.

Já no contexto da validade externa, este estudo foi conduzido utilizando exclusivamente o GitHub Copilot (baseado no modelo GPT-5 mini) e a linguagem Java. Os resultados podem não ser generalizáveis para outras ferramentas de geração de código, como Amazon CodeWhisperer ou Tabnine, ou para outras linguagens de programação, especialmente aquelas de paradigmas diferentes (como funcionais ou de *script*). Já a amostra de repositórios, embora composta por projetos populares e maduros de código aberto, não representa a totalidade do ecossistema de *software*. Os resultados podem variar em projetos de código fechado, em domínios de negócio específicos, ou em projetos com diferentes níveis de maturidade arquitetural e padrões de codificação.

7. Conclusão e Trabalhos Futuros

Este trabalho se propôs a investigar os impactos da utilização de ferramentas de Inteligência Artificial, especificamente o GitHub Copilot, na manutenibilidade de *software*. Por meio de um estudo quantitativo aplicado em 23 repositórios Java de código aberto, onde 115 novas funcionalidades foram implementadas a partir de *issues*, foi possível mensurar a variação em métricas de qualidade de código e a introdução de dívida técnica. Os resultados revelam um cenário complexo, que aponta tanto para a capacidade da IA de se integrar a arquiteturas existentes quanto para os riscos associados à sua adoção acrítica.

A análise demonstrou que o código gerado por IA tende a preservar a estrutura de herança (DIT) e a coesão interna das classes (LCOM*) em um nível majoritariamente neutro. Este achado sugere que, para modificações localizadas e bem definidas, a IA é capaz de produzir código que se acopla adequadamente às responsabilidades existentes de uma classe. Contudo, o mesmo não se aplica às métricas de acoplamento (CBO e RFC) e complexidade (WMC), que apresentaram uma clara tendência de deterioração. O aumento dessas métricas indica que o código gerado, embora funcional, tende a criar mais dependências entre classes e a possuir uma lógica interna mais intrincada.

Corroborando essa tendência, a análise de *code smells* revelou que uma parcela das implementações introduziu novos débitos técnicos, com uma predominância de problemas de criticidade média e alta. A presença desses *smells* não é apenas um indicador de má qualidade de design, mas representa um custo futuro concreto. O esforço médio de remediação de 38,1 minutos por *issue* problemática quantifica essa dívida, funcionando como uma ”taxa”de refatoração que pode, em larga escala, neutralizar os ganhos de produtividade almejados. Este resultado está alinhado com as descobertas de Siddiq et al. (2022), que também identificaram a propagação de *code smells* em código gerado por IA, reforçando a necessidade de uma revisão humana criteriosa.

Conclui-se que a utilização de IA para geração de código gera um paradoxo, pois, embora acelere o desenvolvimento de novas funcionalidades, também pode introduzir uma dívida técnica oculta que compromete a manutenibilidade a longo prazo. As ferramentas de IA, no seu estado atual, atuam com maior eficácia como assistentes de implementação de baixo nível, mas ainda carecem da visão arquitetural necessária para otimizar o design do sistema em uma escala macro. A decisão de adotá-las deve, portanto, ser acompanhada de uma estratégia de governança de código que inclua revisões rigorosas e um monitoramento contínuo das métricas de qualidade aqui analisadas.

Trabalhos futuros devem comparar os resultados com ferramentas como Amazon CodeWhisperer e Tabnine, verificando se os impactos são exclusivos do GitHub Copilot. Sugere-se também investigar linguagens como Python e JavaScript de diferentes paradigmas e avaliar se o uso de instruções detalhadas, considerando restrições arquitetônicas e de qualidade, é capaz de mitigar *code smells* e melhorar a manutenibilidade das soluções.

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-vitor-lany`

Referências

Aniche, M. (2016). Ck: Code metrics for java code by means of static analysis. `https://github.com/mauricioaniche/ck`. Acessado em: 18 de maio de 2025.

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.

Devi, A., Charaya, S., and Maan, M. (2023). Factors influencing software maintainability prediction using soft computing. In *2023 International Conference on Artificial Intelligence and Smart Communication (AISC)*, pages 1390–1394.

Fenton, N. E. and Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 3rd edition.

Goel, V. and Kaur, A. (2025). Software maintainability datasets collection across android and apache kafka versions. *Procedia Computer Science*, 258:3944–3957. International Conference on Machine Learning and Data Engineering.

Guo, D. e. a. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

Martin, R. C. (2009). *Código Limpo: Habilidades Práticas do Agile Software*. Alta Books, Rio de Janeiro.

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, Pittsburgh, PA, USA.

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)*, pages 103–107, Lisbon, Portugal.

Riaz, M., Mendes, E., and Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, Lake Buena Vista, FL, USA.

Russo, D. (2024). Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(5).

Siddiq, M. L., Majumder, S. H., Mim, M. R., Jajodia, S., and Santos, J. C. S. (2022). An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–82.

Sommerville, I. (2019). *Engenharia de Software*. Pearson, São Paulo, 10 edition.

Times, T. (2025). Big us tech firms to invest $300bn in ai infrastructure – amazon, google, microsoft, meta. *The Times*.