Avaliação Empírica do Impacto da IA Generativa na Eficiência de Depuração de Algoritmos

Benício Mozan Santo Vale[1], Osvaldo Tavares Viana Junior[1]

Escola Superior de Tecnologia – Universidade do Estado do Amazonas (UEA) Manaus - AM – Brazil

{bmsv.snf23,ojunior}@uea.edu.br

Abstract. *This paper investigates the impact of Generative Artificial Intelligence (GAI) on the debugging efficiency of novice to intermediate programmers. Through a controlled experiment with 20 participants, we compared a group using an AI assistant (Gemini) against a control group coding manually. Metrics included implementation time, debugging time, code cyclomatic complexity (measured via Radon), success rate and user perception (via post-task survey). Results indicate that while AI reduces initial coding time by approximately 67%, it significantly increases the relative time spent on debugging and the structural complexity of the code. The findings suggest a cognitive distance phenomenon, understood in this study as a reduced intellectual connection between the programmer and the program logic when the initial construction of the solution is delegated to AI, which hinders the comprehension and correction of logical errors.*

Resumo. *Este artigo investiga o impacto da Inteligência Artificial Generativa (IAG) na eficiência de depuração de programadores iniciantes e intermediários. Por meio de um experimento controlado com 20 participantes, comparou-se um grupo utilizando um assistente de IAG (Gemini) contra um grupo de controle (manual). As métricas incluíram tempo de implementação, tempo de depuração, complexidade ciclomática (medido por meio do Radon), taxa de sucesso e percepção subjetiva de dificuldade (via questionário pós-tarefa). Os resultados indicam que, embora a IAG reduza o tempo inicial de codificação em cerca de 67%, ela aumenta significativamente o tempo relativo gasto na depuração e a complexidade estrutural do código. Os achados sugerem um fenômeno de distanciamento cognitivo, entendido neste estudo como a redução do vínculo do programador com a lógica da solução quando a construção inicial do código é delegada à IAG, o que dificulta a compreensão e a correção de erros lógicos.*

1. Introdução

A engenharia de *software* contemporânea atravessa um ponto de inflexão, impulsionado pela integração acelerada de ferramentas de Inteligência Artificial Generativa (IAG) aos fluxos de desenvolvimento. Soluções como GitHub Copilot e Gemini prometem ganhos expressivos de produtividade e também vêm sendo utilizadas como apoio à aprendizagem, ao oferecer explicações conceituais e exemplos de código (Silva Junior et al., 2023). Contudo, essa delegação da escrita de código introduz um paradoxo: ao externalizar a etapa de codificação, corre-se o risco de promover um “distanciamento cognitivo” entre o desenvolvedor e a lógica subjacente do sistema. Neste estudo, o termo

designa a redução do vínculo intelectual do estudante com a lógica, a estrutura e o propósito do código quando a solução inicial é produzida por uma ferramenta generativa. Embora Philbin (2023) não utilize essa expressão literalmente, sua discussão oferece base conceitual para o fenômeno ao defender que aprender a programar é uma atividade de alta carga cognitiva, dependente da construção de modelos mentais robustos e da compreensão efetiva dos programas. Nessa perspectiva, quando a IAG fornece uma solução pronta e reduz excessivamente o esforço cognitivo inicial, o estudante pode concluir a tarefa mais rapidamente, mas sem consolidar compreensão suficiente para validar, explicar e depurar a lógica posteriormente, fato apontado também pelo experimento controlado apresentado no trabalho de Kazemitabaar et al. (2023).

A habilidade de depuração (*debugging*) constitui um exercício investigativo complexo, que exige do programador a construção de um modelo mental robusto sobre o funcionamento do *software*. Nesse sentido, a literatura recente alerta que a facilidade da geração automática de sintaxe pode atrofiar competências analíticas fundamentais (Bang e Dang, 2024) e tornar a manutenção de *software* mais custosa a longo prazo pela introdução de complexidade estrutural desnecessária e *code smells*, isto é, sinais recorrentes no código que indicam problemas de legibilidade, projeto ou manutenção, como redundâncias, funções excessivamente complexas e estruturas desnecessárias (Yetiştiren et al., 2023). Estudos preliminares indicam ainda um viés de automação, no qual desenvolvedores tendem a superestimar a correção de códigos gerados por IAG, resultando em sessões de depuração mais longas e frustrantes devido à falta de familiaridade com a estrutura gerada (Vaithilingam et al., 2022; Hashmi et al., 2024).

Neste contexto, o presente trabalho apresenta os resultados de um experimento controlado realizado com 20 estudantes da área de computação. Fundamentado em dados preliminares de projetos de iniciação científica, o estudo visa preencher a lacuna de evidências empíricas sobre o tema. O objetivo central é avaliar, de forma quantitativa e qualitativa, se o uso de código gerado por IAG penaliza o desempenho humano na correção de falhas e se impacta negativamente a qualidade estrutural da solução final entregue.

2. Trabalhos Relacionados

A investigação sobre a interação entre humanos e IAG na programação tem crescido, fornecendo a base teórica para este estudo. Nguyen e Nadi (2022) avaliaram empiricamente a qualidade das sugestões do GitHub Copilot utilizando problemas algorítmicos. Seus resultados indicaram que a ferramenta apresenta dificuldades significativas com a complexidade lógica, gerando códigos com baixa taxa de acerto em diversas linguagens e falhando frequentemente em testes de unidade. Este achado corrobora a premissa de que a geração automática não implica necessariamente em código correto ou livre de falhas.

Yetiştiren et al. (2023) focaram na qualidade do *software* gerado por IAG, analisando métricas de manutenibilidade e a presença de *code smells*. A pesquisa demonstrou que as ferramentas de IAG tendem a violar boas práticas, adicionando variáveis mal nomeadas e funções redundantes, o que corrobora a hipótese de que o aumento da produtividade pode ocorrer às custas da qualidade do código.

Complementarmente, Vaithilingam et al. (2022) ofereceram uma perspectiva qualitativa ao relatar que depurar código de IAG é comparável a "depurar o código de

outra pessoa". Segundo os autores, isso ocorre porque, ao não participar da construção lógica inicial, o programador é forçado a realizar uma engenharia reversa mental para compreender as sugestões. Esse processo aumenta a carga cognitiva e interrompe o fluxo de desenvolvimento, uma vez que a validação da corretude exige escrutínio detalhado de trechos que parecem plausíveis, mas podem conter falhas sutis.

Contrapondo a literatura inicial (Nguyen e Nadi, 2022), Licorish et al. (2025) demonstraram que códigos gerados por LLMs[1] apresentam complexidade ciclomática superior às soluções humanas quando avaliados via Radon[2]. Os autores atribuem esse fenômeno ao *over-engineering* e à inclusão de estruturas redundantes, o que corrobora diretamente os dados aqui apresentados sobre a verbosidade do código gerado por IAG.

Para sistematizar o posicionamento deste estudo frente à literatura existente, a Tabela 1 apresenta uma síntese comparativa dos trabalhos relacionados. A Tabela destaca os focos e conclusões predominantes nas pesquisas anteriores e evidencia os diferenciais metodológicos desta investigação, especificamente a união de métricas de engenharia de *software* com a análise do esforço humano no contexto do Google Gemini.

Tabela 1. Comparativo de Trabalhos Relacionados e Diferenciais da Pesquisa

Estudo	**Foco Principal**	**Conclusão Relevante**	**Diferencial Desta Pesquisa**
Nguyen e Nadi (2022)	Corretude Funcional (Copilot)	A ferramenta falha frequentemente em testes de unidade e lógica.	Foca no esforço humano para corrigir os erros, não apenas na falha da máquina.
Vaithilingam et al. (2022)	Fatores Humanos (Qualitativo)	Depurar código de IAG é cognitivamente custoso ("código de terceiros").	Quantifica essa dificuldade através de métricas de tempo (Td[3]/Tt[4]) em um experimento controlado.
Yetiştiren et al. (2023)	Qualidade de Código	IAG introduz Dívida Técnica (*Technical Debt*) e *Code Smells*.	Avalia o impacto prático dessa dívida técnica na eficiência de programadores iniciantes.
Licorish et al. (2025)	Complexidade Estrutural	IAG tende ao *over-engineering*, gerando maior Complexidade Ciclomática.	Valida o fenômeno no Google Gemini e correlaciona a complexidade com a queda de desempenho.
Este trabalho	Eficiência de Depuração	Distanciamento Cognitivo: Ganho na escrita é penalizado pela depuração.	Único a unir análise estática (Radon) com métricas de tempo e percepção no Gemini.

[1] *Large Language Model*
[2] Ferramenta de código aberto para análise estática em Python. Documentação oficial disponível em: https://radon.readthedocs.io/
[3] Tempo de depuração.
[4] Tempo total.

3. Metodologia

Esta pesquisa foi estruturada sob a óptica de uma pesquisa experimental controlada com abordagem mista (quantitativa e qualitativa), visando estabelecer uma relação de causa e efeito entre o uso do assistente de código Gemini e a eficiência de depuração. A escolha experimental justifica-se pela necessidade de isolar o uso da IAG, a variável independente, e observar seu impacto direto sobre as variáveis dependentes: tempo de implementação, tempo de depuração e complexidade ciclomática do código resultante. Diferentemente de estudos observacionais, esse delineamento permite atribuir diferenças de desempenho à ferramenta com maior grau de confiança, e não a fatores externos ou aleatórios.

A natureza híbrida da pesquisa foi adotada para capturar o fenômeno estudado em duas dimensões complementares: a vertente quantitativa foca na mensuração objetiva de métricas de engenharia de *software*, enquanto a qualitativa investiga a percepção de dificuldade e a confiança na solução gerada, elementos cruciais para validar hipóteses cognitivas como o “distanciamento cognitivo”. Para garantir a validade interna, o estudo foi conduzido em ambiente laboratorial controlado, com protocolo padronizado que abrange desde a seleção e nivelamento dos participantes até a coleta de dados e a aplicação das métricas de avaliação, descritas nas subseções 3.1 a 3.5.

3.1. Participantes

A amostra foi composta por 20 estudantes de cursos de graduação em Computação da UEA, onde cada um dos participantes assinou o Termo de Consentimento Livre e Esclarecido (TCLE), garantindo a ética da pesquisa, a voluntariedade da participação e o anonimato dos dados coletados. Os participantes foram selecionados por conveniência e nivelados para garantir um perfil de conhecimento de iniciante a intermediário. A mediana de idade da amostra foi de 21 anos, sendo 12 participantes das fases iniciais (até o 2º período, classificados como iniciantes) e 8 das fases intermediárias (3º ao 4º período). Embora essa distribuição não comprometa a validade da análise, a predominância de iniciantes deve ser considerada na interpretação dos resultados.

Destaca-se, contudo, que a amostra não se apresentou perfeitamente balanceada entre os níveis de experiência, com predominância de participantes nas fases iniciais do curso. Embora essa distribuição não comprometa a validade da análise realizada, ela constitui um fator relevante de contexto e deve ser considerada na interpretação dos resultados, uma vez que a experiência prévia em programação pode influenciar o desempenho nas tarefas propostas.

3.2. Estrutura Experimental e Tarefas

As tarefas propostas consistiram na implementação de dois algoritmos fundamentais da ciência da computação: um gerador da sequência de Fibonacci e um verificador de números primos. Os participantes foram divididos em dois grupos de 10 indivíduos, onde a Tabela 2 apresenta a descrição de cada grupo. A seleção destes problemas não foi aleatória, mas estratégica para o experimento. Ambos exigem o domínio de estruturas de repetição (*loops*) e condicionais aninhadas, além de apresentarem casos de borda (*edge cases*) críticos.

Tabela 2. Descrição dos Grupos Experimentais

Grupo	Descrição
Grupo Experimental (Com IAG)	Utilizou a ferramenta Gemini para gerar o esboço inicial do código, sendo responsável por testar e depurar a solução.
Grupo de Controle (Sem IAG)	Desenvolveu e depurou a solução manualmente, sem acesso a assistentes de codificação.

Para a sequência de Fibonacci, a ambiguidade na definição do ponto de partida (índice 0 ou 1) e o tratamento de entradas negativas são fontes comuns de erro. Já na verificação de primos, a necessidade de tratar números como 1 (não primo), 2 (único par primo) e números negativos desafia a capacidade de generalização do aluno. Essas nuances são frequentemente onde as ferramentas de IAG alucinam ou geram código tecnicamente funcional, mas que falha na especificação exata do teste, forçando o estudante a uma depuração minuciosa que expõe o distanciamento cognitivo.

3.3. Ambientes e ferramentas

A execução das tarefas ocorreu em laboratórios da UEA, onde todos os computadores disponibilizados aos participantes possuíam especificações de *hardware* e *software* idênticas, padronização fundamental para eliminar diferenças de desempenho da máquina como fatores de confusão nos resultados cronometrados.

No que tange ao ambiente de desenvolvimento de *software* (IDE), padronizou-se o uso do Visual Studio Code (VS Code) para todos os participantes. A escolha dessa ferramenta justifica-se por sua ampla adoção no mercado e pela familiaridade prévia dos estudantes, evitando que a curva de aprendizado da interface gráfica influenciasse negativamente o tempo de implementação (T_i). A linguagem de programação adotada foi o Python (versão 3.12), selecionada devido à sua sintaxe limpa e legibilidade, características que facilitam a análise da lógica algorítmica por programadores iniciantes e intermediários, conforme o perfil da amostra.

Para o Grupo Experimental, a ferramenta de Inteligência Artificial Generativa utilizada foi o Google Gemini 2.5 Pro, acessado por meio da interface *web* padrão. Sua adoção no experimento se justifica pela disponibilidade concreta no contexto institucional da pesquisa, uma vez que todos os estudantes participantes possuíam acesso ao plano Pro em decorrência de uma parceria com a UEA, bem como por sua relevância técnica em tarefas de geração de código, justificado através de resultados expressivos em *benchmarks* amplamente utilizados, como *LiveCodeBench*, *Aider Polyglot* e *SWE-bench Verified* (Comanici et al., 2025).

Para a coleta e análise das métricas de qualidade de código, utilizou-se a biblioteca Radon para Python. Esta ferramenta calcula a Complexidade Ciclomática (CC), definida a partir do índice de McCabe[5], que fundamentalmente atua como um indicador inverso de qualidade: quanto maior o índice, maior o esforço necessário para testes e maior a probabilidade de erros.

[5] Métrica clássica de engenharia de *software* que estima o número de caminhos linearmente independentes no fluxo de controle do programa e, por consequência, o esforço potencial de teste e manutenção (McCabe, 1976).

A classificação automatizada do Radon divide a complexidade em uma escala de seis níveis de risco, que vai de A a F: o *Rank A* (1–5) e o *Rank B* (6–10) indicam complexidade baixa a moderada, enquanto os níveis subsequentes C (11–20), D (21–30), E (31–40) e F (41+) sinalizam complexidade alta a muito alta, alertando para trechos de código instáveis e críticos para manutenção. A execução foi realizada via linha de comando (CLI) após a finalização de cada tarefa, gerando relatórios comparativos entre os grupos.

Por fim, a mensuração dos tempos — Tempo de Implementação (Ti) e Tempo de Depuração (Td) — foi realizada mediante cronometragem assistida. O início da contagem deu-se no momento da leitura do enunciado da tarefa e a transição para a fase de depuração foi marcada pela primeira execução do código (seja ele manual ou gerado pela IAG) que resultasse em erro ou saída incorreta. O encerramento do Tempo Total (Tt) ocorreu somente após a validação manual da solução pelos pesquisadores. Adotou-se um protocolo rigoroso de verificação (*human-in-the-loop*), onde os autores submetiam o código do participante a uma bateria padronizada de entradas de teste diretamente no terminal da IDE. Este processo manual garantiu que não apenas a saída final fosse correta, mas que a lógica não contivesse 'gambiarras' ou falsos positivos que poderiam passar despercebidos em *scripts* automáticos simples. Foram testados, presencialmente, casos de sucesso (entradas padrão), casos de borda (ex: n=0, n=1, n=2) e tratamento de erros (ex: entradas negativas), sendo a tarefa considerada concluída apenas após a aprovação em todos os critérios deste roteiro pré-definido.

Cada participante teve um tempo máximo de 30 minutos por tarefa para concluir a atividade. Caso a solução não atendesse aos critérios de validação dentro desse limite, a tarefa era encerrada e registrada com pendência de correção.

3.4. Instrumento de Coleta de Dados Qualitativos e Perceptivos

Além das métricas automáticas de código e tempo, foi aplicado um questionário estruturado pós-experimento (*Post-Task Survey*) para capturar a dimensão subjetiva do “distanciamento cognitivo”. O instrumento foi desenhado para ser respondido imediatamente após a conclusão das tarefas, mitigando o viés de memória.

O formulário foi dividido em duas seções: (i) Avaliação Quantitativa Subjetiva, utilizando escala Likert de 5 pontos para mensurar dificuldade, confiança e esforço mental percebido; e (ii) Questões Abertas, destinadas a coletar relatos detalhados sobre os obstáculos enfrentados durante a etapa de depuração. A Tabela 3 detalha a estrutura das perguntas aplicadas aos participantes do Grupo Experimental (Com IAG) e do Grupo de Controle (Manual), com adaptações contextuais onde necessário.

Tabela 3. Estrutura do Questionário de Percepção Pós-Tarefa

ID	Pergunta / Enunciado	Tipo de Resposta	Objetivo da Métrica
Q1	De 1 a 5, quão difícil foi completar a tarefa proposta?	Escala Likert (1: Muito Fácil a 5: Muito Difícil)	Mensurar a Dificuldade Percebida global da tarefa.
Q2	De 1 a 5, quão difícil foi depurar o código da tarefa proposta?	Escala Likert (1: Muito Fácil a 5: Muito Difícil)	Mensurar a dificuldade de entender a lógica depois de ter feito o código.

Q3	Você sentiu que tinha controle total sobre o fluxo do algoritmo? Justifique.	Texto aberto (qualitativo)	Investigar a sensação de Autoria e Controle.
Q4	Descreva o maior obstáculo encontrado durante a fase de correção de erros.	Texto aberto (qualitativo)	Identificar gargalos específicos (ex: sintaxe vs. lógica).
Q5	(Apenas Grupo IAG) O quanto você sentiu que entendeu a lógica gerada pela IAG antes de testá-la?	Escala Likert (1: Não entendi nada a 5: Entendi completamente)	Avaliar o nível de Compreensão Lógica prévia à execução.

As respostas abertas das questões Q3 e Q4 foram categorizadas manualmente pelos pesquisadores, sem apoio de IAG, por meio de leitura independente, consolidação dos temas recorrentes e posterior agrupamento em categorias semânticas. Essa decisão visou evitar que a própria ferramenta avaliada interferisse na interpretação qualitativa dos dados. As respostas foram categorizadas em grupos semânticos (ex: “Dificuldade de Sintaxe”, “Lógica Obscura”, “Excesso de Confiança”) para correlacionar com os tempos de depuração registrados cronometricamente. Essa triangulação de dados permitiu verificar se a percepção subjetiva dos alunos condizia com a complexidade ciclomática aferida pelo Radon, além de permitir cruzar os relatos de “falta de controle” com os tempos de depuração registrados.

3.5. Métricas de Avaliação

Para quantificar o desempenho e a experiência, foram definidas as seguintes métricas:

- Tempo de Implementação (T_i): Minutos cronometrados até a obtenção da primeira versão funcional (livre de erros de sintaxe).
- Tempo de Depuração (T_d): Minutos dedicados exclusivamente à correção de falhas lógicas após a primeira execução.
- Tempo Total (T_t): Soma de $T_i + T_d$.
- Taxa de Sucesso: Porcentagem de casos de teste aprovados.
- Complexidade Ciclomática (CC): Medida da complexidade estrutural (caminhos independentes), calculada automaticamente via biblioteca Radon.
- Dificuldade Percebida: Escala Likert (1-5) respondida após o experimento.

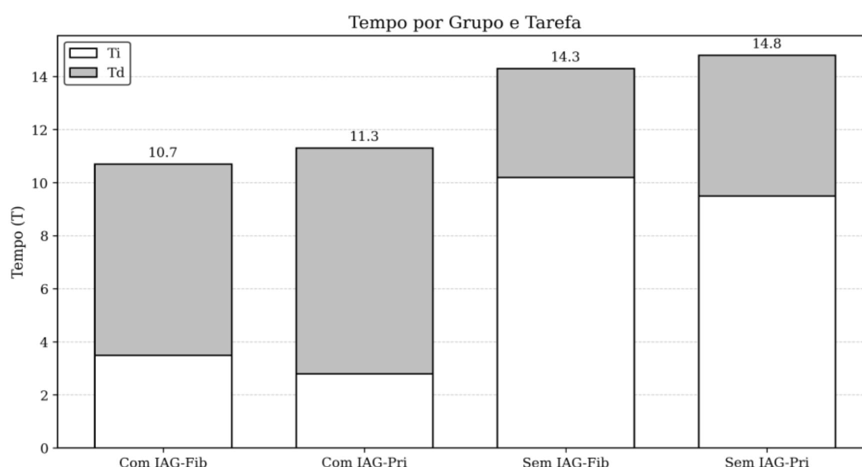
4. Resultados e análise

Nesta seção, apresentamos e discutimos os dados consolidados do experimento controlado com os 20 participantes. Para efeitos de comparação, mantivemos a divisão por tarefa e grupo, destacando médias e razões entre tempos de implementação e de depuração.

4.1. Análise de Tempo e Eficiência

A Figura 1 resume os tempos médios obtidos pelos dois grupos nas tarefas combinadas. Os dados revelam uma assimetria notável. O grupo com IAG foi significativamente mais rápido na implementação (T_i), com redução de aproximadamente 65–70 % no tempo de escrita inicial em relação ao grupo comparado. No entanto, o tempo gasto em depuração (T_d) foi superior no grupo com IAG. Observa-se que, para o grupo com IAG, a depuração consumiu a maior parte do tempo total (entre 67 % e 75 %), enquanto no grupo manual a depuração representou apenas cerca de um terço do tempo.

Figura 1. Comparativo de Tempos Médios (em minutos)



Nesta rodada, a redução do tempo total (T_t) foi de aproximadamente 24%, o que demonstra que a alta eficiência na implementação (cerca de 67%) foi compensada de forma mais significativa pelo esforço ampliado na depuração do que o observado preliminarmente. Além disso, observa-se que as tarefas de verificação de números primos exigiram mais tempo de depuração em ambos os grupos, sugerindo uma maior complexidade lógica em comparação com a geração da sequência de Fibonacci.

4.2. Qualidade do Código e Taxa de Sucesso

A Tabela 4 apresenta as métricas de qualidade aferidas. A análise via Radon alinha-se às observações de Yetiştiren et al. (2023) sobre a baixa manutenibilidade do código gerado por máquinas.

Tabela 4. Qualidade do Código e Sucesso

Grupo	Taxa de Sucesso (%)	Complexidade Ciclomática (Radon)	Erros Lógicos Residuais
Com IAG	85%	6,4 (*Rank B*)	1,2
Sem IAG	95%	3,1 (*Rank A*)	0,3

O indicador de erro lógico residual corresponde à média de falhas de lógica que permaneceram após a etapa de depuração, identificadas durante a bateria final de validação aplicada pelos pesquisadores. Para cada solução, contabilizou-se o número de comportamentos incorretos observados nos casos de teste padronizados (incluindo casos de borda e tratamento de entradas inválidas). Em seguida, calculou-se a média desse total dentro de cada grupo, o que resultou em 1,2 erros residuais no grupo com IAG e 0,3 no grupo sem IAG.

O código gerado com auxílio de IAG apresentou uma complexidade ciclomática média de 6,4, quase o dobro do código manual (3,1). A inspeção das soluções evidenciou

que a IAG frequentemente sugeriu estruturas de controle redundantes, incluindo condicionais e laços aninhados desnecessários, que aumentaram o número de caminhos independentes. Ademais, estudos em larga escala que comparam código gerado por *LLMs* com código humano corroboram que a IAG é mais propensa à inserção de defeitos e vulnerabilidades de alta criticidade (Cotroneo et al., 2025).

Além disso, a taxa de sucesso foi menor no grupo com IAG (85 % versus 95 %), e os poucos erros lógicos residuais encontrados após a depuração concentraram-se em falhas de lógica sutis, como erros na definição de condições de parada e manipulação inadequada de índice. A relação entre complexidade ciclomática e número de erros sugere que soluções mais complexas tendem a abrigar mais oportunidades de defeito, o que corrobora o alerta de que produtividade não pode ser o único critério para avaliar a utilidade de assistentes de IAG.

4.3. Dificuldade Percebida

A aplicação do instrumento detalhado na Seção 3.4 permitiu uma análise sobre onde reside a verdadeira complexidade da programação assistida. Os dados consolidados das respostas quantitativas e a síntese dos relatos abertos estão apresentados na Tabela 5.

Tabela 5. Comparativo de Dificuldade Percebida e Relatos (médias)

Grupo	Dificuldade da tarefa (1-5)	Dificuldade de depuração (1-5)	Principais Relatos
Com IAG	2,1	3,8	Incluíram dificuldade em entender a lógica gerada.
Sem IAG	3,5	2,5	Maior controle sobre a estrutura do código.

Ao analisar Q1 e Q2, observa-se um paradoxo interessante no Grupo com IAG: embora tenham considerado a tarefa global mais fácil (média 2,1) devido à geração rápida do código, reportaram uma dificuldade de depuração significativamente maior (3,8) do que o Grupo Manual (2,5). Esse dado confirma estatisticamente que a IAG transfere a carga de trabalho da escrita para a correção, onde o estudante é menos eficiente.

A questão exclusiva do grupo experimental (Q5) elucidou a causa raiz desse gargalo. Houve uma correlação direta entre a baixa compreensão prévia e o tempo gasto: participantes que responderam “1” ou “2” em compreensão da lógica gerada foram os responsáveis pelos maiores tempos de depuração (T_d). Isso valida a visão de Hashmi et al. (2024) sobre a frustração gerada pela falta de domínio do código.

Nas questões qualitativas Q3 (Controle) e Q4 (Obstáculos), a disparidade na sensação de autoria foi evidente. Enquanto o grupo manual justificou ter controle total por ter “escrito cada linha de código”, os relatos do grupo com IAG em Q4 frequentemente citaram, de forma sintetizada, “variáveis desconhecidas” e “funções bagunçadas” como maiores obstáculos. Essa percepção de falta de controle alinha-se perfeitamente com a metáfora de “depurar código de terceiros” de Vaithilingam et al. (2022) e reforça que a aceitação passiva da sugestão da IAG (Velho, 2025) cria um distanciamento cognitivo que penaliza a etapa de correção de erros.

5. Considerações Finais

Os resultados corroboram a hipótese do “distanciamento cognitivo”: a eficiência inicial da IAG gera uma ilusão de competência que se dissolve na depuração. A elevada complexidade ciclomática no grupo com IAG (média de 6,4 vs. 3,1 no manual) indica que ferramentas generativas priorizam a eficácia sintática imediata em detrimento da legibilidade, resultando em *over-engineering* e estruturas redundantes, risco também alertado por Licorish et al. (2025). Embora haja um ganho de 67% na velocidade de implementação, este não compensa o esforço desproporcional de correção. Estabelece-se um paradoxo onde a ferramenta insere uma “taxa cognitiva” adicional, alienando o programador da lógica do próprio código e reproduzindo a sensação de “depurar o código de outra pessoa” (Vaithilingam et al., 2022).

Diante deste cenário, os achados sugerem uma revisão curricular: o ensino tradicional, focado na sintaxe e construção “do zero”, mostra-se insuficiente. O foco deve deslocar-se para a leitura, interpretação e validação lógica. Sem mediação pedagógica, o uso de IAG pode favorecer a atrofia de competências analíticas fundamentais (Bang e Dang, 2024). Assim, a IAG deve ser incorporada através de estratégias que preservem a compreensão lógica, como atividades de explicação de código e identificação de casos de borda. Como reforça Velho (2025), a integração da IAG exige postura crítica, sob pena de transformar o aluno num mero “operador de *prompt*” incapaz de validar a eficiência da solução.

Recomenda-se a transição para metodologias ativas que combatam a passividade, como a “engenharia reversa assistida” (gerar código via IAG e comentá-lo linha a linha) e exercícios de refatoração para simplificar a complexidade de sugestões do Gemini. Conclui-se que, embora a IAG acelere a produção bruta, ela deve ser posicionada como tutor, e não como executor. Para assegurar que os fundamentos da lógica de programação sejam de fato assimilados, recomenda-se que as avaliações somativas ocorram em ambientes controlados e sem o auxílio de geradores de código. Tal medida visa garantir que o estudante exercite sua autonomia cognitiva e capacidade de resolução de problemas sem 'muletas' tecnológicas. Por outro lado, o potencial da IAG deve ser explorado na fase de aprendizagem como uma ferramenta de apoio conceitual (Silva Junior et al., 2023), atuando como um tutor capaz de explicar teorias e esclarecer dúvidas sintáticas, desde que o aluno permaneça como o autor principal da solução lógica.

Este estudo apresenta limitações, como a amostra reduzida (20 participantes) e a ausência de estratificação por nível de experiência, o que limita a precisão do impacto sobre tempos de execução e qualidade do código. Além disso, o foco exclusivo no Google Gemini e em tarefas específicas (Fibonacci e primos) restringe a extrapolação dos dados. Trabalhos futuros devem adotar delineamentos comparativos com quatro subgrupos (iniciantes/intermediários com e sem IAG), variar o nível de dificuldade das tarefas e incluir métricas de retenção de conhecimento a longo prazo para aprofundar a compreensão dos impactos pedagógicos da IAG, além de utilizar outras LLMs na elaboração do experimento, como o ChatGPT, Claude, DeepSeek, entre outros.

6. Referências

Bang, K. and Dang, M. (2024) "Impact of Generative AI on Learning Programming", Dissertação de Mestrado, Chalmers University of Technology, Gothenburg, Sweden.

Carrie Anne Philbin. (2023). Exploring the Potential of Artificial Intelligence Program Generators in Computer Programming Education for Students. ACM Inroads 14, 3 (September 2023), 30–38. https://doi.org/10.1145/3610406.

Comanici, Gheorghe et al. (2025) Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. arXiv preprint arXiv:2507.06261

Cotroneo, D., Improta, C. and Liguori, P. (2025) "Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity", arXiv preprint, arXiv:2508.21634.

Hashmi, N. et al. (2024) "Generative AI's impact on programming students: frustration and confidence across learning styles", Issues in Information Systems, 25(3), pp. 371-385.

Kazemitabaar, M. et al. (2023) "Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming", In: CHI Conference on Human Factors in Computing Systems, Hamburg. ACM, pp. 1–23.

Licorish, S. A. et al. (2025) "Comparing Human and LLM Generated Code: The Jury is Still Out!", arXiv preprint, arXiv:2501.16857.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.

Nguyen, N. and Nadi, S. (2022) "An Empirical Evaluation of GitHub Copilot's Code Suggestions", In: 19th International Conference on Mining Software Repositories (MSR), Pittsburgh. ACM, pp. 1–5.

Silva Junior, S. M. et al. (2023) "ChatGPT no auxílio da aprendizagem de programação: Um estudo de caso", In: Anais do XXXIV Simpósio Brasileiro de Informática na Educação (SBIE), Passo Fundo. pp. 1375-1384.

Vaithilingam, P., Zhang, T. and Glassman, E. L. (2022) "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models", In: CHI Conference on Human Factors in Computing Systems, New Orleans. ACM, Article 332.

Velho, M. A. B. (2025) "Investigando o Impacto da Inteligência Artificial Generativa no Aprendizado em Programação", Monografia de Conclusão de Curso. Universidade Federal do Rio Grande do Sul, Porto Alegre.

Yetiştiren, B., Özsoy, I., Ayerdem, M. and Tüzün, E. (2023) "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT", arXiv preprint, arXiv:2304.10778.

7. Declaração sobre uso de Inteligência Artificial

Declaro que não fiz uso de ferramentas de IA na elaboração deste artigo.