Avaliação da Performance e Corretude na Geração de Código Através de Técnicas de Engenharia de *Prompt*: Um Estudo Comparativo

Gabriel Trevisan Damke
Universidade Tecnológica Federal do Paraná (UTFPR)
Medianeira-PR, Brasil
gabrieldamke@alunos.utfpr.edu.br

Daniel Mahl Gregorini
Universidade Tecnológica Federal do Paraná (UTFPR)
Medianeira-PR, Brasil
danielgregorini@alunos.utfpr.edu.br

Luana Copetti
Universidade Tecnológica Federal do Paraná (UTFPR)
Medianeira-PR, Brasil
luanacopetti@alunos.utfpr.edu.br

***Abstract*—*Prompt* engineering is a relatively new process that plays a crucial role in the effectiveness of language models, including in tasks such as code generation. This research aims to compare the performance of different *prompt* engineering techniques in code generation. The study evaluates the techniques based on two main metrics: correctness and performance of the generated code. A small dataset was manually created with 12 exercises of different difficulty levels, with trade-offs between time and space. Performance was measured by the ability to obtain the answer with optimal asymptotic complexity given a specific constraint. The data were analyzed using the Pass@k metric, where k was defined as 1 and 3. Two distinct language models were defined to obtain the results: Meta Llama 3 8B and Google Gemma 7B.**

***Keywords*—*Prompt* Engineering; Artificial Intelligence; Code Generation.**

***Resumo*—A Engenharia de *Prompt* é um processo relativamente novo que desempenha um papel crucial na eficácia dos modelos de linguagem, inclusive em tarefas como a geração de código. Essa pesquisa busca comparar a performance de diferentes técnicas de engenharia de *prompt* na geração de código. O estudo avalia as técnicas com base em duas principais métricas: corretude e performance do código gerado. Um pequeno *dataset* foi manualmente criado com 12 exercícios de diferentes níveis de dificuldade, com *trade-offs* entre tempo e espaço. A performance foi mensurada pela capacidade de obter a resposta com complexidade assintótica ótima dado uma restrição específica. Os dados foram analisados pela óptica da métrica Pass@k onde k foi definido como 1 e 3. Foram definidos dois modelos de linguagem distintos para obter os resultados: Meta Llama 3 8B e o Google Gemma 7B.**

***Palavras-chave*—Engenharia de *Prompt*; Inteligência Artificial; Geração de Código.**

I. INTRODUÇÃO

A engenharia de *prompt* é o processo de estruturação das instruções que podem ser processadas e executadas por um modelo generativo de linguagem natural, como o GPT (*Generative Pre-trained Transformer*) [1]. Esses modelos são capazes de gerar textos, responder perguntas e inclusive gerar soluções de código de programação, mas a qualidade e precisão dessas respostas dependem de como os *prompt*s são estruturados. O *prompt* é um trecho de linguagem natural que descreve uma ou mais tarefas que o modelo de inteligência artificial deve executar [2].

No cenário atual, a geração automática de código por meio de modelos de linguagem se tornou uma ferramenta poderosa e frequentemente utilizada por desenvolvedores e engenheiros de *software*, pois a capacidade de automatizar a geração de trechos de código pode diminuir o tempo total de criação de *software* em relação a uma abordagem de programação manual. Nesse contexto, a engenharia de *prompt* demonstra ser uma ferramenta essencial para maximizar o potencial desses modelos, aumentando a precisão do código gerado e possibilitando que o modelo proponha soluções eficientes de complexidade de tempo e espaço. Existem diferentes técnicas de engenharia de *prompt* que podem ser utilizadas, cada uma com características e aplicações específicas.

Uma dessas técnicas é o *zero-shot prompting*, no qual o modelo realiza uma tarefa sem qualquer exemplo específico fornecido, dessa forma, o modelo confia apenas no conhecimento obtido durante o treinamento prévio. Essa abordagem é conveniente em casos em que não há exemplos disponíveis ou quando a tarefa é relativamente simples [3].

Já o *few-shot prompt*ing é uma técnica que fornece alguns exemplos para a tarefa desejada antes de solicitar a execução de uma nova instância da tarefa. A utilização dessa técnica é especialmente útil em casos nos quais é necessário que o modelo aprenda padrões personalizados específicos [3].

Outra técnica é o *chain-of-thought prompt*ing, que permite capacidades de raciocínios complexas através de orientar o modelo a analisar o modelo em etapas, dividindo a tarefa de problemas a subproblemas ou etapas intermediárias [4]. A abordagem é eficaz em problemas que requerem raciocínio lógico, pois o modelo é capaz de explorar e justificar cada passo do processo de resolução para fornecer uma solução final.

Além das técnicas de engenharia de *prompt*, as métricas para avaliação da eficácia dos modelos de linguagem também são objeto de estudo e demonstram ser essenciais para uma análise adequada da performance dos modelos. Uma dessas métricas é o Pass@k, que mede a capacidade do modelo de fornecer uma resposta correta dentro de suas primeiras "k" tentativas, conforme proposto por Chen *et al* [5].

Nesse estudo, busca-se avaliar a eficácia de diferentes técnicas de engenharia de *prompt* na geração de código

otimizado por meio de uma abordagem experimental que envolve a criação e análise de exercícios de programação. Esses exercícios foram desenvolvidos manualmente e apresentam diferentes soluções com complexidades de tempo e espaço variadas.

II. METODOLOGIA

Na primeira etapa, foram desenvolvidos manualmente exercícios com o objetivo de cumprir uma ampla gama de complexidade, desde problemas simples, que requerem uma abordagem com soluções matemáticas e lógicas comuns, até problemas mais complexos, envolvendo algoritmos avançados e considerações de otimização. A criação manual destes exercícios foi realizada com o propósito de apresentar ao modelo desafios inéditos, ou seja, que não são baseados no conjunto de dados pré-existentes que foi utilizado no treinamento dos modelos. Embora não seja possível garantir a ausência de dados similares no pré-treinamento devido à falta de informações sobre as fontes originais, o uso de modelos com datasets de treinamento relativamente pequenos, junto a abordagens criativas, ajuda a mitigar esse risco. Os problemas foram divididos em: fácil, médio e difícil.

Foi definido que problemas fáceis são todos aqueles que possuem uma resolução direta e intuitiva, no qual o caminho para a solução é claro e não exige estratégias complexas ou o uso de algoritmos avançados. A solução segue um padrão previsível, sendo alcançada através de procedimentos diretos, como *loops* simples ou operações condicionais básicas. Dessa forma, não é necessário, para esses problemas, a aplicação de técnicas sofisticadas, como otimização de desempenho ou manipulações avançadas de estruturas de dados.

Os problemas classificados como médios foram definidos como aqueles que requerem uma compreensão mais aprofundada a respeito dos princípios algoritmicos e demandam o uso de estratégias de resolução que vão além do básico. Nesse caso, a solução não é imediatamente óbvia e pode exigir o uso de múltiplas abordagens. Além disso, os problemas médios podem envolver a análise de *trade-offs*, ou seja, a necessidade de balancear diferentes aspectos de uma solução. Como Hennessy e Patterson apontam, “projetar uma arquitetura exige fazer *trade-offs* porque diferentes objetos de design, como desempenho e custo, muitas vezes estão em conflito” [6]. Assim, as soluções para tais problemas exigem uma análise detalhada das alternativas disponíveis e uma avaliação de como cada uma afeta o desempenho geral da solução.

Os problemas difíceis foram definidos como aqueles que apresentam um alto grau de dificuldade na formulação da solução, exigindo a aplicação de algoritmos mais avançados, ou seja, aqueles que envolvem várias técnicas de manipulação de dados, como o uso de grafos e técnicas de programação dinâmica. Além disso, exigem uma capacidade de pensamento mais abstrata, exigindo a decomposição do problema em componentes conceituais que possam ser manipulados individualmente. Por fim, a combinação de diferentes técnicas para resolver subproblemas interdependentes, como a aplicação paralela ou sequencial de técnicas de programação dinâmica, algoritmo de grafos e heurísticas de otimização.

Na segunda etapa, foram selecionadas três técnicas para serem analisadas: *zero-shot prompt*ing, *few-shot prompting* e *chain-of-thought prompt*ing. Para cada técnica, uma versão do exercício foi desenvolvida.

No *zero-shot prompt*ing, o modelo recebeu apenas o enunciado da questão, sem nenhum exemplo adicional. Para o *few-shot prompt*ing, foram desenvolvidas duas versões com enunciados diferentes e suas devidas soluções ótimas em termos de complexidade de espaço e tempo, mas com abordagens semelhantes. No *chain-of-thought prompt*ing, o modelo recebeu um exemplo de linha de raciocínio em um exercício semelhante, que incluía a melhor solução em termos de complexidade de tempo e espaço, seguido da pergunta a ser solucionada.

Na terceira etapa foram definidas as métricas de análise de desempenho, tanto para a avaliação da corretude quanto da capacidade de geração de código otimizado. A métrica Pass@k foi aplicada com k=1 e k=3. O valor de k=1 foi escolhido para verificar se a primeira solução é ótima, enquanto k=3 permite uma avaliação mais abrangente, levando em conta a natureza não determinística dos modelos de linguagem.

Para verificar a corretude do código, foram desenvolvidos casos de teste específicos. Esses casos inserem diferentes entradas na função retornada pelo modelo e buscam abranger todas as possibilidades para validar o funcionamento da solução em diferentes cenários. A corretude de cada exercício é então quantificada pela porcentagem de casos de teste que a solução resolve corretamente. O resultado final da corretude de cada modelo é determinado pela média ponderada das porcentagens das corretudes obtidas em todos os exercícios. Para adaptar tal análise para a métrica Pass@k, foi considerado analisar a corretude para exercícios otimizados para tempo.

Para verificar a performance do código, é necessário considerar que podem haver *trade-offs* entre complexidade de espaço e tempo, assim, a melhor solução é relativa, pois o fato do algoritmo ser otimizado para tempo ou para espaço depende do contexto de uso. Além disso, estabelecer um *benchmark* preciso é complexo devido à diversidade dos exercícios e à volatilidade do tamanho de entrada, especialmente em algoritmos com complexidades assintóticas elevadas, como O(n!). A performance desses algoritmos pode variar drasticamente entre pequenas e grandes entradas, como n=1 e n=100. Ademais, a metodologia utilizada para determinar se o modelo foi capaz de identificar e implementar a solução ótima tanto em termos de eficiência de espaço quanto de tempo para cada problema proposto consistiu em analisar a porcentagem de respostas ótimas geradas em relação ao total de questões propostas.

Na quarta etapa, foram definidos os modelos de linguagem a serem utilizados nos testes. Foram escolhidos o Llama 3 8B da Meta e o Gemma 7B do Google, esses quais são modelos *open-source* de geração de texto.

Na quinta etapa, um algoritmo foi especialmente desenvolvido para automatizar a inferência das respostas necessárias. O código em questão realizou requisições nos devidos modelos através da API (*Application Programming Interface*) disponibilizada pela plataforma Groq Cloud. O *prompt* de usuário utilizado foi obtido através de um JSON (*Javascript Object Notation*) em que os exercícios de programação previamente feitos estavam armazenados, com adição a isso, foi explicitamente mencionado que o código deveria ser gerado em Python e foi especificado o tipo de otimização assintótica que deveria ser produzida. Os parâmetros de temperatura e limite de *tokens* foram definidos como 1 e 1024, respectivamente.

Na sexta etapa, os resultados gerados pelo algoritmo anteriormente mencionado foram manual e individualmente analisados. Foi desenvolvido um pequeno sistema para preenchimento de um formulário que possibilita a geração de um JSON com os resultados de corretude e capacidade de otimização para cada um dos modelos, considerando os parâmetros k=1 e k=3.

Na sétima e última etapa, os dados foram condensados e avaliados através de um sistema de análise desenvolvido especialmente para receber um JSON com os dados da etapa anterior. O sistema em questão permite a comparação percentual de desempenho de técnicas e modelos para diferentes valores de k de forma intuitiva.

III. RESULTADOS PRELIMINARES

Os resultados preliminares demonstram uma melhoria significativa em todos os casos ao utilizar técnicas mais avançadas em relação ao *zero-shot prompting*. A melhoria relativa varia de aproximadamente 20%, no contexto de corretude, a até aproximadamente 90%, no contexto de complexidade de tempo.

As diferenças entre diferentes técnicas se tornam evidentes a partir da análise de performance agregada por dificuldade. Conforme ilustrado na Figura 1, a técnica *zero-shot* apresenta uma performance satisfatória em exercícios de dificuldade fácil, mas a medida em que a dificuldade aumenta, a eficiência é reduzida significativamente, com as respostas fornecidas sendo majoritariamente incapazes de aplicar as técnicas necessárias para cumprir com a instrução do *prompt*.

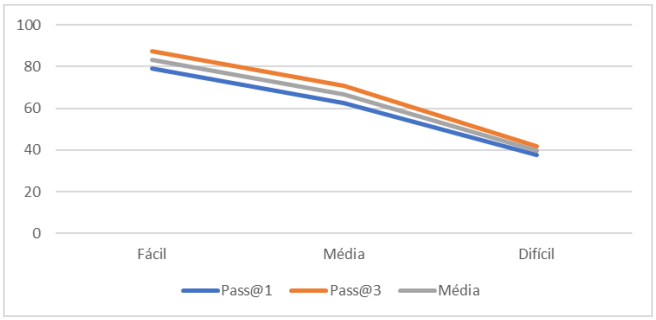

Fig. 1. Desempenho do *Zero-Shot Prompting* em Diferentes Níveis de Dificuldade: Métricas Pass@1, Pass@3 e Média (%). Fonte: Autoria Própria

O *Few-Shot Prompting* manteve altos níveis de corretude e eficiência em diferentes níveis de dificuldade. Como demonstrado na Figura 2, embora o Pass@1 diminua em tarefas mais complexas, o Pass@3 continua elevado, demonstrando que os modelos conseguem gerar soluções corretas com algumas tentativas.

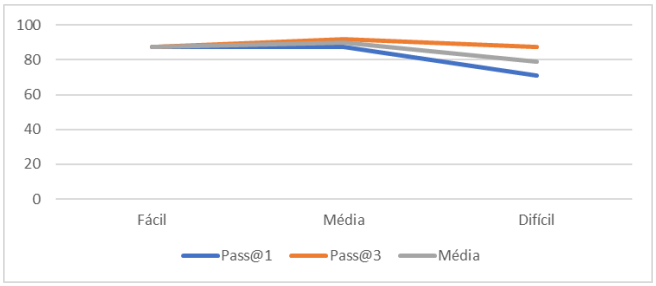

Fig. 2. Desempenho do Few-Shot *Prompt*ing em Diferentes Níveis de Dificuldade: Métricas Pass@1, Pass@3 e Média (%). Fonte: Autoria Própria

O *chain-of-thought* teve desempenho inferior ao *few-shot*, principalmente em exercícios difíceis. Como indicado na Figura 3, A análise das respostas sugere uma sobrecarga de raciocínio que, embora aumente a corretude, reduz a eficiência das soluções.

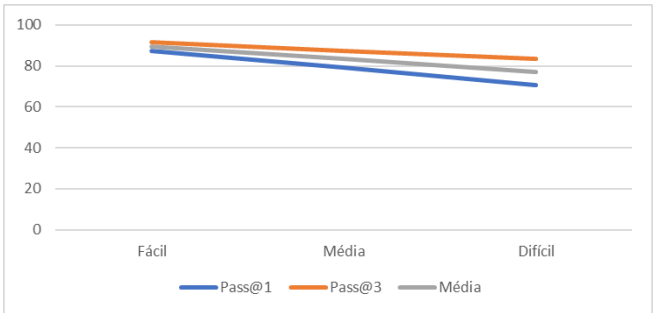

Fig. 3. Desempenho do Chain-of-Thought *Prompt*ing em Diferentes Níveis de Dificuldade: Métricas Pass@1, Pass@3 e Média (%). Fonte: Autoria Própria

TABELA I
ANÁLISE COMPARATIVA DE DESEMPENHO DOS MODELOS META LLAMA 3 8B E GOOGLE GEMMA 7B SOB DIFERENTES TÉCNICAS DE PROMPTING. EM VERMELHO, A MELHORIA RELATIVA DA TÉCNICA DE MELHOR DESEMPENHO EM RELAÇÃO AO ZERO-SHOT PROMPTING

Técnica	Desempenho agregado dos modelos Meta Llama 3 8B e Google Gemma 7B					
	Corretude		Tempo		Espaço	
	Pass@1	Pass@3	Pass@1	Pass@3	Pass@1	Pass@3
Zero-Shot *prompt*ing	74.9	83.30	41.66	50.00	62.50	66.67
Few-Shot *prompt*ing	91.67	100.00	79.16	83.33	75.00	83.33
CoT *prompt*ing	83.33	91.67	75.00	87.50	83.33	87.50
Melhoria Relativa	**22.38%**	**20.04%**	**90.01%**	**75.00%**	**33.32%**	**31.24%**

A análise supracitada indica que a otimização da complexidade assintótica de tempo constitui o principal ganho de desempenho observado. Mesmo ao considerar três diferentes métricas de resposta, as técnicas avançadas de engenharia de *prompt* demonstram uma melhoria relativa substancial. Além disso, registrou-se ganhos significativos em corretude e na otimização do uso de espaço, os quais também contribuem de forma relevante para o desempenho geral dos modelos.

IV. CONSIDERAÇÕES FINAIS

Este estudo comparativo demonstrou que técnicas de *prompt*ing aumentam significativamente a corretude e a eficiência de espaço. Embora preliminares, os resultados indicam potencial para futuras otimizações na arquitetura de LLMs e incentivam projetos com estratégias inovadoras de engenharia de *prompt*, visando maior eficiência dos modelos de linguagem. Os dados e materiais do estudo estão disponíveis em: <https://github.com/gabrieldamke/promptengineering-data>.

REFERÊNCIAS

[1] L. Reynolds and K. McDonell, "*Prompt* programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021. ACM, 2021, pp. 1-7.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, 2019. [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*. NeurIPS, 2020, pp 1-25.

[4] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought *prompt*ing elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2024, pp. 24824–24837.

[5] M. Chen *et al*., "Evaluating Large Language Models Trained on Code," in *arXiv*. CoRR, 2021. Available: https://arxiv.org/abs/2107.03374.

[6] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed. Waltham, MA, USA: Morgan Kaufmann, 2011