Prompt Engineering of ChatGPT to Improve Generated Code & Runtime Performance Compared with the Top-Voted Human Solutions

Ashraf Elnashar, Max Moundas, Douglas C. Schimdt, Jesse Spencer-Smith, Jules White
{ashraf.elnashar, maximillian.r.moundas, d.schmidt, jesse.spencer-smith, jules.white}@vanderbilt.edu
Department of Computer Science, Vanderbilt University, Nashville, Tennessee, USA

Abstract

This paper presents the results of a study comparing the runtime performance of the best performing coding solution selected from 100 solutions generated with ChatGPT to the top-voted human-produced code on Stack Overflow. These results show that selecting from the best of 100 solutions generated by ChatGPT is competitive or better than the top-voted human solution on Stack Overflow for the range of problems that we tested. Moreover, the results indicate that prompting multiple times for code and selecting the best of many generated solutions is a promising autonomous coding aid to help human software engineers find the best solutions for performance-critical code sections.

1 Introduction

Emerging trends, challenges, and opportunities. Large-language models (LLMs) [4], such as ChatGPT [3] and Copilot [1], have the ability to generate complex code to meet a set of natural language requirements [6]. Software developers can generate human descriptions of desired functionality or requirements and ask for code in a variety of languages from Python to Java to Clojure. Already, these tools are being integrated into popular Integrated Development Environments (IDEs), such as IntelliJ [9] and Visual Studio.

LLMs are now easily accessible through the Internet and within IDEs, and developers are increasingly leveraging them to obtain guidance on how to produce code samples. In many cases, the questions and code samples that developers use these LLMs for are the same questions and code samples they previously would have sought help on via discussion forums. For example, Stack Overflow is one of the most popular forums where developers ask questions and get guidance on code samples.

There has been significant discussion and research about the ramifications of using LLMs to generate code with respect to the quality of the code from a security and defect perspective [1, 2, 12]. In particular, LLM-based tools can produce poor quality code due to their ability to "hallucinate" convincing text or code that is fundamentally flawed although it appears correct. In addition, LLMs were trained on human-produced code in open-source projects that may have vulnerabilities or do not demonstrate best practices. Much discussion on the code quality of these tools has therefore focused on functional correctness and security.

Although there are certainly risks to using these tools before their capabilities are fully understood, there are also clear productivity benefits for developers in specific areas. For example, LLMs can help to automate repetitive, tedious, or boring coding tasks and often perform these tasks faster than a developer. This productivity boost is particularly apparent when coding tasks involve APIs or algorithms that developers are unfamiliar with and thus require study to get up to speed before performing the tasks. These APIs and algorithms are often included in the LLM's training set, allowing it to generate code for them swiftly and accurately.

In addition, a key benefit related to code performance is how LLMs can be used to prompt for many different potential solutions that can be automatically benchmarked to identify the fastest solution. This approach can also be performed for other quality metrics (e.g., memory or disk utilization), though we focus on performance for this paper. In the near future, developers will likely use LLM-based tools like Code Inspector and Auto-GPT to generate numerous solutions for each query and provide detailed performance analysis for each solution.

Determining when to leverage these tools or not involves fully exploring and documenting the pros, cons, and risks of their usage. Security and functional correctness are clearly important points of consideration, but must also be supplemented with additional analyses. Likewise, other quality attributes, such as memory consumption, long-term maintainability, and modularity, must also be analyzed.

Open Question: How does LLM-generated code with varying prompts compare to human-produced code in terms of runtime performance? One quality attribute that has limited analysis in the context of LLMs is runtime performance of generated code. In particular, will there be a reduction, improvement, or no net change in overall software runtime performance as these tools are adopted? Human-generated hand-optimized code *may* be faster than LLM-generated code based on common approaches encountered in training data. Conversely, however, LLM-generated code may be consistently perform better than what the average human developer produces. Exactly where we will end up in this overall spectrum and where we are with current LLM-based tools is not well understood.

One challenge in answering these question is the range of different ways to approach and answer it. For example, LLMs are stochastic and may generate different code samples for the same "prompt"[1] each time the LLM is invoked. Although we and others have documented prompt patterns [15], there is considerable variations in how prompts are worded and the code generated in response.

Another open question, therefore, is should analysts look at the most common structure that gets produced vs. the best or worst code, etc.? Similarly, which of the many human-produced solutions to a problem should be compared against and which is most representative of the average developer? Likewise, what skill level of developer should be considered? There are many different dimensions to this question that must be explored.

The wording of a prompt influences the quality of an LLM output [15]. We therefore need to understand how prompt wording influences the quality of generated code. In particular, we need to know if varying the wording yields a higher probability of producing faster code.

Experiment: An initial comparison of human produced code example on Stack Overflow to a ChatGPT-generated solutions using a variety of prompting strategies. To start answers these questions, we analyzed them from the perspective of "what happens to code performance when humans apply the fastest of 100 solutions generated by ChatGPT instead of a human answer on Stack Overflow." This narrower exploration of the questions mimics the behavior of developers who want an answer and are willing to generate 100 solutions from ChatGPT and benchmark them. While our approach just provides an initial exploration, it provides a meaningful starting point for exploration of this topic.

In the past, developers would likely have selected human-written solutions on Stack Overflow. In the future, we envision coding tools supporting developers by generating a large number of potential solutions and quantitatively evaluating them along a number of dimensions to aid the developer in selection.

This paper presents initial data from experiments comparing the runtime performance of human-written Stack Overflow solutions to the fastest of 100 ChatGPT-generated solution to coding questions asked on Stack Overflow. Each solution was inserted into a test harness that exercised the code with progressively larger input sizes. The overall runtime of the code was benchmarked for each solution and then compared. This paper's preliminary findings indicate that when prompted 100 times, an LLM can generate Python code at least as efficient as that produced by humans for a sample of 15 Stack Overflow questions.

Paper organization. The remainder of this paper is organized as follows: Section 2 discusses our experiment setup; Section 3 analyzes the results from our comparison of top Stack Overflow coding solutions and ChatGPT-produced solutions; Section 4 describes threats to the validity of these results; Section 5 compares and contrasts this work to related research in the area; and Section 6 presents key lessons learned and future directions of this research.

2 Experiment Setup

All our analysis was done on code samples in Python since (1) it is relatively easy to extract and experiment with stand-alone code samples in Python compared to other languages, (2) ChatGPT appears to generate more correct code in Python vs. less popular languages, such as Clojure, and (3) Python is a popular language in domains, such as Data Science, where developers may have more familiarity and comfort with LLMs.

The problem set was manually curated from Stack Overflow by browsing questions related to Python. We searched Stack Overflow for questions pertaining to categories like array questions and linked list questions, since these questions are easy to test performance at increasing input sizes. We then analyzed each question and its candidate solutions to select question/solution pairs that could be isolated and inserted into our test harness. We avoided questions that relied heavily on third-party libraries (a potential threat to validity) and instead focused on solutions built on core libraries and capabilities within Python.

Wherever possible, we selected the top-voted solution as the comparison. In some cases, multiple languages were present in the solutions and we selected the first Python solution, mimicking a developer that is looking for the first solution in their target language. These decisions are discussed in Section 4.

For each selected question, the title of the question posted on Stack Overflow was extracted as a prompt for ChatGPT-3.5 Turbo through the API.[2] it is important to note that this decision meant that ChatGPT was not provided the full information in the question, which may have

[1] A "prompt" is the natural language input to the LLM [10]

[2] This decision was made based on ChatGPT-3.5's availability during the study, as ChatGPT-4 was not accessible for this study, though it may produce even better results.

handicapped it in providing better performing solutions. All the original Stack Overflow posts, human code solutions, and ChatGPT-produced code solutions, along with our entire set of questions and genereated answer, can be accessed in our Github repository: `github.com/elnashara/CodePerformanceComparison`. We encourage readers to replicate our results and submit issues and pull requests for possible improvements.

For each code sample, we measured the runtime performance using Python's *timeit* package. Code samples were provided with small, medium, and large inputs. The inputs were progressively increased in size to show the effects of scaling on the generated code. What constituted small, medium, and large was problem-specific. For each input size, we generated 100 random inputs of the given size to test with. In addition, for each input, we tested the given code 100 times on the input using the Python *timeit* package.

2.1 Overview of the Coding Problems

A total of 15 problems from Stack Overflow were selected, grouped into two classes: one related to arrays and the other related to linked lists. The problems were as follows: **P1**: identify missing number(s) in an unsorted array, **P2**: detect a duplicate number in an array that is not sorted, **P3**: given an unsorted array, find the indices of the k smallest numbers, **P4**: count pairs of elements in an array with a given sum, **P5**: find duplicates in a list, **P6**: remove list duplicates, **P7**: implement the Quicksort algorithm, **P8**: reverse a list or iterate over it in reverse, **P9**: count the frequency of elements in an unordered list, **P10**: find the maximum product subarray, **P11**: identify the middle element of a linked list in one traversal, **P12**: detect if a linked list has a loop or cycle, **P13**: reverse a linked list in Python, **P14**: find the length of a linked list in Python, **P15**: create Pascal's triangle in Python with given number of rows.

2.2 Prompts strategies

Throughout this experiment, we apply various prompting strategies to generate Python code with ChatGPT, including (1) **Naive approach**, which uses the title from Stack Overflow as the prompt, *e.g.*, "How to count the frequency of the elements in an unordered list", (2) **Ask for speed approach**, which adds a requirement for speed at the end of the prompt, *e.g.*, "How to count the frequency of the elements in an unordered list, where the implementation should be fast", (3) **Ask for speed at scale approach**, which provided more detailed information about how the code should be optimized for speed as the size of the list grows, *e.g.*, "How to count the frequency of the elements in an unordered list, where the implementation should be fast as the size of the list grows", (4) **Ask for the most optimal time complexity**, which should prioritize achieving the most optimal time complexity possible, *e.g.*, "How to count the frequency of the elements in an unordered list, where implementation should have the most optimal time complexity possible", and (5) **Ask for the chain-of-thought**, which generates coherent text by providing a series of related prompts, *e.g.*, "Please explain your chain of thought to create a solution to the problem: How to count the frequency of the elements in an unordered list First, explain your chain of thought. Next, provide a step by step description of the algorithm with the best possible time complexity to solve the task. Finally, describe how to implement the algorithm step-by-step in the fastest possible way."

ChatGPT was prompted 100 times with each prompt per coding problem, which yielded up to 100 different coding solutions to the problem per prompt. In practice, fewer than 100 unique coding solutions were produced since ChatGPT generated logically equivalent programs. However, we tested the performance of all generated code. For the results, we did not remove duplicate solutions. If two different prompts had identical solutions, we benchmarked each and left the results with the expectation that 100 timing runs on 100 different inputs would average out any negligible differences in timing.

3 Analysis of Experiment Results

The results from our experiments evaluating the performance of code provided by Stack Overflow and prompting ChatGPT 100 times for fifteen coding problems with three different input sizes (1,000, 10,000, and 100,000) are shown in Figures 1, 2 and 3. Figure 4 illustrates the minimum average performance across all input sizes.

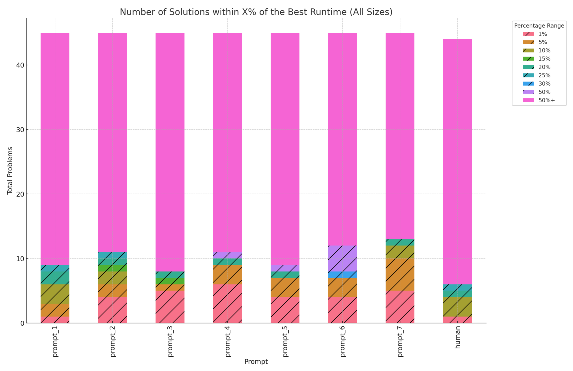

Figure 1. Number of Solutions within X% of the Best Runtime (Input Size 1,000)

These figures show the number of problems for each prompt where the best of the 100 solutions generated by each prompt was within 1%, 5%, etc. of the best solution found across all prompts and the human. For each problem, a total of up to 701 solutions were benchmarked (7 prompts * 100 solutions per prompt + 1 human solution). The best performing solution across all solutions was used as the "Best Runtime" solution in the figures against which other solutions were compared. Figures 1, 2, 3 and 4 show

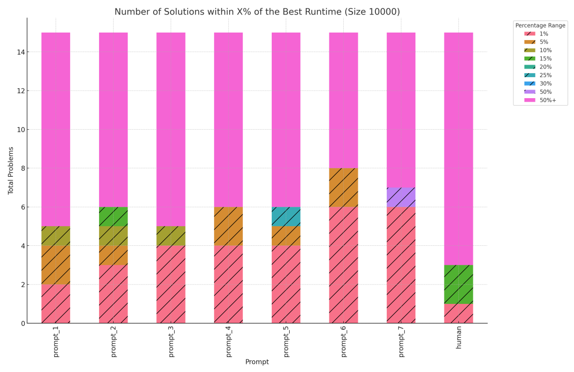

Figure 2. Number of Solutions within X% of the Best Runtime (Input Size 10,000)

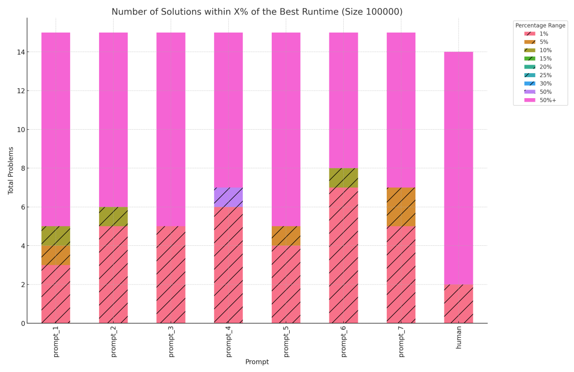

Figure 3. Number of Solutions within X% of the Best Runtime (Input Size 100,000)

how ChatGPT selects the best performing solution from 100 prompts with different input sizes that is competitive to the human Stack Overflow solution in nearly all cases.

The human solution was the fastest solution on only one of the problems. We used the title of the question as the input to ChatGPT. All the code samples produced code with respect to the title of the Stack Overflow post. Since we directly translated the titles into prompts for ChatGPT, however, there may have been additional contextual information in the question that could have been used by ChatGPT to further improve its solution.

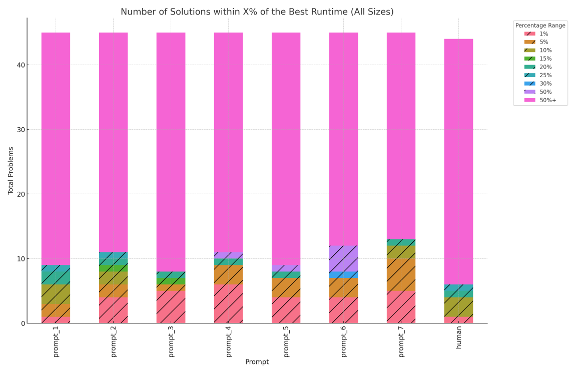

Figure 4. Number of Solutions within X% of the Best Runtime (All Input Sizes)

4 Threats to Validity

Although our results are promising, they are based on a relative small overall sample size. Clearly, more work on a larger sample size is needed. In particular, the software engineering and LLM communities will benefit from a large-scale set of benchmarks that associate (1) code needs (expressed as natural language requirements), questions, specifications, and rules with (2) highly optimized human code, as well as associated benchmarks and interfaces. These communities can then apply such benchmarks to measure and validate LLM coding performance over time to ensure the communities are headed in the right direction regarding the development and use of these tools.

Prompt construction was a significant threat to validity. We chose to only use the title of questions in Stack Overflow and did not provide additional information from the full body of the question. We made this decision to eliminate the possibility that ChatGPT used any provided code as the basis for its answer. We did not want ChatGPT completing/improving fundamentally flawed code. This choice in prompt design risked that ChatGPT was deprived of information that it could have used to produce better solutions.

Another area of risk is in the variety of coding problems that we analyzed. The problems were relatively narrow in scope and data structure type. A much wider range of problem types is needed to ensure that hidden risks regarding specific problem structures don't exist. There may be classes of problems that trigger poor performing hallucinations or code structures that we are not aware of yet. This risk is particularly problematic when attempting to generalize the overall meaning of our results.

A further threat to the validity of our results is the inherent question and code sample selection bias in the study. Since the questions and answers were manually selected, partially to focus on problems and code samples that could easily be tested and benchmarked, we may have inappropriately influenced the problem types selected and not chosen samples representative of what developers would ask in practice. We also recognize that there are other additional threats to validity beyond those discussed above.

5 Related Work

An area that may significantly effect the performance of code produced by LLMs is *prompt engineering*, which is the study of designing natural language inputs to LLMs to solve different problems, such as using outside tools [17] or tapping into LLM capabilities [14]. Naive prompting cannot solve problems in a variety of domains, such as mathematics [7], and proper prompt structure is essential to achieve good results.

We expect a similar result for performance. Well-designed prompts will likely produce better code. Our prompts, sourced directly from Stack Overflow questions, have potential for future enhancements to generate more efficient code. For this study, we did not apply our prior prompt engineering research and instead looked solely at direct input of Stack Overflow questions as prompts.

A body of initial research has looked at bugs and security issues in LLM-generated code [5, 7, 8, 11]. Some work has looked specifically at the security of human vs. LLM-produced code [2]. Some prompt engineering work [13, 16] has also looked at how to interact with an LLM to fix bugs and can likely inform future work on interacting with an LLM to improve performance.

6 Concluding Remarks

The results from our experiments show that prompting and automatically benchmarking generated code solutions to select the best is an effective strategy for leveraging LLMs to produce fast code. When prompted 100 times, all our prompts were competitive to the top-voted human solutions on Stack Overflow. The highest performing prompt employed chain-of-thought prompting.

A key question for future work is if/how other code quality metrics can be integrated to allow considering multiple dimensions of code quality beyond performance. Future LLM-based tools may allow developers to define a range of metrics and automatically prompt for solutions until a quality threshold is met, a prompt limit is hit, or too much time has passed. The ability to search many more coding solutions is a key attribute of ChatGPT-based code generation as shown in the results.

7 Acknowledgements

ChatGPT Code Interpreter was used to generate the code for the data visualizations and filter the data sets.

References

[1] Github copilot · https://github.com/features/copilot.

[2] O. Asare, M. Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741*, 2022.

[3] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, et al. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

[4] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[5] A. Borji. A categorical archive of chatgpt failures. *arXiv preprint arXiv:2302.03494*, 2023.

[6] A. Carleton, M. H. Klein, J. E. Robert, E. Harper, R. K. Cunningham, D. de Niz, J. T. Foreman, J. B. Goodenough, J. D. Herbsleb, I. Ozkaya, and D. C. Schmidt. Architecting the future of software engineering. *Computer*, 55(9):89–93, 2022.

[7] S. Frieder, L. Pinchetti, R.-R. Griffiths, T. Salvatori, T. Lukasiewicz, P. C. Petersen, A. Chevalier, and J. Berner. Mathematical capabilities of chatgpt. *arXiv preprint arXiv:2301.13867*, 2023.

[8] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam. Chatgpt and software testing education: Promises & perils. *arXiv preprint arXiv:2302.03287*, 2023.

[9] J. Krochmalski. *IntelliJ IDEA Essentials*. Packt Publishing Ltd, 2014.

[10] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[11] M. Nair, R. Sadhukhan, and D. Mukhopadhyay. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive*, 2023.

[12] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[13] D. Sobania, M. Briesch, C. Hanna, and J. Petke. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*, 2023.

[14] E. A. van Dis, J. Bollen, W. Zuidema, R. van Rooij, and C. L. Bockting. Chatgpt: five priorities for research. *Nature*, 614(7947):224–226, 2023.

[15] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[16] C. S. Xia and L. Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

[17] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.