Engineering Educators Bringing the World Together
2025 ASEE Annual Conference & Exposition
Palais des congrès de Montréal, Montréal, QC • June 22–25, 2025
Paper ID #47024

An Evaluation of Prompt Engineering Strategies by College Students in Competitive Programming Tasks

Sita Vaibhavi Gunturi, Pennsylvania State University, Harrisburg, The Capital College
Dr. Jeremy Joseph Blum, Pennsylvania State University, Harrisburg, The Capital College

Dr. Jeremy Blum is an associate professor of Computer Science at the Pennsylvania State University, Harrisburg, PA, USA. Prior to joining Penn State Harrisburg, Dr. Blum worked as a research scientist at the Center for Intelligent Systems Research at the George Washington University. Dr. Blum received a D.Sc. in Computer Science and an M.S. in Computational Sciences, both from the George Washington University, as well as a B.A. in Economics from Washington University. His research interests include computer science education and transportation safety.

Dr. Tyler S. Love, University of Maryland Eastern Shore

Dr. Love is Professor and Director of Career and Technology Education Studies for the University of Maryland Eastern Shore (UMES) at the Baltimore Museum of Industry. He is also the Coordinator for Technology and Engineering (T&E) education. Dr. Love earned his master's and Ph.D. in Integrative STEM Education from Virginia Tech. His bachelors degree is in Technology Education from UMES. He previously taught T&E courses in Maryland's Public School System. He is nationally recognized for his work related to the safer design of makerspaces and collaborative STEM labs. Dr. Love is an Authorized OSHA Trainer for General Industry. He has also served on committees at state and national levels that developed P-12 engineering education standards. In recognition of his work, Dr. Love was awarded the 2022 ASEE mid-Atlantic section Best Paper Award, and has received numerous awards from the International Technology and Engineering Educators Association.

An Evaluation of Prompt Engineering Strategies by College Students in Competitive Programming Tasks

Sita V. Gunturi[1], Jeremy J. Blum[1], Tyler S. Love[2]
[1]Pennsylvania State University, Harrisburg
[2]University of Maryland Eastern Shore

Abstract

Generative AI, powered by Large Language Models (LLMs), has the potential to automate aspects of software engineering. This study implemented a monostrand conversion mixed-methods approach to examine how computer science students utilize generative AI tools during a competitive programming competition across multiple campuses. Participants used tools such as ChatGPT, GitHub Copilot, and Claude and submitted transcripts documenting their interactions for analysis. Drawing from prompt engineering literature, the study mapped six key strategies to 14 areas of best practices for competitive programming. These practices included clarifying instructions, one-shot and few-shot prompting, chain-of-thought prompting, feedback to refine solutions, and leveraging of LLM meta-capabilities. The transcripts were analyzed through a directed content analysis to assess adherence to these practices and then converted to descriptive statistics. Findings revealed significant variability in adherence, with an average compliance rate of 34.2% across practices. While simpler practices achieved adherence rates as high as 98%, eight practices saw minimal or no usage. These results highlight that students often adopt basic prompt engineering techniques but struggle with more complex strategies, suggesting the need for structured prompt engineering instruction in computer science curricula to maximize the potential of generative AI tools.

1 Introduction

Generative AI is transforming software engineering and problem-solving. Despite growing interest in this technology, there is limited research on the extent to which students efficiently use these tools in real-world scenarios. This study addresses that gap by analyzing student interactions with LLMs during a competitive programming competition held across multiple college campuses. It evaluates adherence to prompt engineering best practices, including clarity of instructions, chain-of-thought prompting, and iterative feedback, and examines how these strategies impact success. In this study, the term “best practices” refers to guidelines that have been established for optimizing AI interactions during problem-solving tasks, for example in (Open AI, 2024; Google, 2024).

An IRB-approved plan guided data collection from the competition, where teams of three undergraduate students were encouraged to use generative AI to solve programming problems. Over 100 students participated. After the competition, students voluntarily submitted transcripts documenting their interactions with AI tools. These transcripts were examined using a directed content analysis (Hsieh and Shannon 2005) to assess how well students followed prompt engineering best practices.

The study findings reveal significant variability in adherence to best practices in prompt engineering, with an average compliance rate of less than 50%. Some of the practices were widely adopted, while others were rarely used, exposing gaps in students' understanding of effective LLM utilization. These results highlight the importance of incorporating prompt engineering principles into computer science (CS) curricula to prepare students for the demands of AI-assisted workflows.

2 Literature Review

Large language models (LLMs) have made significant advances in natural language processing. They excel in tasks such as code generation, question answering, and reasoning. These capabilities make them valuable in competitive programming, where accuracy and efficiency are critical. As a result, prompt engineering has become an essential skill for optimizing the use of generative AI tools like Claude, GitHub Copilot, and ChatGPT. A range of best practices for prompt engineering have been developed by both researchers and the developers of LLMs.

These LLMs have begun to be incorporated into CS education. ChatGPT, for example, has provided instructional support for beginning students, providing template code to get students started and facilitate the identification and correction of programming errors (Yilmaz and Yilmaz 2023). Github's Copilot has been used in similar ways to provide scaffolding for novice programmers (Prather *et al.* 2023). The literature also demonstrates that CS educators require support and training to help scaffold novice programmers and not just integrate generative AI for basic uses, but as a tool for personalized learning, adaptive feedback, and co-creative educational experiences (Zhai 2024). Moreover, previous research has found that barriers to participating in competitive programming events can be lowered using generative AI (Jayachandran *et al.* 2024a).

Careful prompt engineering is especially important in competitive programming. Clear and contextually relevant prompts are essential for generating accurate and efficient solutions. LLMs are sensitive to prompt structure and content (Chang *et al.* 2024). Poorly designed prompts can lead to irrelevant or suboptimal outputs. Prompts act as task instructions, defining parameters and goals, which directly impact the quality of LLM responses (Lertbanjongngam *et al.* 2022). For example, specifying a programming language in the prompt not only ensures that the model generates code in the required syntax, but actually can improve system output (Jayachandran and Blum 2024).

General or zero-shot prompting refers to prompting where the large language models are provided with a task without examples or advanced strategies. For some time now, models that have been fine-tuned with examples relevant to the desired task often perform well with zero-shot prompts (Wei *et al.* 2021). Moreover, when these prompts are sufficiently detailed, the LLMs can often meet precise requirements (OpenAI 2024). However, if these prompts are unsuccessful, prompt engineering guidelines suggest that users utilize additional strategies including chain-of-thought, few shot prompting, and LLM-based meta-prompting (Google 2024).

Wei *et al.* introduced Chain-of-Thought (CoT) prompting enhances reasoning by encouraging LLMs to follow logical sequences (2022). This method involves structuring the prompt to break down a problem into smaller, step-by-step reasoning tasks. For example, a CoT prompt might ask the model to "explain each step before providing the solution." This approach reduces errors by making reasoning steps explicit and is especially useful for solving complex problems that require precision and logical progression (White *et al.* 2023). Indeed, researchers found that adding the simple phrase "Let's think step by step" to a prompt can significantly enhance the LLM performance in reasoning tasks (Kojima *et al.* 2022).

Few-shot learning allows LLMs to adapt to new tasks by providing minimal examples within the prompt (Brown *et al.* 2020). This capability helps the model understand the task's context and requirements without requiring extensive retraining. For instance, a few-shot prompt for contest programming could include sample inputs and expected outputs or an example of a correctly solved problem, followed by a new problem for the model to solve.

Corrective feedback involves iteratively refining model outputs by providing targeted suggestions or corrections. For example, if an LLM generates incorrect or suboptimal code, the user can revise the prompt to include specific instructions like "Ensure the solution handles edge cases" or "Ensure that the solution runs in linear time." This iterative approach improves output quality by steering the model toward the desired solution. Previous research and OpenAI's guidelines support this strategy, highlighting its value in resolving logical flaws in generated code (White *et al.* 2023; OpenAI 2024).

The stochastic nature of LLMs significantly affects their behavior during task execution. LLMs rely on probabilistic sampling methods, such as nucleus sampling (top-p) and temperature tuning, to generate outputs. These methods introduce variability, enabling diverse responses rather than converging on a single answer. Techniques like majority voting can leverage this variability to improve accuracy, especially in quantitative tasks (Lewkowycz *et al.* 2024). While stochasticity benefits open-ended tasks, it poses challenges for precision-demanding problems, such as competitive programming. Controlling randomness through parameters like top-p and temperature can balance creativity and precision (OpenAI, 2023). Lower temperatures yield deterministic results, while higher values encourage exploration. Understanding and managing stochastic behavior is essential for optimizing LLM utility.

Meta-prompting enables LLMs to self-refine prompts and align responses with user intent. This strategy involves designing prompts that instruct the model to adapt its responses based on specific user feedback. For example, a meta-prompt might include instructions like "Reframe this question to clarify the desired outcome." By automating parts of the prompt creation process, meta-prompting allows users to focus on high-level goals rather than micromanaging individual tasks (Brown *et al.* 2020).

3 Methodology

This study aimed to evaluate how students apply prompt engineering strategies identified from the literature when using generative AI tools. The analysis looked for evidence of student use of these strategies, including clarity of instructions, chain-of-thought prompting, few-shot and zero-shot learning, corrective feedback, managing stochastic sampling, and meta-prompting.

As part of an IRB-approved study, participants were recruited from multiple campuses to participate in a generative AI-assisted competitive programming competition. The competition ran for two hours, during which teams of three undergraduate students were tasked with solving a problem set specifically designed to require interaction with generative AI systems such as ChatGPT, GitHub Copilot, and Claude. The problem set included challenges that could not be trivially solved by generative AI but instead were designed to test both the students' problem-solving skills and their ability to effectively engage with AI tools through well-constructed prompts. More details about the problem set and links to examples of the problem sets are provided in section 3.3 of this paper.

Following the competition, students were invited to voluntarily submit transcripts documenting their interactions with the generative AI systems during the contest. These transcripts provided detailed records of how students formulated prompts, incorporated feedback from the AI, and iteratively refined their queries and solutions.

To assess adherence to the prompt engineering best practices identified from the literature, the transcripts were analyzed qualitatively using a directed content analysis method. Directed content analyses use existing theories or research to identify key concepts and extend a framework or theory (Hsieh and Shannon 2005). The framework that guided the directed content analysis was derived from the strategies and best practices identified in the literature review. Key aspects from each transcript were first categorized into prompt engineering strategies, such as the clarity and relevance of prompts, the use of logical reasoning (e.g., chain-of-thought prompting), and the application of corrective feedback to refine AI-generated outputs. Additionally, the extent to which students leveraged stochastic behaviors and meta-prompting techniques was examined. The prompt engineering strategies were then mapped to a set of 14 best practices for competitive coding scenarios identified from the literature. Lastly, these qualitative best practices were converted to descriptive statistics to analyze the frequency of each best practice applied by the participating student teams. This monostrand conversion mixed-methods approach (Teddlie and Tashakkori, 2006) allowed the researchers to closely examine the prompt engineering practices that were implemented by the participants during the competition and identify which specific practices were used more or less frequently.

3.1 Prompt Engineering Strategies

The literature review identifies a set of prompt engineering strategies that are particularly relevant to competitive programming tasks. These strategies were selected based on their potential to enhance the performance of LLMs in generating accurate, efficient, and contextually appropriate code solutions.

1. *Provide Clear Directions:* Prompts should include a context for the answer that is being provided. It should for example include constraints, desired output format, programming language, and examples. For instance, instead of asking, "Write a program that adds two numbers," a more effective prompt would be, "You are a competitor in a programming contest. Write a Python program that takes two space-separated integers from standard input and prints the sum of the integers to standard output."

2. *Break Complex Procedures into Steps:* LLMs will tend to provide better solutions for smaller steps. Moreover, this approach reduces cognitive load for the user, increasing the likelihood that the user can identify and fix a flaw in the approach used by the LLM. For example, rather than having the LLM directly produce a result, a user might ask the LLM to first provide an outline for a solution approach. After ensuring that the approach seems sound, the user could then ask for code to be generated.

3. *Leverage Chain-of-Thought Prompting:* Chain-of-Thought prompting guides the model through logical reasoning, ensuring thorough and consistent outputs. Phrases like "Let's solve this step-by-step" prompt the model to break down its response into intermediate steps. For a contest programming problem, a prompt could tell the LLM to first describe the algorithm, then to reason about its correctness, and finally, when it is sure the approach is correct, to write the code.

4. *Provide Feedback to Improve Output:* Providing feedback to improve output is a key to successful, iterative refinement. Users can correct errors or make suggestions to enhance the solution. For example, if the model provides an incorrect response, a user can provide details for where the solution is failing, for example, indicating if the approach is too slow, whether it causes a runtime error, or whether it fails for small or large test cases.

5. *Leverage the stochastic nature of LLMs:* Leveraging the stochastic nature of LLMs involves exploring multiple potential solutions by generating several outputs for the same problem. If the first solution provided by an LLM is incorrect, simply starting a fresh chat and providing the prompt again may provide a correct answer

6. *Use the LLM as a Prompt Engineer:* If one is in doubt about what an effective prompt might look like, the LLM can be used to develop a good prompt. For example, if the output from LLMs has not worked, one might prompt: “Your output for programming

problems has produced a solution that is inefficient, misses edge cases, and considers only some of the possible inputs. Given the following meta-prompt, which is the specification of a programming problem, generate a new prompt that could potentially lead to better results. Aim for prompts that are clear, concise, and can guide the model effectively. {{Meta-Prompt}}"

3.2 Mapping Strategies to Best Practices

The strategies from the literature review were then mapped to a set of best practices for competitive coding scenarios. A set of 14 best practices were identified, which covered the six strategies from the previous section. These best practices include:

1. *Start a fresh chat:* Beginning a new chat ensures that contextual noise from prior conversations does not affect the model's responses. This helps maintain focus on the current problem and prevents unintended interference.

2. *Provide the entire, relevant portion of the problem statement:* Providing the entire problem statement directly to the model ensures that all relevant details are included. In the problem set, each problem had a section of text that was not relevant to the problem, and therefore, should not be provided as part of the prompt. This irrelevant or extraneous information should be removed to avoid confusing the model.

3. *Clarify the problem statement*: Ambiguities in the problem can hinder the model's performance. By clarifying unclear aspects, users help the model interpret tasks accurately and generate appropriate solutions.

4. *Specify constraints and limitations:* The constraints on input or memory are required so that the model can provide a solution that can handle potential input values and an algorithm that is efficient enough given the potential complexity of input.

5. *Include sample test cases:* Providing sample input and output from the problem statement helps the model understand the expected input-output behavior. Moreover, ChatGPT would run the generated code against the test cases. In the event that the code did not pass a test case, ChatGPT automatically refines its output in an attempt to pass the provided testcases.

6. *Include additional test cases:* Adding custom examples can address nuanced scenarios that may not be covered by the given examples. This provides additional guidance tailored to specific edge cases.

7. *Request specific programming language:* Specifying the programming language ensures that the generated solution is technically compatible with the requirements of the task, and previous research has shown that specifically requesting a language can improve the system output.

8. *Ask for comments or explanations:* Requesting code comments or detailed explanations helps users understand the logic behind the model's output. This is especially useful for verifying correctness and learning from the solutions generated. In addition, this output provides context which can help the LLM produce better output.

9. *Iterative improvement:* Follow-up questions refining the model's responses. Users can address errors or ask for enhancements, such as optimizing algorithms or handling edge cases.

10. *Create multiple chats for the same question:* Starting new chats for the same problem allows users to explore alternative solutions. This resets the context for the prompts and leverages the stochastic nature of the LLM output, which can lead to diverse and potentially better implementations.

11. *Prompt model to adopt a persona:* Instructing the model to emulate a specific domain expert, such as a senior developer or a CS professor, can improve task performance. This approach leverages the model's ability to adopt domain-specific expertise.

12. *Specify steps required to complete a task:* Breaking the problem into discrete steps ensures logical, step-by-step reasoning. For example, one could instruct the model to first define an approach, then reason about its correctness, then provide test cases, and write and test the code.

13. *Chain-of-Thought prompting:* Tests the model's capabilities relying on limited additional context in prompts, that steers the model to perform better.

14. *Meta-prompting*: Users can prompt the model to refine or create a better prompt for solving a given problem. For instance, asking the model, "Can you suggest a better way to ask this question?" allows the system to enhance the interaction itself.

As shown in Table 1, the authors map the best practices into the strategies. As shown in the table, many of the best practices apply to multiple strategies. In addition, most of the best practices have at least some elements of providing clear instructions for the LLM.

Table 1: Mapping of Best Practices to Prompt Engineering Strategies

		Best Practice													
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Strategy	Provide Clear Directions	X	X	X	X	X	X	X	X			X	X	X	
	Break Complex Procedures into Steps									X			X		
	Leverage Chain-of-Thought Prompting								X					X	
	Provide Feedback to Improve Output			X		X	X			X					
	Leverage the stochastic nature of LLMs										X				
	Use the LLM as a Prompt Engineer														X

3.3 Contest Environment and Problem Set

The contest problem set consisted of 12 problems, and teams had two hours to solve these problems. These problems were evenly divided between easy problems, medium problems, and difficult problems. Each problem was worth 100 points. If a team passed only a subset of testcases, their score for that problem would be proportional to the proportion of testcases passed. There was no penalty for making multiple submissions.

The “easy” problems could occasionally be solved by ChatGPT, using GPT-4; however, most of the solutions generated by ChatGPT would fail multiple testcases. Nonetheless, if users tried multiple attempts to generate the solutions there was a high likelihood that they could receive a correct solution. Moreover, the solutions generated were likely to be mostly correct, and therefore, with further prompting, the students would be able to get the LLM to correct the mistake.

With the medium difficulty problems, the initial solutions from GPT-4 were likely to have more problems that the students would need to work through. Some of these problems contained partial information, in an effort to encourage students to complete the information using the LLM.

The solutions generated by LLMs for the difficult problems were more limited in their utility. Typically, these solutions were either logically flawed, or they were not efficient enough to solve the testcases within the given time constraints. Nonetheless, the LLMs could quickly provide code to handle the input and output, and they could provide ideas for solution approaches.

This range of difficulty and specific learning objectives are illustrated in the following sample of problems.[1] The following six problems illustrate the range of challenges and goals for the problem set.

- *We Are*: This problem involves writing a simple program to handle a predefined call-and-response interaction, the Penn State's iconic "WE ARE" chant, which student should know well. This problem contained only partial information about the chant, which testing against LLMs at the time would require students to provide the model with the missing information in order to pass all test cases.
- *54 States and Counting:* This problem tasks students with implementing a lookup system that reflects historical and newly proposed changes to U.S. state names, abbreviations, and capitals. Only partial information was provided about current state abbreviations and capitals, to encourage students to use the LLMs to retrieve a complete listing of this information. Moreover, testing revealed that LLMs would typically produce code with bugs due to the introduction of new states and the redefinition of some of the abbreviations. The goal was to encourage students to either manually find and fix these bugs or to work with the LLM to fix the code.
- *RickRoll:* This problem involves transforming a square matrix using a sequence of operations named Rick, Roll, and RickRoll. The operations were described using images, which at the time LLM's were not able to correctly interpret. The goal was to encourage students to describe the image content as part of a prompt. In addition, the rotation operations changed both the location and orientation of the symbols. Testing revealed that LLMs would produce code that could handle the location changes but not the orientation changes, and the goal was to see if students were able to fix this code, either through interaction with the LLM or manually.
- *Crypto Trading Bots*: This problem required students to recognize that the solution required a first-in, first-out queue augmented to support a median operation, with operations that ran in logarithmic or better time. During testing the LLMs provided solutions that ran in linear time, and the goal was for the students to interact with the LLM to request a more efficient approach than the initial response provided using a basic prompt.
- *race fast safe car*: This problem required the LLM to identify the minimum number of characters which need to be added to an input string in order to make it a palindrome. In testing, the LLMs made a variety of errors. Some of these errors included problems with edge cases that could be resolved through the use of few-shot prompting that included these edge cases as additional examples.
- *Evil Chutes*: This problem involved a game that could be modelled as a Markov chain, with a target value that was obtainable via matrix inversion. However, because of the nature of the transition matrix, the best approach involved a straightforward, and fast back substitution. In testing the LLM failed to find the faster approach with a general prompt, and the goal was to see if students could use a more sophisticated prompting approach to find the best approach.

[1] A repository with these sample problems can be accessed at: https://www.hackerrank.com/challenge-repository-1745082795

4 Results

This study employed a monostrand conversion mixed-methods approach, qualitatively analyzing transcripts from the programming contest and converting those results into quantitative data. After the contest was over, students were asked to volunteer to provide their LLM transcripts, with an incentive that they would be entered into a drawing for $50 gift cards.

Sixty (60) transcripts were voluntarily provided by teams. These transcripts were then analyzed for evidence that students followed best practices in their interactions with the LLMs. Note that the success rate for problems ranged from 0% to 33%, indicating that teams typically required multiple submissions before arriving at a completely correct solution.

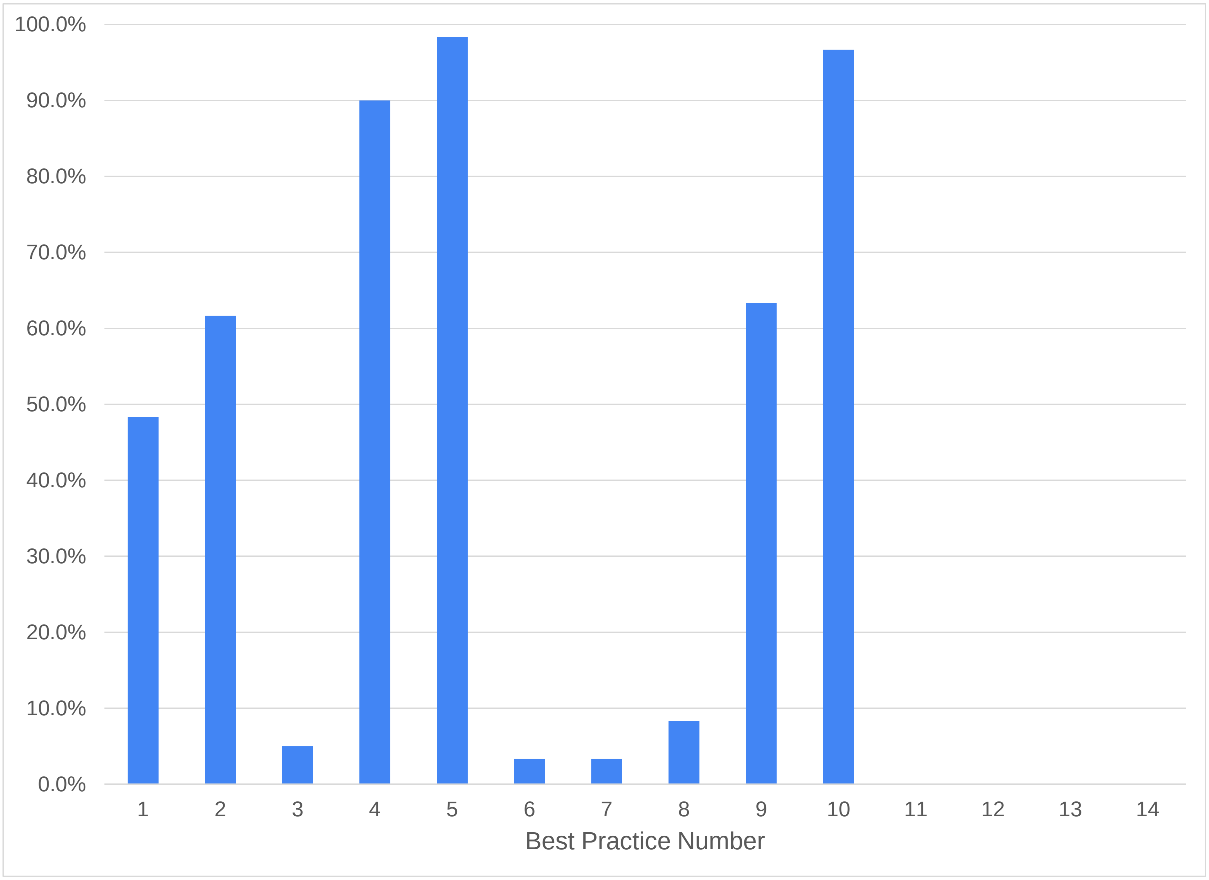

Figure 1: Percent of Teams Utilizing the Best Practices By Number.

As shown in Figure 1, some of the best practices were common in many of the transcripts. However, the majority of best practices were found in few or no transcripts.

Three of the best practices were evident in at least 90% of the transcripts. The most common best practice, present in all but one transcript (98%), was best practice 5, *include sample test cases*. Since the sample test cases were part of the problem statement, as long as teams copied and pasted the entire problem statement, the sample testcases would be included. The second most common best practice, evident in all but two transcripts (97%), was the *use of multiple chats* for a single problem. The third most common practice, *specify constraints and limitations,* was present in 90% of the transcripts. As with the sample test cases, these constraints are a part of the problem definition, so simply copying and pasting the entire problem would provide the LLM with the problem constraints.

The next three most common best practices appeared in about half of the transcripts. Sixty-three percent (63%) of teams attempted to use *iterative improvement* to try to improve the initial solution provided by the LLM. Sixty-two percent (62%) of transcripts had evidence that teams *provided the entire, relevant portion of the problem statement*. These teams removed the portion of the problem statement that was not pertinent to the task. Forty-eight percent (48%) of teams *started a fresh chat* for each challenge. Interestingly, this means that more than half of the transcripts showed evidence of attempting to solve distinct problems while previous problems were in the context provided to the LLM.

Four of the best practices appeared rarely, showing up between three and eight percent of the transcripts. These included *asking for comments or explanations* to try to understand issues in the solution, which occurred in 8% of the transcripts. Teams in only 5% of the transcripts *clarified the problem statement*. Teams specified *included additional test cases* only 3% of the time, and they *request specific programming language* only 3% of the time.

The four best practices were not evident in any of the transcripts. None of the teams *prompted the model to adopt a persona, specified steps required to complete a task,* used *Chain-of-Thought prompting,* or tried *meta-prompting*.

The limited evidence of best practice use surprised the researchers. Given that the majority of practices appeared rarely, the results support the need for integrating some prompt engineering education into the curriculum. This also prompts further research into how prepared and confident CS educators are in teaching these skills to students to enhance their use of generative AI tools.

5 Conclusions and Future Work

This study examined the use of generative AI tools in a competitive programming context, evaluating the use of prompt engineering best practices among teams of undergraduate CS students. While participants demonstrated basic prompt engineering skills, such as including sample test cases and specifying constraints, more complex strategies like chain-of-thought prompting and meta-prompting were absent. The average compliance rate of 34.2% across best practices highlights a gap in students' ability to fully leverage the capabilities of LLMs. These findings emphasize the need for structured, targeted curricula and instruction in prompt engineering. Further research is needed to investigate why students did not apply certain best practices that were rarely or never implemented.

Moreover, the study revealed that while generative AI tools can assist in solving simpler problems and providing components of solutions for more difficult problems, their effectiveness is highly dependent on the quality of user prompts. The observed variability in adherence underscores the importance of explicitly teaching prompt engineering strategies to maximize the utility of AI tools in competitive and professional environments. It is imperative that future studies investigate how CS instructors are integrating generative AI best practices into their instruction to inform additional training/professional development efforts so instructors can help students develop and apply these skills.

The results of this study open several avenues for future research and curriculum development. In particular, the authors are pursuing the design and integration of prompt engineering modules into CS courses. Such modules could include hands-on exercises, best practice demonstrations, and assessments to enhance students' proficiency in using LLMs effectively. These efforts can also help provide resources that will prompt CS educators to incorporate more best practice instruction into their teaching.

Acknowledgements

Review and approval for this study and all procedures was obtained from the Penn State Human Research Protection Program, under STUDY00024086. This research was funded by the Institute for Computational and Data Sciences at Penn State University through the Inter-Institutional Partnerships for Diversifying Research (IPDR) Program.

References

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20). Curran Associates Inc., Red Hook, NY, USA, Article 159, 1877–1901.

Chang, Y., Wang, X., Wang, J., Wu, Y., Zhu, K,, Chen, H., Yang, L., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P., Yang, Q. & Xie, Xing. (2024). A Survey on Evaluation of Large Language Models. CM Trans. Intell. Syst. Technol. 15, 3, Article 39 (June), 45 pages. https://doi.org/10.1145/3641289

Gemini for Google Workspace. 2024. *Prompting guide 101: A quick-start handbook for effective prompts, October 2024*, https://services.google.com/fh/files/misc/gemini-for-google-workspace-prompting-guide-101.pdf, accessed 4/19/2025.

Hsieh, H-F., & Shannon, S. E. (2005). Three Approaches to Qualitative Content Analysis. Qualitative Heath Research. 15, 9, 1277-1288. https://doi.org/10.1177/1049732305276687

Huang, J., Gu, S., Hou, L., Wu, Y., Wang, X., Yu, H. & Han, Jiawei. (2023). Large Language Models Can Self-Improve. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 1051–1068, Singapore. Association for Computational Linguistics.

Jayachandran, D., Maldikar, P., Love, T. S., & Blum, J. J. (2024). Leveraging Generative Artificial Intelligence to Broaden Participation in Computer Science. Proceedings of the 2024 AAAI Spring Symposium Series: Increasing Diversity in AI Education and Research. (31st), (pp. 486-492). https://ojs.aaai.org/index.php/AAAI-SS/article/view/31262

Jayachandran, D., & Blum, J. J. (2024). A Large Language Model Pipeline to Automate the Solution of Competitive Programming Problems. ASEE Mid-Atlantic Section Spring Conference. https://peer.asee.org/a-large-language-model-pipeline-to-automate-the-solution-of-competitive-programming-problems

Kojima, T., Gu, S., Reid, M., Matsuo, Y. & Iwasawa, Y. (2022). Large Language Models are Zero-Shot Reasoners. In Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22). Curran Associates Inc., Red Hook, NY, USA, Article 1613, 22199–22213.

Lertbanjongam, S., Chinthanet, B., Ishio, T., Kula, R., Leelaprute, P., Manaskasemsak, B., Rungsawang, A. & Matsumoto, K. (2022). An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode. In 2022 IEEE 16th International Workshop on Software Clones (IWSC), Limassol, Cyprus, pp. 10-15, doi: 10.1109/IWSC55060.2022.00010.

OpenAI. (2024) *Prompt Engineering Guide. OpenAI Documentation,* https://platform.openai.com/docs/guides/prompt-engineering, accessed 1/15/2025.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., Wu, Y., Neyshabur, B., Gur-Ari, G. & Misra, V. (2024). Solving Quantitative Reasoning Problems with Language Models. In Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22). Curran Associates Inc., Red Hook, NY, USA, Article 278, 3843–3857.

OpenAI Developer Forum (2023). Cheat Sheet: Mastering Temperature and Top-p in ChatGPT API." OpenAI Community, https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683, accessed 1/15/2025.

Prather, J.; Reeves, B. N.; Denny, P.; Becker, B. A.; Leinonen, J.; Luxton-Reilly, A.; Powell, G.; Finnie-Ansley, J.; and Santos, E. A. (2023). “It’s Weird That It Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers. ACM Transactions on Computer-Human Interaction, 31(1): 1-31. Doi.org/10.1145/3617367.

Teddlie, C., and Tashakkori, A. (2006). A general typology of research designs featuring mixed methods. Research in the Schools, 13(1), 12–28.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., ... & Le, Q. V. (2021). Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35, 24824-24837.

White, J., Fu, Quchen., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J. & Schmidt, Douglas. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT, arXiv Prepreints, 10.48550/arXiv.2302.11382.

Yilmaz, R.; and Yilmaz, F. G. K. (2023). Augmented intelligence in programming learning: Examining student views on the use of ChatGPT for programming learning. Computers in Human Behavior: Artificial Humans, 1(2): 1-7, doi.org/10.1016/j.chbah.2023.100005.

Zhai, X. (2024). Transforming teachers' roles and agencies in the era of generative AI: Perceptions, acceptance, knowledge, and practices. Journal of Science Education and Technology. https://doi.org/10.1007/s10956-024-10174-0