From Words to Code: Do NLP Prompting Strategies Generalize to Code Generation?

Erin Woo[1⋆], Sangyeop Yeo[2], Hyungkook Jun[2], Sangcheol Kim[2], Seung-won Hwang[3], and Yu-Seung Ma[2]

[1] Korea National University of Science and Technology, Daejeon 34113, South Korea
[2] Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea
[3] Seoul National University, Seoul 08826, South Korea
{yrwoo,sangyeop,hkjun,sheart,ysma}@etri.re.kr
seungwonh@snu.ac.kr

Abstract. Prompt engineering plays an important role in optimizing the performance of Large Language Models (LLMs). Although various prompting techniques have achieved substantial gains in natural language processing (NLP), their transferability to code generation remains underexplored. This paper presents an empirical study that examines the effects of NLP-inspired prompting methods on code generation using five LLMs and two established benchmarks, HumanEval and LiveCodeBench. We find that instruction-based prompting produces surprisingly limited or inconsistent improvements in programming contexts, while reasoning-based prompting delivers stronger and more stable performance. Our analysis identified two forms of reasoning-based prompting—procedural and goal-oriented—each addressing different aspects of reasoning and both effective for code generation. Building on these findings, we explore prompt designs that more fully leverage the reasoning capabilities of advanced LLMs, achieving measurable improvements in code generation accuracy. These results emphasize the continuing importance of prompt design in advancing AI-assisted programming.

Keywords: Large Language Model · Code Generation · Prompt Engineering.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of natural language processing (NLP) tasks, and their use has expanded rapidly into software engineering tasks, including code generation, testing, and summarization [15, 36, 19, 26]. These advances have stimulated growing interest in employing LLMs as automated programming assistants, or even as agents capable of performing software development tasks autonomously.

⋆ A project page is available at https://github.com/eyrwoo/FromWords2Code

E. Albert and C. Pasareanu (Eds.): FASE 2026, LNCS 16504, pp. 43–63, 2026.
https://doi.org/10.1007/978-3-032-22774-4_3

A critical factor influencing the performance of LLMs is **prompting**—the design of input instructions that guide the model's reasoning and output. In NLP, various prompting strategies [4, 30, 29, 27, 28] such as few-shot prompting, chain-of-thought prompting, and self-consistency have shown significant effects in improving task performance and reliability. Naturally, researchers have adapted these prompting paradigms to programming-related tasks, hoping to obtain similar benefits in software engineering domains [9, 20, 22].

However, it remains unclear whether prompting strategies that are effective in natural language contexts also yield comparable improvements in programming contexts. Programming languages differ fundamentally from natural languages in their formal syntax, compositional semantics, and execution-based evaluation criteria. Therefore, a prompting strategy that enhances reasoning coherence in NLP tasks may not necessarily improve—or might even hinder—the structural and functional quality of generated code. For instance, while minor ambiguities in natural language may be tolerable, similar ambiguities in source code can result in syntax or runtime errors [6, 21]. These differences raise our research question for AI-assisted software development: **Do prompting techniques designed primarily for NLP generalize effectively to programming tasks?**

To address this question, we conduct an empirical study on NLP-inspired prompting methods for code generation. We evaluate 22 prompting techniques across five LLMs—LLaMA3.1 8B Instruct, LLaMA3.1 70B Instruct, Qwen2.5 Coder 32B Instruct, Deepseek-R1-Distill-LLaMA-70B, and GPT-4o [10, 17, 12, 2]—using two established benchmarks, HumanEval [7] and LiveCodeBench [18].

The contributions of this paper are below:

- We systematically evaluate existing NLP-inspired prompting techniques on code generation tasks using five LLMs and two benchmarks.
- We show that instruction-based prompting yields limited or inconsistent improvements in code generation, whereas reasoning-based prompting achieves more stable and reliable performance across models and benchmarks.
- We design prompts that enable reasoning-specialized LLMs to integrate both procedural and goal-oriented forms of reasoning, leveraging their complementary strengths for more effective code generation.

The rest of this paper is organized as follows: Section 2 reviews related work, Section 3 describes our methodology, Section 4 presents results and analysis, and Section 5 concludes with implications and future directions.

2 Related Works

A variety of prompting strategies have been developed in natural language processing to improve the reasoning capability of large language models without additional fine-tuning. Few-shot prompting [4] provides minimal in-context examples for task adaptation. Chain-of-Thought (CoT) prompting [30] encourages step-by-step thinking, and Tree-of-Thought [33] extends it by structuring reasoning into hierarchical steps. ReAct [34] and Chain-of-Knowledge [24] integrate

external knowledge retrieval to ground reasoning in a factual context. Role-play prompting [31] guides models through specific reasoning personas, while self-consistency [29] enhances reliability by aggregating multiple reasoning paths. Beyond these, various other prompting approaches [15, 36] have been introduced to strengthen model reasoning and decision-making.

Motivated by the success of these prompting techniques in NLP, researchers have attempted to adapt them for programming tasks to achieve similar improvements in reasoning and reliability. Huang et al. [16] proposed CodeCoT, which encourages models to think step by step for code generation by extending the chain of thought paradigm, while Yang et al. [32] suggested a framework that automatically generates high-quality reasoning traces to improve the reliability of CoT-based code generation. In addition, some studies have approached code generation by emphasizing explicit planning and requirement analysis. Self-Planning [20] encourages models to formulate task plans before producing code. AceCoder [23] and ArchCode [13] guide models to analyze and translate problem requirements into code generation steps, ensuring alignment between specifications and implementation. SEK [11] focuses on extracting and emphasizing key functional elements from requirements, enabling models to attend to the most critical aspects during code generation.

While these studies indicate that certain NLP prompting principles show potential applicability to programming, the transition from natural language to code remains nontrivial. To this end, we systematically evaluate a range of NLP-inspired prompting strategies to examine whether their effectiveness transfers to programming contexts. This investigation is further motivated by our preliminary observations that prompting methods often regarded as effective—such as few-shot prompting—do not consistently yield improvements in code generation.

3 Methodology

This section describes how we adapt existing NLP-inspired prompting techniques for code generation and evaluated their effectiveness. We first introduce the models used in our experiments, followed by the benchmarks and performance metrics, and finally outline the prompts employed in the study.

3.1 Models

We used five LLMs with different sizes, architectures, and specializations to reflect the diversity of current LLMs.

- **LLaMA3.1 (8B, 70B) Instruct:** Two models at small and medium scales, representing general-purpose instruction-tuned LLMs.
- **Qwen2.5 Coder 32B Instruct:** A code-specialized model optimized for multilingual programming tasks, trained with extended code datasets.
- **GPT-4o (2024-11-20):** A large-scale multimodal model with advanced architecture and strong overall performance across text and code tasks.
- **DeepSeek-R1-Distill-LLaMA-70B:** A reasoning-focused model distilled from a larger expert reasoning model, DeepSeek-R1.

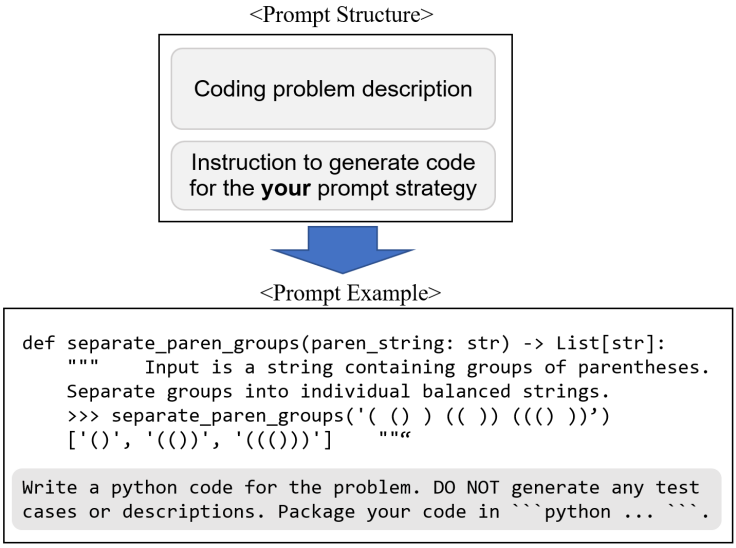


Fig. 1 A prompt example for a coding problem from HumanEval.

3.2 Datasets and Evaluation Metric

We use two widely recognized benchmarks for code generation:

- **HumanEval** [7]: A set of 164 Python coding tasks where each task provides a code snippet, a docstring description, and test cases.
- **LiveCodeBench** [18]: A larger and more diverse benchmark featuring tasks from various programming challenge sources. Our set contains 400 problems (cutoff May 2023–March 2024) with text descriptions and test cases.

To quantify performance, we measure the functional correctness of generated solutions using the *pass*@k metric [7]. The *pass*@k metric calculates how often at least one of the model's top-k generated solutions successfully passes all unit tests. We focus mainly on *pass*@1 as a representative measure of the accuracy of the model.

3.3 Experimental NLP Prompts

Code generation prompts typically consist of instructions placed before or after the problem description. As shown in Fig. 1, the HumanEval prompt includes a code snippet beginning with `def` and a gray instruction block reading "Write a Python code...". This example represents the widely used basic prompting.

For our experiments, prompt templates were drawn from the publicly available ATLAS benchmark [5], which was originally designed for evaluating prompting strategies in NLP tasks. ATLAS encompasses diverse prompting strategies, including affirmative directives (e.g., "do"), chain of thought, and interactive

prompts. By leveraging this well-organized collection, we ensured objectivity, transparency, and reproducibility across experiments. The original intent and linguistic structure of the prompts were preserved wherever possible, with minor adaptations to fit the code generation context.

Table 1 presents the instructions utilized in our experiments. We categorized the prompts into two main groups: *instruction-based prompting* and *reasoning-based prompting*. The reasoning-based prompting refers to cases where the LLM autonomously extracts additional information and uses it to generate code. Among the 26 referenced instructions, 20 were employed, maintaining the original numbering for clarity in the Table 1. Four instructions were excluded for the reasons below. Since code generation using LLMs is typically evaluated without user intervention, we adjusted prompts that required user interaction by enabling the LLM to handle the interaction autonomously. Prompts that could not be automated or that were incompatible with code generation tasks were excluded from the experiments: prompts #15, #22, #24, and #26. The evaluation compared the performance of the Basic Prompt with the proposed prompts. The Basic Prompt already incorporates some elements of the 26 principled instructions, such as removing polite expressions like "please" (specifically, prompt #1), which were unnecessary and excluded. Prompt #23, which involves generating code across multiple files, was also excluded due to the single-file structure of the evaluation datasets. Appendix A provides examples of the problem descriptions with the Basic Prompt applied to HumanEval and LiveCodeBench tasks for reference.

4 Results and Discussion

4.1 Initial Evaluation on LLaMA3.1 70B Instruct and HumanEval

We began by conducting a preliminary evaluation using the LLaMA3.1 70B Instruct model on the HumanEval benchmark, applying all prompts listed in Table 1 to this single model–dataset combination due to the high computational costs involved. To assess both deterministic and diverse output scenarios, experiments were performed with Greedy decoding (temperature = 0) and Nucleus sampling (temperature = 0.8) [14]. The baseline Basic Prompt achieved *pass*@1 scores of 79.04% (Nucleus) and 78.04% (Greedy).

Fig. 2 summarizes the results for the 21 prompts detailed in Table 1. The color gradient represents changes in *pass*@1 performance relative to the Basic Prompt, where red denotes improvement and blue indicates a decrease in performance.

Reasoning-based prompts showed clear advantages, improving performance by up to 8.55 percentage points with prompt #21 ("Write the list of what information you should consider to solve this problem."). This suggests that reasoning-based prompting helps models clarify requirements and structure intermediate reasoning steps, thereby providing richer contextual grounding for code generation. In contrast, instruction-based prompting yielded only modest or inconsistent gains—typically within 1–2 percentage points—adding overhead

Table 1 An overview of the prompts. The prompt numbers are the same as the principles listed in ATLAS [5], with the Basic Prompt.

Category	# Prompts	**Description**
Basic Prompt		A basic instruction such as "Write a Python code ...".
Instruction -based prompting	2	Integrate the intended audience in the prompt, e.g., the audience is a software developer.
	4	Employ affirmative directives such as "do", while steering clear of negative language like "don't".
	5	When you need clarity or a deeper understanding of a coding problem, utilize the following prompts: "Write the code and then explain the code."
	6	Add "I will give you a reward if you can solve the problem."
	7	Implement example-driven prompting (Use few-shot prompting).
	8	When formatting your prompt, start with "### Description", followed by "### Instruction". Subsequently, present your content. Use one or more line breaks to separate instructions, examples, questions, context, and input data.
	9	Incorporate the following phrases: "Your task is ..." and "You MUST ...".
	10	Incorporate the following phrases: "You will be penalized."
	11	Use the phrase "Write a code given in a natural, human-like manner" in your prompts.
	12	Use leading words like writing "Let's think step by step."
	13	Add to your prompt the following phrase: "Ensure that your answer is unbiased and avoids relying on stereotypes."
	16	Assign a role, "Professional Programmer", to the large language models.
	17	Use delimiter such as " ".
	18	Repeat a specific word or phrase multiple times within a prompt, such as "coding" or "code".
	20	Use output primers, which involve concluding your prompt with the beginning of the desired output. Utilize output primers by ending your prompt with the start of the anticipated response: such as "Answer: "
Reasoning -based prompting	3	Break down complex tasks into a sequence of simpler prompts in an interactive conversation.
	14	Allow the model to elicit precise details and requirements from you by asking you questions until it has enough information to provide the needed output (for example, "From now on, I would like you to ask me questions to ...").
	19	After instructing to generate a chain of thought (CoT), provide the self-CoT as an additional context.
	21	To write an information/resource or any type of text that should be detailed: "Write the list of what information you should consider to solve this problem."
	25	After instructing to generate the clear requirements that the model must follow in order to solve the problem, in the form of the keywords, regulations, hint, or instructions, provide the self-requirements as an additional context.

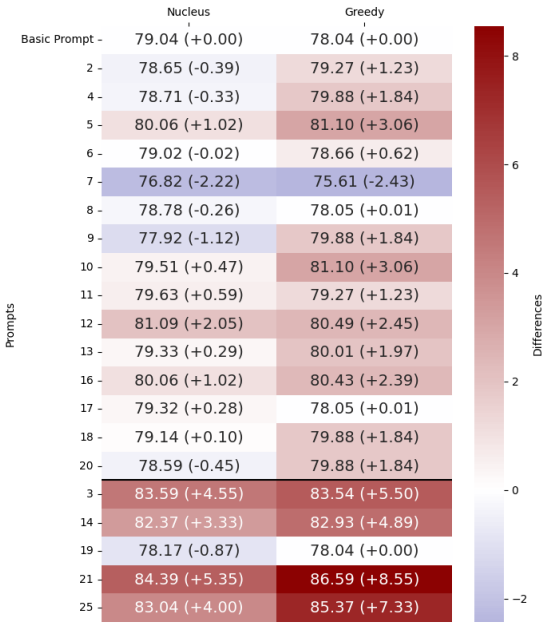


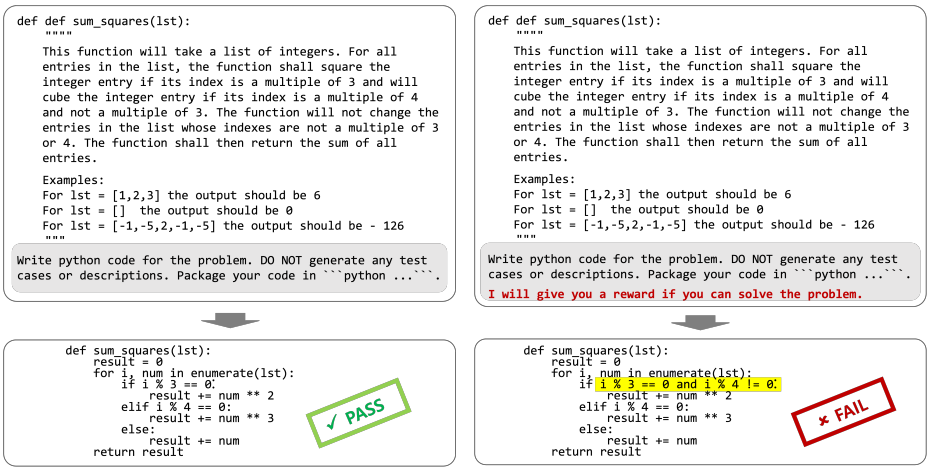
Fig. 2 Results of the initial experiment using HumanEval on LLaMA3.1 70B Instruct model.

without substantial benefit. Given these modest returns, simpler formulations such as the Basic Prompt may offer a more practical and efficient choice, particularly when token or computational costs are limited.

Across the 21 prompts, most instruction-based variants underperformed relative to the Basic Prompt. Explicit constraints, such as penalizing or conditional phrasing, often degraded results. As shown in Fig. 3, prompt #6 ("I will give you a reward if you can solve the problem.") generated erroneous code, whereas the Basic Prompt correctly solved the same problem, suggesting that excessive instruction can distract models from the actual code generation process.

Prompts #7 and #19 are particularly noteworthy: both are well-known techniques in NLP but failed to transfer effectively to code generation. Prompt #7 (few-shot) showed significant decline, while prompt #19 (chain of thought) offered no measurable gain and occasionally produced code that did not satisfy the requirements. As illustrated in Fig. 4, prompt #19 produced incorrect logic, whereas prompt #25, which explicitly guided the model to reason about and apply requirements, generated the correct output. This suggests that encouraging the model to clarify its reasoning goals and requirements—rather than merely following step-by-step procedures—helps align its reasoning process with programming semantics.

Overall, these initial observations underscore the challenges of transferring prompting strategies from NLP to code generation tasks. The results provide



(a) Correct answer with Basic Prompt. (b) Bad answer with Prompt #6.

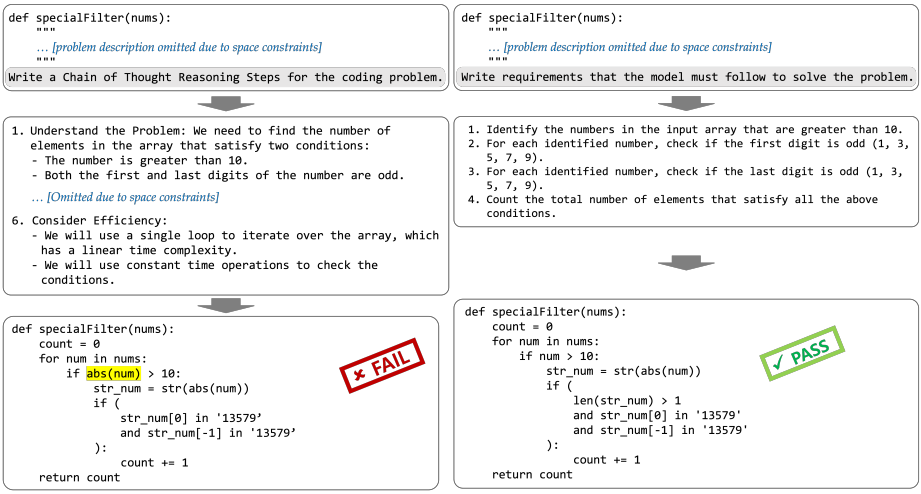
Fig. 3 Comparison of Basic Prompt and instruction-based prompts on HumanEval. The LLaMA3.1 70B Instruct model generated correct code with the Basic Prompt (a), whereas Prompt #6 (b) produced an incorrect implementation (highlighted line).

preliminary evidence that NLP-inspired prompting techniques do not generalize effectively to code generation, highlighting the need for broader evaluation across diverse models. Accordingly, we expand our analysis to include models of varying sizes, architectures, and levels of specialization. To ensure consistency, we use the HumanEval benchmark to systematically assess whether the observed trends persist across this extended set of models.

4.2 Instruction-based Prompting vs. Reasoning-based Prompting

This subsection extends our experiment across four LLMs and two benchmarks: HumanEval and LiveCodeBench. The results are averaged over three runs. We reduced the scope to 11 prompts, selected from an initial pool of 21, prioritizing those most relevant to code generation tasks to balance evaluation depth with computational cost. The selection ensured balanced representation across both instruction- and reasoning-based categories to maintain fair comparison.

HumanEval Results Fig. 5 presents the performance outcomes of various prompting strategies evaluated across four models on the HumanEval benchmark. While certain prompts (e.g., #6, #10, #12) resulted in modest improvements for models such as LLaMA3.1 8B Instruct, the overall gains from prompt engineering were relatively limited for high-performing models like Qwen2.5



(a) Bad answer with Prompt #19. (b) Correct answer with Prompt #25.

Fig. 4 Comparison between Prompt #19 and Prompt #25 from HumanEval (reasoning-based prompting).

Coder and GPT-4o. These models already achieved baseline accuracies exceeding 90% with the Basic Prompt, leaving limited room for further improvement through more sophisticated prompts.

Interestingly, smaller models such as LLaMA3.1 8B tended to benefit more noticeably than larger ones from explicit or instruction-based prompts, suggesting that direct guidance may be advantageous for models with constrained reasoning capabilities. In contrast, prompts incorporating procedural or goal-oriented reasoning (e.g., #21 and #25) exhibited comparatively stronger performance with larger models like LLaMA3.1 70B.

The limited variability observed in prompting effectiveness across the benchmark likely stems from the simplicity and self-contained nature of HumanEval tasks. Consequently, the following section presents experiments on LiveCodeBench —a more complex and demanding benchmark—to more comprehensively evaluate the potential benefits of advanced prompt engineering.

LiveCodeBench Results LiveCodeBench includes coding competition problems that are more complex and realistic than those in HumanEval. Fig. 6 summarizes the experimental results, showing that reasoning-based prompts generally outperformed the Basic Prompt across all evaluated models. In contrast, instruction-based prompts exhibited higher variability and less robust performance. For instance, prompts such as #8 and #12 produced slight improvements for LLaMA models but showed notable declines (up to -7.15%) with GPT-4o, indicating limited generalizability in code generation tasks.

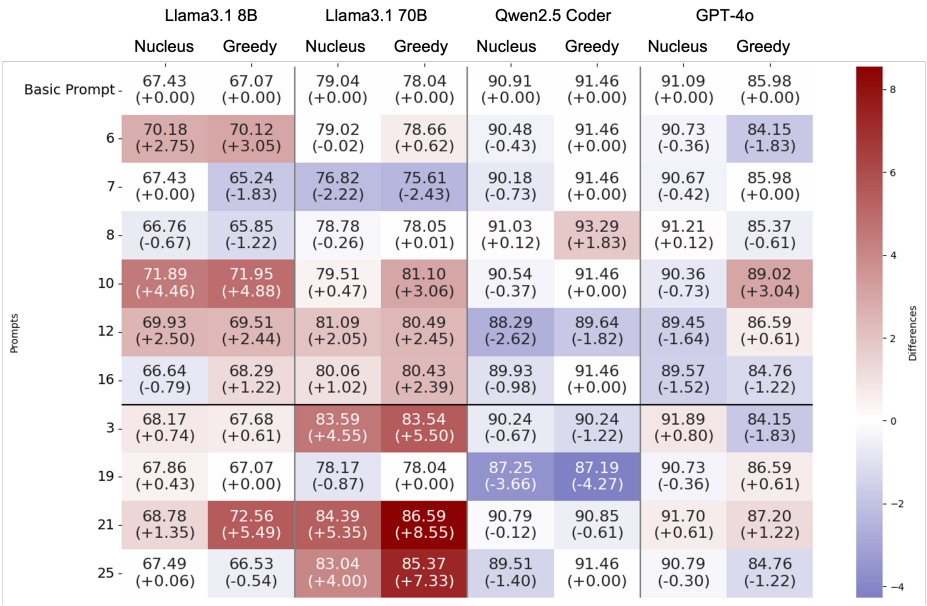


Fig. 5 Results on HumanEval. The heatmap represented *pass*@1(%), with the values in parentheses indicating the changes relative to the Basic Prompt (red = improvement, blue = decline).

Reasoning-based prompts, on the other hand, yielded more consistent gains, with some achieving up to a 7 percentage point improvement (e.g., Prompts #3, #21, and #25). These results indicate that leveraging supplementary information—such as requirements generated by the LLM itself through reasoning—is an effective approach for generating accurate code.

The inherent complexity of coding problems often demands systematic reasoning, structured task decomposition, and precise contextual nuances. Consequently, instruction-based prompts, which typically involve superficial modifications such as rephrasing, role assignment, or reward framing, may not effectively guide the model through these challenges. In contrast, reasoning-based prompts provide richer contextual grounding and procedural cues, making them better suited for navigating the intricacies of coding tasks.

Analysis on Difficulty Level The LiveCodeBench benchmark dataset is categorized into three difficulty levels: Easy, Medium, and Hard. This section analyzes the performance of four models across each difficulty level, as summarized in Table 2. Cases where a prompt achieved higher performance than the Basic Prompt are highlighted in bold. The top-3 prompts within each difficulty level (i.e., the three highest-performing prompts among the eleven evaluated for that level) are underlined for each model.

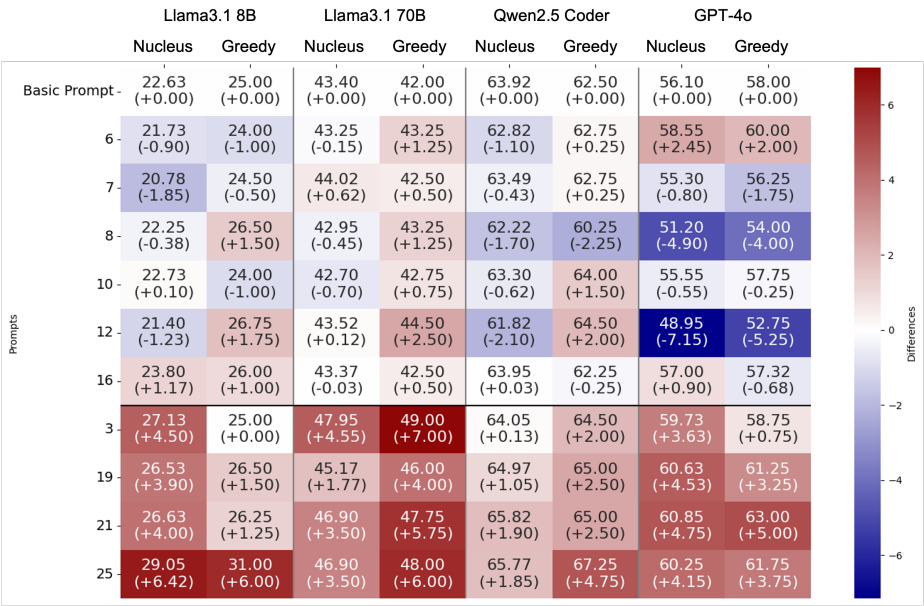


Fig. 6 Results of LiveCodeBench Overall. The heatmap represented *pass*@1(%), with the values in parentheses indicating the changes relative to the Basic Prompt (red = improvement, blue = decline).

Reasoning-based prompting showed stable and substantial improvements in the Easy and Medium categories across all models. The fact that most of the top-3 results belong to reasoning-based prompts suggests that such strategies are more effective than instruction-based methods even in realistic and diverse programming tasks, as represented by LiveCodeBench. For Hard category problems, inconsistent results likely stemmed from their unfamiliarity and difficulty for the model, requiring deeper reasoning beyond the scope of prompt engineering alone, suggesting the need for additional advancements or resources.

Taken together, results from the HumanEval and LiveCodeBench benchmarks reaffirm that reasoning-based prompting enhances model performance across both simple and complex programming tasks. This demonstrates that the benefits of prompt engineering stem not from surface-level instruction tuning, but from encouraging structured reasoning processes that better reflect programming semantics.

4.3 Integrating Procedural and Goal-Oriented Reasoning for Code Generation in Reasoning LLMs

Recent LLMs incorporate explicit reasoning mechanisms in their training and architecture. Models such as DeepSeek-R1 [12], Llama-3.1-Nemotron-Ultra-253B-v1 [3], Gemini 2.5 [8], and GPT-o1 [25] are designed to generate intermediate

Table 2 The results of LiveCodeBench by difficulty for each model with Nucleus sampling. Bold indicates *pass*@1(%) values higher than Basic Prompting, and the top-3 results per difficulty are underlined.

Category	# Prompts	LLaMA3.1 8B Instruct			LLaMA3.1 70B Instruct			Qwen2.5 Coder 32B Instruct			GPT-4o		
		Easy	Medium	Hard	Easy	Medium	Hard	Easy	Medium	Hard	Easy	Medium	Hard
Instruction -based prompting	Basic Prompt	48.31	10.24	5.22	76.90	31.25	13.22	89.44	60.24	30.56	79.65	53.75	23.33
	6	44.37	**11.07**	**5.89**	76.54	30.77	**13.99**	**89.51**	60.06	25.89	**84.37**	**54.29**	**25.78**
	7	42.11	**10.65**	**6.00**	**77.04**	**31.94**	**14.55**	**90.00**	60.06	28.11	74.01	**56.37**	**23.78**
	8	46.27	**10.95**	**5.44**	75.56	31.13	**13.55**	88.38	58.57	27.78	72.82	49.11	21.00
	10	47.68	**11.01**	5.22	76.33	31.07	11.33	**90.00**	**60.36**	26.67	79.01	**53.99**	21.44
	12	44.37	**10.42**	**5.67**	76.83	**31.54**	**13.33**	**89.65**	58.21	24.67	70.00	48.51	16.56
	16	**49.01**	**12.02**	**6.00**	76.05	31.13	**14.66**	**90.21**	**60.89**	28.22	**79.72**	**55.65**	**23.67**
Reasoning -based prompting	3	**51.55**	**17.02**	**7.44**	**81.33**	**36.66**	**16.33**	**90.07**	60.18	30.22	**84.72**	**57.98**	**23.56**
	19	**49.93**	**16.43**	**8.44**	**78.23**	**33.75**	**14.33**	**92.32**	**61.85**	27.67	**87.32**	**60.06**	19.56
	21	**50.21**	**15.83**	**9.56**	**80.14**	**35.29**	**16.11**	**91.48**	**63.10**	30.44	**88.38**	**58.81**	21.22
	25	**55.49**	**17.79**	**8.33**	**80.56**	**34.28**	**17.33**	**91.12**	**63.39**	29.66	**87.25**	**57.97**	21.77

Table 3 Results of recent LLMs evaluated on the LiveCodeBench benchmark leaderboard, including model size specifications and *pass*@1 scores, based on the same cutoff dataset, May 2023–March 2024.

Model	**Size**	*Pass*@1
DeepSeek-R1-preview	671B	86.2
Llama-3.1-Nemotron-Ultra-253B-v1	253B	84.0
Gemini 2.5 Pro	undisclosed (200–300B)*	94.2
GPT-o1-2024-12-17	undisclosed (200B)*	90.1

* Estimates, not officially confirmed.

reasoning steps before producing final outputs. These reasoning-focused models have achieved high functional accuracy in programming benchmarks, as shown in Table 3 based on the LiveCodeBench leaderboard [1].

Reasoning-oriented prompts often take the form of either procedural, step-by-step reasoning (*how*) or goal-oriented reasoning that foregrounds objectives and constraints (*what*) without prescribing procedures [35]. In our earlier experiments (Sections 4.1 to 4.2), the most effective prompts—#3, #14, #19, #21, and #25 (Table 1)—spanned both categories: prompt #19 induced step-by-step procedures, whereas prompts #3, #14, #21, and #25 emphasized sub-goals and objectives. These results show that both reasoning styles can support effective code generation, each emphasizing different aspects of reasoning.

We next investigated how modern reasoning-specialized LLMs employ these two forms of reasoning during code generation. Specifically, we examined whether such models inherently integrate both procedural and goal-oriented reasoning, or whether they tend to rely predominantly on one form. Examining this distinction would help reveal how current reasoning mechanisms contribute to structured problem solving in programming tasks.

To explore this question, we analyzed samples from the training dataset of Llama-3.1-Nemotron-Ultra-253B-v1 [3]. We initially expected that both procedural and goal-oriented reasoning would be reflected in the data. However, as illustrated by an example in Fig. 7, the dataset primarily emphasizes sequential reasoning, guiding the model to produce detailed step-by-step inferences. Given the nature of this training data, it is not surprising that the model's reasoning behavior is largely procedural, with little evidence of reasoning about goals or requirements—an element that our earlier experiments identified as beneficial for code generation.

Motivated by this observation, we examined whether explicitly prompting reasoning-focused models to articulate goals could complement their procedural reasoning and thereby improve code generation performance. To test this idea, we conducted an experiment using the DeepSeek-R1-Distill-LLaMA-70B model on the LiveCodeBench benchmark. Specifically, we modified only the initial in-

<think>

... [Omitted due to space constraints]

So the plan is:
For each index i (0-based?), split into a_i as follows:
If i is even (i%2 ==0), then a_i =0, b_i =s_i.
Else, a_i =s_i, b_i=0.
Wait, no. Wait, according to the previous example, in the even indices (0,2,4) in 0-based, which are the first, third, fifth elements in 1-based. Then a_i is 0, and b_i is s_i. For odd indices (1,3,5 in 0-based?), which are 2nd,4th,6th in 1-based, a_i is s_i, and b_i is 0.

... [Omitted due to space constraints]

Then check if the sum for a and b is acceptable.
Wait, but we don't need to check, because according to the reasoning, the sum will always be acceptable.

... [Omitted due to space constraints]

</think>

```python
... [Answer]
```

Fig. 7 A training sample from the Llama-3.1-Nemotron-Ultra-253B-v1 dataset [3], containing procedural reasoning with actions generated sequentially in response to defined conditions.

Table 4 *Pass*@1 results(%) on the Deepseek-R1-Distill-LLaMA-70B model for HumanEval and LiveCodeBench. LiveCodeBench results were the overall performance (cutoff May 2023–March 2024).

Prompt	**HumanEval**	**LiveCodeBench**
Basic Prompt	92.68	81.75
Tailored Prompt	**93.90**	**83.00**

struction from "Write a Python code for the problem." (basic prompt) to "First, define the objectives to solve the problem, and then write the Python code based on them." (tailored prompt).

This tailored prompt resulted in measurable performance improvements, as summarized in Table 4, improving *pass*@1 accuracy from 92.68% to 93.90% on HumanEval and from 81.75% to 83.00% on LiveCodeBench.

This result indicates that explicitly instructing reasoning-focused models to articulate problem goals further improves their performance. Consequently, they highlight the importance of developing prompting strategies tailored explicitly for programming tasks, combining procedural reasoning with clearly defined objectives to fully leverage the advanced reasoning capabilities of modern LLMs.

5 Limitations

This study provided an empirical examination of NLP-inspired prompting strategies for code generation, but several limitations should be acknowledged.

First, our experiments involved five LLMs and two widely used benchmarks, HumanEval and LiveCodeBench. Although this setup did not cover all programming domains, it combined models of different scales with both standard and large-scale benchmarks, providing a meaningful basis for comparative evaluation. Nonetheless, future studies on more diverse programming tasks and alternative benchmarks would help validate the generality of these observations, and extending goal-oriented prompting beyond DeepSeek to other LLMs would further test whether the effects generalize across models.

Second, our evaluation relied primarily on the $pass@1$ metric, which indicated whether generated code passed functional tests. While this metric captured functional correctness effectively, it did not reflect other important aspects such as readability, maintainability, runtime efficiency, or code style.

Third, we examined a curated set of prompting methods from the ATLAS benchmark. This selection offered diverse coverage but remained finite; consequently, newer techniques—such as multi-turn self-refinement or reflection-based prompting—were not explored in our setup.

Finally, the prompting methods evaluated in this study did not incorporate execution feedback, human feedback, or iterative error correction (e.g., running intermediate code and refining it). We excluded such feedback to maintain objectivity, reproducibility, and full automation of the experiments, as human involvement could introduce subjectivity and variability across runs.

6 Conclusion

This study investigated how prompting strategies originally developed for natural language processing transfer to code generation tasks. Through systematic experiments with five LLMs and two widely used benchmarks, we observed that instruction-based prompting provides only limited or inconsistent benefits in programming contexts, whereas reasoning-based prompting consistently leads to stronger and more stable performance. These findings highlight that prompt effectiveness depends strongly on the alignment between reasoning behavior and the structure of the target task.

Our analysis further revealed that both procedural reasoning—emphasizing step-by-step problem solving—and goal-oriented reasoning—focusing on clarifying problem objectives—contribute to successful code generation. However, current reasoning-focused LLMs exhibit a tendency to rely predominantly on procedural reasoning while underutilizing goal-oriented reasoning. To address this imbalance, we designed prompts that explicitly encourage models to define problem goals before generating code, while maintaining procedural reasoning during synthesis. This approach resulted in measurable gains in code generation accuracy, demonstrating that balanced reasoning can lead to more coherent and

functionally correct code outputs. These results suggest that even reasoning-capable models require careful prompt design to fully exploit their reasoning potential. Thoughtful prompt engineering thus remains a critical factor in advancing the reliability and effectiveness of AI-assisted programming.

Acknowledgements This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00995, Automated reliable source code generation from natural language descriptions)

References

1. LiveCodeBench Leaderboard (2025), https://livecodebench.github.io/leaderboard.html, accessed: 2025-04-24
2. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
3. Bercovich, A., Levy, I., Golan, I., Dabbah, M., El-Yaniv, R., Puny, O., Galil, I., Moshe, Z., Ronen, T., Nabwani, N., et al.: Llama-nemotron: Efficient reasoning models. arXiv preprint arXiv:2505.00949 (2025)
4. 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.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems (NeurIPS) 2020 (2020)
5. Bsharat, S.M., Myrzakhan, A., Shen, Z.: Principled instructions are all you need for questioning llama-1/2, gpt-3.5/4. arXiv preprint arXiv:2312.16171 (2023), code available at https://github.com/VILA-Lab/ATLAS
6. Casalnuovo, C., Sagae, K., Devanbu, P.: Studying the difference between natural and programming language corpora. Empirical Software Engineering **24**(4), 1823–1868 (2019)
7. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
8. DeepMind, G.: Gemini 2.5: Our most intelligent ai model. Tech. rep. (2025), https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/
9. Ding, H., Fan, Z., Guehring, I., Gupta, G., Ha, W., Huan, J., Liu, L., Omidvar-Tehrani, B., Wang, S., Zhou, H.: Reasoning and planning with large language models in code development. In: Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. p. 6480–6490. KDD '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3637528.3671452, https://doi.org/10.1145/3637528.3671452
10. Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al.: The llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024)

11. Fan, L., Chen, M., Liu, Z.: SEK: Self-explained keywords empower large language models for code generation. In: Che, W., Nabende, J., Shutova, E., Pilehvar, M.T. (eds.) Findings of the Association for Computational Linguistics: ACL 2025. pp. 6249–6278. Association for Computational Linguistics, Vienna, Austria (Jul 2025). https://doi.org/10.18653/v1/2025.findings-acl.324, https://aclanthology.org/2025.findings-acl.324/
12. Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al.: Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948 (2025)
13. Han, H., Kim, J., Yoo, J., Lee, Y., Hwang, S.w.: Archcode: Incorporating software requirements in code generation with large language models. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 13520–13552 (2024)
14. Holtzman, A., Buys, J., Du, L., Forbes, M., Choi, Y.: The curious case of neural text degeneration. arXiv preprint arXiv:1904.09751 (2019)
15. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H.: Large language models for software engineering: A systematic literature review **33**(8) (2024)
16. Huang, D., Bu, Q., Qing, Y., Cui, H.: Codecot: Tackling code syntax errors in cot reasoning for code generation. arXiv preprint arXiv:2308.08784 (2023)
17. Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al.: Qwen2.5-coder technical report. arXiv preprint arXiv:2409.12186 (2024)
18. Jain, N., Han, K., Gu, A., Li, W.D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., Stoica, I.: Livecodebench: Holistic and contamination free evaluation of large language models for code. arXiv preprint arXiv:2403.07974 (2024)
19. Jha, S.K., Jha, S., Ewetz, R., Velasquez, A.: Co-synthesis of code and formal models using large language models and functors. In: MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM). pp. 215–220 (2024). https://doi.org/10.1109/MILCOM61039.2024.10773930
20. Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., Jiao, W.: Self-planning code generation with large language models. ACM Transactions on Software Engineering and Methodology **33**(7) (Sep 2024). https://doi.org/10.1145/3672456, https://doi.org/10.1145/3672456
21. Kong, A., Zhao, S., Chen, H., Li, Q., Qin, Y., Sun, R., Zhou, X., Wang, E., Dong, X.: Better zero-shot reasoning with role-play prompting. arXiv preprint arXiv:2308.07702 (2023)
22. Li, J., Li, G., Li, Y., Jin, Z.: Structured chain-of-thought prompting for code generation. ACM Transactions on Software Engineering and Methodology **34**(2) (Jan 2025). https://doi.org/10.1145/3690635, https://doi.org/10.1145/3690635
23. Li, J., Zhao, Y., Li, Y., Li, G., Jin, Z.: Acecoder: An effective prompting technique specialized in code generation. ACM Trans. Softw. Eng. Methodol. **33**(8) (Nov 2024). https://doi.org/10.1145/3675395, https://doi.org/10.1145/3675395
24. Li, X., Zhao, R., Chia, Y.K., Ding, B., Joty, S., Poria, S., Bing, L.: Chain-of-knowledge: Grounding large language models via dynamic knowledge adapting over heterogeneous sources. arXiv preprint arXiv:2305.13269 (2023)
25. OpenAI: Openai o1 system card. Tech. rep., OpenAI (2024), https://openai.com/index/openai-o1-system-card/
26. Peng, Q., Zhang, C., Mangal, R., Pasareanu, C., Jia, L.: Random Perturbation Attack on LLMs for Code Generation . In: 2025 IEEE/ACM

4th International Conference on AI Engineering – Software Engineering for AI (CAIN). pp. 285–287. IEEE Computer Society, Los Alamitos, CA, USA (Apr 2025). https://doi.org/10.1109/CAIN66642.2025.00052, https://doi.ieeecomputersociety.org/10.1109/CAIN66642.2025.00052
27. Qiao, S., Ou, Y., Zhang, N., Chen, X., Yao, Y., Deng, S., Tan, C., Huang, F., Chen, H.: Reasoning with language model prompting: A survey. In: Rogers, A., Boyd-Graber, J., Okazaki, N. (eds.) Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 5368–5393. Association for Computational Linguistics, Toronto, Canada (Jul 2023). https://doi.org/10.18653/v1/2023.acl-long.294, https://aclanthology.org/2023.acl-long.294/
28. Sahoo, P., Singh, A.K., Saha, S., Jain, V., Mondal, S., Chadha, A.: A systematic survey of prompt engineering in large language models: Techniques and applications (2025), https://arxiv.org/abs/2402.07927
29. Wang, X., Wei, J., Schuurmans, D., Le, Q.V., Chi, E.H., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. In: The Eleventh International Conference on Learning Representations (2023), https://openreview.net/forum?id=1PL1NIMMrw
30. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems **35**, 24824–24837 (2022)
31. Xu, B., Yang, A., Lin, J., Wang, Q., Zhou, C., Zhang, Y., Mao, Z.: Expertprompting: Instructing large language models to be distinguished experts. arXiv preprint arXiv:2305.14688 (2023)
32. Yang, G., Zhou, Y., Chen, X., Zhang, X., Zhuo, T.Y., Chen, T.: Chain-of-thought in neural code generation: From and for lightweight language models. IEEE Transactions on Software Engineering (2024)
33. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., Narasimhan, K.: Tree of thoughts: Deliberate problem solving with large language models. Advances in Neural Information Processing Systems **36** (2024)
34. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synergizing reasoning and acting in language models. In: International Conference on Learning Representations (ICLR) (2023)
35. Yeo, S., Hwang, S.W., Ma, Y.S.: Chain of Grounded Objectives: Concise goal-oriented prompting for code generation. In: 39th European Conference on Object-Oriented Programming (ECOOP 2025). pp. 35–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2025)
36. Zheng, Z., Ning, K., Zhong, Q., Chen, J., Chen, W., Guo, L., Wang, W., Wang, Y.: Towards an understanding of large language models in software engineering tasks. Empirical Software Engineering **30**(2) (Dec 2024). https://doi.org/10.1007/s10664-024-10602-0, https://doi.org/10.1007/s10664-024-10602-0

A Examples of Prompts

LiveCodeBench prompt was standardized with function declarations extracted directly from the ground-truth code for consistency. Examples of problem descriptions on HumanEval and LiveCodeBench are provided in Table 5.

Table 5 Examples of Problem Descriptions on HumanEval and LiveCodeBench

HumanEval Prompt

```
def separate_paren_groups(paren_string: str) -> List[str]:
    """
    Input is a string containing groups of parentheses.
    Separate groups into individual balanced strings.
    >>> separate_paren_groups('( () ) (( )) (() ))')
    ['()', '(())', '((()))']
    """

Write a Python code for the problem.
DO NOT generate any test cases or descriptions.
Package your code in ```python ... ```.
```

LiveCodeBench Prompt

```
You are given a string s consisting only of uppercase English
 letters. You can apply some operations to this string where,
 in one operation, you can remove any occurrence of one of
 the substrings "AB" or "CD" from s. Return the minimum
 possible length of the resulting string that you can obtain.
Note that the string concatenates after removing the
 substring and could produce new "AB" or "CD" substrings.

Example 1:
Input: s = "ABFCACDB"
Output: 2
Explanation: We can do the following operations:
- Remove the substring "ABFCACDB", so s = "FCACDB".
- Remove the substring "FCACDB", so s = "FCAB".
- Remove the substring "FCAB", so s = "FC".
So the resulting length of the string is 2.
It can be shown that it is the minimum length thatwe can
 obtain.

Example 2:
Input: s = "ACBBD"
Output: 5
Explanation: We cannot do any operations on the string so the
             length remains the same.

Constraints:
1 <= s.length <= 100
s consists only of uppercase English letters.

class Solution:
    def minLength(self, s: str) -> int:\n

Write a Python code for the problem.
DO NOT generate any test cases or descriptions.
Package your code in ```python ... ```.
```