
ENHANCING COMPUTER PROGRAMMING EDUCATION WITH LLMs: A STUDY ON EFFECTIVE PROMPT ENGINEERING FOR PYTHON CODE GENERATION

A PREPRINT

Tianyu Wang* Mercy University Math & Computer Science Department 555 Broadway, Dobbs Ferry, NY 10522, USA twang4@mercy.edu	Nianjun Zhou* IBM Research 1101 Kitchawan Road, Route 134 Yorktown Heights, NY 10598, USA jzhou@us.ibm.com
Zhixiong Chen Mercy University Math & Computer Science Department 555 Broadway, Dobbs Ferry, NY 10522, USA zchen@mercy.edu	

July 9, 2024

ABSTRACT

Large language models (LLMs) and prompt engineering hold significant potential for advancing computer programming education through personalized instruction. This paper explores this potential by investigating three critical research questions: the systematic categorization of prompt engineering strategies tailored to diverse educational needs, the empowerment of LLMs to solve complex problems beyond their inherent capabilities, and the establishment of a robust framework for evaluating and implementing these strategies. Our methodology involves categorizing programming questions based on educational requirements, applying various prompt engineering strategies, and assessing the effectiveness of LLM-generated responses. Experiments with GPT-4, GPT-4o, Llama3-8b, and Mixtral-8x7b models on datasets such as LeetCode and USACO reveal that GPT-4o consistently outperforms others, particularly with the "multi-step" prompt strategy. The results show that tailored prompt strategies significantly enhance LLM performance, with specific strategies recommended for foundational learning, competition preparation, and advanced problem-solving. This study underscores the crucial role of prompt engineering in maximizing the educational benefits of LLMs. By systematically categorizing and testing these strategies, we provide a comprehensive framework for both educators and students to optimize LLM-based learning experiences. Future research should focus on refining these strategies and addressing current LLM limitations to further enhance educational outcomes in computer programming instruction.

Keywords Prompt Engineering, Large Language Models (LLMs), Computer Programming, Code Quality Evaluation, Education

1 Introduction

In the rapidly evolving field of Artificial Intelligence (AI), Large Language Models (LLMs) [Bommasani et al., 2021, Touvron et al., 2023], a breakthrough in AI, have shown remarkable potential in a variety of applications, including

*These authors contributed equally to this work.

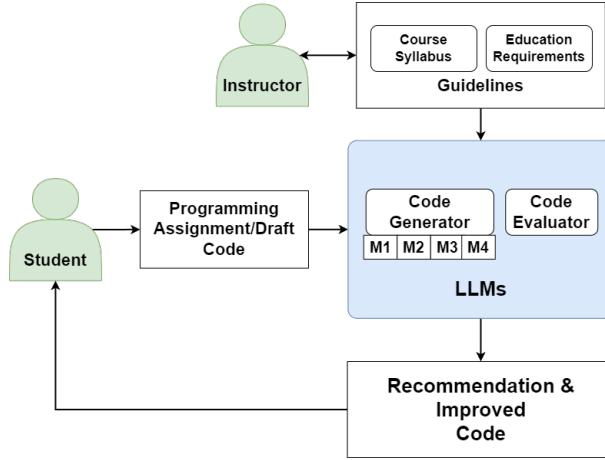


Figure 1: Flowchart illustrating the users in code generation and evaluation process

natural language processing, content creation, and more recently, in code generation. Numerous LLM models, such as ChatGPT and LLaM, have shown their capability to cater to a variety of task domains, as Fig 1 shown, ranging from question answering to the generation of code snippets, [Radford et al., 2019, Bommasani et al., 2021, Ouyang et al., 2022, Liu et al., 2023a, Touvron et al., 2023]. Furthermore, in the evolving landscape of computer science education, integrating advanced technologies has become increasingly pivotal. Many Integrated Development Environments (IDEs) have equipped with LLMs to generate code in software development tools, such as VS Code². As the demand for proficient programmers grows, so does the necessity for innovative and effective teaching methods. These usages of tools draw attention to developing software vulnerabilities and security concerns [Asare et al., 2023, Pearce et al., 2022a, Dakhel et al., 2023a].

This paper presents significant contributions to the field of AI-assisted programming education by focusing on the optimization of Python code generation with LLMs for educational applications. Our research addresses critical aspects of how LLMs can be effectively utilized to create personalized and adaptive learning environments that cater to diverse educational needs. We systematically investigate and categorize prompt engineering strategies, tailoring them to specific educational objectives and problem types. By doing so, we enable educators and students to leverage LLMs in a way that maximizes their problem-solving potential and instructional effectiveness.

Our contributions are threefold:

- 1. Categorization of prompt engineering strategies.** We develop a comprehensive categorization of prompt engineering strategies that align with various educational requirements, ranging from foundational knowledge and skills to competition-level challenges and advanced problem-solving tasks. This categorization provides a structured approach for educators to customize prompts and optimize learning pathways for their students.
- 2. Explore prompt engineering impact on LLM performance in code generation.** We propose a robust framework designed to explore and validate various prompt engineering strategies, assessing their impact on LLM performance in generating Python code. Our findings underscore the effectiveness of different prompt strategies in guiding students through complex problem-solving processes, thereby enhancing the role of LLMs as facilitators in the educational journey.
- 3. Provide a general prompt guidelines for different educational purpose.** Following the evaluation of prompt engineering strategies using various datasets, we present comprehensive guidelines for educators. These guidelines facilitate the systematic use of LLMs to generate superior code and optimize LLM-based learning experiences. By ensuring the efficient application of diverse prompt strategies, these guidelines aim to enhance educational outcomes in computer programming instruction across different educational requirements.

By advancing the application of LLMs in Python code generation for educational purposes, our research contributes to the broader goal of integrating AI technologies into educational practices, paving the way for more dynamic, personalized, and effective learning experiences in the field of computer programming.

²<https://code.visualstudio.com/>

2 Related Work

Recent research in the educational sector has prominently featured the application of Large Language Models (LLMs) to enhance learning outcomes, particularly in the programming domain Denny et al. [2023a]. This review synthesizes findings from key studies that illustrate the diverse roles LLMs play in education, from interactive assistance in computer science courses to the evaluation of programming skills. Murr et al. [2023] focused on the effectiveness of LLMs in generating code, emphasizing the critical role of prompt specificity. Many studies have demonstrated the efficacy of integrating AI code generators in introductory programming courses, as evidenced by research conducted by Finnie-Ansley et al. [2022], Hellas et al. [2023], and Kazemitaar et al. [2023]. In addition, Kiesler and Schiffner [2023] assessed the capabilities of ChatGPT-3.5 and GPT-4 in solving introductory Python programming tasks sourced from CodingBat. Pearce et al. [2022b] explored the application of LLMs in reverse engineering tasks and exhibited promising results. Supporting discussions on LLMs' application in programming education, particularly with development assistant, additional references such as Asare et al. [2023], Pearce et al. [2022a], Dakhel et al. [2023a] and Denny et al. [2023b] provide insights into the integration of AI tools within software development environments. These studies collectively underscore the transformative impact of LLMs on programming education, suggesting avenues for future research in optimizing their use for educational enhancement and addressing broader software development challenges Chen et al. [2021].

While LLMs were impressively successful in generating code for different purpose in education and production, they stumbled when confronting real-world security and risks concerns. Bommasani et al. [2021] examine the potential and risks associated with LLMs and highlights their content creation capabilities and warns about potential issues like bias, misinformation, and homogenization. Dakhel et al. [2023b] leveraged LLMs for generating unit tests in software development. To overcome the inaccurate response from LLM, a new discipline or guideline called 'Prompt Engineering', which includes specific strategies for maximizing the capability of LLM, applies to us with modification [Reynolds and McDonell, 2021, Liu et al., 2023a]. delve into methodologies for leveraging the inherent capabilities of narratives and cultural anchors to intricately encode intentions and strategies, thereby facilitating a structured breakdown of problems into their constituent elements prior to reaching conclusions Reynolds and McDonell [2021]. Expanding upon this notion, introduce a novel approach termed least-to-most prompting, which systematically deconstructs complex issues into manageable sub-problems, addressing them sequentially to enhance problem-solving efficiency in LLMs Zhou et al. [2022]. Complementing these insights, demonstrate the natural emergence of reasoning capabilities within sizable LLMs through what is known as chain-of-thought prompting. This technique involves the use of select demonstrations that guide the model through a thought process, thereby facilitating the comprehension and solving of tasks Wei et al. [2022]. In a similar vein, propose an innovative "Ask Me Anything" (AMA) prompting strategy that iteratively employs the LLM itself to reformulate task inputs into a more effective question-and-answer format, thereby significantly augmenting the performance of LLMs Arora et al. [2022]. Additionally, explore the potential of refining language models' task execution and instruction-following capabilities through the integration of human feedback, marking a significant step towards more interactive and adaptive LLMs Ouyang et al. [2022]. The study by White et al. [2023a] on prompt pattern catalog to enhance prompt engineering with ChatGPT provides a comprehensive overview of best practices and patterns in prompt engineering, highlighting its importance in optimizing LLM outputs for specific tasks.

The code quality generated by LLMs holds paramount importance in applications spanning educational contexts and real-world production environments and many code evaluation studies have been introduced. Hendrycks et al. [2021] unveiled APPS, a benchmark specifically designed for code generation tasks. This benchmark assesses the capability of models to interpret arbitrary natural language specifications and produce Python code that meets the specified requirements. Furthermore, Chen et al. [2021] introduced HumanEval, an innovative evaluation set aimed at measuring the functional correctness of programs synthesized from docstrings. Xu et al. [2022] conducted a comprehensive assessment of the largest code-generating models available, spanning multiple programming languages. They introduced a novel model, PolyCoder, which demonstrated superior performance in generating C programming code, outperforming its counterparts. Liu et al. [2023b] developed EvalPlus, a comprehensive framework for the evaluation of code synthesis. This framework is meticulously designed to benchmark the functional correctness of code generated by LLMs with a high degree of rigor. White et al. [2023b] presented a more systematic methodology for the cataloging of software engineering patterns. This study classifies various patterns and delves into numerous prompt strategies that have been employed to enhance code quality and system design. In a comparative study, Murr et al. [2023] analyzed the efficacy of code produced by different LLMs across 104 customized Python challenges, utilizing widely recognized metrics such as the pass rate for assessment.

Beyond conventional methodologies, deep learning techniques have increasingly been applied to the evaluation of code. Kanade et al. [2020] explored the capabilities of a finely tuned CuBERT model, revealing that it surpasses traditional methods in source code evaluation. This advantage was observed even with limited training and a smaller number

of labeled examples. Ciniselli et al. [2021] presented an empirical study employing a RoBERTa model to assess its efficiency in code completion tasks from various angles. The findings indicate that BERT-based models are a promising avenue for enhancing code completion capabilities. Wang and Chen [2023] proposed an approach for the automatic assessment of code quality using a BERT model that has been meticulously fine-tuned with specific datasets, offering a groundbreaking perspective on the evaluation of code quality.

3 Problem Statement

In computer programming education, LLMs and prompt engineering hold promise for personalized instruction. By harnessing LLMs' capability to comprehend and respond to natural language prompts, we can create adaptive learning environments that dynamically adjust to a education requirement and problem-solving approach. However, a pivotal question persists: **can we fully exploit this potential to enhance educational outcomes?**

This paper investigates this very issue by focusing on three key research questions:

1. **Can we systematically categorize prompt engineering strategies tailored to various educational requirements and question types?** This categorization would enable educators and students to customize prompts according to specific problem types and educational objectives, thereby maximizing the effectiveness of computer programming instruction. Understanding these categories is crucial for developing targeted and efficient learning pathways.
2. **Do different strategies empower LLMs to address problems beyond their immediate solution capabilities?** This exploration aims to unlock the potential for LLMs to guide students through complex or unfamiliar problem-solving processes. By identifying strategies that extend LLM capabilities, we can enhance their role as facilitators in the learning journey, providing support even in challenging scenarios.
3. **Can we establish a robust framework for testing the effectiveness of various prompt engineering strategies and provide comprehensive guidelines for their implementation?** This framework would allow educators to systematically evaluate and optimize LLM-based learning experiences, ensuring maximum benefit for students. By developing and validating these guidelines, we can offer a structured approach to enhance educational practices and outcomes through LLMs.

By addressing these questions, this research aims to pave the way for a future of personalized and adaptive coding education, empowering students to approach problems with confidence and a deeper understanding of core programming concepts.

4 Methodology

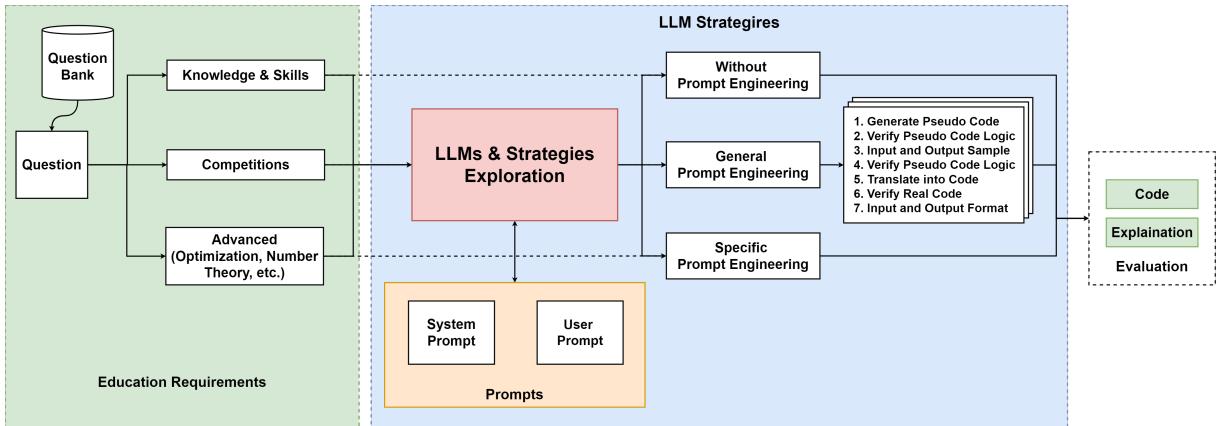


Figure 2: Conceptual Diagram Highlighting the Interaction Between LLMs and Prompt Engineering

To address our research problem, we designed a comprehensive model, as illustrated in Fig 2. This model outlines the process of categorizing questions, applying various prompt engineering strategies, and evaluating the effectiveness of LLM-generated responses. The methodology is structured into three primary steps:

Table 1: Prompt Configurations for Model

Model	System Prompt	User Prompts and Chains
base		Question llm -> Code
example (1-shot)	See Ap- pendix A	Question llm -> Code Example -> Code
dynamic example		Question llm -> Dynamic Related Example -> Code
guide		Question llm -> Code General Guide ³ -> Code
multi		Question llm -> multi-step chat guide llm -> Code
all-in-one		Question All-in-one llm -> Code

1. **Categorization of Questions Based on Educational Requirements.** The first step involves categorizing computer programming questions according to different educational requirements. This categorization helps in tailoring prompt engineering strategies to the specific needs of learners. We classify the questions into three distinct levels:

- **Knowledge and Skills:** This category focuses on fundamental algorithms and data structures, preparing students for coding interviews. It aims to build a solid foundation in programming by familiarizing students with essential concepts and techniques.
- **Competitions:** This category is designed to enhance programming skills for higher-level competitions. It includes problems that challenge students to think critically and innovatively, helping them gain recognition that can benefit their academic and professional careers.
- **Advanced Complex Problems:** This category addresses advanced topics such as algorithm design, optimization, and number theory. It is intended for students who have a strong grasp of basic concepts and are looking to tackle more sophisticated and intricate problems.

2. **Exploration of Prompt Engineering Strategies.** In the next step, we explore different prompt engineering strategies on the categorized questions. There are three major prompt engineering strategies that we employ:

- **Without Prompt Engineering:** In this approach, we present the original question to the LLM without any additional guidance. This strategy tests the LLM’s inherent ability to comprehend and solve problems without external aid.
- **General Prompt Engineering:** This method involves providing some general prompt templates to help the LLM understand the problem better. For example, we include the requirements of question constraints, test cases, and guidelines to assist the LLM in breaking down the problem into logical sub-problems. This approach serves as a framework to improve the LLM’s problem-solving capabilities.
- **Specific Prompt Engineering:** This strategy provides detailed instructions on how to address the problem, rather than using a generic approach. For instance, we may instruct the LLM to consider extreme scenarios or treat the problem as an optimization task. This method is particularly useful for challenging and complex problems that require specialized solutions.

3. **Evaluation of LLM Responses.** After applying the appropriate prompt engineering strategies, we evaluate the LLM-generated responses using different metrics. The evaluation process involves:

- **Correctness:** Assessing the correctness of the generated code. We verify if the code meets the problem requirements and constraints.
- **Effectiveness:** Measuring the overall effectiveness of the LLM’s code. This includes testing the code against efficiency in terms of time and memory complexity. More details about evaluation can be found at Section 5.3.

By following this structured methodology, we aim to systematically investigate the potential of LLMs and prompt engineering in enhancing computer programming education. The insights gained from this research will provide valuable guidelines for educators to optimize LLM-based learning experiences for maximum student benefit.

4.1 General Prompt Engineering

Prompt engineering is crucial in the development and optimization of LLMs due to its significant impact on the performance and accuracy of these models. The choice of prompts directly influences the output generated by LLMs, as they determine how the model interprets and responds to the given input. By experimenting with and testing different prompts for the same question, researchers can identify the most effective phrasing and structure, thereby enhancing the model’s ability to generate relevant, coherent, and contextually appropriate responses. This iterative process of

prompt refinement ensures that the LLMs are not only robust but also adaptable to a wide range of applications, thereby maximizing their utility in various domains. As Table 1 shown, to explore and compare different prompts in LLMs, we use the following prompt settings in our study.

- **Prompt 1 (base):** This configuration serves as our baseline. In this setup, the model is presented with the fundamental structure of a problem statement. The objective is to assess the model’s inherent capability to understand and generate responses with minimal guidance.
- **Prompt 2 (example (1-shot)):** In this prompt, the model is initially provided with the basic structure of a problem statement. Subsequently, we introduce a single of high-quality code example to the LLM. This configuration aims to evaluate the impact of enhanced prompt structures on the model’s output, particularly focusing on how the inclusion of an example influences the generated responses.
- **Prompt 3 (dynamic example):** Similar to prompt 2, this prompt initially presents the problem statement. However, instead of using a static, high-quality code example, the LLM is tasked with generating a related but distinct example. The goal here is to assess the model’s adaptability and ability to produce a code solution based on dynamically generated examples.
- **Prompt 4 (guide):** This prompt incorporates general coding guidelines alongside the problem statement. The intention is to guide the LLM towards generating code that adheres to broad quality standards. Details regarding the content of these general guidelines are provided in Appendix B.
- **Prompt 5 (multi):** This configuration employs a multi-step conversational prompt strategy with LLMs. Initially, it provides a multiple turns of interaction, where the model responds to ongoing inputs that build on previous context. Instructors or students evaluate each suggestion, providing feedback to guide the LLM towards optimal solutions. This approach usually suits problems requiring sustained engagement like tutoring or solving complex problems. Specifically,
 1. **Generate Pseudo Code:** The initial step involves generating pseudo code based on the question. This pseudo code acts as an intermediate representation of the solution, outlining the logical steps necessary to solve the problem without delving into specific syntax. The generated pseudo code serves as the foundation for subsequent verification and refinement processes.
 2. **Verify Pseudo Code Logic:** In the second step, the generated pseudo code is subjected to a logic verification process. This step ensures that the pseudo code accurately reflects a viable solution to the problem by identifying and correcting any logical errors. The verified pseudo code provides a reliable blueprint for generating sample inputs and outputs.
 3. **Input and Output Sample:** Following the verification of pseudo code logic, the LLM takes sample input and output pairs from original question. These samples are used by LLM to better understand the question and play the important role for testing the correctness of the logical flow in pseudo code.
 4. **Verify Code Logic:** The focus here is on ensuring that the code from previous step behaves as expected when provided with the sample inputs, thereby producing the correct outputs. This verification process is essential for validating the logical coherence of the solution before final translating into real programming code.
 5. **Convert into Code:** After checked the logic of pseudo code, this steps focus on the implementation of translating pseudo code into real programming code.
 6. **Verify Code Logic:** The focus here is on ensuring that the translated code behaves as expected when provided with the pseudo code logic, sample inputs, thereby producing the correct outputs.
 7. **Input and Output Format:** The final step involves refining the code to adhere to the specified input and output format requirements. This step ensures that the code meets the problem’s specifications in terms of structure and presentation. The output from this step is the finalized code, which is expected to solve the given problem accurately.
- **Prompt 6 (all-in-one):** This model includes all the prompt configurations from Prompt 5 (multi). However, instead of using a multi-step conversational process, all prompt settings are consolidated into one single prompt. The objective is to compare the efficacy of multi-step prompting versus an all-in-one approach.

By employing these diverse prompts, our study systematically investigates the influence of different prompt structures on the performance of LLMs, thereby providing insights into the optimal strategies for prompt engineering in the context of LLM-driven code assessment and guidance.

4.2 Problem-Specific Prompts

These are often unusual coding problems, commonly seen in coding competitions, complex optimization problem, or even number theory problems, where a universal guide prompt is ineffective. Detailed, problem-specific prompts are required to navigate the intricacies of such problems. For example, advanced dynamic programming challenges or problems involving intricate mathematical concepts typically require bespoke prompts that provide in-depth, step-by-step guidance tailored to the specific problem at hand. We will discuss this category in details in section 6.2.

5 Experiments

5.1 Dataset

Numerous sources and coding question banks (benchmarks) are available to assess the capabilities of AI code generation. The majority of these code questions are publicly accessible, having been extensively analyzed and discussed, thus serving as training data for most large language models (LLMs). Identifying a unique, diverse, and challenging coding dataset that has not been extensively used in training LLMs remains a significant challenge. In this study we used two data source as our dataset: 1) LeetCode 2) USACO.

5.1.1 LeetCode Dataset

We start with LeetCode⁴ as our experiment dataset, due to its comprehensive range of well-defined and real-world-relevant programming problems. The structured nature of these problems enhances the AI's ability to process and generate solutions, providing a consistent benchmark for performance analysis. Furthermore, LeetCode holds significant educational value; it offers students practical applications of theoretical concepts, enhancing learning through exposure to diverse problem-solving techniques. For our experiments, we selected and compiled the first 100 questions from LeetCode (1c100), including both questions and test cases.

A Sample of LeetCode Question

Problem Content: Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.

Test Input: nums = [2, 7, 11, 15], target = 9

Test Output: [0,1]

5.1.2 USACO Dataset

Given that the LeetCode dataset has been widely published and discussed and extensively utilized for training by many current LLMs, Brown et al. [2020], we aimed to identify a less commonly used dataset for our analysis. In this context, we have identified the United States of America Computing Olympiad (USACO)⁵, an online computer programming competition that serves as a qualifier for the International Olympiad in Informatics in the USA, as a valuable data source. USACO provides a diverse set of coding problems, categorized by difficulty levels: Bronze, Silver, Gold, and Platinum. This dataset is particularly valuable because it is less commonly used as a training dataset by major LLMs, thereby offering a fresh and underutilized resource for evaluation. For our study, we selected relatively easy questions from the Bronze category, randomly picking 20 questions from different years spanning 2016 to 2023. This dataset is subsequently referred to as usaco20.

⁴<https://leetcode.com/>

⁵<https://usaco.org/>

A Sample of USACO Question

Problem Content: Farmer John has lost his prize cow Bessie, and he needs to find her! Fortunately, there is only one long path running across the farm, and Farmer John knows that Bessie has to be at some location on this path. If we think of the path as a number line, then Farmer John is currently at position x and Bessie is currently at position y (unknown to Farmer John). If Farmer John only knew where Bessie was located, he could walk directly to her, traveling a distance of $|x - y|$. Unfortunately, it is dark outside and Farmer John can't see anything. The only way he can find Bessie is to walk back and forth until he eventually reaches her position. Trying to figure out the best strategy for walking back and forth in his search, Farmer John consults the computer science research literature and is somewhat amused to find that this exact problem has not only been studied by computer scientists in the past, but that it is actually called the "Lost Cow Problem" (this is actually true!). The recommended solution for Farmer John to find Bessie is to move to position $x + 1$, then reverse direction and move to position $x - 2$, then to position $x + 4$, and so on, in a "zig zag" pattern, each step moving twice as far from his initial starting position as before. As he has read during his study of algorithms for solving the lost cow problem, this approach guarantees that he will at worst travel 9 times the direct distance $|x - y|$ between himself and Bessie before he finds her (this is also true, and the factor of 9 is actually the smallest such worst case guarantee any strategy can achieve). Farmer John is curious to verify this result. Given x and y , please compute the total distance he will travel according to the zig-zag search strategy above until he finds Bessie.

Testing Format:

INPUT FORMAT (file `lostcow.in`): The single line of input contains two distinct space-separated integers x and y . Both are in the range $0 \dots 1,000$.

OUTPUT FORMAT (file `lostcow.out`): Print one line of output, containing the distance Farmer John will travel to reach Bessie.

Testing Case:

SAMPLE INPUT:3 6 SAMPLE OUTPUT:9

5.2 LLM Models

In this study, we recognize the vast and rapidly evolving landscape of LLMs, which includes numerous models of significance. Given the impracticality of listing all of them, we focus on the most popular and practically usable models, both open-source and closed-source. Our comparative analysis includes the following prominent LLMs: ChatGPT (gpt-4-turbo)⁶, ChatGPT (gpt-4o)⁷, LLAMA (llama3-8b)⁸, and Mistral (mixtral-8x7b)⁹. Detailed configurations of these models are delineated in the Appendix under section A.

5.3 Evaluation Methods

For the LeetCode dataset, the website¹⁰ provides all test cases and expected results for each question. Given that LeetCode questions are widely used in coding interviews, it is crucial to employ a comprehensive set of evaluation criteria that capture both functional correctness and code efficiency. Therefore, we utilize three primary indicators: pass rate (correctness), time spent, and Pylint score, each selected for their distinct contributions to a holistic evaluation.

1. **Pass Rate:** The pass rate serves as a fundamental measure of correctness and reliability. It quantifies the proportion of test cases that a solution correctly handles, directly reflecting its ability to meet the problem's specifications under various scenarios. This metric is crucial as it directly correlates with the primary goal of any coding solution—its accuracy and functionality.
2. **Time Spent:** This metric evaluates the efficiency of the problem-solving process. It encompasses the duration from initiating the coding of the solution to its successful execution and debugging. Monitoring time spent is essential for understanding the complexity and efficiency of the solution from a practical standpoint. In competitive programming, where time efficiency is as critical as correctness, this metric provides insights into the algorithm's performance under time constraints.
3. **Pylint Score:** As a widely recognized tool for code analysis in Python, Pylint assesses code quality based on a set of coding standards and heuristics. The Pylint score is indicative of the maintainability, readability, and structural quality of code. High scores suggest adherence to Python coding conventions and best practices,

⁶<https://openai.com/research/gpt-4>

⁷<https://openai.com/index/hello-gpt-4o/>

⁸<https://llama.meta.com/llama3/>

⁹<https://mistral.ai>

¹⁰<https://leetcode.com/problemset/>

Submitted; Results below show the outcome for each judge test case																													
1	*	10.7mb 69ms	2	*	10.7mb 69ms	3	*	10.7mb 69ms	4	*	10.7mb 70ms	5	*	10.7mb 69ms	6	*	10.7mb 69ms	7	*	10.7mb 69ms	8	*	10.7mb 69ms	9	*	10.7mb 70ms	10	*	10.7mb 69ms

Figure 3: A screenshot of the USACO evaluation system displaying user submission results (all pass).

which are pivotal for long-term code maintenance and clarity. By including this metric, the evaluation encompasses not only the functional aspects but also the quality of the coding practices employed.

For USACO dataset, their website provides a user submission evaluation system¹¹. All submitted code solutions are evaluated and scored against a set of predetermined test cases, considering not only correctness but also time and memory usage (Fig. 3). In this study, we use the USACO website to submit code generated by large language models (LLMs). A passing example is defined as code that successfully passes all test cases on their website.

At last, our experiment was conducted within a meticulously controlled and isolated virtual setting. By adopting this method, we guarantee the results' reliability and uniformity, offering a transparent and impartial evaluation of each LLM's effectiveness in addressing coding challenges.¹²

6 Result Analysis

6.1 LeetCode Results

Table 2: Model Comparisons (Accuracy (%), the higher, the better)

	base	example (1-shot)	dynamic example	guide	multi	all-in-one
gpt-4	98%	99%	99%	99%	99%	97%
gpt-4o	97%	98%	88%	97%	100%	96%
llama3-8b	94%	93%	85%	79%	74%	74%
mixtral-8x7b	75%	66%	66%	75%	67%	74%

Table 3: Model Comparison (Time Spent (ms), the lower, the better)

	base	example (1-shot)	dynamic example	guide	multi	all-in-one
gpt-4	4095	3988	3999	3994	3959	3984
gpt-4o	4604	4431	4589	4430	4371	4129
llama3-8b	4325	4173	4238	4216	4142	4196
mixtral-8x7b	4180	4097	4017	3954	3939	4032

In our experiments, we evaluated the performance of various models using different prompting strategies on easily solvable coding problems. The results, summarized in Table 2, indicate that GPT-4 and GPT-4o consistently outperform Llama3-8b and Mixtral-8x7b in terms of pass rate. Notably, GPT-4o achieved a perfect pass rate of 100% with the "multi" prompt strategy, highlighting its adaptability and efficiency. GPT-4 also demonstrated high reliability with pass rates predominantly at 99%, except for minor variations in the base scenario (98%) and all-in-one prompt (97%). Llama3-8b and Mixtral-8x7b showed lower adaptability, with Llama3-8b performing best in the base scenario (94%) and declining in more complex prompts.

Regarding time efficiency, as shown in Table 3, GPT-4 required the least time to execute across all prompts, with the shortest time recorded at 3959 milliseconds for the "multi" prompt. This suggests that GPT-4 is both effective and efficient. GPT-4o, while achieving the highest pass rate, exhibited slightly longer execution times, ranging from 4371 milliseconds ("multi") to 4604 milliseconds ("base"). Llama3-8b and Mixtral-8x7b generally required more time than GPT-4 but less than GPT-4o.

Pylint scores, reported in Table 4, reveal that all models achieved high scores, indicating good adherence to coding standards. GPT-4 generally exhibited the highest Pylint scores, especially under the "multi" prompt, scoring 9.66.

¹¹<https://usaco.guide/general/usaco-faq?lang=py>

¹²We repeatedly tested using different high-quality code examples, and the results remained stable with no significant performance variations.

Table 4: Model Comparisons (Pylint Score, the higher, the better)

	base	example (1-shot)	dynamic example	guide	multi	all-in-one
gpt-4	9.59	9.58	9.63	9.56	9.66	9.38
gpt-4o	9.34	9.49	9.25	9.00	9.62	9.52
llama3-8b	9.18	9.14	10.00	8.64	8.41	7.24
mixtral-8x7b	9.05	9.17	8.27	8.04	8.27	7.74

GPT-4o displayed variability in scores, with the highest being 9.62 ("multi") and the lowest 9.00 ("guide"). Llama3-8b and Mixtral-8x7b showed lower and more variable scores, with particularly low scores in the "all-in-one" and "guide" prompts, respectively. Notably, the "dynamic example" prompt resulted in a perfect score of 10.00 for Llama3-8b, though this was an outlier.

In summary, our findings indicate that while the base prompt and GPT-4 family models (GPT-4, GPT-4o) already exhibit exceptional performance, the multi-step prompt strategy offers limited improvement. This is primarily due to the widespread public discussion of questions like those on LeetCode, and the fact that the data used in training most LLMs, such as Common Crawl, already includes these questions and their solutions Brown et al. [2020]. Consequently, for simpler problems, more complex prompting strategies provide only incremental benefits, and using LLMs without additional prompts is sufficient.

6.2 USACO Results

Table 5: Results of USACO Dataset (Pass Example & Accuracy)

	Solvable (base prompt)	Solvable (multi prompt)	Solvable¹³ (multi+spec prompt)	Currently Not Solvable
Question ID (cpid)	639, 737, 761, 766, 807, 939	639, 641, 735, 737, 739, 760, 761, 766, 807, 939, 1228	639, 641, 735, 737, 739, 760, 761, 766, 783, 785, 787, 807, 939, 1228, 1323	644, 738, 808, 1035, 1131
Count #	6	11	15	5
% (out of 20)	30%	55%	75%	25%

We summarize all the pass examples in Table 5. Given multi-step prompt (`multi`) and GPT-4o model demonstrated superior performance in our previous experiments on the LeetCode dataset, we compare three prompts (`base`, `multi`, and `spec`) with the GPT-4o model on the USACO dataset.

The base prompt configuration solved only 6 out of 20 problems (30%). This indicates that LLMs struggle to generate valid code solutions without any form of prompt engineering. The low success rate suggests that minimal guidance in problem statements does not sufficiently leverage the capabilities of LLMs for effective problem-solving in computer programming.

In contrast, the multi prompt configuration showed significant improvement, solving 11 out of 20 problems (55%). The Question IDs exclusive to the multi prompt (questions: 641, 735, 739, 760, 1228) are highlighted in bold in the table. This improvement underscores the advantages of multi-step conversational prompts. These prompts enhance the LLMs' contextual understanding, allow for iterative refinement, ensure logical coherence, and encourage user engagement and feedback. This method is particularly well-suited for complex problem solving as it incorporates thorough verification and validation processes, making it superior to the without-prompt (`base`) approach.

The multi+spec prompt configuration further extends the LLMs' capabilities, solving 15 out of 20 problems (75%). This approach integrates specific question-related prompts (highlight in bold), such as analyzing the original code's time complexity and suggesting more efficient approaches (question 785), or listing all possible corner cases in a problem (question 783). Although these specific prompts cannot be generalized into a single prompt applicable to all questions, tailored prompts provide more detailed scenarios and considerations, thereby augmenting the LLMs' problem-solving abilities for complex tasks.

Despite these advancements, some problems remain unsolved by the LLMs, even with enhanced prompts (questions: 644, 738, 808, 1035, 1131). These challenging problems typically require complex logical reasoning, sequential decision-making, and optimization under constraints. They involve maintaining context across multiple stages, understanding

¹³Including solvable multi prompt and partial solvable spec prompts.

intricate relationships between variables, and handling combinatorial tasks. For example, ensuring a farm remains fully connected after each barn closure or optimizing cow milking time based on pairings involves high-order planning and sophisticated problem-solving skills. These tasks often surpass the capabilities of current LLMs due to limitations in context retention, logical reasoning, and handling numerical and combinatorial complexities.

The results of our study indicate that prompt engineering significantly enhances the problem-solving capabilities of LLMs in the context of computer programming education. The multi-step prompts, in particular, demonstrate the potential to guide LLMs through complex problem-solving processes by providing a structured and iterative approach. We will discuss the usages of different strategies in prompt engineering in the next section.

7 Discussion

7.1 LLM Model in Code Generation

The results from our experiments indicate a clear recommendation for the use of the GPT-4 and its optimized variant GPT-4o as the preferred LLMs for educational purposes in computer programming instruction. The consistently high performance of these models across various prompting strategies highlights their adaptability, efficiency, and robustness. GPT-4o, in particular, demonstrated a perfect pass rate in the "multi" prompt strategy, suggesting its superior capability in understanding and generating accurate responses in complex problem-solving scenarios. The slightly higher execution times of GPT-4o compared to GPT-4 are a reasonable trade-off for its enhanced performance and adaptability. Therefore, we recommend GPT-4o as the primary model for educational settings, especially where complex problem-solving and iterative refinement processes are critical.

7.2 Prompt Strategies Based on Educational Requirements

Furthermore, to maximize the effectiveness of LLMs in different educational contexts, we propose tailored prompt strategies for three distinct educational requirements: regular knowledge and skills, competition preparation, and advanced problem-solving.

1. **Knowledge and Skills:** For foundational learning and skill-building in computer programming, no-prompt engineering approaches suffice. This strategy involves presenting the LLM with questions that include problem content, constraints, and test cases. Since most fundamental computer science knowledge and exercises are already well-represented in LLM training datasets, this structured prompting method effectively aids in developing essential skills necessary for class exercise, basic algorithmic problem-solving and coding interviews. The method's systematic nature supports the learners in acquiring a solid foundation in programming concepts and techniques.
2. **Competition Preparation** (e.g., USACO): In the context of preparing for programming competitions, such as USACO, the "Multi-Step Conversational Prompt" strategy is most effective. This approach allows for a dynamic interaction between the LLM and the student, where multiple turns of feedback and refinement are possible. The steps involved in generating pseudo code, verifying logic, and iteratively refining the solution are particularly beneficial for complex and competitive problems that require critical and innovative thinking. This strategy enhances the LLM's contextual understanding and ensures logical coherence, making it ideal for high-stakes competitive scenarios.
3. **Advanced Problem-Solving:** For tackling advanced and complex problems, the "Specific Prompt Engineering" strategy is recommended. This method provides detailed instructions and considerations specific to the problem at hand, such as treating the problem as an optimization task or considering extreme scenarios. The focused guidance helps in addressing sophisticated topics like algorithm design, optimization, and number theory. This strategy is essential for students who already have a strong grasp of basic concepts and are looking to deepen their understanding and tackle more intricate problems. The integration of tailored prompts, as seen in the "multi+spec" prompt configuration, further extends the LLMs' capabilities, making it suitable for advanced educational objectives.

7.3 Future work

As LLMs continue to advance, the use of multi-round prompts with various strategies remains vital in addressing complex problems. Future research will target several pivotal areas to augment LLM capabilities through prompt engineering. Initially, we will extend our testing of prompt engineering strategies on more challenging datasets, including more competition and number theory problems, to evaluate and refine our approaches across a broader

spectrum of difficult issues. Furthermore, our research will emphasize improving context retention, enhancing logical reasoning, and better managing numerical and combinatorial complexities. Developing sophisticated prompt engineering techniques tailored to these challenging problem types is essential for optimizing the educational potential of LLMs. By systematically categorizing and testing various prompt engineering strategies, we aim to establish a robust framework incorporating retrieval-augmented generation (RAG), multi-agents system, and Plan-and-Execute tools for educational implementation. This framework will provide educators with comprehensive guidelines to optimize LLM-based learning experiences, ultimately enhancing educational outcomes in computer programming instruction.

8 Conclusion

This paper explores the potential of prompt engineering in large language models (LLMs) to enhance educational outcomes in computer programming instruction. Our research is driven by three key questions: the systematic categorization of prompt engineering strategies tailored to educational requirements, the empowerment of LLMs to solve complex problems, and the establishment of a robust framework for testing and implementing these strategies.

Our findings indicate that the GPT-4 and GPT-4o models outperform other LLMs such as Llama3-8b and Mixtral-8x7b in terms of pass rates, execution times, and adherence to coding standards. The GPT-4o model, in particular, demonstrated a successfully pass rate with the "multi" prompt strategy, highlighting its superior adaptability and efficiency. These results lead us to recommend GPT-4o as the preferred model for educational purposes in computer programming.

We propose tailored prompt strategies based on educational requirements. For foundational learning and skill-building, such as LeetCode, asking question directly without prompt engineering is suffice, providing structured guidance that helps students grasp essential concepts and techniques. For competition preparation, such as USACO, the "Multi-Step Conversational Prompt" strategy proves beneficial, facilitating dynamic interaction and iterative refinement that enhance contextual understanding and problem-solving skills. For advanced problem-solving, the "Specific Prompt Engineering" strategy is ideal, offering detailed instructions that address complex topics like algorithm design and optimization.

Our study also highlights the significant role of prompt engineering in maximizing the potential of LLMs in educational contexts. By categorizing and testing various strategies, we have established a robust framework for their implementation, providing educators with comprehensive guidelines to optimize LLM-based learning experiences. Despite the advancements, certain complex problems remain challenging for current LLMs, suggesting the need for further research to enhance context retention, logical reasoning, and handling of numerical and combinatorial complexities.

In conclusion, prompt engineering significantly enhances the capabilities of LLMs in computer programming education. The tailored strategies we propose align with specific educational objectives, from foundational learning to advanced problem-solving. The superior performance of GPT-4 and GPT-4o confirms their suitability for a wide range of educational applications. By adopting and refining these strategies, educators can significantly improve educational outcomes, providing students with a more effective and personalized learning experience in computer programming.

References

- Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023a.
- Owura Asare, Meiyappan Nagappan, and N Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering*, 28(6):129, 2023.

- Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh 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, 2022a.
- Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203: 111734, 2023a.
- Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. Computing education in the era of generative ai. *arXiv preprint arXiv:2306.02608*, 2023a.
- Lincoln Murr, Morgan Grainger, and David Gao. Testing llms on code generation with varying levels of prompt specificity. *arXiv preprint arXiv:2311.07599*, 2023.
- James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, pages 10–19, 2022.
- Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. Exploring the responses of large language models to beginner programmers' help requests. *arXiv preprint arXiv:2306.05715*, 2023.
- Majeed Kazemitaabar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. Studying the effect of ai code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–23, 2023.
- Natalie Kiesler and Daniel Schiffner. Large language models in introductory programming education: Chatgpt's performance and implications for assessments, 2023.
- Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering?, 2022b.
- Paul Denny, Brett A Becker, Juho Leinonen, and James Prather. Chat overflow: Artificially intelligent models for computing education-renaissance or apocalypses? In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 3–4, 2023b.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective test generation using pre-trained large language models and mutation testing. *arXiv preprint arXiv:2308.16557*, 2023b.
- Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2021.
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.
- Simran Arora, Avanika Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, Frederic Sala, and Christopher Ré. Ask me anything: A simple strategy for prompting language models. *arXiv preprint arXiv:2210.02441*, 2022.
- Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023a.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.
- Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023b.

- Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*, 2023b.
- Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR, 2020.
- Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of bert models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 108–119. IEEE, 2021.
- Tianyu Wang and Zhixiong Chen. Analyzing code text strings for code evaluation. In *2023 IEEE International Conference on Big Data (BigData)*, pages 5619–5628. IEEE, 2023.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

A Prompts in This Research

Within the domain of LLMs, such as the entity engaged in the current interaction, the terminologies `system prompts` and `user prompts` delineate distinct categories of input stimuli instrumental in directing the model's output generation.

- **System Prompt:** The system prompt is used to define the persona that LLM will play. In our application, it is part of the system's design to help guide the LLM in generating responses or performing code generation to ensure it plays as a sophisticated programmer professional or mentor.
- **User prompts:** They are the prompts specifying the questions, commands, or statements that users input into the system to seek a response from the LLM (played as a programming tutor). In essence, this encompasses the textual or verbal input furnished by the user to the model.

The following figures presents illustrative examples of prompts utilized within the scope of our manuscript.

System Prompt: Guru for Python Programming

You are an AI code assistant. Provide your solution in Python 3. Include only the code, without explanations or comments.

Figure 4: System Prompt

User Prompt: Question

Here is the question: {question}

User Prompt: Example

You are tasked as a professional coding assistant. Your assignment involves receiving a segment of code, designated as `{input_code}`. Your responsibility is to refine and enhance this code, ensuring the output is of high quality. For reference, an example of the high-quality code expected is provided: `{high_quality_code}`. Show me code only.

Figure 5: User Prompt (Base and Example)

B General Guidelines for High-Quality Python Code

1. **Use of Standard Libraries:** The code effectively utilizes Python's standard libraries such as `re` (for regular expressions), `heapq` (for heap queue algorithms), `bisect` (for array bisection algorithms), and `math`.
Best Practice: Always prefer standard libraries for common algorithms. Do not import unnecessary libraries.
2. **Use Proper Data Structures:** Fundamental building blocks for linked lists and binary trees. These structures are essential for representing hierarchical or sequential data. `Stack`: Implements a basic data structure, like `Node`, `ListNode`, `TreeNode`: a stack, a queue, a linked list, or a binary tree.
Best Practice: Create and choose the right data structure based on the problem requirements for efficiency in both time and space complexity.
3. **Itertools Usage:** The use of `itertools` for permutations, combinations, and cartesian product is a sign of advanced Python knowledge. These functions are crucial for solving combinatorial problems.
Best Practice: Leverage `itertools` to write cleaner and more efficient looping constructs.
4. **Heap Queue (`heapq`):** The `heapq` module is used for implementing priority queues. This is vital in algorithms where you need to repeatedly access the smallest or largest element.

User Prompt: Dynamic Example

You are tasked as a professional coding assistant. Your task is to interpret coding challenges within a specific category, marked as category You are tasked with generating a Python code example that relates closely to the provided question but is distinctively different. The question is marked as: question
Your generated code example should focus on optimizing both processing speed and memory usage, thus enhancing the code's overall effectiveness and efficiency. For each category, you should provide:

- Best Practices: Outline the essential coding practices and principles to follow.
- Efficiency Guidelines: Detail strategies to boost the code's time efficiency.
- Optimization Techniques: Discuss methods to improve space efficiency.
- Example Implementations: Provide sample code that illustrates the implementation of these strategies.

Your contributions are vital in ensuring that these code examples are comprehensive, precise, and tailored to meet the unique demands of each category, thereby facilitating the development of advanced coding solutions.

User Prompt: General Guide

You are tasked as a professional coding assistant. Your assignment involves receiving a segment of code, designated as {input_code}. Your responsibility is to refine and enhance this code, ensuring the output is of high quality. For reference, a guideline of high-quality code is provided: {general_guide}

Figure 6: User Prompt (Dynamic Example, General Guide)

Best Practice: Use heapq for priority queue implementation if necessary instead of a sorted list for better performance.

5. **Functional Programming:** Usage of functools like reduce, cache, and lru_cache indicates a functional approach to problem-solving. This is efficient for operations that benefit from memoization or reducing a list.
Best Practice: Employ functional programming concepts where applicable to make the code more concise and readable.
6. **Math and Random Modules:** The inclusion of math operations and random for random number generation is suitable for problems involving math computations and stochastic processes.
Best Practice: Understand and utilize the vast array of functions provided by these modules to simplify complex calculations.
7. **Efficiency and Optimization:** The use of bisect for binary searches and heapq for efficient element access in priority queues shows an understanding of algorithmic efficiency.
Best Practice: Always consider time and space complexity; optimize code where necessary but avoid premature optimization.
8. **Number Theory:** When solving problems related to number theory, it is essential to leverage mathematical properties and efficient algorithms. Utilize functions from the math module for operations like finding greatest common divisors (gcd), least common multiples (lcm), and performing modular arithmetic.
Best Practice: Use built-in functions and algorithms for common number theory operations. Optimize prime-related computations using the Sieve of Eratosthenes for prime generation and employ efficient algorithms for primality testing and factorization to handle large numbers effectively.

C Hyper-parameters in LLMs

In this research paper, we choose and compare three popular LLMs models: gpt-4-turbo, gpt-4o, meta-llama-3-8b-instruct and open-mixtral-8x7b. The details configurations can be found at Table 6. We set the maximum token limit to 4096, allowing for extensive and detailed responses. Our experiments were conducted with a single response output ($n = 1$), ensuring focused and specific replies and each LeetCode question will only generate one code solution. To balance creativity with relevance, we use the default temperature 0.7, a moderate setting that encourages a mix of predictable and innovative responses. The top_p parameter was set to 1, enabling the model

User Prompt: Category

You are tasked as a professional coding assistant. Your primary task involves analyzing coding questions presented in the following format: {question} Your duty is to categorize each question into the most relevant categories. Possible categories include:

- Data Structures: This includes Arrays, Strings, Linked Lists, Stacks, Trees, Heaps, Graphs, Hash Tables, etc.
- Algorithms: Categories include Sorting, Searching, Dynamic Programming, Recursion, Backtracking, and Divide and Conquer techniques.
- Problem Solving: This covers Mathematical problems, Simulation and Implementation challenges, Two Pointers techniques, Sliding Window approaches, and Greedy Algorithms.
- Graph Algorithms: Focuses on Traversal techniques, Depth-First Search (DFS), and Breadth-First Search (BFS).
- Optimization and Miscellaneous Techniques: Includes Recursion, Iterative Solutions, Functional Programming, Bit Manipulation, and String Operations.

Your responsibility is to ensure each question is properly sorted into its appropriate category for further analysis and discussion.

Figure 7: User Prompt (Category Question)

Table 6: Hyper-Parameter of LLMs

Parameter	Models
Engine	gpt-4-turbo, gpt-4o, meta-llama-3-8b-instruct, open-mixtral-8x7b
Max Token	4096
n (responses)	1
Temperature	0.7
Top P	1
Frequency Penalty	0
Presentation Penalty	0.6

to consider the full range of possible next words. We applied a frequency penalty of 0.0 to prevent repetition and a presence penalty of 0.6, encouraging the model to introduce new concepts and ideas throughout the conversation. This configuration was pivotal in achieving the desired balance between coherence, relevance, and novelty in the model's responses.

User Prompt: Specific Guide

You are tasked as a professional coding assistant. Your task is to interpret coding challenges within a specific category, marked as {category}. You are expected to produce detailed recommendations and strategies that focus on enhancing the code's effectiveness and efficiency in terms of both processing speed and memory usage. For each category, you should provide:

- Best Practices: Outline the essential coding practices and principles to follow.
- Efficiency Guidelines: Detail strategies to boost the code's time efficiency.
- Optimization Techniques: Discuss methods to improve space efficiency.
- Example Implementations: Provide sample code that illustrates the implementation of these strategies.

Your contribution is essential in ensuring these guidelines are thorough, accurate, and adapted to the specific requirements of each category, supporting the development of superior coding solutions.

User Prompt: Specific Guide to Code

You are tasked as a professional coding assistant. Your task is to address a coding question presented as: {question}. Your responsibility is to generate high-quality Python code in response to this query. For reference, a guideline of high-quality code is provided: {specific_guide}

Figure 8: User Prompt (Category and Guide)

User Prompt: Multi-round for Complex Problem

1. Help me generate the pseudo code for this problem: {question}
2. {response} Please double check your logic with the problem description. Ensure your logic is correct.
3. {response} Here is the sample input and output. Can you verify your logic again with the sample input/output. {Input} {Output}
4. {response} Can you convert this into Python code
5. {response} Double check the Python code to ensure the logic correction.
6. {response} Here is the format needed for more testing. Can you update the code for the format requirement: {Format}

Figure 9: User Prompt (Multi-round Guide)