Proceedings of the 2nd International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/73/20240367

An empirical study of prompt mode in code generation based on ChatGPT

Huaiyu Guo

Capital University of Economics and Business, Flower-Town, Fengtai District, Beijing 100070, China.

koheyo77@gmail.com

Abstract. In recent years, with the continuous advancement of technologies such as Large Language Models (LLMs) and Chat Generative Pre-trained Transformer (ChatGPT), an increasing number of developers have turned to AI-assisted code generation. However, in the context of code generation, simple question-and-answer approaches may not yield the desired results. To address this challenge, we introduce prompt engineering as a means to construct efficient prompting methods for guiding models in generating the intended code. This paper empirically explores the impact of different prompting methods on code-generation tasks. We introduce several prompt-sensitive code tasks in our experiments and assess the effectiveness of various prompt methods in terms of the quality of generated code. Ultimately, we find that guiding the model from a specific role perspective yields the best results, while other methods exhibit varying degrees of effectiveness. This research provides valuable insights into the application of prompt engineering in code generation, encouraging future efforts to further optimize prompting methods and enhance the accuracy and practicality of generated code.

Keywords: Large Language Model, ChatGPT, Prompt Engineering, Code Generation

1. Introduction

The development of Large Language Models (LLMs) has injected new vitality into the field of Natural Language Processing (NLP). Models such as the GPT series have demonstrated outstanding performance in generating various forms of text, including articles, translations, conversations, and code. However, in practical code generation tasks, simple question-and-answer approaches may not yield sufficiently accurate results. To enhance the quality of generated code, the introduction of prompt engineering has become a highly regarded direction. Prompt engineering aims to guide models in generating code that meets expected standards and is functionally complete by constructively designing prompts. However, existing research lacks systematic and empirical analysis to validate the practical effectiveness of prompt engineering in code generation.

This paper seeks to address the shortcomings in existing work by conducting empirical research to explore how different prompting methods can guide the behavior of large language models in code generation tasks. Our research is divided into several steps. First, an overview of the development of large language models and their role in code generation will be provided. Second, it will delve into the characteristics and applications of Chat Generative Pre-trained Transformer (ChatGPT). Subsequently, the concept and methods of prompt engineering and analyze its role in code generation will be introduced. Finally, an experimental design to systematically evaluate the effects of various prompting methods in code generation tasks will be proposed.

Through this research, the use of prompt engineering methods in code generation tasks and exploring the effects of different prompting methods from multiple perspectives will be systematically investigated. By quantitatively analyzing and experimentally comparing these methods, we summarize the impact of various prompt attributes on the quality of generated code, providing a deeper understanding and guidance for prompt engineering in code generation tasks. Furthermore, we aim to drive the development of prompt engineering in the field of natural language generation, offering more accurate and efficient methods for practical applications.

2. Background

2.1. LLM and ChatGPT

Large Language Models (LLMs) have become a vital research direction in the fields of artificial intelligence and Natural Language Processing (NLP) [1]. They are employed for understanding and generating natural language text. ChatGPT (Chat Generative Pre-trained Transformer) is a representative LLM developed by the OpenAI team. Built upon the Transformer architecture [2], ChatGPT is an evolution of GPT-1 [3], GPT-2 [4], and GPT-3 [5]. Before the advent of GPT, traditional NLP models relied on extensive annotated data for supervised training, often encountering issues related to data quality and generalization to new tasks. To address these problems, GPT-1 used unsupervised learning as the pre-training objective for supervised models, employing a left-to-right generative objective for pre-training, followed by supervised fine-tuning for downstream tasks. Building upon this architecture, GPT-2 increased the model's scale and improved its zero-shot learning capabilities. GPT-3 further expanded the model's parameters to 175 billion and introduced Few-Shot learning and In-Context Learning capabilities through simple task examples. Subsequently, GPT-3.5, trained with 200 billion parameters, introduced ChatGPT, which rapidly gained millions of users.

2.2. Prompt Engineering

As the design of pre-training and fine-tuning becomes increasingly complex, researchers seek lightweight, versatile, and efficient methods. Prompt engineering has emerged as a discipline that focuses on developing and optimizing prompts for Large Language Models (LLMs) to enhance their applicability in various scenarios and research fields.

Prompt engineering is an emerging field that, as people explore different prompting methods, has spawned practical techniques. In addition to the methods mentioned earlier, Zero-shot-CoT [6], with the simple phrase "Let's think step by step," has been widely applied in prompting. The emergence of CoT has also inspired the use of code as an intermediate reasoning step, leading to the development of Program-Aided Language Models (PAL Models) [7]. By introducing code into prompts, ChatGPT is guided to use code for computation and reasoning, making it easier to arrive at correct answers. The upcoming experiments are based on these excellent methods to investigate their performance in code generation tasks.

3. Study Design

The text of your paper should be formatted as follows:

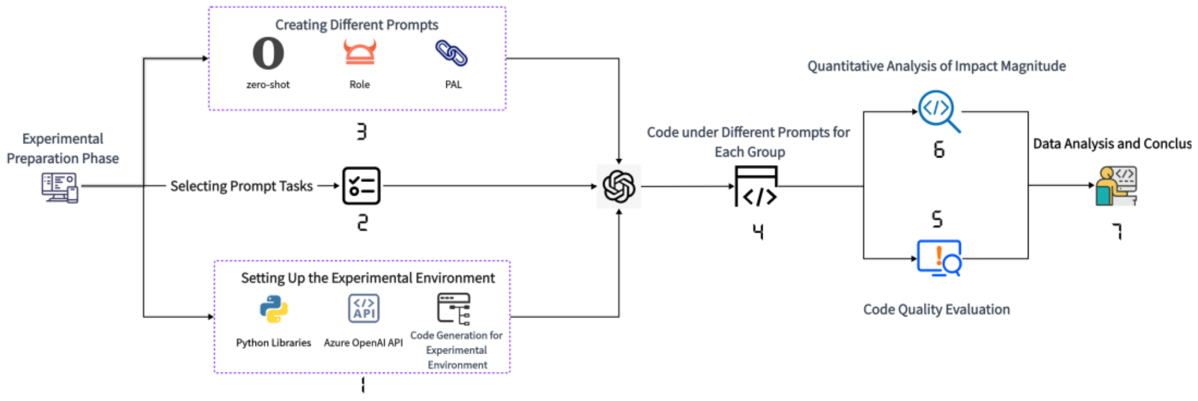

Creating Different Prompts
zero-shot
Role
PAL
3
Experimental Preparation Phase
Selecting Prompt Tasks
2
Setting Up the Experimental Environment
Python Libraries
Azure OpenAI API
Code Generation for Experimental Environment
1
Code under Different Prompts for Each Group
4
Quantitative Analysis of Impact Magnitude
6
5
Code Quality Evaluation
Data Analysis and Conclus
7

Figure 1. Flow chart of the experiment

Before conducting experiments, the experimental procedure was designed as illustrated in Figure 1. Initially, in the experiment preparation phase, the necessary Python libraries were installed, and the environment variables for the Azure OpenAI API were configured to set up the experimental environment. In Step 2, based on the research objectives, three prompt-sensitive code tasks were selected. In Step 3, mainstream prompt techniques were employed to construct different prompts for each task, ensuring the ability to guide ChatGPT in generating code that meets the requirements. Following that, in Step 4, the OpenAI API was invoked to utilize ChatGPT for code generation. For each constructed prompt, a batch of different code examples was generated. In Step 5, the generated code examples were executed to verify if they could run smoothly and meet the basic requirements of the tasks. In Step 6, based on the results of quality assessments, quantitative standards were formulated. In Step 7, statistical methods and charts were employed to analyze the experimental data.

4. Implementation and Comparison

In order to explore the impact of different prompt attributes on the quality of generated code in code generation tasks, this experiment involved a total of five different prompt methods applied to three prompt-sensitive code generation tasks, namely Snake Game, Sokoban Game, and Tetris Game. Figure 2 displays the outcomes of various prompt methods. Taking the prompt code for the Snake Game as an example, we will provide a detailed explanation of the prompt process for each group.

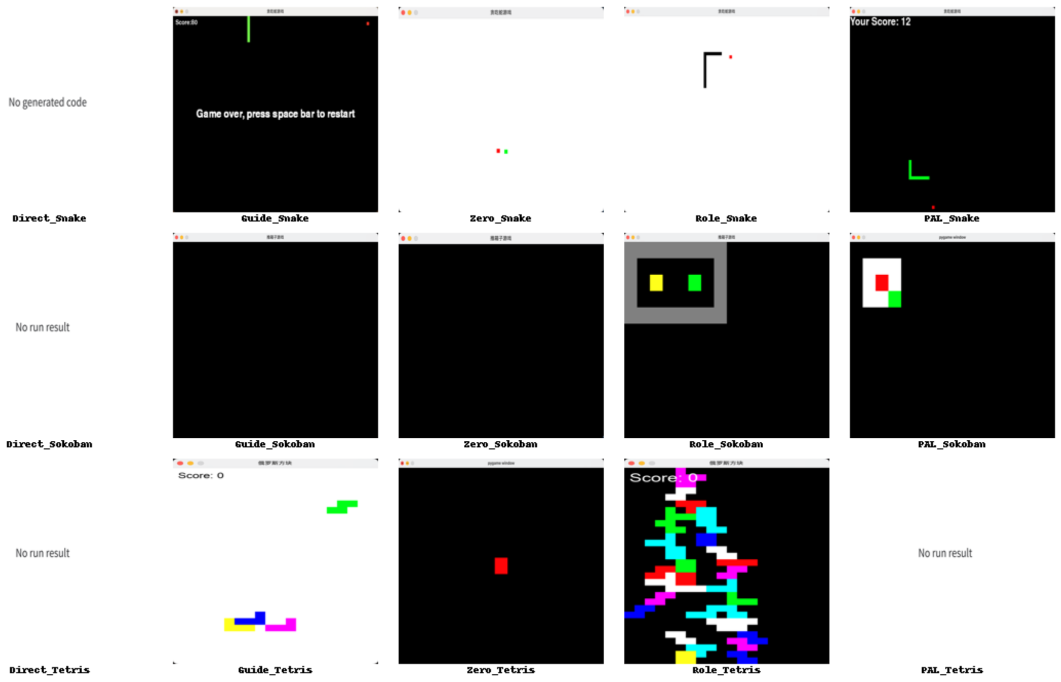


Figure 2. Results of running 5 cue groups in 3 code tasks, each column for each group of results, each row corresponding to each code task

4.1. Direct Prompt

In this initial prompt group, this group simply outlines the code task without providing any specific additional descriptions. Taking the Snake Game as an example, our prompt is as follows: "Using the Python language to help me fulfill the requirements of writing a Snake game, please provide a code example." In the Snake Game, the generated results only include the concept without specific code. In the Sokoban Game, the generated code is incomplete and still requires the developer to fill in the missing parts. In the Tetris Game, although code is provided, it contains errors and cannot run successfully. In our experimental results chart, a vertical comparison reveals that this is the only group with no successful outcomes, and the purpose of this group of experiments is to establish a baseline for evaluating the quality of generated code.

4.2. Guide words

Building upon the Direct Prompt group, this group added Python code prompts, specifically "import," at the end of the task prompts. The result was that the Snake Game could run but still exhibited issues. The problem lies in the fact that the Snake Game lacks movement speed, relying solely on keyboard inputs for movement, and the restart function using the spacebar after death does not function correctly. In the Sokoban Game, the game can be successfully launched, but it fails to continue running. Similarly, in the Tetris Game, there are missing functionalities, and the game stops after a few blocks have descended. In the results chart, it can be observed that compared to the Direct Prompt group, this group was able to generate complete program code and run the program successfully, with a noticeable improvement.

4.3. Zero-shot CoT

This experimental group employed the Zero-shot-CoT Prompt method, which involves adding "Let's think by step" at the end of the Direct Prompt group's prompts. In the generated results, the Snake Game could run, but its basic functionality was not fully completed, as it could not consume the food.

The Sokoban Game exhibited results consistent with the group that used the original prompts. The Tetris Game only completed a portion of the task. From both the complexity of the generated code and the final execution results, this group closely resembled the group that used the original prompts.

4.4. Role

This group employed the role-playing Prompt technique, where ChatGPT assumed the role of a Python game developer, guided by prompts such as: "I want you to play the role of a Python game developer. I will provide you with the game requirements, and you will develop the game." In terms of the generated results, the Snake Game developed by this group appeared to be the most refined. Unlike other groups with missing functionalities, the Snake Game exhibited no issues with movement speed and the restart function. The Sokoban Game was operable within a basic visual interface, offering playability. The Tetris Game, in particular, was the only group that could continuously stack blocks in the interface and complete a full game sequence. In terms of code complexity, this group was notably more complex compared to previous task groups. They defined more functions, especially in the Sokoban task, where the code volume experienced a significant increase. When compared vertically to other groups, this group demonstrated the highest code generation quality and the best execution results among all groups.

4.5. PAL group

In the Prompting, this group described the game's thought logic and added some simple code prompts within the steps, thus employing the PAL (Program-Aided Language Models) Prompting method. Initially, a description of the rules was incorporated into the Prompt: "You can move the snake using the arrow keys or the WASD keys. Your goal is to make the snake eat the food. Each time the snake eats the food, it grows longer and you earn points. If the snake collides with the game boundaries or itself, it will die, and the game will be over." Subsequently, some brief code snippets were added, as depicted in Figure 7.

Based on the results generated from these three task groups, the Snake Game had the best performance. The generated code exhibited flawless execution and closely resembled the results of the Role-Playing group, even completing the scoring functionality as an additional feature based on our prompts. However, the results for the other tasks were less satisfactory. While the Sokoban Game managed to display the game interface, subsequent operations were not feasible. The Tetris Game, on the other hand, experienced difficulties running due to certain code errors. On the whole, the performance of the PAL group exhibited instability.

~~~~
problem = f"""
Write a snake game
"""
prompt = f"""
Your task is to use the Python
language to help me complete
the requirements in the triple
quotes. Please provide a code
example.
Requirements:
```{problem}```
"""
response =
get_completion(prompt)
print(response)
~~~~

Figure 3. Prompts for Direct prompts group

~~~~
prompt = f"""
Your task is to use the Python
language to help me complete
the requirements in the triple
quotes. Please provide a code
example.
Requirements:
```{problem}```
import
"""

response =
get_completion(prompt)
print(response)
~~~~

Figure 4. Prompts for guide words group

~~~~
prompt = f"""
Your task is to use the Python
language to help me complete the
requirements in the triple quotes.
Please provide a code example.
Let's think step by step to ensure
we get the final result.
Requirements: ```{problem}```
"""

response =
get_completion(prompt)
print(response)
~~~~

Figure 5. Prompts for the zero-shot group

```
messages =   [
{'role':'system', 'content':'I want you to play the role of a Python game developer. \
I will provide you with game requirements, and you will develop the game.'},
{'role':'user', 'content': prompt}    ]

prompt = f"""
Your task is to use the Python language to help me complete the requirements in the triple quotes.
Please provide a code example.

Requirements: ```{problem}```
"""

response =
get_completion_from_messages(messages, temperature=0)
print(response)
```

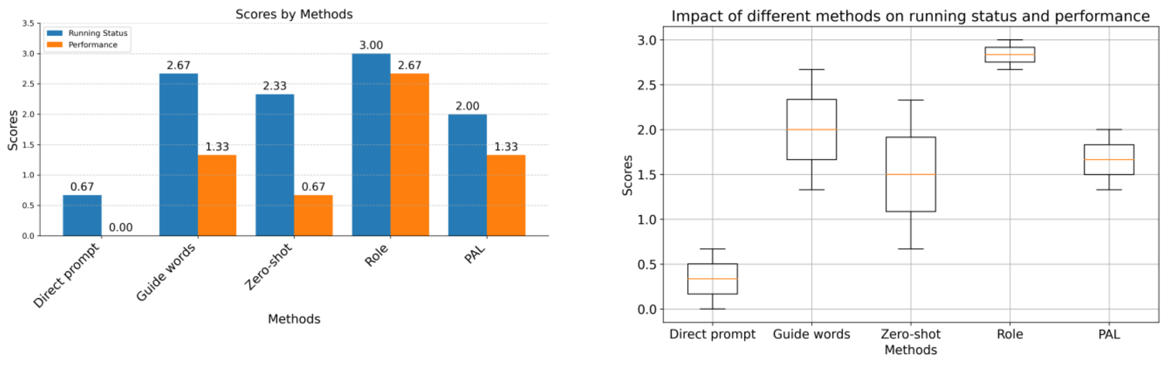
Figure 6. Prompts for role group

```
problem = f"""
Write a Snake game.
The rules of the Snake game are as follows: Move the snake up, down, left, or right using the arrow keys or WASD keys to make it eat food. Every time the snake eats food, its length increases and it earns points. If the snake hits the game border or itself, it dies and the game ends.

Game design ideas:
import pygame
import time
import random

pygame.init()
window_width = 800
window_height = 600
snake_block = 10
snake_speed = 15
def Score(score)
def our_snake
def message
"""
```

Figure 7. Prompts for PAL group

Based on the experimental results outlined above, this study conducted the following quantitative analyses of different prompting attributes: 1) Runtime Performance: Based on whether the generated code from each group could run successfully, we categorized it into four classes: runnable and playable, runnable but not playable, not runnable, and no code provided. We assigned scores of 3, 2, 1, and 0 to these categories in ascending order and then calculated the statistics. 2) Functional Performance: For the generated code that could run, we evaluated the completion of functional aspects of the code tasks, such as movement operations and restart functionality. We established different standards for the completeness of functionality, including incomplete, partially complete, mostly complete, and fully complete.

As shown in Figure 8, when comparing the data for the five groups, the Role group exhibited higher performance in both performance metrics compared to the other groups, with a significant advantage in functional performance. Apart from the Direct Prompt group, the differences in data among the other groups were relatively small.



(a)Score comparison for each cue group (b) Combined performance for each cue group

Figure 8. Comparison among cue groups

In order to illustrate the comprehensive performance of each group, a combination of runtime and functional performance data was plotted in Figure 8(b). The figure demonstrates that the Role group consistently exhibited significantly higher average performance than the other groups. The narrow spread of the Role group's data points indicates its relatively stable performance with minimal fluctuations.

Overall Assessment: Based on the aforementioned metrics, this study conducted a comprehensive evaluation of different Prompt methods, ranking their effectiveness in code generation tasks. Comparing both runtime and functional performance, the Role group achieved the highest performance, followed by the Guide Words group and the PAL group. The Zero-shot group performed the least effectively. The performance of these three groups was relatively similar.

Through the analysis presented above, we can draw conclusions about the sensitivity of different prompting attributes in code generation tasks and provide quantitative insights for the utilization of different methods.

5. Discussion

Footnotes should be avoided whenever possible. If required they should be used only for brief notes that do not fit conveniently into the text.

5.1. Results Analysis

Based on the experimental results, the following outcomes were summarized:

1) **Prompt Attributes Affect Generated Results:** During the experiments, it was evident that different prompting attributes significantly influenced the quality and functional performance of the generated results in code generation tasks. 2) **The superiority of the Role-playing Prompt Method:** The Role-playing Prompt method performed exceptionally well, generating high-quality code with strong functional performance. In code generation tasks, guiding the model from the perspective of a specific role, such as a developer, helped achieve code closer to the expected result.

5.2. Future Directions

Based on the experiments and results analysis presented above, we intend to make improvements and explore the following areas in this research:

1) **Improvements in Experimental Design:** While the Role-playing Prompt method performed well, there is room for improvement and exploration of additional Prompt methods and further optimization of Prompt architectures. In similar tasks, more refined Prompt designs, incorporating key code snippets and guiding words, could help generate more accurate and functional code. Additionally, due to the limited samples of code tasks and prompting attributes, coupled with significant variations in code, the

evaluation dimensions for generated results were limited, and there was a margin for error. To address this, more samples should be added to establish more detailed and accurate evaluation metrics and standards. 2) **Exploration of Multiple Prompt Attributes:** Given that this experiment considered only single attributes' impact, combining various prompting attributes is also an avenue for exploration. Combining different prompting attributes may have the potential to maximize code quality and functional achievement.

6. Conclusion

This study conducted empirical research to investigate the impact of different prompting attributes on code-generation tasks. By employing various Prompt methods in code generation tasks, we assessed the performance in terms of runtime and functional capabilities. The experimental results highlighted the significant influence of prompting attributes on the quality of generated code. Among the tested Prompt methods, the Role-playing method consistently outperformed others, consistently delivering the highest-quality generated code and excellent functional performance. Additionally, we noted the potential of the PAL method but acknowledged the need for careful Prompt construction to ensure logical coherence and suitable code prompts. Our research contributes to a better understanding and enhancement direction on the Prompt Engineering in code generation tasks.

References

[1] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., ... & Wen, J. R. (2023). A survey of large language models. arXiv preprint arXiv:2303.18223.
[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
[3] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training.
[4] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI blog, 1(8), 9.
[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. Advances in neural information processing systems, 33, 1877-1901.
[6] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. Advances in neural information processing systems, 35, 22199-22213.
[7] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., ... & Neubig, G. (2023, July). Pal: Program-aided language models. In International Conference on Machine Learning (pp. 10764-10799). PMLR.