

**APPLICATION AND OPTIMIZATION OF PROMPT ENGINEERING
TECHNIQUES FOR CODE GENERATION IN LARGE LANGUAGE
MODELS**

CHUNG-YU WANG

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

APRIL 2025

© Chung-Yu Wang, 2025

Abstract

Large Language Models have demonstrated remarkable capabilities across various domains, particularly in code generation and task-oriented reasoning. However, their accuracy and reliability in generating correct solutions remain a challenge due to the lack of task-specific prior knowledge and the limitations of existing prompt engineering techniques. Current state-of-the-art approaches, such as PAL, rely on manually crafted prompts and examples but often produce suboptimal results. Additionally, while numerous prompt engineering techniques have been developed to improve performance, selecting the most effective technique for a given task remains difficult since different queries exhibit varying levels of complexity.

This work presents an integrated approach to enhance the application and optimization of prompt engineering for code generation. First, it introduces TITAN, a novel framework that refines language model reasoning and task execution through step-back and chain of thought prompting. TITAN eliminates the need for extensive manual task-specific instructions by leveraging analytical and code-generation capabilities, achieving state of the art zero-shot performance in multiple tasks. Second, it proposes Pet-Select, a prompt engineering agnostic model that classifies queries based on code complexity and dynamically selects the most suitable prompt engineering technique using contrastive learning. This approach enables Pet-Select to optimize prompt selection, leading to improved accuracy and significant reductions in token usage.

Comprehensive evaluations across diverse benchmarks, including HumanEval, MBPP, and APPS, demonstrate the effectiveness of TITAN and PET-Select. TITAN achieves up to 7.6 percent improvement over existing zero-shot methods, while Pet-Select enhances pass@1 accuracy by up to 1.9 percent and reduces token consumption by 49.9 percent. This work represents a significant advancement in optimizing prompt engineering for code generation in large language models, offering a robust and automated solution for improving performance in complex and diverse programming tasks.

Acknowledgements

This thesis would not have been possible without the support and encouragement of many remarkable individuals, to whom I am deeply grateful. First and foremost, I would like to express my heartfelt appreciation to my parents, whose unwavering support has been a constant source of strength. No matter where I am, they encourage me to pursue my passions without hesitation. Their presence at every stage of my life means the world to me.

I am profoundly grateful to my supervisor, Professor Hung Viet Pham, for his invaluable guidance, encouragement, and support throughout these years. His mentorship has been instrumental in both my academic and personal growth, and I feel incredibly fortunate to have had the opportunity to work with him. I also extend my sincere thanks to my supervisory committee member, Dr. Hadi Hemmati, whose insightful feedback, guidance, and support have been essential to my research.

Throughout my master's journey, I had the privilege of collaborating with talented and dedicated researchers. I am especially grateful to my friends and colleagues, including Jiho Shin, Alireza Daghighfarsoodeh, Nima Shiri Harzevili, Hamed Taherkhani, YingHang Ma, WenHao Zhu, Xuchen Tan, and Liching Cheng. Their friendship, support, and kindness have made this experience truly enjoyable and rewarding.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	vi
List of Figures	vii
1 Introduction	1
1.1 Motivation	1
1.2 Research Hypothesis	3
1.3 Thesis Overview	4
1.4 Thesis Contribution	4
1.5 Thesis Organization	5
2 Application of Prompt Engineering for Code Generation	6
2.1 Introduction & Background	6
2.2 Approach	8
2.2.1 Challenges	10
2.2.2 Script generation with step-back prompting and zero-shot chain of thought prompting	11
2.3 Experimental Setup	13
2.3.1 Task-oriented datasets	13
2.3.2 Mathematical and symbolic reasoning datasets	15
2.3.3 Baselines	16
2.3.4 Experiment Details	16
2.4 Result and Discussion	17
2.4.1 RQ1: How does TITAN compare to the state-of-the-art zero-shot prompting approaches with code generation?	17
2.4.2 RQ2: How does TITAN compare to the state-of-the-art few-shot prompt- ing approaches with code generation?	18

2.4.3	RQ3: Can self-consistency help improve TITAN?	22
2.4.4	RQ4: How each component contribute to TITAN’s overall performance?	23
2.5	Related Work	25
2.5.1	Prompt engineering with code generation	25
2.5.2	Traditional prompt engineering without code generation	25
2.5.3	Code generation with LLMs	26
2.6	Conclusion	26
3	Optimization of Prompt Engineering for Code Generation	27
3.1	Introduction	27
3.2	Background	29
3.2.1	Prompt Engineering Challenges	29
3.2.2	Automated Prompt Engineering	29
3.3	Approach	30
3.3.1	Ranked PET Dataset Construction	30
3.3.2	Fine-tuning CodeBERT Embedding Model	32
3.3.3	Training Selection Model	35
3.4	Experimental Setup	35
3.4.1	Prompt Engineering Techniques (PETs) for code generation	35
3.4.2	Code complexity metrics	37
3.4.3	Experiment Settings	38
3.5	Result	39
3.5.1	RQ1. How do PET-Select compare to single PETs and baselines?	39
3.5.2	RQ2. How do the different components contribute to PET-Select’s performance	43
3.5.3	RQ3. How well PET-Select rank PETs for each query?	44
3.5.4	Discussion	45
3.6	Limitation & Threats to validity	47
3.6.1	Internal validity	47
3.6.2	External validity	47
3.6.3	Construct validity	48
3.6.4	Limitation	48
3.7	Related work	48
3.7.1	Code Complexity Prediction	48
3.7.2	Automated Prompt Engineering	49
3.8	Conclusion	49
4	Conclusions and Future work	50
4.1	Thesis Findings and Contributions	50
4.2	Future Work	51
	Bibliography	52

List of Tables

2.1	Datasets overview	13
2.2	The templates to create the four task-oriented datasets	14
2.3	Comparison of TITAN accuracy (%) to PAL Zero-shot, evaluated across eleven benchmarks using two GPT models. The best accuracy scores for each dataset and model are bold.	18
2.4	Comparison of TITAN accuracy (%) to other Few-shot code generation approaches, evaluated across eleven benchmarks using two GPT models. The * symbol indicates the accuracy is from original papers. The best accuracy scores for each dataset and model are bold.	19
2.5	TITAN self-consistency integration on GSM8K, Multiarith, and True/False datasets utilizing GPT-4 with majority voting from three samples.	22
2.6	TITAN Ablation Study	23
3.1	The prompting techniques used in the experiments. The ‘Iteration’ column specifies whether the technique requires multiple rounds of interaction with LLMs. The ‘Template’ column outlines the specific prompt template used in the experiments.	36
3.2	Pass@1 accuracy and token usage on HumanEval, MBPP, and APPS across different models. <i>Acc</i> refers to Pass@1 accuracy (%), and #Token is the average token usage.	40
3.3	Pass@1 accuracy and token usage on HumanEval+ and MBPP+ across different LLMs. <i>Acc</i> refers to Pass@1 accuracy (%), and #Token is the average token usage.	42
3.4	Ablation of PET-Select components on HumanEval, MBPP, and APPS across different models.	43
3.5	Ranking effectiveness (MRR and nDCG) of selection methods on HumanEval, MBPP, and APPS across different models.	44
3.6	Selection result of PET-Select for example instances. ✗ indicates the technique is selected correctly by PET-Select, while ✓ indicates it select incorrectly.	46

List of Figures

2.1	TITAN Overview	9
2.2	An example prompt where TITAN successes while PAL fails.	20
2.3	An example prompt where both PAL and TITAN fail.	21
2.4	An example where both reasoning phases together help TITAN generate the correct response.	24
3.1	PET-Select Overview.	31
3.2	An example that demonstrates contrastive learning on a single anchor query.	34

Chapter 1

Introduction

1.1 Motivation

Software engineering (SE) is a key field that focuses on designing, building, testing, and maintaining software systems in a clear and organized way [1]. As the demand for innovative and reliable software solutions continues to grow, SE has become central to many industries, including transportation, healthcare, education, entertainment, and finance [2]. SE aims to address the challenges of complexity, scalability, and quality in software development, ensuring that systems are robust, maintainable, and capable of meeting user needs. By incorporating principles of engineering, computer science, and project management, SE provides a structured framework to guide the development lifecycle, from initial requirements gathering to deployment and maintenance [3]. Over the years, SE has evolved as both a scientific discipline and a professional practice, gaining increasing attention from researchers and industry professionals [4].

Recently, the emergence of Large Language Models (LLMs), such as ChatGPT [5, 6], Gemini [7], and similar systems, has transformed the landscape of software engineering by enhancing how developers interact with tools and processes. LLMs are advanced AI systems trained on vast datasets comprising natural language, programming languages, technical documentation, and software development practices [8]. This extensive training enables these models to perform a variety of tasks that were previously time-consuming, error-prone, or required significant expertise. They can analyze natural language inputs to provide intelligent suggestions, automate repetitive tasks, streamline workflows, and even generate complex documentation with minimal human intervention [9].

In software engineering, LLMs have already demonstrated their value in several areas [10]. For instance, they support code completion, where they predict and suggest code snippets as developers type, reducing effort and boosting productivity [11]. They assist in debugging by identifying potential issues in code and suggesting fixes. Additionally, they facilitate API generation and usage by helping developers understand library functions, generating boilerplate code, and providing tailored examples [12]. Beyond coding, these models aid in

project management by automating the creation of user stories, test cases, and reports [13, 14].

This thesis is primarily focused on code generation, which refers to the ability of LLMs to produce functional code snippets, algorithms, or even entire applications based on natural language descriptions or high-level problem statements [15]. This capability has been made possible through advancements in deep learning techniques and the availability of large-scale training datasets, which include source code repositories, documentation, and programming best practices. These models are capable of understanding the syntax and semantics of various programming languages, such as Python, Java, C++, and JavaScript, as well as frameworks and libraries relevant to specific domains.

However, their performance on code generation remains a significant challenge. Despite their potential, current LLMs often struggle with several challenges as listed below:

- **Limitation to accurately perform procedural or numerical tasks:** LLMs rely on their training corpus to generate responses, which makes them struggle with queries requiring numerical calculations or procedural execution since the answers are not always present in their training data [16, 17]. Existing prompt engineering techniques have limitations such as Chain-of-Thought (CoT) prompting can help break down tasks into logical steps, but it still suffers from LLMs' inability to execute numerical operations accurately [18]. Program-Aided Language Models (PAL) attempt to generate executable code but rely heavily on manually crafted prompt templates and few-shot examples, which require significant human effort and expertise to construct [19, 20, 21]. Existing solutions perform better with few-shot learning, where hand-picked examples guide the model. However, selecting and ordering these examples correctly is non-trivial and can drastically affect performance [22, 23].
- **Gap of selecting the most suitable Prompt Engineering Technique (PET) for a given query in code generation tasks:** Selecting the optimal prompt engineering technique (PET) for code generation remains a significant challenge, as no single PET is universally effective for all queries. Different PETs, such as zero-shot, few-shot, chain-of-thought, and self-debugging, are designed to enhance LLM performance in various ways, yet each has its limitations. While some PETs excel in handling complex problems, they can be inefficient for simpler queries, whereas others struggle with more intricate tasks [24]. Additionally, interactive PETs like self-refinement and self-debugging, which require multiple iterations, can improve code generation but come at a high computational cost and do not always provide benefits for straightforward queries [25]. Applying such complex techniques unnecessarily increases token consumption and execution time, reducing efficiency. Furthermore, existing automated PET selection methods [26, 27] lack adaptability and fail to leverage multi-stage interactions effectively. Many rely on LLM feedback during execution to determine the best PET, making them expensive and impractical for real-world applications.

This thesis focuses on addressing these limitations through advancements in prompt engineering, which offers a scalable and adaptable way to improve LLMs without retraining

their underlying architectures [28]. This thesis explores two novel frameworks that leverage and enhance prompt engineering techniques to improve the performance of LLMs in code generation tasks. The first framework, TITAN, utilizes step-back prompting and chain-of-thought reasoning to generate executable scripts for solving task-oriented problems. Building on this, the second framework, PET-Select, introduces a dynamic method for selecting the most appropriate prompt engineering techniques based on query complexity, thereby optimizing accuracy and reducing token usage.

Together, these frameworks demonstrate the critical role prompt engineering can play in improving the capabilities of LLMs for code generation tasks. By developing new techniques (TITAN) and enhancing their application (PET-Select), this thesis aims to establish a comprehensive approach to advancing the use of LLMs in software engineering.

1.2 Research Hypothesis

In this thesis, the research hypothesis is formulated based on previous studies and findings as fundamental concepts. A series of experiments are conducted to validate the hypothesis and demonstrate its effectiveness. The research hypotheses for TITAN and PET-Select are listed below:

- The research hypothesis proposed in TITAN is based on the assumption that script generation through step-back and zero-shot Chain-of-Thought (CoT) prompting can enhance the reasoning capabilities of large language models (LLMs) without requiring hand-crafted data or few-shot prompting techniques. The research hypothesize that by integrating structured script generation, which includes explicit extraction of inputs and steps, TITAN can improve natural language reasoning tasks and outperform existing methods that rely on prompt engineering alone.
- The research hypothesis in the PET-Select posits that code complexity can serve as a reliable proxy for selecting the most appropriate prompt engineering technique (PET) for code generation tasks using large language models (LLMs). PET-Select hypothesize that problem complexity can be inferred from the complexity of the generated code and used to classify queries as simple or complex. By applying contrastive learning, PET-Select aims to improve the differentiation between these queries, enabling a more effective selection of PETs. Furthermore, the PET-Select proposes that a PET-agnostic selection model can outperform any single PET by dynamically choosing the best technique for each query. This hypothesis is tested by evaluating whether PET-Select improves pass@1 accuracy while reducing token usage across different benchmarks and LLMs.

1.3 Thesis Overview

This section offers an overview of the works presented in this thesis, which is organized into two main chapters. Each chapter addresses a distinct problem and is designed to be self-contained. Additionally, each chapter includes a summary of related works relevant to its specific objective.

- **Chapter 2: Application of Prompt Engineering for Code Generation:** In chapter 2, we propose TITAN, an application of prompt engineering for code generation. The chapter explores the use of step-back and zero-shot Chain-of-Thought (CoT) prompting to enhance the reasoning capabilities of large language models (LLMs) without relying on hand-crafted data or few-shot prompting techniques. The core hypothesis suggests that structured script generation, which explicitly extracts inputs and steps, can improve natural language reasoning tasks and outperform conventional prompt engineering methods. The study also examines the role of self-consistency techniques in enhancing the model’s reliability while acknowledging the associated computational costs .
- **Chapter 3: Optimization of Prompt Engineering for Code Generation:** In chapter 3, we propose PET-Select, an optimization approach of prompt engineering for code generation. The chapter introduces a PET-agnostic selection model that leverages code complexity as a proxy for classifying queries and selecting the most appropriate prompt engineering technique (PET) for code generation. The hypothesis asserts that problem complexity can be inferred from the generated code’s complexity and that contrastive learning can enhance differentiation between simple and complex queries, leading to more effective PET selection. PET-Select aims to outperform any single PET by dynamically adapting to query complexity, achieving improved pass@1 accuracy and reduced token usage across multiple benchmarks and LLMs.

1.4 Thesis Contribution

This thesis makes the following contributions:

- A novel framework designed to enhance LLMs’ ability to solve task-oriented problems in a zero-shot manner by extracting inputs and procedure steps through step-back and chain-of-thought prompting.
- Four newly introduced datasets, such as Finding, Counting, True/False, and Generative, were created to better evaluate LLMs’ capability in handling task-oriented challenges.
- A rigorous assessment of TITAN’s performance across eleven datasets using two widely used LLMs, GPT-3.5 Turbo and GPT-4.

- A new approach that dynamically selects the most effective prompt engineering techniques for code generation tasks.
- An extensive evaluation of PET-Select’s performance on five widely used benchmark datasets across three different LLMs.
- A combination of quantitative and qualitative analyses, along with an ablation study, to provide deeper insights into PET-Select’s decision-making process and the impact of its components.

1.5 Thesis Organization

We organized this thesis as follows. Chapter 2 elaborates on the application of prompt engineering for code generation. Chapter 3 describes the optimization approach of prompt engineering for code generation. Lastly, chapter 4 concludes this thesis and discusses future research directions.

Chapter 2

Application of Prompt Engineering for Code Generation

2.1 Introduction & Background

Large Language Models (LLMs) like GPT have significantly advanced the field of Natural Language Processing (NLP) through their proficiency in a wide range of tasks, including text generation [29, 30], translation [31, 32], summarization [33], and answering questions [34, 35]. These models are trained on vast datasets, enabling them to produce text that is both coherent and contextually appropriate [36]. However, when it comes to handling basic, task-oriented problems that involve numerical calculations or step executions, LLMs often fall short [16, 17].

This is an inherent weakness due to the way LLMs are designed and constructed. LLMs are trained on large text corpora and finely tuned for linguistic content creation and processing [37]. LLMs construct answers from text corpora and often face difficulties when the answers are not present directly in the training dataset but require precise numerical operations or step executions [18]. For instance, LLMs' limitation in counting tasks is evident in a simple test. If GPT-4 [5] using greedy decoding (0.0 temperature), is asked: "Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?", the answer would be: "Ed still had 22 more marbles than Doug". LLMs have a tendency to provide approximations or incorrect counts. This highlights the necessity for specialized prompt improvement to address these kinds of task-oriented challenges.

Recent research has explored agent-based frameworks and reasoning augmented models as promising alternatives for enhancing the performance of large language models on complex tasks. These approaches typically combine the reasoning capabilities of language models with external tools or structured planning components to enable multi step problem solving. For example, tool using agents such as ReAct [38] and Toolformer [39] integrate language generation with real time tool invocation, while reasoning based methods like Tree

of Thoughts [40] guide models through deliberate multi branch reasoning processes. Although these methods demonstrate strong performance on complex or multi turn problems, they often require additional infrastructure, external APIs, or orchestration layers, which introduces practical challenges for deployment.

In contrast, our proposed framework, TITAN, improves the reasoning ability of language models through interventions at the prompt level only, without the need for external tools or modifications to the system. This makes TITAN a lightweight yet effective solution, especially for task oriented problems where interpretability and ease of use are important. While agent based and reasoning augmented approaches may offer greater general purpose capabilities, TITAN is more cost effective and better suited for the specific demands of numerical and deterministic tasks.

Another approach is to utilize prompt engineering [41, 42, 43, 44, 45] to improve LLMs’ performance on specific tasks. This process involves crafting well-defined, strategically structured prompts to guide models towards specific outcomes. For instance, Chain-of-Thought (CoT) [46], one of the prompting techniques, breaks down problems into intermediate, textual steps to facilitate problem understanding and reveal steps toward potential solutions. CoT could be applied to the above problems to help LLM derive steps to complete the task. However, CoT still struggles with task-oriented problems as it still inherits the LLMs’ weakness of not having direct access to the numerical answers [18]. To address this, prior work proposes Program-Aided Language Models (PAL)[47]. PAL employs the generation of code as an intermediate step in the reasoning process to bridge the task execution gap.

Nevertheless, these prompt techniques perform optimally when used with few-shot prompting, in which hand-picked problem examples and answers are provided to help LLMs correctly comprehend and solve problems. However, crafting these examples for few-shot prompting is non-trivial and is an extra burden for the users without any related experience [19, 20, 21]. Using wrong examples [22] or wrongly ordering examples [23] for few-shot prompting will affect performance dramatically.

To address the drawbacks of crafting few-shot prompts for task-oriented problems, we introduce TITAN, a novel prompting framework to address the challenges of task-oriented problems while requiring no user effort. **Distinct from PAL, which directly generates scripts, TITAN incorporates two additional intermediate reasoning stages: input extraction and step extraction. These additional reasoning stages facilitate the generation of accurate scripts without the need for labeled examples.** Specifically, TITAN applies step-back prompting [41] to extract the inputs and their specifications for each task, making it the first framework to incorporate step-back prompting into code generation tasks. Input extraction helps LLMs overcome their tendency to perform poorly when given meaningless variables or when there is a misunderstanding of the problem’s inputs. In parallel, TITAN extracts procedure steps to complete the task by utilizing CoT prompting, which has been shown to aid LLMs in accurately comprehending problems by breaking them down into steps [46, 48]. Finally, TITAN combines the information from these two additional

reasoning stages to generate the most accurate scripts that are capable of producing precise answers.

To assess the effectiveness of TITAN, we evaluate it across seven distinct prior datasets [49, 47, 50, 51, 52, 53] containing mathematical and symbolic reasoning tasks and four additional task-oriented benchmarks we constructed in this work. We addressed the following research questions as our result:

- **RQ1:** How does TITAN compare to the state-of-the-art zero-shot prompting approaches with code generation?
- **RQ2:** How does TITAN compare to the state-of-the-art few-shot prompting approaches with code generation?
- **RQ3:** Can self-consistency help improve TITAN?
- **RQ4:** How does each component contribute to TITAN’s overall performance?

This chapter makes the following contributions:

- A new TITAN framework that can improve LLMs’ ability to solve task-oriented problems in a zero-shot manner by extracting inputs and procedure steps using step-back and chain-of-thought prompting respectively.
- Four task-oriented datasets consist of Finding, Counting, True/False, and Generative problems that can be used to better assess the capacity of LLMs in addressing task-oriented challenges.
- An extensive evaluation of TITAN on eleven datasets employing two different popular versions of LLMs (GPT-3.5 Turbo and GPT-4).

The rest of this chapter is organized as follows. Section 2.2 presents the methodology of this study. Section 2.3 provides the details of the experiment conducted for the study. Section 2.4 presents the results. Section 2.5 presents the related studies. Section 2.6 concludes this chapter.

2.2 Approach

In this work, we introduce TITAN, a novel approach to improve LLMs responses to task-oriented prompts. This is achieved by generating code through the use of a universal prompt template coupled with zero-shot learning techniques. TITAN significantly differs from prior prompt-specific strategies [47, 54, 55, 56, 57, 58, 59] since it provides a general and effective way to boost the performance of LLMs in performing various tasks without the need for individual prompt adjustments. TITAN enhances LLMs in two key ways: firstly, by utilizing the inherent abilities of LLMs to dissect and analyze complex queries; and secondly, by employing the code generation ability of LLMs to create executable scripts that produce the answers.

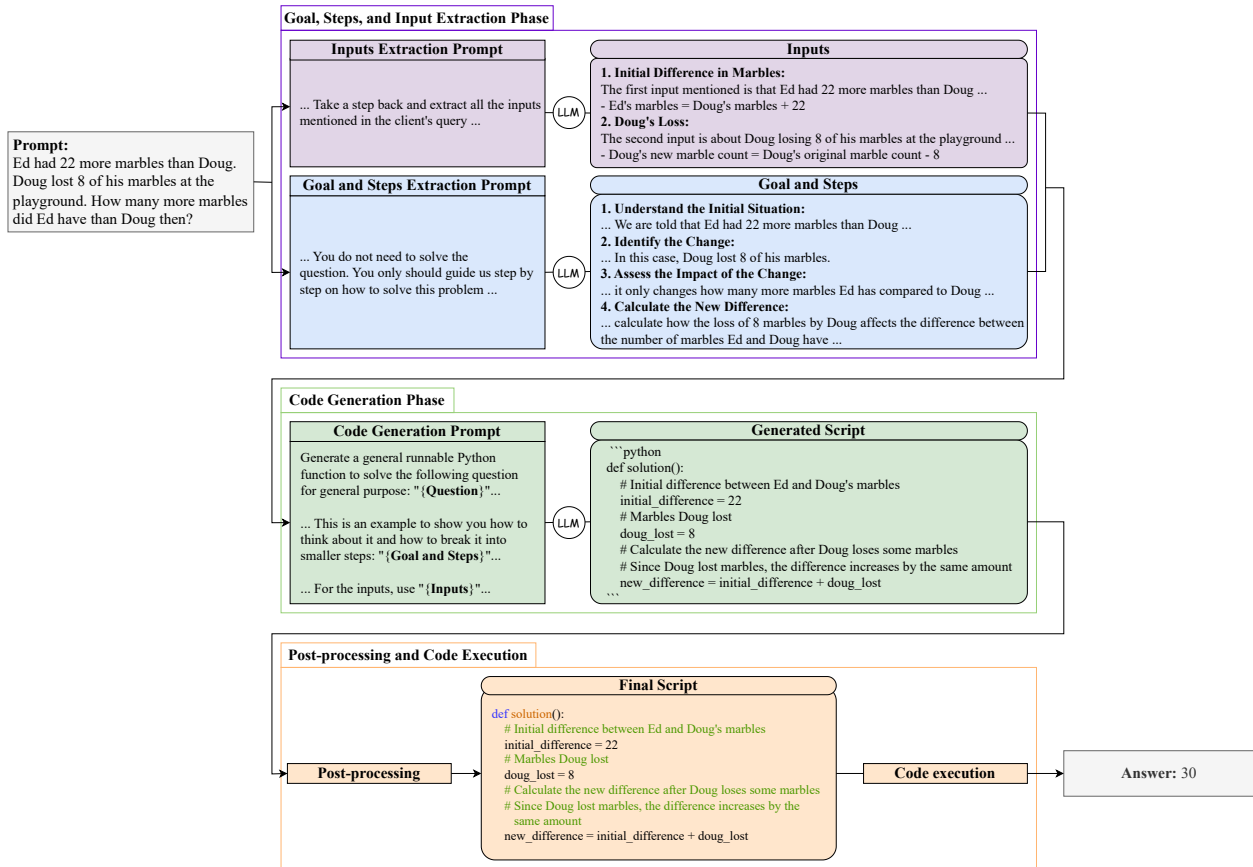


Figure 2.1: TITAN Overview

2.2.1 Challenges

As discussed in the Introduction, task-oriented prompts that involve numerical numbers (e.g., Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?) are not a direct fit for LLMs [18]. LLMs excel at analyzing and understanding complex queries by learning prior knowledge from a large corpus of text data [60, 61]. However, for LLMs to correctly respond to a query, the answer must be composed of such prior knowledge [62]. In the case of task-oriented prompts, the answer often is the result of an execution of some procedure described by the prompt. Hence in most cases, the result does not directly exist in the prior knowledge.

Conversely, LLMs have shown proficiency in generating code from natural language descriptions [63, 64, 65, 66, 67], as their training datasets contain numerous coding instances that the LLMs can utilize. By combining the analysis ability of LLMs with the task execution function of computer programs, we can create a system that is capable of precisely performing complex task-oriented prompts.

Prior approaches improve task-oriented prompts with code generation by relying heavily on task-specific prompts, requiring extensive manual effort and domain expertise. Such approaches [47, 54, 56, 58] employ uniquely crafted prompt templates tailored to individual tasks, along with few-shot learning techniques, to create scripts that can produce the desired response. Thus, for a specific task, the user would need to design a prompt template and provide some learning examples to get the most accurate responses from LLMs.

PAL is one of the approaches that is similar to our work. However, there are two main differences between TITAN and PAL. The first difference is that PAL generates code without extracting input information from the problem description. This often causes the language model to produce incorrect code due to the use of inappropriate or assumed inputs. This issue is also acknowledged in their original paper. Without proper input extraction, the model may fail to identify user-customized inputs and may misinterpret the user’s intent. In contrast, TITAN performs input extraction to provide the language model with clearer input information. This helps the model better understand the user’s intent and results in more accurate code generation. The second difference is that PAL relies on few-shot prompting, whereas TITAN uses zero-shot prompting. This reduces the effort required from the user by removing the need to supply additional examples.

Figure 2.1 shows an example of a step-by-step execution of TITAN. Specifically, given the input prompt “Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?”, which asks TITAN to perform a subtraction task of “Marbles Ed had” and “Marbles Doug had after he lost some of them at the playground”. TITAN performs step-back analysis to extract the specification of the inputs (i.e., the initial difference in marbles and the amount that Doug had after the accident). At the same time, TITAN performs chain-of-thought (CoT) analysis to extract the procedure (i.e., Understanding the situation and the changes after the accident) as well as the expected output (i.e., calculating the new difference). By combining these two pieces of information, TITAN was able to generate the solution script that correctly outputs the expected result.

2.2.2 Script generation with step-back prompting and zero-shot chain of thought prompting

To reduce the reliance on human effort in utilizing LLMs for task-oriented prompts, TITAN leverages two recently proposed prompt engineering techniques: step-back prompting [41] and zero-shot chain-of-thought prompting [48]. Step-back prompting helps TITAN analyze the query to identify the relevant inputs and their requirements. Additionally, zero-shot chain-of-thought prompting analyzes the query to extract the relevant steps and procedures to perform the task while requiring no additional input from the user. By integrating these two processes, TITAN could produce a script that faithfully represents the original task, which in turn yields a precise response.

Extracting inputs and specification phase: TITAN employs step-back prompting to identify the input requirements from the initial prompt. This process involves querying the LLM to identify and outline the specific inputs needed for each task. By taking a step back, TITAN enables the model to focus on the essential inputs required for successful code execution in the later stage.

Step-back prompting consists of two stages: abstraction and reasoning. Instead of asking the question directly, the abstraction stage asks the LLM a step-back question about a related higher-level concept or principle [41]. In the reasoning stage, the LLM is asked to reason about the high-level concept or principle facts found after the step-back question. In this work, TITAN utilizes the abstraction stage to extract inputs and their specifications from the original prompt. It then performs the reasoning stage when generating code based on the extracted inputs.

As demonstrated in Figure 2.1, TITAN performs the abstraction stage by asking a step-back question “... Take a step back and extract all the inputs mentioned in the client’s query ...” to extract the inputs from the original prompt. Specifically, by directing the LLM to focus on a higher concept (i.e., inputs and their specifications) without directly addressing the original prompt, TITAN can extract accurate inputs that help the LLM in the later code generation phase. In this example, TITAN extracts the two inputs (i.e., “22” and “8”) and their specifications (i.e., “initial difference in marbles” and “Doug’s loss”). These are then integrated into the generated code later as `initial_difference = 22` and `doug_lost = 8`.

Extracting the goal and procedure steps phase: To improve the precision of the code generation step, TITAN applies zero-shot chain-of-thought prompting to delve deeper into the goal of the task and the logical steps needed to achieve it while requiring no additional examples. TITAN prompts the LLM to articulate a step-by-step reasoning process, effectively mapping out the pathway from problem statement to solution. As shown in Figure 2.1, TITAN uses the prompt “... should guide us step by step on how to solve this problem ...” to extract the procedure steps needed to address the original prompt. By encouraging the LLM to express explicitly the thought process, TITAN can extract a detailed goal and the methodological procedure required to complete the task. This process clarifies the objective to ensure the generated code aligns closely with the intended outcome. By utilizing the

zero-shot variant of chain-of-thought, TITAN eliminates the need for additional examples that prior work such as PAL requires to perform optimally.

Figure 2.1 shows that by instructing the LLM to outline steps toward a goal without directly seeking the final answer, TITAN can clarify the main objective “Calculate the New Difference”. Additionally, chain-of-thought prompting helps TITAN uncover logical thinking to solve the problem “It only changes how many more marbles Ed has compared to Doug”. This phase guides the LLM to better understand the final goal and process during the code generation phase. This is demonstrated in the final generated code `new_difference = initial_difference + doug_lost` which matches the extracted steps.

Code generation phase: Combining the information extracted in the two previous phases, TITAN prompts the LLM to generate code based on clearly defined inputs, the articulated goal, and well-reasoned steps. This code generation phase combines the inputs of step-back prompting and the steps from chain-of-thought prompting to synthesize a coherent and functional code output. Specifically, TITAN employs the prompt “Generate a general Python function to solve the following question for general purpose: {question}” to ask LLMs to generate a Python function for the question. Followed up by the prompts “This is an example to show you how to think about it and how to break it into smaller steps: “{The Output from Goal and Steps Extraction}”” and “ For the inputs, use “{The Output from Inputs Extraction}”” to aggregate the outputs from previous phase into code generation phase. In this way, TITAN is able to guide the LLM to produce code that is logically aligned with the task’s objectives. For example, TITAN obtained the generated script (in Figure 2.1) given the original prompt “Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?” This example demonstrates the effectiveness of step-back and chain-of-thought prompting in helping LLM generate code with the correct inputs `initial_difference = 22` and `doug_lost = 8`, and precise representation of the task `new_difference = initial_difference + doug_lost`.

Post-processing and code execution: Since LLMs return free-form responses, TITAN employ a rigorous extraction and validation process. Specifically, TITAN utilizes regular expressions to extract the generated code from the responses. The regular expressions target consistent formatting markers (i.e., “Python”) for code generated by LLMs to ensure consistent extraction accuracy. Once the code is extracted, additional post-processing is required such as importing required packages and fixing indentation errors. The code is then executed automatically and the output is extracted as the final response to the user prompt. To get the result, we first set up rules to identify math questions from the code’s output. If the output matches these rules, we take and return this part. If not, we just clean up the output and use that as the final answer. This makes sure we always have a neat and relevant result. For example, The Final Script in Figure 2.1 is the final generated code that is executed and the output is extracted as 30 (i.e., the value of the variable `new_difference`).

Table 2.1: Datasets overview

Dataset	N	Input	Output
GSM8K [49]	1319	Question	Number
GSMHard [47]	1319	Question	Number
SVAMP [50]	1000	Question	Number
ASDIV [51]	2096	Question	Number
AddSub [52]	395	Question	Number
MultiArith [52]	600	Question	Number
Penguins [53]	149	Table + Text + Question	Number + Text
Finding	660	Question	Text
Counting	1100	Question	Number
True/False	500	Question	Binary
Generative	1100	Question	Text + List

2.3 Experimental Setup

2.3.1 Task-oriented datasets

Drawing from the prior study [18], when faced with a more challenging question on prime numbers that were sufficiently common to have representation on the Internet, the system performed adequately at that time. Hence, we create four task-oriented datasets from scratch to thoroughly evaluate TITAN performance. These include simple task-oriented prompts which we found to be particularly difficult for LLMs to directly address. This new dataset will benefit future research, given the current trend in prompt construction towards decomposition techniques, such as Least-to-Most Prompting [45] and Decomposed Prompting [68]. Our dataset includes decomposition tasks such as finding, counting, true/false questions, and generative tasks.

Table 2.1 shows the summary of these datasets. Each dataset includes multiple task templates that can be used to generate the prompts and the expected responses. Table 2.2 shows the complete template sets for the task-oriented datasets.

Finding Dataset: Given that LLMs have been shown to be strong in natural language tasks, it was a surprise to see LLMs (including GPT-4), perform poorly on simple Finding tasks where the prompt asks the models to identify specific patterns and letters within some given text. This Finding dataset includes 1100 queries that ask for a pattern or a word as a response, we demonstrate prompt templates for generating the Finding dataset in Table 2.2. These types of templates evaluate LLMs’ capacity to discern patterns amidst varying textual contexts.

Counting Dataset: Counting tasks is not a natural fit for LLMs. As demonstrated in the Introduction, GPT models can make mistakes when performing counting tasks. In this dataset, we include 1100 queries that require numerical responses. We form the dataset with various prompt templates stated in Table 2.2. This dataset is designed to test LLMs’

Table 2.2: The templates to create the four task-oriented datasets

Dataset	Prompt Template	Response
Finding	Choose the word from the three options provided that does not have “{word}“ within it. The single word given is: “[’{word}’, ’{word}’, ’{word}]”.	Word
	Taking into account that “{letter}” is identical to “{letter}“, seek out the word among these three that has the most unique letter count. The words are “[’{word}’, ’{word}’, ’{word}]”.	
	Assuming “{word}” has precisely one “{word}”, identify from the list below the word(s) that also contain exactly one “{word}”. The list includes: “[’{word}’, ’{word}’, ’{word}]”.	
	Among the three words listed, select the one that initiates with “{letter}”. The words for consideration are “{words}”.	
Counting	Excluding words that have fewer than four letters, how many words, spaced apart by ‘space’, exist in this sentence? The input is: {sentence}.	Number
	How many numeric characters are found in “{word}”?	
	What is the count of “{letter}” in “{word}” when ignoring uppercase letters?	
	What is the total number of distinct letters in “{word}”, disregarding case?	
True/False	How many vowels can be found in the “{word}”	0 / 1
	If there is a space in “{word}”, is there any space in “{word}”? If there is a space return “1”, otherwise return “0”.	
	Is there a capitalization difference between “{word}” and “{word}”? If there is a difference return “1”, otherwise return “0”.	
	Does this sentence has more than 3 spaces? “{sentence}” If there are more than 3 spaces return “1”, otherwise return “0”.	
Generative	Is there any repeated word in the following sentence? “{sentence}” If there are repeated words return “1”, otherwise return “0”.	Word
	If we assume the letter “{letter}” is equal to the letter “{letter}”, is there any spelling difference between “{word}” and “{word}”? If there is a difference return “1”, otherwise return “0”.	
	Take the first letter of each word within the specified sentence, join these letters to construct and return a new word. Words are spaced apart. The input is: “{sentence}”	
	Switch the initial two letters of the word provided and return the word thus generated. The input is: “{word}”	
Generative	Replace the final letter of the given word with an ‘s’ and return the newly formed word. The input is: “{word}”	List
	Capitalize the first character of the given word and return the word with the adjustment. The input is: “{word}”	
	Replace the first letters of the words with each other and return the adjusted versions as the response. The words are: “[’{word}’, ’{word}’, ’{word}’, ...]”	

ability to handle counting-related challenges such as enumerating numbers, unique letters, and words.

True/False Dataset: True/False dataset includes tasks requiring LLMs to discern the veracity of statements related to natural language processing. The dataset includes queries that describe the natural language processing tasks on sentences or words. The models are required to respond with a binary digit. The dataset consists of 500 samples with auto-generated answers from five prompt templates listed in Table 2.2. These templates encompass five distinct tasks, focusing on identifying aspects such as spacing, capitalization, repetition of words, and spelling differences within natural language sentences or words. Utilizing binary responses reduces the complexity of response composition and focuses the evaluation on the models’ ability to analyze and perform natural language tasks.

Generative Dataset: Performing generative tasks is generally considered a strong point for LLMs which are generative in nature. LLMs are known to generate surprisingly well-structured responses for various creative prompts. However, procedural generative tasks that require strict procedure steps to complete are different from the typical free-form creative tasks that LLMs are known for. For example, in a procedural generative task, the LLM could be tasked with creating a word from the first letter of each word in a sentence. We construct a generative dataset that aims to assess the capability of models to produce novel responses by following a given procedure and inputs. This dataset comprises 1100 queries, formatted as prompts which ask the models to generate a new word as a response in various ways. There are five distinct categories of prompts listed in Table 2.2 that challenge various aspects of textual manipulation, including word swapping, capitalization adjustments, and modifications to the letters at the end of words. This dataset seeks to thoroughly explore the LLM’s generative potential and its ability to understand and manipulate language constructs in novel ways.

2.3.2 Mathematical and symbolic reasoning datasets

Following prior work [47, 56], we evaluate TITAN on mathematical and symbolic reasoning datasets such as GSM8K, GSMHard [49], SVAMP [50], MAWPS [52], PENGUINS [53, 47], and ASDIv[51].

GSM8K and GSMHard are expansive collections of high-quality, linguistically varied grade school mathematics word problems, meticulously curated by adept problem composers. Each problem within the dataset necessitates a solution process that spans between two to eight steps, predominantly involving a series of fundamental arithmetic operations (i.e., addition, subtraction, multiplication, and division) to deduce the conclusive answer.

SVAMP (i.e., Simple Variations on Arithmetic Math Word Problems) dataset presents itself as a challenge set specifically designed for elementary-level Math Word Problems (MWP). An MWP is defined by a concise narrative in Natural Language, delineating a scenario or state of the world, which culminates in posing a query regarding one or more unknown quantities.

MAWPS is an online compendium dedicated to Math Word Problems (MWP) and serves as a comprehensive dataset for the evaluation of diverse algorithms.

PENGUINS delineates a task framework that integrates a tabular dataset of penguins, augmented with supplementary descriptors in natural language. The primary objective within this framework is to deduce answers to queries concerning the attributes of the penguins based on the provided dataset and descriptions.

ASDIv corpus (i.e., Academia Sinica Diverse MWP Dataset) is another MWP dataset which consists diverse language patterns and problem types, constituting an English MWP collection. This corpus is designed for the assessment of the proficiency of various MWP solvers.

2.3.3 Baselines

PAL [47] is the state-of-the-art approach that focuses on task-oriented prompts. PAL employs a code interpreter for problem reasoning with few-shot prompting while TITAN utilizes zero-shot learning. To gain a deeper understanding of how our approach compares to the state-of-the-art we also include PAL’s zero-shot prompting technique (PAL ZS) as our second baseline. PAL ZS utilizes the same prompt template as PAL but without the use of examples.

Recently, a few variants of PAL have been proposed such as Model Selection [56] and X-of-Thoughts [57]. Distinct from TITAN, which employs a two-step intermediate process for script creation, the Model Selection approach alternates between using CoT or PAL based on which yields more accurate outcomes as determined by language model evaluations. Meanwhile, X-of-Thought adopts the framework consisting of planning and verification phases, selecting the optimal method from among CoT, Program-of-Thought (PoT), and Equation-of-Thought (EoT) for problem-solving. Since these variants are also using code generation, we adopt their result as our baseline.

Our comparison does not include another variant, Code-based Self-Verification (CSV) [55], due to its dependence on the GPT-4 Code Interpreter and its evaluation solely on the MATH dataset. The dataset does not align with the types of problems TITAN aims to address.

Follow prior work, we also adopt the most recent GPT-4 model (i.e., gpt-4-0125-preview) and a less powerful GPT-3.5 model (i.e., gpt-3.5-turbo-0125) as our backend language model.

2.3.4 Experiment Details

We replicate PAL’s result by executing PAL code from their GitHub using the same version of GPT-3.5 and GPT-4 as stated in the original paper. To run PAL on our task-oriented datasets, we form the PAL few-shot prompt with four examples by randomly selecting one example from each task-oriented dataset, otherwise, we keep all PAL’s templates the same.

The zero-shot version of PAL uses the same templates but without any examples. Since PAL didn’t provide a zero-shot prompt, we developed the zero-shot version of PAL inspired

by another related study, PoT [54]. We adopt their prompt format by first posing the problem, followed by a request for the LLMs to complete a Python function named “solution()” without adding any examples. The function name and the return type are the same as the PAL few-shot version. For a fair comparison, we utilize the same metric used by PAL, which adopts exact match scores for evaluation. Unless specified differently, all experiments by default employ greedy decoding, adjusting the temperature of the language models to 0.0.

2.4 Result and Discussion

In this section, we compare TITAN against state-of-the-art code generation with zero-shot (RQ1) or few-shot (RQ2). In RQ3, we evaluate if self-consistency could help improve TITAN at a significant cost. Finally, we perform an ablation study (RQ4) to evaluate the contribution of each component in TITAN (i.e., input extraction and step extraction).

2.4.1 RQ1: How does TITAN compare to the state-of-the-art zero-shot prompting approaches with code generation?

Similar to prior work, TITAN employs script generation to solve mathematical and task-oriented problems. Hence, we first assess the effectiveness of TITAN when comparing state-of-the-art approaches with code generation in the zero-shot scenario. Existing code generation methods such as PAL, Model Selection (MS), and X-of-Thought (XoT), all utilize few-shot prompting. To evaluate how TITAN is compared to the prior work in the zero-shot scenario, we create a baseline PAL ZS (PAL zero-shot version) by incorporating a zero-shot prompts template [54] into PAL to be our zero-shot baseline.

Table 2.3 compares TITAN’s accuracy to other state-of-the-art zero-shot approaches (Row Approach) with code generation across 11 datasets (Columns *Acc*). The table is divided into two sections each corresponding to a version of GPT (e.g., GPT-3.5 and GPT-4). It includes the Δ column to show the relative performance differences between TITAN and PAL ZS. For instance, TITAN paired with GPT-4 outperforms the state-of-the-art zero-shot approach PAL ZS in all 11 datasets with a margin as big as 10.3% on the Penguin dataset. When a weaker LLM model such as GPT-3.5 is used, TITAN still outshines PAL ZS on 8 over 11 datasets with margins as big as 35.2% in the case of the Penguin dataset. The reason weaker LLM models fall short on some datasets is that the less advanced GPT model is unable to leverage the additional stages that TITAN offers. In scenarios involving simple datasets, inundating simpler models with excessive information can lead to incorrect responses. [24]

The average row specifies the average performance of each approach across 11 datasets. On average, TITAN achieved state-of-the-art zero-shot performance with significant margins of 7.6% and 3.9% when used on GPT-3.5 and GPT-4 respectively. Overall, the weaker LLM models such as GPT-3.5 benefit more from the additional stages that TITAN utilizes. In contrast, the stronger models such as GPT-4 have better reasoning capability build-in and

Table 2.3: Comparison of TITAN accuracy (%) to PAL Zero-shot, evaluated across eleven benchmarks using two GPT models. The best accuracy scores for each dataset and model are bold.

LLM	GPT-3.5			GPT-4		
	PAL ZS	TITAN		PAL ZS	TITAN	
Metric	<i>Acc</i>	<i>Acc</i>	Δ	<i>Acc</i>	<i>Acc</i>	Δ
GSM8K	76.6	84.2	$\uparrow 7.6$	93.6	95.3	$\uparrow 1.7$
GSMHard	61.8	69.6	$\uparrow 7.8$	74.1	78.2	$\uparrow 4.1$
ASDIV	85.3	91.4	$\uparrow 6.1$	92.7	97.2	$\uparrow 4.5$
SVAMP	82.8	84.3	$\uparrow 1.5$	94.0	94.8	$\uparrow 0.8$
AddSub	93.1	89.8	$\downarrow 3.3$	95.7	97.7	$\uparrow 2.0$
Multiarith	97.3	96.8	$\downarrow 0.5$	96.8	98.7	$\uparrow 1.9$
Penguin	59.1	94.3	$\uparrow 35.2$	87.2	97.5	$\uparrow 10.3$
Finding	93.8	98.4	$\uparrow 4.6$	95.5	99.8	$\uparrow 4.3$
Counting	89.1	87.8	$\downarrow 1.3$	87.5	89.8	$\uparrow 2.3$
True/False	59.2	76.7	$\uparrow 17.5$	84.4	93.8	$\uparrow 9.4$
Generative	84.9	94.1	$\uparrow 9.2$	98.7	99.9	$\uparrow 1.2$
Average	80.3	87.9	$\uparrow 7.6$	90.9	94.8	$\uparrow 3.9$

hence benefit less from step-back and chain-of-thought prompting. This result highlights TITAN’s ability to significantly boost the performance of weaker language models.

When looking at the GPT-4 result, TITAN improvement on PAL ZS is relatively small (around 2%) on simpler datasets with small numbers and easy questions such as AddSub, Multiarith, and Counting. On the other hand, on more challenging datasets involving large numbers, tables, and complicated questions such as GSMHard, True/False, and Penguin datasets, TITAN significantly outperforms PAL-ZS (by 4.1%, 9.4%, and 10.3% respectively). This suggests that the TITAN is particularly suitable for complex task-oriented prompts.

Finding 1: TITAN outperforms the state-of-the-art zero-shot approach by 7.6% and 3.9% when paired with GPT-3.5 and GPT-4. In some cases, the performance gain can be as big as 35.2% and 10.3% with GPT-3.5 and GPT-4 respectively.

2.4.2 RQ2: How does TITAN compare to the state-of-the-art few-shot prompting approaches with code generation?

In this RQ, we assess the broader effectiveness of TITAN when comparing state-of-the-art approaches with code generation that also utilizes few-shot prompting such as PAL few-shot

Table 2.4: Comparison of TITAN accuracy (%) to other Few-shot code generation approaches, evaluated across eleven benchmarks using two GPT models. The * symbol indicates the accuracy is from original papers. The best accuracy scores for each dataset and model are bold.

LLM	GPT-3.5								GPT-4					
	PAL		MS*		XoT*		TITAN		PAL		MS*		TITAN	
Approach	X		X		X		✓		X		X		✓	
Zero-shot	X		X		X		✓		X		X		✓	
Metric	<i>Acc</i>	<i>TΔ</i>	<i>Acc</i>	<i>TΔ</i>	<i>Acc</i>	<i>TΔ</i>	<i>Acc</i>	<i>AΔ</i>	<i>Acc</i>	<i>TΔ</i>	<i>Acc</i>	<i>TΔ</i>	<i>Acc</i>	<i>AΔ</i>
GSM8K	81.0	↓3.2	82.6	↓1.6	83.3	↓0.9	84.2	↑0.9	94.8	↓0.5	95.6	↑0.3	95.3	↓0.3
GSMHard	64.1	↓5.5	-	-	63.4	↓6.2	69.6	↑5.5	70.9	↓7.3	-	-	78.2	↑4.1
ASDIV	86.4	↓5.0	89.4	↓2.0	-	-	91.4	↑2.0	92.1	↓5.1	93.5	↓3.7	97.2	↑3.7
SVAMP	84.6	↑0.3	84.3	-0.0	83.6	↓0.7	84.3	↓0.3	94.6	↓0.2	93.7	↓1.1	94.8	↑0.2
AddSub	92.7	↑2.9	90.6	↑0.8	90.5	↑0.7	89.8	↓3.3	97.2	↓0.5	95.7	↓2.0	97.7	↑0.5
Multiarith	98.7	↑1.9	98.7	↑1.9	97.3	↑0.5	96.8	↓1.9	98.8	↑0.1	99.0	↑0.3	98.7	↓0.3
Penguin	96.6	↑2.3	-	-	-	-	94.3	↓2.3	96.6	↓0.9	-	-	97.5	↑0.9
Finding	97.7	↓0.7	-	-	-	-	98.4	↑0.7	99.8	-0.0	-	-	99.8	-0.0
Counting	84.6	↓3.2	-	-	-	-	87.8	↑3.2	88.5	↓1.3	-	-	89.8	↑1.3
True/False	76.0	↓0.7	-	-	-	-	76.7	↑0.7	95.6	↑1.8	-	-	93.8	↓1.8
Generative	87.7	↓6.4	-	-	-	-	94.1	↑6.4	99.4	↓0.5	-	-	99.9	↑0.5
Average	86.4	↓1.5	-	-	-	-	87.9	↑1.5	93.5	↓1.3	-	-	94.8	↑1.3

version, Model Selection (MS), and X-of-Thought (XoT). The details of each baseline are discussed in Sec 2.3.3. We replicate PAL on the original and our task-oriented datasets, we employ the code provided in the original study.

Table 2.4 compares TITAN’s accuracy to other state-of-the-art Few-shot approaches with code generation. The asterisk “*” symbol indicates approaches that are not possible to replicate (i.e., due to missing executable source code) and for which the accuracy values are taken from the original papers. This is why the Table does not show results for MS and XoT on our proposed task-oriented datasets which is indicated by the dash (“-”).

In Table 2.4, for each combination of LLM and dataset, the best accuracy value is highlighted in bold, indicating state-of-the-art performance. The $A\Delta$ columns illustrate the improvement that TITAN makes when compared to all baselines. For example, TITAN achieves state-of-the-art performance on 8 over 11 datasets when paired with GPT-4 LLM while only marginally losing to few-shot approaches (which needed additional non-trivial examples) on three datasets GSM8K, Multiarith, and True/False (by 0.3%, 0.3% and 1.8% respectively as indicated by the $A\Delta$ column).

Overall, when paired with the most advanced LLM (i.e., GPT-4), TITAN provides the most enhancement among all approaches across most datasets. This can be seen by looking at the $T\Delta$ columns which indicate the difference between each baseline and TITAN. Only

<p>Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?</p>	Prompt	Answer
		30
<pre>more_marbles_ed = 22 marbles_lost_doug = 8 # Since Ed's marbles are not affected by Doug losing his, the # difference remains the same. result = more_marbles_ed</pre>	PAL	Incorrect
		<pre></> → 22</pre>
<pre>initial_difference = 22 doug_lost = 8 # Calculate the new difference after Doug loses some marbles # Since Doug lost marbles, the difference increases by the same amount new_difference = initial_difference + doug_lost</pre>	TITAN	Correct
		<pre></> → 30</pre>

Figure 2.2: An example prompt where TITAN succeeds while PAL fails.

in four cases where PAL and MS (which requires non-trivial examples) outperform TITAN with marginal gaps (i.e., 0.1%, 0.3%, 0.3%, and 1.8%).

On average, TITAN with GPT-4 achieves an average performance boost of 1.3% compared to PAL (which unlike TITAN requires few-shot examples). When evaluated on GPT-3.5, TITAN performs better on average than alternative few-shot approaches, achieving an average accuracy enhancement of 1.5% over PAL. These results indicate that TITAN achieves state-of-the-art performance in most cases even against few-shot approaches that require non-trivial examples.

When moving from GPT-3.5 to GPT-4, the average performance gain TITAN provides over PAL is only 0.2% (between 1.5% and 1.3%) which is significantly smaller than the 3.7% (between 7.6% and 3.9%) gain TITAN provides over PAL ZS. This might be because the more advanced language models (i.g., GPT-4) benefit more from few-shot prompting and hence are harder to enhance with step-back and chain-of-thought prompting that TITAN employs. Nonetheless, TITAN enhances the performance of these models across the board.

Finding 2: Overall, TITAN achieves state-of-the-art performance in most (8 out of 11) cases when paired with GPT-4 while only marginally losing to few-shot approaches (which needed additional non-trivial examples) on three datasets by small margins. Overall, TITAN outperforms PAL by 1.3%.

To better demonstrate how TITAN was able to effectively generate correct responses and illustrate TITAN’s limitations we include two examples where: **1:** TITAN is correct while PAL fails and **2:** both PAL and TITAN fail.

Case 1: Figure 2.2 shows an example from the ASDIV dataset where TITAN can combine correct reasoning from the input and step extraction phase to generate the correct code.

<p>Alex is getting ready to attend an event that she has hosted and wants to make sure that she has enough seats for everyone. She invites 100 people via email and each invited person says that they will also invite 2 of their friends. She then calls 10 of her friends to invite them too and 8 of them say they will be bringing their spouses. How many seats will Alex need?</p>	Prompt	Answer 319
<pre> people_invited_email = 100 friends_invited_by_email = people_invited_email * 2 friends_invited_by_call = 10 spouses_coming = 8 total_seats_needed = people_invited_email + friends_invited_by_email + friends_invited_by_call + spouses_coming result = total_seats_needed </pre>	PAL	Incorrect → 318
<pre> email_invited = 100 friends_per_person = 2 total_email_invited = email_invited * (1 + friends_per_person) phone_invited = 10 spouses = 8 total_friends_attending = phone_invited + spouses total_seats_needed = total_email_invited + total_friends_attending </pre>	TITAN	Incorrect → 318

Figure 2.3: An example prompt where both PAL and TITAN fail.

Specifically, TITAN is able to identify the inputs `initial_difference` and `doug_lost` as well as the importance of reasoning: `#Since Doug lost marbles, the difference increases by the same amount which is realized by the important extracted step: new_difference = initial_difference + doug_lost`. This is further illustrated by Figure 2.1 where the extraction of goals and steps yields the correct logical steps: “Calculate the New Difference” On the other hand, without either of these reasoning phases, PAL fails to generate the correct responses. Specifically, PAL identifies the input name wrong `more_marbles_ed` which hinders the integration reasoning into the solution. This example showcases the distinct differences between PAL and TITAN and highlights TITAN’s profound grasp of the problem context due to the reasoning phases.

Case 2: Figure 2.3 shows an example where both PAL and TITAN fail to generate the correct answer. Both methods correctly reason about the composition of the answer by counting the email invitations, the email invitations’ friends, the phone invitations, and their spouses. However, both fail to account for one additional seat for Alex herself. This example might expose TITAN’s inherent weakness from the underlying GPT model (i.e., GPT-4) where the GPT model does not link the number of seats needed to the number of attendants and instead links to the number of invitees mentioned in the prompt. We suspect that this can be due to GPT’s autoregressive mechanism (i.e., tokens are generated

Table 2.5: TITAN self-consistency integration on GSM8K, Multiarith, and True/False datasets utilizing GPT-4 with majority voting from three samples.

	TITAN	TITAN + SC@3
GSM8K	95.3	95.6 \uparrow 0.3
Multiarith	98.7	99.5 \uparrow 0.8
True/False	93.8	95.4 \uparrow 1.6

sequentially) that makes each token’s production reliant heavily on the outcomes generated before it [69]. This highlights unique challenges and flaws in the way GPT-4 learns and encodes these processes [18].

2.4.3 RQ3: Can self-consistency help improve TITAN?

Self-consistency [70] is an agnostic strategy that can be applied to CoT prompting approaches to improve the underlying approach performance. One downside of self-consistency is that it requires many duplicated queries to be sent to the LLM which can have diminishing returns [25]. Since TITAN incorporates the CoT we hypothesize that incorporating self-consistency could further improve TITAN’s performance. However, due to the elevated cost, we do not incorporate self-consistency in TITAN by default and instead use the temperature of 0.

In this RQ3, we want to test if self-consistency can help improve TITAN further and if the benefit warrants the cost. Specifically, we apply self-consistency on TITAN for three datasets (GSM8K, Multiarith, and True/False) to see if self-consistency can help TITAN achieve state-of-the-art performance. As indicated by the original paper [70], we set the LLM’s temperature to a nonzero value to retrieve distinct response samples. In this case, we set the GPT-4 temperature to 0.7 before running TITAN three times. The final response is subsequently chosen by majority voting.

In Table 2.5, TITAN + SC3 (self-consistency with three samples) indicates the accuracy (%) of TITAN when self-consistency is integrated with three responses. When comparing the vanilla TITAN, self-consistency helps enhance TITAN by 0.3% on GSM8K, 0.8% on Multiarith, and 1.6% on True/False at the cost of triple the number of queries. If the cost of self-consistency is not a factor, TITAN + SC3 would achieve state-of-the-art performance on GSM8K and Multiarith while getting within 0.2% of the best approach on True/False. Overall, the result suggests the adoption of self-consistency not only bolsters TITAN’s robustness but also achieves superior performance relative to greedy decoding (i.e., setting the LLM temperature to 0.0 which is TITAN’s default setting) at a significant cost.

Finding 3: Overall, Self-consistency can improve TITAN performance with significant cost. Specifically, with triple the cost, TITAN + SC@3 is shown to improve TITAN by achieving state-of-the-art performance on GSM8K and Multiarith.

Table 2.6: TITAN Ablation Study

Dataset	W/o Input Extraction	W/o Step Extraction	TITAN
GSM8K	95.6 \uparrow 0.3	93.4 \downarrow 1.9	95.3
GSMHard	77.5 \downarrow 0.7	76.1 \downarrow 2.1	78.2
ASDIV	95.8 \downarrow 1.4	95.3 \downarrow 1.9	97.2
SVAMP	94.5 \downarrow 0.3	92.9 \downarrow 1.9	94.8
AddSub	97.7 -0.0	97.7 -0.0	97.7
Multiarith	98.5 \downarrow 0.2	98.3 \downarrow 0.5	98.7
Penguins	95.3 \downarrow 2.2	94.6 \downarrow 2.9	97.5
Finding	98.5 \downarrow 1.5	98.3 \downarrow 1.3	99.8
Counting	93.6 \uparrow 3.8	88.6 \downarrow 1.2	89.8
True/False	94.5 \uparrow 0.7	95.1 \uparrow 1.3	93.8
Generative	99.0 \downarrow 0.9	99.1 \downarrow 0.8	99.9
Average	94.5 \downarrow 0.3	93.5 \downarrow 1.4	94.8

2.4.4 RQ4: How each component contribute to TITAN’s overall performance?

Input and step extraction are two important phases that help TITAN generate precise scripts that can be used to generate correct responses. In this RQ, we perform an ablation study with GPT-4 to see how each component contributes to TITAN’s overall performance.

Table 2.6 shows how turning off one of the main reasoning components (Column W/o Input Extraction and W/o Step Extraction) can reduce the effectiveness of TITAN. The \downarrow and \uparrow indicate that removing the component reduces or increases the effectiveness of TITAN respectively. For example, removing the step extraction phase reduces TITAN accuracy on the Penguins dataset by 2.9%. On all datasets, removing the step extraction phase reduces TITAN’s performance significantly with an average reduction in performance of 1.5%. The result suggests that step extraction significantly contributes to TITAN’s performance.

On the other hand, removing the input extraction phase hinders TITAN’s performance in a few cases. For example, on the Counting dataset, TITAN gains 3.8% without the input extraction phase. This could be because, for the counting tasks, there are fewer inputs and including additional information about the input complicates the script generation prompt and affects the final script. This result suggests that input extraction might have less contribution to the overall success of TITAN in correctly crafting the answers. However, in a broader term, input extraction is still a necessary component of TITAN as removing it reduces TITAN performance by an average of 0.3%.

On AddSub, removing either component does not trigger significant accuracy changes (i.e., changes larger than 0.1%). This suggests that each reasoning phase already provides enough information to enhance the code generation. This is evident in Table 2.4, where

<p>Tim gets a promotion that offers him a 5% raise on his \$20000 a month salary. It also gives him a bonus worth half a month's salary. How much money will he make in a year?</p>	Prompt	Answer 262500
<pre>original_salary = 20000 raise_amount = 0.05 new_monthly_salary = original_salary * (1 + raise_amount) bonus = 0.5 * original_salary total_annual_salary_without_bonus = new_monthly_salary * 12 total_annual_salary = total_annual_salary_without_bonus + bonus</pre>	W/o Input Extraction	Incorrect 262000
<pre>monthly_salary = 20000 annual_salary = monthly_salary * 12 raise_amount = monthly_salary * 0.05 bonus = monthly_salary / 2 total_annual_income = annual_salary + raise_amount + bonus</pre>	W/o Step Extraction	Incorrect 251000
<pre>original_monthly_salary = 20000 raise_percentage = 0.05 months_in_year = 12 raise_amount = original_monthly_salary * raise_percentage new_monthly_salary = original_monthly_salary + raise_amount bonus = new_monthly_salary / 2 total_annual_salary_without_bonus = new_monthly_salary * months_in_year total_annual_salary_with_bonus = total_annual_salary_without_bonus + bonus</pre>	TITAN	Correct 262500

Figure 2.4: An example where both reasoning phases together help TITAN generate the correct response.

TITAN outperforms PAL (without such reasonings) by 0.5% on the AddSub dataset. Overall, both components are essential to the overall performance of TITAN as removing either reduces the overall performance of TITAN.

This is further illustrated by the example in Figure 2.4, where without either input and step extraction, TITAN failed to utilize the correct input or follow the precise procedure. Specifically, without input extraction, the framework fails to distinguish between the `original_salary` and the `new_monthly_salary`, leading to incorrect bonus calculations. On the other hand, without step extraction, the model fails to recognize the correct steps to compute the `raise_amount` (i.e., should be across 12 months) and `bonus` (should be on raised salary). When utilizing both reasoning phases, TITAN was able to correctly extract precise inputs and procedural steps to complete the task in a clear and precise manner.

Finding 4: Step extraction phase plays a significant role in enhancing TITAN’s performance while in some cases input extraction can hinder the performance of the framework. However, both components are essential to the overall performance of TITAN as removing either reduces the overall performance of TITAN.

2.5 Related Work

2.5.1 Prompt engineering with code generation

There has been prior work that utilizes code generation to address the gap in LLMs’ execution ability when it comes to task-oriented prompts. PAL[47] introduces an innovative method for tackling mathematical problems through script generation combined with few-shot prompting. Further improvement is made by a Model Selection technique (MS) [56] which employs both PAL and CoT (Chain-of-Thought) in tandem by selecting the best response between them.

X-of-Thoughts (XoT) [57] represents another code generation strategy, focusing on resolving mathematical and algebraic equations by dynamically switching among different prompting methods. Another distinct method, CSV [55], approaches the resolution of mathematical challenges through coding, which heavily relies on the GPT-4 Code Interpreter. Its efficacy is evaluated exclusively on the MATH dataset [71], indicating a specialized focus on coding solutions for mathematical issues.

In contrast, TITAN diverges significantly from these methods by adopting a zero-shot learning technique, which enhances its capacity for generalization across diverse problem sets. TITAN aims to solve reasoning questions without relying on any hand-crafted data, few-shot learning techniques, or human annotators.

2.5.2 Traditional prompt engineering without code generation

The exploration of concepts such as Chain-of-Thought [46] and Step-Back [41] has revealed the capacity of LLMs for zero-shot learning [48] primarily through their ability to process

information in a step-by-step manner. Prior work such as PHP [72], Self-Contrast [58], and Boosting-of-Thought (BoT) [59] has been developed to uncover the reasoning paths through post-hoc strategies of prompt engineering, aiming to enhance the models’ problem-solving capabilities by mimicking human-like [31] reasoning processes. These techniques use self-verification methods. For example, BoT iteratively generates, assesses, and refines thoughts using the model’s self-evaluation. Self-Contrast exploits diverse solutions to enrich reasoning, while PHP refines reasoning paths with LLM outputs toward the correct answer.

Because the mechanisms and effectiveness of self-correction in LLMs are not understood comprehensively [25], we do not include such an iterative mechanism in TITAN. Recent research indicates that LLMs are not yet capable of self-correction [25]. Distinct from such interactive methods which require human annotations or many more query rounds, TITAN does not require manual effort to incorporate evaluation information into the reasoning process.

2.5.3 Code generation with LLMs

Recently, research in using LLMs for code generation [73, 74, 75, 76] has made significant advancements. These advancements often come from training these models with more specific examples of code, which makes them better at specific coding tasks [74]. Another line of research is guided code generation [77, 78] where LLMs are utilized to efficiently create code. Specifically, Willard et al. [79] introduce a system that guides LLMs to produce text using rules and structures from programming languages, reducing unnecessary steps in creating code sequences. Another recent work, SynCode [80], is a framework that improves how LLMs understand and generate code by focusing on the rules of programming languages. This approach helps create more accurate code by filtering out mistakes and focusing on the correct coding syntax. TITAN differs from these code generation research in the general nature of TITAN’s input (i.e., prompt) where TITAN is designed to improve the overall question/answer capability of LLMs by utilizing script generation and not to solve general software engineering problems related to code generation.

2.6 Conclusion

In this work, we introduced TITAN, a novel approach for natural language reasoning that leverages script generation through the extraction of inputs and steps by utilizing Step-Back and Chain-of-Thought prompting respectively. We evaluate TITAN on 11 datasets with a comprehensive comparison with prior state-of-the-art. Unlike preceding approaches that predominantly rely on few-shot prompting techniques, TITAN employs a zero-shot prompting strategy, thereby eliminating the requirement for hand-crafted data. Our findings demonstrate that TITAN exhibits superior performance in a diverse set of tasks. Furthermore, the integration of TITAN with self-consistency further enhances its efficacy (with some additional cost), thereby underscoring the potential of TITAN as a robust solution for advanced natural language reasoning challenges.

Chapter 3

Optimization of Prompt Engineering for Code Generation

3.1 Introduction

Recently, Large Language Models (LLMs) have shown their promising performance in various software engineering tasks, such as unit test case generation [81, 13, 82], automated bug repair [83, 84], and API specification [85]. Especially for code generation from natural language descriptions, LLMs demonstrate their impressive capability where code is generated with natural language descriptions [86, 64, 87, 88].

Given that most of the state-of-the-art LLMs are all closed-source, the most popular way to enhance the LLM's ability to generate accurate and reliable code is to utilize various prompt engineering techniques (PETs) [89, 90]. For example, Some techniques ask LLMs to provide reasoning steps for solving problems [91, 43, 92], while others utilize LLMs to refine their output by prompting them to review and improve the code they generate [44, 93]. In addition to these strategic PETs, some frameworks have been proposed that leverage LLMs [26] or retrieve relevant instances from databases to automatically generate optimal prompts for questions [94], a process known as automated prompt engineering.

Despite numerous studies focused on crafting the optimal prompt, not a single technique is optimally applicable to every query and the task of selecting a correct PET is not trivial. This is because of two key reasons: (1) interactive prompting techniques might be too costly and do not always provide the promised benefit especially when applied to simpler queries [25, 24, 95, 96], and (2) existing automatic prompt engineering does not utilize the multiple rounds of responses which is associated with the success of iterative PETs [72, 44]. Not to mention, all of the auto prompt engineering techniques are not easily extended.

Prior work [56] has proposed frameworks to select the most appropriate PET for a given query based on feedback from LLMs. However, these approaches focus on reasoning tasks and require implementation alongside language model execution, where the best answer is selected based on the outputs of various techniques. This makes it less practical and quite costly.

One possible solution is to assess the complexity of the problem presented in the prompt and use it to select the appropriate PET, eliminating the need for multiple queries to language models. This solution is based on prior studies [96] that indicate that the single-round PETs (e.g., Zero-shot, Zero-shot CoT, Few-shot, Few-shot CoT, Persona) are more suited to simpler problems whereas some multi-round PETs (e.g., Self-planning, Self-refine, Progressive Hint, Self-debug) are designed to tackle more complex problems with multiple rounds of reasoning. However, directly extracting the problem complexity from the description (i.e., the prompt) is not trivial. The code complexity of the solution could be a good proxy to estimate the problem’s complexity. Using this complexity proxy, we could apply contrastive learning to derive the embedding for phrases in the problem description which can better associate simple problems with direct PETs and complex problems with sophisticated PETs.

Equipped with such insights, we propose a general low-cost solution, PET-Select, a PET agnostic selection model that is not dependent on the pool of available PETs and can be easily adaptable and extendable to the ever-growing list of available advanced PETs. PET-Select integrates query complexity by using generated code complexity as a proxy using contrastive learning [97]. Specifically, by incorporating generated code complexity, PET-Select can differentiate each query between simple and complex problems (i.e., requiring simple or complex code) which can help PET-Select select the appropriate PET that targets the relevant level of problem. Furthermore, we incorporate a wide range of PETs representing various categories [98] including those PETs that have multi-round interactions with language models.

We evaluate PET-Select on five popular code generation benchmark datasets, HumanEval, HumanEval+, MBPP, MBPP+, and APPS. To ensure a fair evaluation, we apply 5-fold cross-validation with 80% training and 20% testing sets. Our evaluation on GPT-3.5 Turbo, GPT-4o, and DeepSeek-V3 shows that PET-Select achieves an improvement of up to 1.9% in terms of pass@1 accuracy when compared with an individual PET while using as little as 49.9% fewer tokens on the HumanEval with GPT-3.5 Turbo. Our quantitative and qualitative results also demonstrate that PET-Select effectively selects appropriate techniques for each code generation query. This paper makes the following contributions:

- PET-Select, a novel approach that automatically selects the most optimal prompting engineering techniques for each code generation query.
- An evaluation of PET-Select on five widely used benchmark datasets using three LLMs.
- Quantitative and qualitative analyses and an ablation study that provide insights into how PET-Select selects the appropriate PET and the contribution of each component.

3.2 Background

3.2.1 Prompt Engineering Challenges

With the increasing number of prompting techniques being proposed and achieving state-of-the-art results on various benchmark datasets, a question arises: “Can we apply the most advanced prompting techniques to every question?” Unfortunately, the answer may be no. The first and most obvious issue is that using these advanced prompting techniques for every question is costly, as they often require multiple rounds of interactions with language models or involve crafting lengthy prompts with numerous examples. The second, and less well-known issue is that applying advanced prompting techniques to simpler questions can sometimes lead to incorrect answers. A recent study [24] experimented on a variant of GSM8K, where all the answers to the questions in the dataset were explicitly stated in the questions themselves and could be obtained without any calculations. Surprisingly, the accuracy improves when language models are restricted from performing any calculations or reasoning steps, compared to when no instructions are specified. This suggests that unnecessary calculations and over-reasoning can lead to incorrect answers. There is another study [25] suggests that language models are not able to self-correct themselves. Self-correction is defined as a scenario where the model attempts to correct its initial responses purely based on its capabilities, without relying on external feedback. Many advanced prompting techniques leverage the self-correction ability of language models, such as Progressive Hint [72] and Self-refine [44]. However, research has shown that accuracy decreases with each iterative round. This suggests that the model struggles to identify and correct the specific incorrect parts. When the initial answer is correct, the model often changes the correct portion to something incorrect, resulting in a wrong answer.

PET-Select learns to determine whether a question is easy or difficult by predicting the code complexity of the ground-truth code. This allows PET-Select to choose the relatively appropriate prompting techniques for each query, applying simpler techniques to easy problems and more advanced ones to difficult problems. This approach helps prevent over-reasoning and redundant calculations for easy questions while also avoiding situations where the model changes a correct answer to an incorrect one.

3.2.2 Automated Prompt Engineering

Since LLMs are too large to fine-tune for every downstream task, prompt engineering has become a common approach to optimize performance across various tasks, including unseen ones. However, designing effective prompts for each task is a challenging process. Several studies have suggested reliable methods to improve language model performance, such as Chain-of-Thought and Self-correction prompting. Despite this, the question remains whether we can develop a system that automatically generates appropriate prompts for different queries. Previous studies [26, 27] proposed frameworks for automatic instruction generation and selection, where several candidate prompts are generated by LLMs, and the best prompt is chosen from these candidates. Another approach involves retrieving similar queries from a

database and using them to create a more effective prompt [94]. However, these automatic prompt engineering methods primarily focus on crafting a single optimal prompt for a given problem. There is limited research on how to design multi-round prompting, where multiple interactions with language models are used to refine the response. Crafting prompts based on the model’s responses is crucial, as many state-of-the-art prompting techniques rely on self-generated answers to achieve optimal performance. Whether used for correction or evaluation, iterative interactions with language models play a key role in helping them generate better responses.

Technically, prompting technique selection is also a form of automatic prompt engineering, as it involves choosing the relatively appropriate prompt automatically. Unlike previous approaches, prompting technique selection considers whether the prompt should be crafted for a single or multiple iterations, allowing for multiple rounds of interaction. A previous study [56] selects prompting techniques after each execution, which is costly and impractical in real-world applications, particularly when multiple techniques are considered as candidates. PET-Select is the first framework to select prompting techniques prior to execution. It employs a traditional deep learning model with contrastive learning to select the most suitable technique for each question, making it applicable and affordable even without the need to run language models.

3.3 Approach

In this work, we propose PET-Select, a novel method to select suitable prompt engineering techniques (PETs) for each query. Figure 3.1 provides the overview of PET-Select. PET-Select is a supervised learning approach, and since no such record of execution is available for various prompt engineering techniques, we start off by building the data in the Ranked PET Dataset Construction phase (Section 3.3.1). PET-Select’s model consists of two main parts: the Finetuned CodeBERT Embedding Model (Section 3.3.2) and the Selection Model (Section 3.3.3). Finally, we conduct a 5-fold cross-validation evaluation to ensure that PET-Select is correctly evaluated.

3.3.1 Ranked PET Dataset Construction

To train PET-Select, we first need to conduct a study to collect the dataset of execution records of various representative PETs such as Zero-shot, and Few-shot and rank each PET given their performance and cost for each query. Since numerous PETs could be employed for the code generation task, we select the most representative ones by choosing at least one technique from each fundamental strategic design category, such as root techniques, refinement-based techniques, and others, as defined in a recent study [98]. Detailed descriptions and implementations of these prompting techniques are provided in Section 3.4.1.

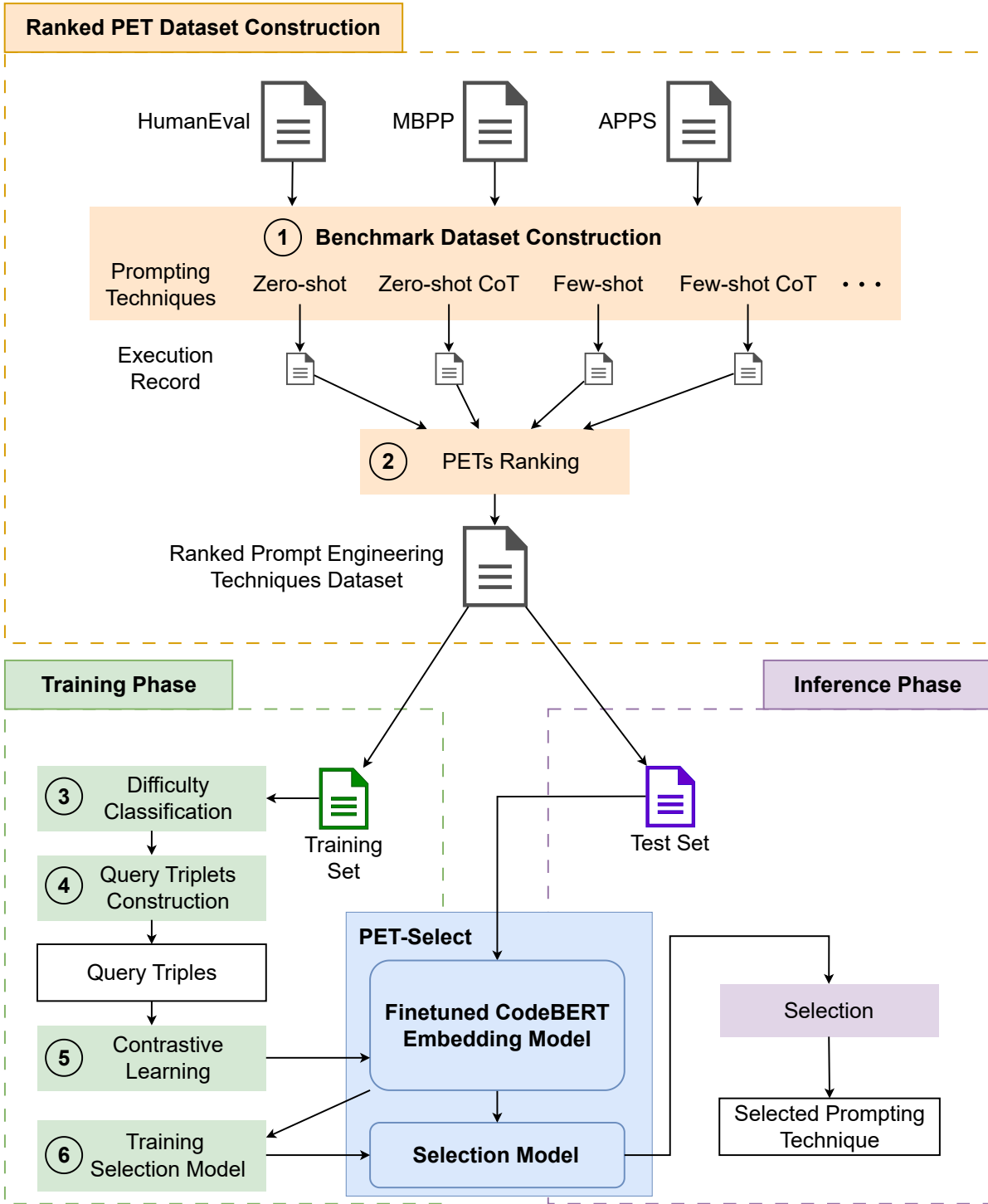


Figure 3.1: PET-Select Overview.

Benchmark Dataset Construction

To construct this dataset, we choose the five most popular code generation benchmark datasets HumanEval, HumanEval+, MBPP, MBPP+, and APPS, and then employ selected PETs on GPT-3.5 Turbo, GPT-4o, and DeepSeek-V3 (Step ①). The responses were recorded along with the cost of the query in terms of the number of input and output tokens. In addition to the query cost, the generated code complexity measured by five metrics is also recorded and normalized. Details of the metrics used are provided in Section 3.4.2.

PETs Ranking

Once every technique has been benchmarked, we select the most appropriate one for each query with the highest R_Score_i (Step ②). Where R_Score_i for technique i is calculated as:

$$R_Score_i = \log(\max_{j=1}^N(T_tokens_j)) \times pass_i - \log(T_tokens_i)$$

Here, T_tokens is the sum of the number of input and output tokens required by PET i , and the $\max(\max_{j=1}^N(T_tokens_j))$ represents the highest number of required tokens across all prompting techniques for that query. The binary number $pass_i$ is 1 (i.e., the generated code passes all test cases) and 0 (i.e., at least one test case failed). Specifically, for techniques that fail to generate test passing code, the formula ensures that the score will be negative, and for successful techniques, the score will be positive. In all cases, the score is always inversely proportional to the number of required tokens. In the end, the technique that generates the correct code while requiring the fewest number of tokens will have the highest score. Since two techniques almost never use the same number of tokens, we can almost always choose the most appropriate one for each query.

After this stage, we will obtain the Ranked PETs Dataset where each entry includes the query string, the generated code, the number of tokens used, the complexity measures, and the most successful PET with the highest R.score as the label.

3.3.2 Fine-tuning CodeBERT Embedding Model

As elaborated in the Introduction, single-round PETs are generally more suited to simpler problems whereas multiple-round PETs are designed to tackle more complex problems with multiple rounds of reasoning [96]. The code complexity of the solution could be a good proxy to estimate the problem’s complexity that can bring insight to better select the appropriate PET. Hence, we incorporate the generated code complexity into our PET-Select decision-making process to achieve the best selection. We accomplish this by tuning CodeBERT [99] embedding model utilizing contrastive learning [97]. Specifically, the tuning process reshapes the embedding space so that prompts with similar generated (i.e., solution) code complexity will be closer while dissimilar prompts are placed farther apart.

Difficulty Classification

To perform contrastive learning, we need to know which are easy prompts and which are hard prompts. We can estimate this by using the code complexity scores measured by five diverse metrics recorded in Section 3.3.1 to classify the problem difficulty. Instead of selecting an arbitrary threshold to distinguish between easy and difficult questions, we utilize the difficulty labels in the APPS dataset to train a simple difficulty classification model consisting of three fully connected layers network with ReLU activation functions (Step ③). The input tensor consists of the five metric scores, while the difficulty labels from the APPS dataset serve as the output. Specifically, “Introductory” questions are assigned easy labels, while “Interview” and “Competition” questions are assigned hard labels. Given the class imbalance in the dataset, the model is trained using weighted binary cross-entropy loss to account for the disproportion between easy and hard instances. This model can then be used on the unseen testing data in the APPS dataset or other datasets such as HunmanEval and MBPP.

Query Triplets Construction

Contrastive learning performs optimization on query triplets each including an anchor query, a positive query, and a negative query [100, 101]. Specifically, anchor queries are the original natural language questions, positive queries are either semantically equivalent to or share the same answer as the anchor queries [97], while negative queries are unrelated to both the anchor and positive queries.

In this work, for a given query (i.e., the anchor query), we select positive queries as those with similar difficulties (same label given by classification model) and negative queries as those with differing difficulties (Step ④). As mentioned previously, the problem difficulties are estimated with the solution code complexity using our difficulty classification model. For instance, given an easy anchor query “*Write a function to find the k th element in the given array.*”, with the difficulty score of 0.42 given by our model, the positive query could be “*Write a function to find the ratio of zeroes in an array of integers.*” with the same difficulty score of 0.42, and the negative query could be “*Write a function to find the maximum product subarray of the given array.*” which is a more challenging problem with a higher score of 0.64.

To simplify the triplet process, we divide the entire training set into two sets of easy and hard problems based on the results of the difficulty classification. We then randomly select a query from the same set as the anchor query to serve as its positive query. Conversely, a query is randomly selected from the opposite set to serve as the corresponding negative query.

Contrastive Learning

Once the query triplets are constructed, we use them to fine-tune the CodeBERT sentence embedding model (Step ⑤). The objective of contrastive learning is to bring queries with

- A** Write a function to find the kth element in the given array.
Complexity Score: 0.42
- P** Write a function to find the ration of zeroes in an array of integers.
Complexity Score: 0.42
- N** Write a function to find the maximum product subarray of the given array.
Complexity Score: 0.64

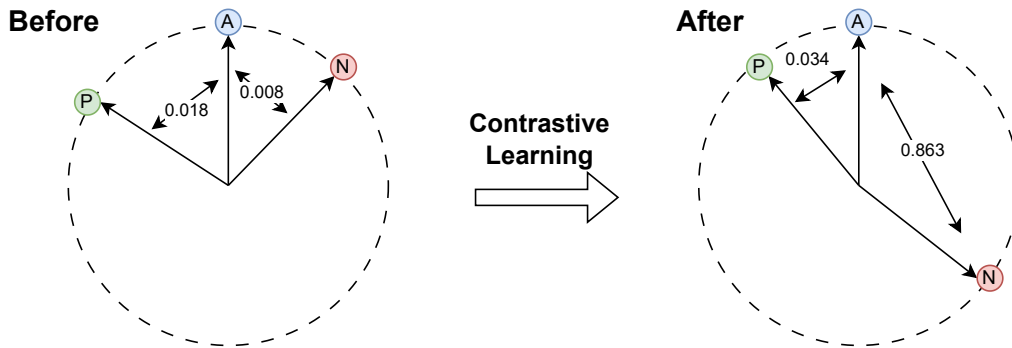


Figure 3.2: An example that demonstrates contrastive learning on a single anchor query.

similar features and complexity closer together while pushing unrelated queries with differing complexity further apart [102]. When constructing the query triplets, we designate an input query as the anchor, treating queries with similar difficulty as positive examples, while those with dissimilar difficulty are used as negative examples. This design allows the model to learn semantic representations by associating anchor queries with their positive counterparts, positioning them closer within the embedding vector space. Conversely, we expect the model to push unrelated queries further apart from the anchor queries. Figure 3.2 illustrates the examples discussed in Section 3.3.2 to show the progress of contrastive learning in PET-Select. Before contrastive learning, the cosine distance between the anchor sentence (blue point) and the positive sentence (green point) is 0.018, which is greater than the distance between the anchor sentence and the negative sentence (red point), measured at 0.008. However, after contrastive learning, the positive sentence moves closer to the anchor sentence in the embedding vector space, with a reduced distance of 0.034, while the negative sentence is pushed farther away, increasing the distance to 0.863.

PET-Select’s model architecture is built with the Sentence Transformer framework, specifically leveraging CodeBERT as a transformer-based model for sentence embedding. First, the pre-trained CodeBERT model is used to extract embeddings for each word in sentences (in the anchor, positive, and negative queries). These word embeddings are aggregated with a pooling layer to create a fixed-size sentence-level embedding. The embedding model is fine-tuned by minimizing a Triplet Loss, which is computed based on the distances between

the anchor-positive and anchor-negative query pairs:

$$L = \max(0, \text{Distance}_{\text{anchor,positive}} - \text{Distance}_{\text{anchor,negative}} + \text{margin})$$

In short, the loss function is to learn an embedding space where semantically similar sentences are clustered together (small distance), and dissimilar sentences are far apart (large distance) [103]. The *margin* is a positive value (by default set to 1 in the model) that defines a minimum gap between the anchor-positive and anchor-negative distances. It ensures that the negative sentence is not simply pushed just outside the positive one but is kept at a meaningful distance. The *max* function ensures the loss is non-negative, meaning if the distance between the negative and the anchor is already sufficiently large, the loss will be zero (i.e., no update is needed for this triplet).

3.3.3 Training Selection Model

Once the embedding is computed, it can be used to extract a sentence embedding for any given query. The embedding will be used as input to PET-Select’s three fully connected layers of neural network with ReLU activation function (Step ⑥). These layers are tasked with multi-label classification (i.e., PET selection). Specifically, the predicted technique is selected based on the highest probability according to the sigmoid function. These layers are trained normally using cross-entropy loss. It is important to note that the data used for training both the sentence embedding model and the selection model is within the training dataset and the model never sees the test set which is set aside to evaluate the model. For evaluation, we also record the probability of each class to calculate the MRR and nDCG metrics (described in Section 3.5) for the results.

3.4 Experimental Setup

In this section, we introduce the setup that we used to conduct our experiments. We first introduce the prompting techniques that are included in PET-Select selection pool, we then discuss the code complexity metrics, and finally, the experimental setting including the code generation datasets and the evaluation metrics.

3.4.1 Prompt Engineering Techniques (PETs) for code generation

Table 3.1 provides a summary of the PETs used in our experiment. To ensure a broad exploration of techniques, we selected at least one from each category as stated in the recent work [98]. These prompting techniques are classified into five categories based on their core concepts: root techniques, refinement-based techniques, decomposition-based techniques, reasoning-based techniques, and priming techniques. The “Iteration” column specifies whether the technique involves iterative interactions with the language models. The “Template” column demonstrates the prompting templates we used for each technique. For techniques with multiple iterations, we provided specific prompting templates for each stage.

Table 3.1: The prompting techniques used in the experiments. The ‘Iteration’ column specifies whether the technique requires multiple rounds of interaction with LLMs. The ‘Template’ column outlines the specific prompt template used in the experiments.

	Iteration	Template
Zero-shot [91]	Single	Only generate the Python code for the following task. {Coding Task} .
Few-shot [91]	Single	Here are some examples of how to generate the code. {Three examples} . How about this task? {Coding Task} .
Zero-shot CoT [48]	Single	Only generate the Python code for the following task. {Coding Task} . Let’s generate the code step by step.
Few-shot CoT [46]	Single	Here are some examples of how to generate the code step by step. {Three examples with reasoning steps} . How about this task? {Coding Task} .
Persona [104]	Single	You are a programming expert, especially good at Python. Please complete the following task in Python: {Coding Task} .
Self-planning [105]	Multiple	Plan Stage: {Three examples of showing the Intent and Plan} How about this intent: {Coding Task} . Implementation Stage: {Coding Task} . Please complete the task with the following plan in Python. {Plan generated by the Plan Stage} .
Self-refine [44]	Multiple	Initial Stage: Only generate the Python code for the following task. {Coding Task} Reflection Stage: Here is a code snippet: {Code generated by Initial Stage} . Please review the code and suggest any improvements or identify any issues. Refinement Stage: Here is a code snippet: {Code generated by Initial Stage} . Based on the following feedback, refine the code: {Feedback generated by Reflection Stage} .
Progressive Hint [72]	Multiple	Initial Stage: Please complete the following task in Python. {Coding Task} . Hint Stage: Please complete the task in Python. The answer is near to: {Code generated by Initial Stage} .
Self-debug [93]	Multiple	Initial Stage: Only generate the Python code for the following task. {Coding Task} Your code should pass the test: {One test case of the Coding Task} . Success Stage: {Code generated by Initial Stage} . Is the code above correct? If not, please fix it. Failure Stage: {Code generated by Initial Stage} . The code above is wrong. Please fix it.

We briefly go through each PET and provide some pros and cons to emphasize that no one PET is optimal for all cases.

Root PETs: Zero-shot and Few-shot Root PETs directly query LLMs for answers. Zero-shot and Few-shot [91] are two examples of root PETs where Zero-shot provides no additional example and Few-shot includes several examples. While it is convenient and requires no domain-specific input, Zero-shot performance may be limited when the model encounters unfamiliar tasks. The added examples in Few-shot PET improve LLMs’ ability to handle unseen tasks but are not trivial to craft [19, 20, 21] and can negatively impact the performance if given incorrectly [22, 23].

Reasoning PETs: Zero-shot/Few-shot Chain-of-Thought (CoT) are reasoning-based techniques that query LLMs to explain intermediate reasoning steps while generating answers [46, 48]. It enables LLMs to produce more coherent and accurate results. The zero-shot and few-shot CoT differ in the presence of examples: zero-shot CoT does not include examples while few-shot CoT offers additional reasoning examples in the query. Despite the performance improvements similar limitations persist: zero-shot CoT can yield unreliable results on unfamiliar tasks, and the need for carefully crafted prompts with examples remains a challenge with few-shot CoT.

Priming PETs: Persona is a PET that LLM is guided to take on a specific identity or personality based on expertise, tone, or role. This “persona” helps make the communication with LLMs consistent, but a too specific persona can lead to restrictive communication [106].

Decomposition PETs: Self-planning involves having the LLMs create a mental blueprint or set of steps before answering a question. This is particularly useful for complex tasks that require a structured approach (e.g., solving math problems) [107]. On the one hand, this can provide structure to the solution but on the other, if the initial plan is incorrect, the entire response may be off track.

Refinement PETs: Self-refine, Progressive Hint, and Self-debug take a different approach by having the LLM interact with its own response after generating it. Specifically, Self-refine [44], Progressive Hint [72], and Self-debug [93] ask the LLM to review its answers, use its answers as hints, and correct its output based on the execution result of test cases. While self-refine can sometimes correct itself, the errors might still pass notice. Progressive Hint also suffers from similar pitfalls where the first hint can be incorrect and create a domino effect. Finally, with the help of the external test cases, self-debug can sometimes correct itself, however, the debugging process is not perfect and sometimes LLM can over-correct itself thus generating the wrong answer.

3.4.2 Code complexity metrics

PET-Select utilize five popular code complexity metrics: Line of Code, Cyclomatic Complexity, Halstead Complexity, Cognitive Complexity, and Maintainability Index [108, 109, 110] to aid with the contrastive learning step: **Line Complexity** is also known as Lines of Code (LOC), which measures the number of lines in a codebase. In this study, Line Complexity is calculated using Physical Lines of Code (PLOC), which excludes comment lines and focuses

solely on the program’s source code. **Cyclomatic Complexity** [111] counts the number of independent paths through the code. Higher cyclomatic complexity indicates more potential paths, increasing the testing effort and potentially reducing maintainability. **Halstead Complexity** [112] evaluates code complexity from both linguistic and mathematical perspectives, based on the number of operators and operands. **Cognitive Complexity** [113] measures how difficult code is for a human to understand by considering factors like nesting depth and control structures such as if, switch, and for loops. Unlike cyclomatic complexity, it focuses on readability and the mental effort required to follow the code. **Maintainability Index** [114] is a composite metric that predicts the ease of maintaining a software system, combining factors like cyclomatic complexity, Halstead complexity, and lines of code. It ranges from 0 (difficult to maintain) to 100 (easy to maintain), with higher values indicating better maintainability. In this study, custom code was used to calculate LOC, the Radon package was used to calculate Cyclomatic Complexity, Halstead Complexity, and Maintainability Index, and Cognitive Complexity was computed with the cognitive-complexity Python package.

3.4.3 Experiment Settings

Benchmark datasets We used five of the most widely used code generation benchmark datasets to train the model and evaluate PET-Select’s performance: HumanEval [74], HumanEval+ [115], MBPP [116], MBPP+ [115], and APPS [71]. Since HumanEval and MBPP already achieve high accuracy with advanced models, we include APPS as a more challenging dataset to evaluate PET-Select’s performance on harder questions. To ensure a fair comparison, we used the sanitized version of MBPP and filtered out all instances from APPS that did not include the ground-truth code. All datasets provide test cases so that generated code can be functionally evaluated and the pass@k metric can be calculated for evaluation.

Ranking Evaluation Metrics Since PET-Select ranks all the prompting techniques based on the probability of softmax layer output, we applied two popular metrics, Mean Reciprocal Rank (MRR) [117, 118] and Normalized Discounted Cumulative Gain (nDCG) [119], to evaluate PET-Select. These metrics are used extensively in the domain of information retrieval and they measure the ability of a system in recommendation tasks. The Mean Reciprocal Rank measures the effectiveness of a system in returning relevant results or answers by focusing on the rank of the first correct. On the other hand, Normalized Discounted Cumulative Gain (nDCG) measures the quality of ranked results based on the relevance of each result and the position in which they appear in a ranking list. With these two metrics, we can thoroughly evaluate PET-Select’s ability to recommend and rank the appropriate PET.

Environmental Settings We select GPT-3.5 Turbo, GPT-4o, and DeepSeek-V3 for our experiments as they present weaker models, state-of-the-art models, and open-source models respectively which allows us to comprehensively evaluate our approach. We utilize a machine with an 8-core processor AMD Ryzen 7 pro 5845 and an NVIDIA RTX3060 to train PET-Select as well as cloud services for LLMs’ queries.

To better evaluate PET-Select we applied 5-fold cross-validation with 80-20 train-test split. Note that the difficulty classification model, sentence embedding model, and PET selection model were only trained on the training set to prevent test data leakage into the sentence embedding model, which could otherwise bias the evaluation of PET-Select. The only exception is when evaluated on HumanEval and MBPP, the entire APPS data can be used for difficulty classifier training as they are separate datasets and there is no risk of data leakage.

We fine-tune the sentence embedding model for fifteen epochs and select the model with the best performance (the highest value of Cosine Accuracy) on the validation set to train the selection model. For the selection model, we train it for 10 epochs and select the model with the best performance (the highest value of nDCG) on the validation set to choose prompting techniques for each instance in the test set.

3.5 Result

In this section, we evaluate PET-Select by exploring three research questions and discussion. RQ1 compares PET-Select performance against vanilla PETs and a random baseline on five code generation benchmarks utilizing three LLMs (Section 3.5.1). In RQ2, we conduct an ablation study to assess the importance of the prompt difficulty classification model and contrastive learning in PET-Select (Section 3.5.2). In RQ3, we further analyze how PET-Select selects the appropriate PET by performing a quantitative analysis of PET-Select’s ranking performance (Section 3.5.3). Finally, we perform a qualitative analysis to provide some insight to support the experimental results (Section 3.5.4).

3.5.1 RQ1. How do PET-Select compare to single PETs and baselines?

In this RQ, we evaluate the performance of PET-Select by comparing it to the baselines, including a single PET model and a random selection baseline. In the first part (RQ1.1), we evaluate PET-Select on HumanEval, MBPP, and APPS using 5-fold cross-validation. Since previous studies [115] suggest that the test suites in HumanEval and MBPP have limited quantity and quality, which may make them insufficient for accurately measuring code generation performance, we also evaluate all techniques on HumanEval+ and MBPP+, enhanced versions of HumanEval and MBPP with automatically generated and higher-quality test suites in RQ1.2.

RQ1.1 Evaluation on HumanEval, MBPP, and APPS dataset

Table 3.2 presents the pass@1 accuracy and token usage for nine PETs, and two selection approaches on the HumanEval, MBPP, and APPS datasets. Three LLMs, GPT-3.5 Turbo, GPT-4o, and DeepSeek-V3, were used. PETs marked with a star, such as Self-planning,

Table 3.2: Pass@1 accuracy and token usage on HumanEval, MBPP, and APPS across different models. *Acc* refers to Pass@1 accuracy (%), and *#Token* is the average token usage.

Model	Dataset	HumanEval		MBPP		APPS	
		Acc	#Token	Acc	#Token	Acc	#Token
GPT-3.5 Turbo	Zero-shot	69.5	235	69.6	147	15.1	489
	Zero-shot CoT	65.9	255	66.7	156	19.0	536
	Few-shot	68.9	903	77.0	667	16.1	2278
	Few-shot CoT	69.5	1235	74.1	944	19.3	2579
	Persona	65.2	280	68.5	175	17.8	513
	Self-planning*	65.2	1354	65.6	961	8.8	2645
	Self-refine*	40.2	1230	49.5	1114	15.4	2146
	Progressive Hint*	69.5	1022	71.4	649	15.5	1856
	Self-debug*	64.6	663	64.3	731	17.6	1272
	Random Selection	62.9	759	66.6	594	15.1	1580
PET-Select	71.4 $\uparrow 1.9$	619 $\downarrow 49.9\%$	77.2 $\uparrow 0.2$	678 $\uparrow 1.6\%$	20.1 $\uparrow 0.8$	1379 $\downarrow 46.5\%$	
GPT-4o	Zero-shot	85.4	295	85.7	185	40.1	648
	Zero-shot CoT	84.1	344	81.7	217	40.7	689
	Few-shot	87.2	1005	83.3	683	40.1	2175
	Few-shot CoT	88.4	1335	77.2	1037	42.9	2656
	Persona	89.6	369	84.7	213	41.5	695
	Self-planning*	85.4	1842	75.9	1434	32.5	3384
	Self-refine*	56.1	1839	78.0	1575	43.0	3223
	Progressive Hint*	90.9	1378	84.9	819	42.6	2574
	Self-debug*	84.8	994	63.0	1011	40.6	1835
	Random Selection	83.5	942	80.2	780	40.8	1995
PET-Select	91.5 $\uparrow 0.6$	1122 $\downarrow 18.6\%$	86.0 $\uparrow 0.3$	236 $\uparrow 27.6\%$	44.4 $\uparrow 1.1$	2742 $\downarrow 14.9\%$	
DeepSeek-V3	Zero-shot	84.8	250	86.5	136	33.9	687
	Zero-shot CoT	88.4	308	84.7	170	38.3	700
	Few-shot	92.1	1188	85.7	670	40.9	2270
	Few-shot CoT	87.2	1576	87.6	979	46.7	2747
	Persona	88.4	341	86.0	192	39.4	759
	Self-planning*	90.9	2132	73.0	1774	36.8	3982
	Self-refine*	68.3	2273	82.3	1938	33.4	3971
	Progressive Hint*	89.6	1113	86.2	654	37.5	2573
	Self-debug*	82.3	917	66.7	1066	43.4	2064
	Random Selection	81.8	1060	83.9	789	38.1	2209
PET-Select	92.7 $\uparrow 0.6$	1216 $\uparrow 2.3\%$	87.8 $\uparrow 0.2$	941 $\downarrow 3.9\%$	46.8 $\uparrow 0.1$	2137 $\downarrow 22.2\%$	

require multiple iterative rounds to arrive at the answer. The ‘Random Selection’ row represents a baseline approach where one of the nine PETs is randomly chosen as the most appropriate for each instance. The overall accuracy and token usage are then calculated based on the selected technique.

In most cases, PET-Select outperforms all individual PETs while requiring fewer tokens in most cases. For example, on the HumanEval dataset, PET-Select achieves an accuracy of 71.4%, which is 1.9% higher than the best of the other techniques while using 49.9% fewer tokens with GPT-3.5 Turbo. With GPT-4o, PET-Select reaches an accuracy of 91.5%, surpassing the best accuracy of other techniques by 0.6% while also reducing token usage by up to 18.6%. A similar result is observed with DeepSeek-V3, PET-Select also achieves a 0.3% increase in accuracy on MBPP, while reducing token usage by 3.9%.

In some cases, PET-Select outperforms all individual PETs but at a small cost of more tokens. For example, on the MBPP dataset, PET-Select achieves an accuracy of 77.2% with GPT-3.5 Turbo, which is 0.2% higher than the best accuracy achieved by Few-shot PET while using 1.6% more tokens. Similarly, on HumanEval, with DeepSeek-V3, PET-Select achieves an accuracy of 92.7%, improving the best PET technique by 0.6% with a 2.3% increase in token usage. This is a reasonable tradeoff for a consistently better performance compared to any single PET.

On the APPS dataset, which is considered more challenging, PET-Select consistently achieves higher accuracy while requiring fewer tokens. Specifically, PET-Select reaches an accuracy of 20.1% while using 46.5% fewer tokens than the best of the other techniques with GPT-3.5 Turbo. With GPT-4o, PET-Select attains an accuracy of 44.4%, surpassing the best of the other techniques by 1.1% while reducing token usage by 14.9%. Additionally, with DeepSeek-V3, PET-Select achieves the highest accuracy at 46.8% while saving 22.2% of tokens.

One thing to note is that random selection performs similarly to an average PET while requiring a higher-than-average number of tokens. This indicates that problem difficulty is a key feature that can help improve the PET selection method like PET-Select, which consistently achieves state-of-the-art performance while requiring a fewer number of tokens in most cases.

RQ1.2 Evaluation on HumanEval+ and MBPP+ datasets

As mentioned, test suites in HumanEval and MBPP have limited quantity and quality, making them potentially insufficient for accurately measuring code generation performance. Hence, in addition to the standard version, we also evaluate PET-Select on the enhanced datasets with more comprehensive test cases, HumanEval+ and MBPP+. Since the questions in HumanEval+ and MBPP+ are similar to those in the original HumanEval and MBPP datasets, we use the same weights from the contrastive model trained on HumanEval and MBPP in our selection model. Table 3.3 presents the Pass@1 accuracy and token usage for HumanEval+ and MBPP+ across three LLMs. PET-Select consistently achieves the highest accuracy across datasets with all models, with an improvement of up to 0.8%

Table 3.3: Pass@1 accuracy and token usage on HumanEval+ and MBPP+ across different LLMs. *Acc* refers to Pass@1 accuracy (%), and *#Token* is the average token usage.

Model	Dataset	HumanEval+		MBPP+	
		Acc	#Token	Acc	#Token
GPT-3.5 Turbo	Zero-shot	64.6	237	65.9	149
	Zero-shot CoT	62.2	252	66.9	160
	Few-shot	61.0	900	70.9	668
	Few-shot CoT	65.2	1231	69.3	941
	Persona	59.1	281	63.5	176
	Self-planning*	62.8	1362	61.9	967
	Self-refine*	33.5	1213	35.7	1109
	Progressive Hint*	67.1	1034	65.1	648
	Self-debug*	51.2	1085	59.5	719
	Random Selection	52.5	807	62.2	597
	PET-Select	67.2 $\uparrow 0.1$	1011 $\downarrow 2.2\%$	71.7 $\uparrow 0.8$	676 $\uparrow 1.2\%$
GPT-4o	Zero-shot	81.1	298	75.9	187
	Zero-shot CoT	82.3	347	74.1	221
	Few-shot	84.8	1006	73.8	684
	Few-shot CoT	82.3	1331	69.0	1042
	Persona	84.8	373	76.2	214
	Self-planning*	81.1	1849	70.4	1437
	Self-refine*	47.6	1831	66.1	1582
	Progressive Hint*	84.1	1367	75.7	831
	Self-debug*	76.8	1869	54.8	1023
	Random Selection	76.9	1083	69.3	781
	PET-Select	85.4 $\uparrow 0.6$	972 $\downarrow 3.4\%$	76.7 $\uparrow 0.5$	273 $\uparrow 27.6\%$
DeepSeek-V3	Zero-shot	81.7	245	76.7	138
	Zero-shot CoT	86.0	299	73.3	172
	Few-shot	87.2	1195	77.8	672
	Few-shot CoT	82.3	1200	76.7	987
	Persona	81.7	350	75.7	192
	Self-planning*	85.4	2177	66.7	1766
	Self-refine*	61.6	2270	66.1	1942
	Progressive Hint*	86.6	1121	76.2	639
	Self-debug*	64.0	1789	58.7	1050
	Random Selection	72.0	1081	73.6	823
	PET-Select	87.8 $\uparrow 0.6$	1298 $\uparrow 8.6\%$	78.6 $\uparrow 0.8$	701 $\downarrow 4.3\%$

Table 3.4: Ablation of PET-Select components on HumanEval, MBPP, and APPS across different models.

Model	Dataset	HumanEval		MBPP		APPS	
	Metrics	Acc	#Token	Acc	#Token	Acc	#Token
GPT-3.5 Turbo	PET-Select w/o difficulty classification	68.4 \downarrow 3.0	786 \uparrow 27.0%	75.4 \downarrow 2.2	701 \uparrow 3.4%	19.1 \downarrow 1.0	1624 \uparrow 17.8%
	PET-Select w/o contrastive learning	65.2 \downarrow 6.2	907 \uparrow 46.5%	77.0 \downarrow 0.2	667 \downarrow 1.6%	19.4 \downarrow 0.7	1516 \uparrow 9.9%
	PET-Select	71.4	619	77.2	678	20.1	1379
GPT-4o	PET-Select w/o difficulty classification	87.2 \downarrow 4.3	1065 \downarrow 5.1%	85.4 \downarrow 0.6	494 \uparrow 109%	43.6 \downarrow 0.8	2329 \downarrow 15.1%
	PET-Select w/o contrastive learning	87.8 \downarrow 3.7	1153 \uparrow 2.8%	84.9 \downarrow 1.1	201 \downarrow 14.8%	41.0 \downarrow 3.4	2364 \downarrow 13.8%
	PET-Select	91.5	1122	86.0	236	44.4	2742
DeepSeek-V3	PET-Select w/o difficulty classification	91.5 \downarrow 1.2	1628 \uparrow 33.9%	86.5 \downarrow 1.3	660 \downarrow 29.9%	45.2 \downarrow 1.6	2214 \uparrow 3.6%
	PET-Select w/o contrastive learning	89.6 \downarrow 3.1	1631 \uparrow 34.1%	87.6 \downarrow 0.2	969 \uparrow 3.0%	46.7 \downarrow 0.1	2372 \uparrow 11.0%
	PET-Select	92.7	1216	87.8	941	46.8	2137

on MBPP+ using DeepSeek-V3 and GPT-3.5 Turbo. Furthermore, in most cases, PET-Select’s token usage is an acceptable tradeoff to achieve state-of-the-art performance. These results indicate that even when evaluated on more rigorous datasets, PET-Select continues to outperform other baselines.

Finding 1: Overall, PET-Select consistently outperforms other baseline approaches across different models on all datasets, achieving up to a **1.9% improvement in accuracy** while using up to **49.9% fewer tokens**. Furthermore, PET-Select maintains its superior performance even when evaluated on datasets with more comprehensive test cases, indicating its robustness and effectiveness in code generation across various scenarios while being PET-agnostic and easy to extend.

3.5.2 RQ2. How do the different components contribute to PET-Select’s performance

In this RQ, we conduct an ablation study on PET-Select to gain insights into how its components contribute to the overall performance. Table 3.4 presents the Pass@1 accuracy and token usage for two different watered-down versions of PET-Select (PET-Select without difficulty classification and PET-Select without contrastive learning).

To evaluate the contribution of the prompt difficulty classification model (described in Section 3.3.2) in selecting the ‘hard’ and ‘easy’ problems, we evaluate the first watered-down version, PET-Select without difficulty classification. Instead of training a classification model to distinguish between easy and hard prompts, we simply set an average code complexity threshold to divide the training set into ‘hard’ and ‘easy’ problems. Specifically, we sum the scores of five complexity metrics, and if the total score is lower than the average summed score, the instance is classified as part of the easy dataset. Conversely, instances with a higher total score are categorized as part of the hard dataset. By removing the classification model, PET-Select’s Pass@1 accuracy drops by up to 4.3%, and token consumption increases by up to twice as much.

Table 3.5: Ranking effectiveness (MRR and nDCG) of selection methods on HumanEval, MBPP, and APPS across different models.

Model	Dataset	HumanEval		MBPP		APPS	
	Metrics	MRR	nDCG	MRR	nDCG	MRR	nDCG
GPT-3.5 Turbo	Random Selection	0.2735	0.7626	0.2733	0.7779	0.1115	0.2475
	PET-Select w/o difficulty classification	0.2972	0.7913	0.2985	0.8021	0.1306	0.2699
	PET-Select w/o contrastive learning	0.2846	0.7795	0.3006	0.8121	0.1308	0.2702
	PET-Select	0.3052 $\uparrow 2.7\%$	0.7982 $\uparrow 0.9\%$	0.3008 $\uparrow 0.06\%$	0.8123 $\uparrow 0.02\%$	0.1336 $\uparrow 2.1\%$	0.2728 $\uparrow 1.0\%$
GPT-4o	Random Selection	0.3017	0.9068	0.2835	0.8517	0.1816	0.4889
	PET-Select w/o difficulty classification	0.3117	0.9211	0.2931	0.8686	0.1969	0.5042
	PET-Select w/o contrastive learning	0.3137	0.9237	0.2932	0.8689	0.1942	0.5028
	PET-Select	0.3195 $\uparrow 1.8\%$	0.9308 $\uparrow 0.8\%$	0.2941 $\uparrow 0.3\%$	0.8700 $\uparrow 0.1\%$	0.1982 $\uparrow 0.7\%$	0.5060 $\uparrow 0.4\%$
DeepSeek-V3	Random Selection	0.3077	0.9179	0.2837	0.8656	0.2006	0.5095
	PET-Select w/o difficulty classification	0.3193	0.9396	0.2934	0.8817	0.2242	0.5349
	PET-Select w/o contrastive learning	0.3192	0.9386	0.2948	0.8836	0.2294	0.5408
	PET-Select	0.3241 $\uparrow 1.5\%$	0.9427 $\uparrow 0.3\%$	0.2976 $\uparrow 0.9\%$	0.8855 $\uparrow 0.2\%$	0.2298 $\uparrow 0.2\%$	0.5414 $\uparrow 0.1\%$

To evaluate the contribution of contrastive learning, in the second watered-down version, we remove the contrastive learning process and instead use the original CodeBERT model as PET-Select’s embedding. PETs for each instance are then selected based on the output of a selection model trained directly on CodeBERT embeddings. In this version, PET-Select without contrastive learning, the Pass@1 accuracy drops by up to 6.2% while token consumption increases by up to 46.5%.

Finding 2: The ablation study indicates that removing any component of PET-Select leads to a decrease in accuracy of up to 6.2% and an increase of up to twice as many tokens, highlighting the importance of these two components in helping PET-Select identify the complexity of the problem and selecting the appropriate PET.

3.5.3 RQ3. How well PET-Select rank PETs for each query?

To better evaluate PET-Select’s PET recommendation ability for each query, we utilize two metrics, Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain (nDCG). Table 3.5 presents the ranking effectiveness of various selection strategies measured by MRR and nDCG which give more insight into how high PET-Select can rank the optimal PET for each query. Since we applied 5-fold cross-validation the MRR and nDCG values are the average results from the test set across five folds.

We observe that the values of MRR and nDCG are correlated with the Pass@1 accuracy. Strategies with higher accuracy tend to have higher values for both metrics. Across all selection strategies, datasets, and models, Random Selection consistently has the lowest MRR and nDCG values, while PET-Select achieves the highest values for both metrics. This indicates that, compared to other selection approaches, PET-Select is more effective at selecting the most appropriate PETs, leading to more correct answers.

We also observe that across all datasets and models, the MRR value is consistently lower than the nDCG value for PET-Select. This occurs because, although the selected PET is the correct technique for solving an instance, it may not be the most token-efficient option. As

a result, the selected PET is not always ranked as the best choice for the instance, leading to a lower MRR value. On the other hand, the nDCG value remains consistently higher than MRR, indicating that PET-Select can reliably select PETs that lead to consistent accuracy improvement. This is evidenced in Table 3.2, and Table 3.3, where PET-Select outperforms other approaches in terms of Pass@1 accuracy across all experiments while sometimes utilizing more tokens than the best of the rest baselines. This result indicates that PET-Select is always effective in selecting the correct technique that is capable of generating the correct code while more improvement can be made to enable PET-Select to select the most token-efficient option.

Finding 3: While PET-Select does not always select the most efficient technique in terms of token usage, it consistently returns the correct answers by choosing techniques that are capable of generating the correct code.

3.5.4 Discussion

In this section, we perform a qualitative analysis of two types of cases where the PET-Select selects the optimum PET and where PET-Select fails to do so. The analysis aims to provide some additional insight to support the experimental results as well as to provide discussions about cases where PET-Select can improve. Table 3.6 lists some example instances in the MBPP and HumanEval dataset along with the complexity score (from our model), the ground-truth PET, and which technique PET-Select chooses. For example, lines 1-6 are queries that PET-Select successfully chooses the optimal PET while lines 7-10 represent cases where PET-Select fails to do so.

To better understand how contrastive learning helps create a useful embedding, we first analyze prompts that require loops in the code to solve (lines 1-6 in table 3.6). Specifically, questions containing the term ‘nested’ (lines 1-3) will likely require code with nested loops (i.e., more complex code) with a higher complexity score (more than 0.6). For these more complex questions, iterative PETs with multiple rounds of reasoning and correction are more likely to generate the correct answer, while basic direct techniques such as Zero-shot tend to answer incorrectly. In these cases, PET-Select successfully selects the appropriate technique, indicating that it learns to embed complexity correctly. Specifically, by placing sentences containing the word ‘nested’ (i.e., hard problems) closer together in the embedding space, PET-Select is able to classify them and select the correct PETs.

In contrast, sentences that do not contain this specific keyword ‘nested’ are pushed further away from those that do. As a result, PET-Select will select relatively basic techniques for those questions. For example, the queries 4-6 in table 3.6 are also related to the list processing tasks, they only require a single loop to solve. In this case, Zero-shot is a more appropriate PET while multi-round PETs are too complex and sub-optimal. Since those questions do not contain the specific keywords that indicate complex problems (e.g., nested), PET-Select selects Zero-shot instead of multi-round PETs as the appropriate technique. The above examples demonstrate that the PET-Select can effectively select the appropriate technique

Table 3.6: Selection result of PET-Select for example instances. **X** indicates the technique is selected correctly by PET-Select, while **✓** indicates it select incorrectly.

	Prompt	Complexity Score	Ground-truth Technique	PET-Select
1	Write a function to remove the nested record from the given tuple.	0.62	Progressive Hint*	Progressive Hint* (✓)
2	Write a function to flatten a given nested list structure.	0.63	Progressive Hint*	Progressive Hint* (✓)
3	Write a function to extract the even elements in the nested mixed tuple.	0.62	Self-debug*	Self-debug* (✓)
4	Write a function to compute the sum of digits of each number of a given list.	0.34	Zero-shot	Zero-shot (✓)
5	Write a python function to convert the given string to lower case.	0.32	Zero-shot	Zero-shot (✓)
6	Write a python function to get the first element of each sublist.	0.32	Zero-shot	Zero-shot (✓)
7	You're given a list of deposit and withdrawal ... starts with zero balance. Your task is to ... falls below zero , and at that point function should return True. Otherwise it should return False.	0.65	Progressive Hint*	Zero-shot (X)
8	... Evaluates polynomial with coefficients xs at point x ... def find_zero (xs: list): """ xs are coefficients of a polynomial. find_zero find x such that poly(x) = 0. find_zero returns only zero point, even if there are many. Moreover, find_zero only takes list xs having even number of coefficients and largest non zero coefficient as it guarantees a solution.	0.67	Progressive Hint*	Zero-shot (X)
9	Given two positive integers a and b, return the even digits between a and b, in ascending order.	0.31	Zero-shot	Progressive Hint* (X)
10	Given two lists operator, ... and the second list is a list of integers. Use the two given lists to build the algebraic expression ...	0.32	Zero-shot	Progressive Hint* (X)

based on code complexity predictions derived from keywords in the queries with the help of contrastive learning. By selecting simpler PET when appropriate, PET-Select not only performs well in all cases but also reduces the overall number of tokens required when compared to multi-round PETs.

Lines 7-10 in table 3.6 illustrate cases where PET-Select fails by selecting the wrong technique for the given questions. Queries 7 and 8 require code with high complexity scores. However, PET-Select incorrectly selects Zero-shot as the appropriate technique. This may suggest that certain keywords (e.g., ‘zero’) in the questions may have inaccurate embedding representations, resulting in the wrong technique being chosen. Similarly, for queries 9 and 10, we observe a reversed pattern where despite the low complexity score, PET-Select selects multi-round PETs instead of single-round PETs. We suspect that some keywords (e.g., ‘zero’, ‘two’, or ‘second’) with incorrect embedding due to mistakes in the contrastive learning model that associate the prompts with the wrong complexity leading to the incorrect selection of PET. To demonstrate this, we replace the keyword ‘zero’ in query 8 with ‘nested’ (a hard keyword) to obtain queries 8_nested. The similarity score between the embedding of 8 and 10 is 0.12 while the similarity score between 8_nested and 10 is reduced significantly to 0.07. This indicates that the keyword ‘zero’ has an incorrect easy embedding which is the opposite of the hard embedding of the keyword ‘nested’. This change in similarity also indicate that the embedding of query 10 is incorrectly similar to a hard query which led to the incorrect selection of iterative PET. By swapping the keyword, PET-Select is able to correctly select Progressive Hint for query 8_nested. One of the possible explanations for these incorrect keyword embeddings is that the number of samples might not have been enough to allow the contrastive learning process to fully learn the complexity embedding and an automatic prompt generation process would be a possible solution to mitigate such issues. This remains future work.

3.6 Limitation & Threats to validity

3.6.1 Internal validity

Our code has been thoroughly reviewed to ensure the implementation is correct, and we have confirmed that the questions in the testing dataset are not present in the question base. We also carefully crafted our prompts for each prompting technique, adhering closely to the guidelines outlined in the original paper for each method. However, the way prompts and examples are crafted may influence the performance of each technique, which in turn can affect the results of PET-Select.

3.6.2 External validity

In our experiment, we use five of the most widely recognized benchmark datasets for code generation, HumanEval, HumanEval+, MBPP, MBPP+, and APPS to demonstrate the

effectiveness of PET-Select, which is primarily designed for Python programming. The performance of PET-Select can be different on prompting technique selection for other programming languages or other kinds of code generation tasks. In addition, we utilize CodeBERT as our embedding model and five representative code complexity metrics across five datasets in our experiments. PET-Select may perform differently with different embedding models, metrics, and data points. Future work is needed to assess the performance of PET-Select using a broader range of embedding models, metrics, and datasets.

3.6.3 Construct validity

We use MRR, nDCG, pass@k, and token usage calculated by the Tiktoken package to measure the performance of PET-Select. Our approach may have different performance under other metrics. In this work, we assume that code generation questions with similar code complexity scores are semantically equivalent when contrastively training our CodeBERT-based sentence embeddings. Future research is needed to validate this assumption using different metrics or features.

3.6.4 Limitation

Recent trends in agent-based approaches show promise in enhancing the performance of each phase. For example, using larger embedding models or applying large language models to classify the difficulty of problems can be beneficial. However, since PET-Select is designed to generate correct answers while minimizing token usage, we choose CodeBERT as our embedding model and employ five representative code complexity metrics in our experiments. To control computational costs, we deliberately avoid the use of large language models in our pipeline. This design choice introduces certain limitations, and future work may consider integrating large language models into the PET-Select framework to further improve its effectiveness.

3.7 Related work

3.7.1 Code Complexity Prediction

Code complexity prediction has emerged as a key area of focus in recent research, with various approaches leveraging machine learning and deep learning techniques. A notable advancement is the application of deep learning models, such as hierarchical Transformers, which process method-level code snippets and aggregate them into class-level embeddings [120]. These models excel in handling longer code sequences, surpassing previous methods through advanced multi-level pre-training objectives that enhance the model’s understanding of complexity-related features. Additionally, studies have explored the effectiveness of GPT-3-based models like GitHub Copilot, highlighting both their strengths and

limitations in zero-shot complexity prediction [121]. While Copilot performs well with linear complexities, specialized deep learning models demonstrate superior overall accuracy.

In contrast, PET-Select diverges from these methods by not concentrating on code complexity prediction directly from natural language queries. Instead, we employ complexity prediction as an intermediate step to determine the appropriate prompting techniques for answering natural language questions.

3.7.2 Automated Prompt Engineering

Automated prompt engineering is an emerging method to adapt large language models (LLMs) for specific tasks by optimizing prompts without altering the model’s core parameters. Techniques like AutoPrompt [122] use gradient-guided search to create prompts for tasks such as sentiment analysis and natural language inference, achieving results comparable to state-of-the-art models without additional fine-tuning. Methods such as prompt tuning [123] and prefix-tuning [124] further improve model efficiency by learning task-specific prompts while keeping the language model frozen, significantly reducing the number of tunable parameters. Additionally, approaches like Prompt-OIRL [125] optimize arithmetic reasoning through offline inverse reinforcement learning, offering cost-effective and scalable prompt recommendations. Although these automated prompt engineering approaches optimize the prompt without executing large language models (LLMs), they do not account for the iterative interaction with the models.

However, these automated prompt engineering methods focus on optimizing a single prompt without accounting for iterative interactions with LLMs throughout the process. In contrast, PET-Select addresses this limitation by incorporating iterative interaction techniques to select the most suitable prompting strategies for code generation tasks.

3.8 Conclusion

In this paper, we introduced PET-Select, a novel system for automatically selecting appropriate prompt engineering techniques (PETs) for code generation tasks based on contrastive complexity predictions. By constructing triplet datasets using a complexity classification model and leveraging them for contrastive learning on a CodeBERT-based sentence embedding model, PET-Select effectively distinguishes between simpler and more complex questions, applying the most suitable techniques accordingly. Compared to approaches that use a single technique for all instances, PET-Select achieves higher accuracy while using fewer tokens. Our evaluation on the HumanEval, HumanEval+, MBPP, MBPP+, and APPS datasets demonstrates that PET-Select not only enhances performance but also reduces computational costs. Future work will focus on refining the model with automatic prompt generation and exploring its application to other domains.

Chapter 4

Conclusions and Future work

Although LLMs demonstrate impressive performance in code generation tasks, ensuring their reliability, effectiveness, and correctness remains a significant challenge. This thesis takes an initial step toward addressing these challenges by systematically analyzing prompt engineering to enhance LLM performance. We propose frameworks designed to improve the code generation capabilities of LLMs. The findings and techniques presented in this work aim to benefit both developers and researchers.

This chapter is organized as follows: we first summarize our findings and contributions based on the presented work in this thesis (Section 4.1). Finally, we discuss some future works (Section 4.2).

4.1 Thesis Findings and Contributions

- **Application of Prompt Engineering for Code Generation:** We introduced TITAN, a novel framework designed to enhance the reasoning capabilities of large language models (LLMs) through step-back and zero-shot Chain-of-Thought (CoT) prompting. By explicitly extracting inputs and procedural steps, TITAN improves task-oriented reasoning without requiring few-shot prompting or manually curated data. Extensive evaluations across multiple benchmarks demonstrate TITAN’s ability to outperform existing prompting strategies, showing its effectiveness in improving reasoning tasks. Additionally, our analysis highlights the benefits of integrating self-consistency techniques, despite the computational overhead they introduce.
- **Optimization of Prompt Engineering for Code Generation:** We presented PET-Select, a PET-agnostic selection model that leverages code complexity as a proxy for problem complexity to dynamically choose the most suitable prompt engineering technique (PET) for code generation. By incorporating contrastive learning, PET-Select effectively distinguishes between simple and complex queries, improving selection accuracy. Our evaluations across five benchmark datasets and three LLMs demonstrate that PET-Select outperforms individual PETs, achieving up to a 1.9% increase in

pass@1 accuracy while reducing token usage by up to 49.9%. Additionally, an ablation study confirms the importance of contrastive learning and difficulty classification in improving PET-Select’s performance.

4.2 Future Work

- **Direction 1: Optimizing Self-Consistency and Expanding TITAN to Adaptive and Multimodal Reasoning:** We aim to explore methods to further reduce computational costs while maintaining performance by optimizing self-consistency techniques. Additionally, we plan to investigate adaptive step-back strategies, where the number of reasoning steps can dynamically adjust based on task complexity. Another promising direction is to extend TITAN to multimodal tasks, integrating textual reasoning with vision and other modalities to expand its applicability across diverse AI challenges.
- **Direction 2: Enhancing PET-Select with Data Augmentation, Automatic Prompt Generation, and Cross-Domain Adaptation:** Future work for PET-Select will focus on enhancing model performance through data augmentation by mutating prompts, allowing the model to learn from diverse variations of the same query and improving its robustness in selecting the optimal PET. Additionally, we aim to refine PET-Select by incorporating automatic prompt generation, enabling it to dynamically create and adapt prompts based on query complexity without relying solely on predefined techniques. This approach will enhance PET-Select’s adaptability and generalization across different code generation tasks. Furthermore, we plan to extend PET-Select’s applicability to other domains, such as natural language reasoning, automated debugging, and mathematical problem-solving, to explore its effectiveness beyond code generation.

Bibliography

- [1] J. Biolchini, P. G. Mian, A. C. C. Natali, and G. H. Travassos, “Systematic review in software engineering,” System engineering and computer science department COPPE/UFRJ, Technical Report ES, vol. 679, no. 05, p. 45, 2005.
- [2] B. Boehm, “A view of 20th and 21st century software engineering,” in Proceedings of the 28th international conference on Software engineering, 2006, pp. 12–29.
- [3] H. Van Vliet, H. Van Vliet, and J. Van Vliet, Software engineering: principles and practice. John Wiley & Sons Hoboken, NJ, 2008, vol. 13.
- [4] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén et al., Experimentation in software engineering. Springer, 2012, vol. 236.
- [5] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat et al., “Gpt-4 technical report,” arXiv preprint arXiv:2303.08774, 2023.
- [6] L. Floridi and M. Chiriatti, “Gpt-3: Its nature, scope, limits, and consequences,” Minds and Machines, vol. 30, pp. 681–694, 2020.
- [7] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican et al., “Gemini: a family of highly capable multimodal models,” arXiv preprint arXiv:2312.11805, 2023.
- [8] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong et al., “A survey of large language models,” arXiv preprint arXiv:2303.18223, 2023.
- [9] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Barnes, and A. Mian, “A comprehensive overview of large language models,” arXiv preprint arXiv:2307.06435, 2023.
- [10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” ACM Transactions on Software Engineering and Methodology, 2023.

- [11] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, “Language models for code completion: A practical evaluation,” in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
- [12] R. Abhyankar, Z. He, V. Srivatsa, H. Zhang, and Y. Zhang, “Apiserve: Efficient api support for large-language model inferencing,” arXiv preprint arXiv:2402.01869, 2024.
- [13] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” IEEE Transactions on Software Engineering, 2023.
- [14] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, “Can chatgpt support developers? an empirical evaluation of large language models for code generation,” in 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), 2024, pp. 167–171.
- [15] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” Advances in Neural Information Processing Systems, vol. 36, 2024.
- [16] L. Yu, W. Jiang, H. Shi, J. Yu, Z. Liu, Y. Zhang, J. T. Kwok, Z. Li, A. Weller, and W. Liu, “Metamath: Bootstrap your own mathematical questions for large language models,” arXiv preprint arXiv:2309.12284, 2023.
- [17] Y. Li, Y. Zhang, and L. Sun, “Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents,” arXiv preprint arXiv:2310.06500, 2023.
- [18] B. Goertzel, “Generative ai vs. agi: The cognitive strengths and weaknesses of modern llms,” arXiv preprint arXiv:2309.10371, 2023.
- [19] H. Dang, L. Mecke, F. Lehmann, S. Goller, and D. Buschek, “How to prompt? opportunities and challenges of zero- and few-shot learning for human-ai interaction in creative applications of generative models,” 2022.
- [20] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, “What makes good in-context examples for gpt-3?” arXiv preprint arXiv:2101.06804, 2021.
- [21] E. Perez, D. Kiela, and K. Cho, “True few-shot learning with language models,” Advances in neural information processing systems, vol. 34, pp. 11 054–11 070, 2021.
- [22] O. Rubin, J. Herzig, and J. Berant, “Learning to retrieve prompts for in-context learning,” arXiv preprint arXiv:2112.08633, 2021.
- [23] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, “Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity,” arXiv preprint arXiv:2104.08786, 2021.

- [24] C.-H. Chiang and H.-y. Lee, “Over-reasoning and redundant calculation of large language models,” [arXiv preprint arXiv:2401.11467](#), 2024.
- [25] J. Huang, X. Chen, S. Mishra, H. S. Zheng, A. W. Yu, X. Song, and D. Zhou, “Large language models cannot self-correct reasoning yet,” [arXiv preprint arXiv:2310.01798](#), 2023.
- [26] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, “Large language models are human-level prompt engineers,” [arXiv preprint arXiv:2211.01910](#), 2022.
- [27] V.-T. Do, V.-K. Hoang, D.-H. Nguyen, S. Sabahi, J. Yang, H. Hotta, M.-T. Nguyen, and H. Le, “Automatic prompt selection for large language models,” [arXiv preprint arXiv:2404.02717](#), 2024.
- [28] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, “Prompt engineering in large language models,” in [International conference on data intelligence and cognitive informatics](#). Springer, 2023, pp. 387–402.
- [29] A. Lu, H. Zhang, Y. Zhang, X. Wang, and D. Yang, “Bounding the capabilities of large language models in open text generation with prompt constraints,” [arXiv preprint arXiv:2302.09185](#), 2023.
- [30] Y. Abdullin, D. Molla-Aliod, B. Ofoghi, J. Yearwood, and Q. Li, “Synthetic dialogue dataset generation using llm agents,” [arXiv preprint arXiv:2401.17461](#), 2024.
- [31] Z. He, T. Liang, W. Jiao, Z. Zhang, Y. Yang, R. Wang, Z. Tu, S. Shi, and X. Wang, “Exploring human-like translation strategy with large language models,” [Transactions of the Association for Computational Linguistics](#), vol. 12, pp. 229–246, 2024.
- [32] L. Wang, C. Lyu, T. Ji, Z. Zhang, D. Yu, S. Shi, and Z. Tu, “Document-level machine translation with large language models,” [arXiv preprint arXiv:2304.02210](#), 2023.
- [33] H. Jin, Y. Zhang, D. Meng, J. Wang, and J. Tan, “A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods,” [arXiv preprint arXiv:2403.02901](#), 2024.
- [34] J. Kim, J. Nam, S. Mo, J. Park, S.-W. Lee, M. Seo, J.-W. Ha, and J. Shin, “Sure: Improving open-domain question answering of llms via summarized retrieval,” in [The Twelfth International Conference on Learning Representations](#), 2023.
- [35] Z. Li, S. Fan, Y. Gu, X. Li, Z. Duan, B. Dong, N. Liu, and J. Wang, “Flexkbqa: A flexible llm-powered framework for few-shot knowledge base question answering,” 2024.

- [36] J. Wu, S. Yang, R. Zhan, Y. Yuan, D. F. Wong, and L. S. Chao, “A survey on llm-generated text detection: Necessity, methods, and future directions,” arXiv preprint arXiv:2310.14724, 2023.
- [37] J. Ahn, R. Verma, R. Lou, D. Liu, R. Zhang, and W. Yin, “Large language models for mathematical reasoning: Progresses and challenges,” arXiv preprint arXiv:2402.00157, 2024.
- [38] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and C. Yuan, “React: Synergizing reasoning and acting in language models,” ICLR arXiv:2210.03629, 2023.
- [39] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, “Toolformer: Language models can teach themselves to use tools,” Advances in Neural Information Processing Systems, vol. 36, pp. 68 539–68 551, 2023.
- [40] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” Advances in neural information processing systems, vol. 36, pp. 11 809–11 822, 2023.
- [41] H. S. Zheng, S. Mishra, X. Chen, H.-T. Cheng, E. H. Chi, Q. V. Le, and D. Zhou, “Take a step back: evoking reasoning via abstraction in large language models,” arXiv preprint arXiv:2310.06117, 2023.
- [42] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R. L. Bras, Y. Choi, and H. Hajishirzi, “Generated knowledge prompting for commonsense reasoning,” 2022.
- [43] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” Advances in Neural Information Processing Systems, vol. 36, 2024.
- [44] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegrefe, U. Alon, N. Dziri, S. Prabhunoye, Y. Yang et al., “Self-refine: Iterative refinement with self-feedback,” Advances in Neural Information Processing Systems, vol. 36, 2024.
- [45] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, “Least-to-most prompting enables complex reasoning in large language models,” 2023.
- [46] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou et al., “Chain-of-thought prompting elicits reasoning in large language models,” Advances in neural information processing systems, vol. 35, pp. 24 824–24 837, 2022.
- [47] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig, “Pal: Program-aided language models,” 2023.

- [48] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” Advances in neural information processing systems, vol. 35, pp. 22 199–22 213, 2022.
- [49] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, “Training verifiers to solve math word problems,” 2021.
- [50] A. Patel, S. Bhattamishra, and N. Goyal, “Are nlp models really able to solve simple math word problems?” 2021.
- [51] S.-Y. Miao, C.-C. Liang, and K.-Y. Su, “A diverse corpus for evaluating and developing english math word problem solvers,” arXiv preprint arXiv:2106.15772, 2021.
- [52] R. Koncel-Kedziorski, S. Roy, A. Amini, N. Kushman, and H. Hajishirzi, “Mawps: A math word problem repository,” in Proceedings of the 2016 conference of the north american chapter of the association for computational linguistics: human language technologies, 2016, pp. 1152–1157.
- [53] M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou *et al.*, “Challenging big-bench tasks and whether chain-of-thought can solve them,” arXiv preprint arXiv:2210.09261, 2022.
- [54] W. Chen, X. Ma, X. Wang, and W. W. Cohen, “Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks,” 2023.
- [55] A. Zhou, K. Wang, Z. Lu, W. Shi, S. Luo, Z. Qin, S. Lu, A. Jia, L. Song, M. Zhan, and H. Li, “Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification,” 2023.
- [56] X. Zhao, Y. Xie, K. Kawaguchi, J. He, and Q. Xie, “Automatic model selection with large language models for reasoning,” arXiv preprint arXiv:2305.14333, 2023.
- [57] T. Liu, Q. Guo, Y. Yang, X. Hu, Y. Zhang, X. Qiu, and Z. Zhang, “Plan, verify and switch: Integrated reasoning with diverse x-of-thoughts,” 2023.
- [58] W. Zhang, Y. Shen, L. Wu, Q. Peng, J. Wang, Y. Zhuang, and W. Lu, “Self-contrast: Better reflection through inconsistent solving perspectives,” arXiv preprint arXiv:2401.02009, 2024.
- [59] S. Chen, B. Li, and D. Niu, “Boosting of thoughts: Trial-and-error problem solving with large language models,” 2024.
- [60] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, “A survey on evaluation of large language models,” ACM Transactions on Intelligent Systems and Technology, 2023.

- [61] Z. Wang, F. Yang, P. Zhao, L. Wang, J. Zhang, M. Garg, Q. Lin, and D. Zhang, “Empower large language model to perform better on industrial domain-specific question answering,” arXiv preprint arXiv:2305.11541, 2023.
- [62] Y. Ge, W. Hua, K. Mei, J. Tan, S. Xu, Z. Li, Y. Zhang et al., “Openagi: When llm meets domain experts,” Advances in Neural Information Processing Systems, vol. 36, 2024.
- [63] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, “Code generation using machine learning: A systematic review,” Ieee Access, vol. 10, pp. 82 434–82 455, 2022.
- [64] J. Shin and J. Nam, “A survey of automatic code generation from natural language,” Journal of Information Processing Systems, vol. 17, no. 3, pp. 537–555, 2021.
- [65] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, 2020, pp. 1433–1443.
- [66] B. Yetiştirilen, I. Özsoy, M. Ayerdem, and E. Tüzün, “Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt,” arXiv preprint arXiv:2304.10778, 2023.
- [67] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan, “Improving chatgpt prompt for code generation,” arXiv preprint arXiv:2305.08360, 2023.
- [68] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, “Decomposed prompting: A modular approach for solving complex tasks,” arXiv preprint arXiv:2210.02406, 2022.
- [69] Z. Xu, Z. Liu, B. Chen, Y. Tang, J. Wang, K. Zhou, X. Hu, and A. Shrivastava, “Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt,” arXiv preprint arXiv:2305.11186, 2023.
- [70] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, “Self-consistency improves chain of thought reasoning in language models,” 2023.
- [71] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt, “Measuring mathematical problem solving with the math dataset,” 2021.
- [72] C. Zheng, Z. Liu, E. Xie, Z. Li, and Y. Li, “Progressive-hint prompting improves reasoning in large language models,” arXiv preprint arXiv:2304.09797, 2023.
- [73] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin et al., “Code llama: Open foundation models for code,” arXiv preprint arXiv:2308.12950, 2023.

- [74] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., “Evaluating large language models trained on code,” arXiv preprint arXiv:2107.03374, 2021.
- [75] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, “InCoder: A generative model for code infilling and synthesis,” arXiv preprint arXiv:2204.05999, 2022.
- [76] Z. Yang, S. Chen, C. Gao, Z. Li, G. Li, and R. Lv, “Deep learning based code generation methods: A literature review,” arXiv preprint arXiv:2303.01056, 2023.
- [77] J. Zhao, Y. Song, J. Wang, and I. G. Harris, “Gap-gen: Guided automatic python code generation,” arXiv preprint arXiv:2201.08810, 2022.
- [78] W. Zheng, S. Sharan, A. K. Jaiswal, K. Wang, Y. Xi, D. Xu, and Z. Wang, “Outline, then details: Syntactically guided coarse-to-fine code generation,” in International Conference on Machine Learning. PMLR, 2023, pp. 42 403–42 419.
- [79] B. T. Willard and R. Louf, “Efficient guided generation for large language models,” arXiv e-prints, pp. arXiv–2307, 2023.
- [80] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, “Improving llm code generation with grammar augmentation,” arXiv preprint arXiv:2403.01632, 2024.
- [81] J. Shin, H. Hemmati, M. Wei, and S. Wang, “Assessing evaluation metrics for neural test oracle generation,” IEEE Transactions on Software Engineering, 2024.
- [82] J. Shin, S. Hashtroudi, H. Hemmati, and S. Wang, “Domain adaptation for code model-based unit test case generation,” in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1211–1222. [Online]. Available: <https://doi.org/10.1145/3650212.3680354>
- [83] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, “A deep dive into large language models for automated bug localization and repair,” Proceedings of the ACM on Software Engineering, vol. 1, no. FSE, pp. 1471–1493, 2024.
- [84] Y. Wei, C. S. Xia, and L. Zhang, “Copiloting the copilots: Fusing large language models with completion engines for automated program repair,” in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 172–184.
- [85] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, “Leveraging large language models to improve rest api testing,” in Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 37–41.

- [86] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” arXiv preprint arXiv:2406.00515, 2024.
- [87] J. Shin, M. Wei, J. Wang, L. Shi, and S. Wang, “The good, the bad, and the missing: Neural code generation for machine learning tasks,” ACM Transactions on Software Engineering and Methodology, vol. 33, no. 2, pp. 1–24, 2023.
- [88] C.-Y. Wang, A. DaghighFarsoodeh, and H. V. Pham, “Task-oriented prompt enhancement via script generation,” arXiv preprint arXiv:2409.16418, 2024.
- [89] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, “Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks,” arXiv preprint arXiv:2310.10508, 2023.
- [90] S. Feng and C. Chen, “Prompting is all you need: Automated android bug replay with large language models,” in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
- [91] T. B. Brown, “Language models are few-shot learners,” arXiv preprint arXiv:2005.14165, 2020.
- [92] L. Yang, Z. Yu, T. Zhang, S. Cao, M. Xu, W. Zhang, J. E. Gonzalez, and B. Cui, “Buffer of thoughts: Thought-augmented reasoning with large language models,” arXiv preprint arXiv:2406.04271, 2024.
- [93] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” arXiv preprint arXiv:2304.05128, 2023.
- [94] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 2450–2462.
- [95] T. McDonald, A. Colosimo, Y. Li, and A. Emami, “Can we afford the perfect prompt? balancing cost and accuracy with the economical prompting index,” arXiv preprint arXiv:2412.01690, 2024.
- [96] D. Budagam, A. Kumar, M. Khoshnoodi, S. KJ, V. Jain, and A. Chadha, “Hierarchical prompting taxonomy: A universal evaluation framework for large language models aligned with human cognitive principles,” To appear, 2025.
- [97] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, “Supervised contrastive learning,” Advances in neural information processing systems, vol. 33, pp. 18 661–18 673, 2020.
- [98] C. Tony, N. E. D. Ferreyra, M. Mutas, S. Dhiff, and R. Scandariato, “Prompting techniques for secure code generation: A systematic investigation,” arXiv preprint arXiv:2407.07064, 2024.

- [99] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” arXiv preprint arXiv:2002.08155, 2020.
- [100] E. Hoffer and N. Ailon, “Deep metric learning using triplet network,” in Similarity-based pattern recognition: third international workshop, SIMBAD 2015, Copenhagen, Denmark, October 12-14, 2015. Proceedings 3. Springer, 2015, pp. 84–92.
- [101] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, “Clear: contrastive learning for api recommendation,” in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 376–387.
- [102] A. v. d. Oord, Y. Li, and O. Vinyals, “Representation learning with contrastive predictive coding,” arXiv preprint arXiv:1807.03748, 2018.
- [103] H. Bredin, “Tristounet: triplet loss for speaker turn embedding,” in 2017 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE, 2017, pp. 5430–5434.
- [104] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, “A prompt pattern catalog to enhance prompt engineering with chatgpt,” arXiv preprint arXiv:2302.11382, 2023.
- [105] X. Jiang, Y. Dong, L. Wang, F. Zheng, Q. Shang, G. Li, Z. Jin, and W. Jiao, “Self-planning code generation with large language models,” ACM Transactions on Software Engineering and Methodology, 2023.
- [106] A. Liu, M. Diab, and D. Fried, “Evaluating large language model biases in persona-steered generation,” arXiv preprint arXiv:2405.20253, 2024.
- [107] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le *et al.*, “Least-to-most prompting enables complex reasoning in large language models,” arXiv preprint arXiv:2205.10625, 2022.
- [108] H. Zhang, X. Zhang, and M. Gu, “Predicting defective software components from code complexity measures,” in 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007). IEEE, 2007, pp. 93–96.
- [109] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, “Comparing design and code metrics for software quality prediction,” in Proceedings of the 4th international workshop on Predictor models in software engineering, 2008, pp. 11–18.
- [110] Y. Shin and L. Williams, “An empirical model to predict security vulnerabilities using code complexity metrics,” in Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 315–317.

- [111] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, “Cyclomatic complexity,” IEEE software, vol. 33, no. 6, pp. 27–29, 2016.
- [112] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, “Software complexity analysis using halstead metrics,” in 2017 International Conference on Trends in Electronics and Informatics (ICEI). IEEE, 2017, pp. 1109–1113.
- [113] G. A. Campbell, “Cognitive complexity-a new way of measuring understandability,” SonarSource SA, p. 10, 2018.
- [114] K. D. Welker, “The software maintainability index revisited,” CrossTalk, vol. 14, pp. 18–21, 2001.
- [115] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” Advances in Neural Information Processing Systems, vol. 36, pp. 21 558–21 572, 2023.
- [116] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le et al., “Program synthesis with large language models,” arXiv preprint arXiv:2108.07732, 2021.
- [117] E. M. Voorhees et al., “The trec-8 question answering track report.” in Trec, vol. 99, 1999, pp. 77–82.
- [118] D. R. Radev, H. Qi, H. Wu, and W. Fan, “Evaluating web-based question answering systems.” in LREC. Citeseer, 2002.
- [119] K. Järvelin and J. Kekäläinen, “Cumulated gain-based evaluation of ir techniques,” ACM Transactions on Information Systems (TOIS), vol. 20, no. 4, pp. 422–446, 2002.
- [120] M. Jeon, S.-y. Baik, J. Hahn, Y.-S. Han, and S.-K. Ko, “Deep learning-based source code complexity prediction,” To appear, 2023.
- [121] M. L. Siddiq, A. Samee, S. R. Azgor, M. A. Haider, S. I. Sawraz, and J. C. Santos, “Zero-shot prompting for code complexity prediction using github copilot,” in 2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE). IEEE, 2023, pp. 56–59.
- [122] T. Shin, Y. Razeghi, R. L. Logan IV, E. Wallace, and S. Singh, “Autoprompt: Eliciting knowledge from language models with automatically generated prompts,” arXiv preprint arXiv:2010.15980, 2020.
- [123] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” arXiv preprint arXiv:2104.08691, 2021.
- [124] X. L. Li and P. Liang, “Prefix-tuning: Optimizing continuous prompts for generation,” arXiv preprint arXiv:2101.00190, 2021.

- [125] H. Sun, “Offline prompt evaluation and optimization with inverse reinforcement learning,” [arXiv preprint arXiv:2309.06553](https://arxiv.org/abs/2309.06553), 2023.