Bridging Code Semantic and LLMs: Semantic Chain-of-Thought Prompting for Code Generation

Yingwei Ma[1], Yue Yu[*,1], Shanshan Li[*,1], Yu Jiang[2], Yong Guo[1],
Yuanliang Zhang[1], YUTAO XIE[3], Xiangke Liao[1]
[1]Nation University of Defense Technology
[2]Tsinghua University
[3]International Digital Economy Academy
Corresponding email: {yuyue, shanshanli}@nudt.edu.cn

ABSTRACT

Large language models (LLMs) have showcased remarkable prowess in code generation. However, automated code generation is still challenging since it requires a high-level semantic mapping between natural language requirements and codes. Most existing LLMs-based approaches for code generation rely on decoder-only causal language models often treate codes merely as plain text tokens *i.e.*, feeding the requirements as a prompt input, and outputing code as flat sequence of tokens, potentially missing the rich semantic features inherent in source code. To bridge this gap, this paper proposes the "Semantic Chain-of-Thought" approach to intruduce semantic information of code, named SeCoT. Our motivation is that the semantic information of the source code (*e.g.*, data flow and control flow) describes more precise program execution behavior, intention and function. By guiding LLM consider and integrate semantic information, we can achieve a more granular understanding and representation of code, enhancing code generation accuracy. Meanwhile, while traditional techniques leveraging such semantic information require complex static or dynamic code analysis to obtain features such as data flow and control flow, SeCoT demonstrates that this process can be fully automated via the intrinsic capabilities of LLMs (*i.e.*, in-context learning), while being generalizable and applicable to challenging domains. While SeCoT can be applied with different LLMs, this paper focuses on the powerful GPT-style models: ChatGPT(close-source model) and WizardCoder(open-source model). The experimental study on three popular DL benchmarks (*i.e.*, HumanEval, HumanEval-ET and MBPP) shows that SeCoT can achieves state-of-the-art performance, greatly improving the potential for large models and code generation.

ACM Reference Format:
Yingwei Ma[1], Yue Yu[*,1], Shanshan Li[*,1], Yu Jiang[2], Yong Guo[1],, Yuanliang Zhang[1], YUTAO XIE[3], Xiangke Liao[1], [1]Nation University of Defense Technology, [2]Tsinghua University, [3]International Digital Economy Academy, Corresponding email: {yuyue, shanshanli}@nudt.edu.cn . 2024. Bridging Code Semantic and LLMs: Semantic Chain-of-Thought Prompting for Code Generation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The core objective of code generation is to automatically generate corresponding program code based on the user's natural language requirements. In recent years, large language models (LLMs) have achieved significant breakthroughs in this field. Whether it is commercial closed-source API models *e.g.*, ChatGPT [23] and Cluade [1], or open-source models *e.g.*, WizardCoder [21], StarCoder [17] and Code Llama [29], they all demonstrate cutting-edge performance. However, as these models grow in size and complexity, a pivotal challenge emerges: imbuing them with the ability to code reason accurately and effectively. Faced with programming needs, human programmers often engage in in-depth reasoning and deduction to solve complex programming problems. For LLMs, this reasoning capability is crucial to improve their practical application value in code generation tasks. Although LLMs have shown strong potential, they still face challenges in conducting deep and detailed reasoning [36], which has prompted the research community to explore innovative strategies on how to enhance their reasoning capabilities.

Instead of fine-tuning, large language models take as input a prompt containing multiple examples (*e.g.*, <requirement, code> pairs) as well as a new requirement [9, 15]. LLM learns how to generate code from these examples and generates corresponding programs for new requirements. To improve code reasoning ability, existing research in this field mainly focuses on designing diverse thought-eliciting prompting strategies to guide and channel their reasoning process. Chain-of-Thought (CoT) prompting [30] is state-of-the-art (SOTA) prompting technology. CoT Prompting first requires LLM to generate CoT and then output the code. CoTs are several intermediate natural language reasoning steps that describe how to write code step by step. The Structured Chain-of-Thought (SCoT) [15] improves CoT by guiding LLM to generate a program structure similar to pseudocode and then output the code. Figure 1 (a) and (b) show a CoT and a SCoT on code generation. However, existing decoder-only causal language modeling (CLM) architectures typically treate codes merely as plain text tokens *i.e.*, feeding the requirements as a prompt input, and outputing CoT and code as flat sequence of tokens, potentially missing the rich semantic features inherent in source code.

```
1. Use the zip function with the unpacking
operator * to transpose the given list of lists.
2. Convert the zipped object into a list.
3. Iterate over the zipped list and convert
each tuple into a list.
4. Return the resulting list of lists.
```

(a) Chain-of-Thought

```
Input: lst: list of lists
Output: merged_list: a list of lists
1: Initialize two empty lists: first_elements and
last_elements
2: for each sublist in lst do:
3:     append the first element of sublist to
first_elements
4:     append the last element of sublist to
last_elements
5: return a list containing first_elements and
last_elements as its two elements.
```

(b) Structured Chain-of-Thought

Figure 1: The comparison of a Chain-of-Thoughts (CoT) and a Structured Chain-of-Thought (SCoT).

Our work. To fill this gap, this paper proposes Semantic Chain-of-Thought technique (named SeCoT), the first approach to guide LLMs to synthesize semantic information of source code for effective code generation. SeCoT is built on the well-known premise that the semantic information of the source code (*e.g.*, data flow and control flow) describes more precise program execution behavior, intention and function [20, 22, 40]. By guiding LLM consider and integrate semantic information, we can achieve a more granular understanding and representation of code, enhancing code generation accuracy. For example, data flow describes the transfer, processing, and storage of data in a program, while control flow describes the execution sequence of control statements in a program, such as if statements, for loops, and $while$ loops. In short, data flow determines how a program operates on data, while control flow determines the order in which these operations are performed. We propose an innovative idea to enable large language models (LLMs) to automatically generate key semantic information such as data flow and control flow. This information will be constructed by interpreting the semantics of the program into a coherent series of intermediate reasoning steps. The core insight is that LLMs are pre-trained on billions of open source code snippet covering valid programs in various domains and rich software analysis information (eg., debugging output, stack traces, and Error analysis reports). Therefore, compared with traditional techniques on leveraging such semantic information require complex static or dynamic code analysis to obtain features such as data flow and control flow, our proposed SeCoT method has the potential to directly exploit the generative capabilities of LLMs to generate these semantic information. However, despite the powerful generation capabilities of LLMs, existing code generation methods often ignore these in-depth semantic information, resulting in insufficient accuracy of code generation. To give full play to the reasoning capabilities of LLM in code generation, we believe that it is necessary to design and introduce a specialized semantic CoT method.

To implement SeCoT, we first build a dataset containing semantic information such as data flow and control flow by mining code requirements and implementations from open source repositories. SeCoT is built on this data set and uses a in-context learning strategy. We provide LLMs with some examples of <requirements, code, semantic information> pairs (few-shot learning [5]) to generate new code snippets and then perform semantic analysis. From this learning strategy, SeCoT can prime the LLMs to generate more effective programs and semantic information by capturing code semantic ingredients within eitherthe local context examples dataset.

We apply SeCoT hints to two popular LLMs (*e.g.*, ChatGPT [23] - a closed-source API model and WizardCoder [21] - an open-source model) and evaluate them on three representative benchmarks (*i.e.*, HumanEval [6], MBPP [2], and HumanEval-ET [11]) . We use unit tests to measure the correctness of the generated program and report $Pass@k$ ($k \in \{1, 3, 5\}$). Based on the experimental results, we obtained three findings. (1) SeCoT hints significantly improve the accuracy of LLM in code generation. Compared with direct generation procedures, SCoT outperformed it by up to 11.72% in HumanEval, 5.71% in MBPP, and 11.97% in HumanEval-ET. (2) SeCoT prompts are effective for different LLMs. The improvement is as high as 11.97% on ChatGPT and 11.87% on WizardCoder. (3) We explore the robustness of SCoT cues. The results show that SCoT prompts are not dependent on specific examples or writing styles.

The main contributions of this paper are summarized as follows:

- **Dimension** This paper opens up a new direction in introducing semantic information of source code into LLMs for effective code generation. This paper shows for the first time that LLM can easily and effectively generate semantic information of code through prompted and in-context learning to achieve a more fine-grained code understanding and representation, enhancing the accuracy of code generation.
- **Technique** This paper proposes Semantic Chain-of-Thought Technology (SeCoT), which prompts LLMs to understand the <requirements, code, semantic analysis> pairs in the example, and then generates correct programs for new requirements and performs semantic analysis step by step. Although SeCoT can be applied to different LLMs, we build our strategy based on the state-of-the-art closed-source and open-source models ChatGPT and WizardCoder. We demonstrate that SeCoT is fully automated and generalizable to improve the code reasoning capabilities of LLMs.
- **Extensive Study** We conduct extensive experiments on three benchmarks. Qualitative and quantitative experiments show that SeCoT prompting significantly outperforms SOTA baselines. We also discuss the contributions of different semantic information and the robustness of SeCoT prompting.

Data Availability. We open source our replication package [xx], including the datasets and the source code of SeCoT prompting,

to facilitate other researchers and practitioners to repeat our work and verify their studies.

2 A MOTIVATED EXAMPLE

In this section, we elucidate our motivation using several practical cases.

Code Semantic Information → Guided Code Understanding and Generation. Figure 2 shows a requirement from a real-world benchmark, along with three intermediate reasoning steps and code implementations generated by ChatGPT. We select three in-context learning strategies [19] to guide ChatGPT [23] to generate intermediate reasoning steps and codes. In Figure 2, the objective of the requirement is to find the deepest nesting level of parentheses in a given string. For example, '(()())' has maximum two levels of nesting, while '((()))' has three. For a given input, if generated code can pass all test cases, it means that the requirements are met. For example, a test case in the example is: *assert parse_nested_parens('(()()) ((())) () ((())()())') == [2, 3, 1, 3]*.

CoT [30] and SCoT [15] guide large language models (LLMs) to generate natural language reasoning steps and pseudo-code-like structured reasoning steps respectively, thereby further guiding the implementation of the code. We found that although the intermediate reasoning steps of both methods correctly parsed the key information in the requirements, *i.e.*, extracted the deepest nesting level of parentheses (highlighted with a green background in the figure 2), errors occurred in the code implementation, without updating max_level information. This shows that even though the reasoning steps may be correct, LLMs still face challenges in code implementation.

Therefore, we proposed SeCoT. By guiding LLM to consider and integrate semantic information of source code, we can achieve a more fine-grained understanding and representation of the code, thereby improving the accuracy of code generation. As shown in Figure 2, SeCoT not only correctly parses the key information in the requirements, but also implements the correct code, *i.e.*, correctly stores and updates the value of max_level during the loop. This instance demonstrates that our Semantic Chain-of-Thought technology can assist large language models in enhancing the correctness of code implementation.

3 APPROACH

In this section, we propose a Semantic Chain-of-Thought approach (SeCoT) to intruduce semantic information of source code into LLMs. A SeCoT is constructed by interpreting the program semantics into a coherent series of intermediate reasoning steps. SeCoT prompting asks LLMs fist to learn in-context examples *i.e.*, <requirement, code, semantic information> pairs and then output the final code and perform semantic analysis step by step, can achieve a more granular understanding and representation of code, enhancing code generation. In the subsections, we first describe the design of our SeCoT and further show the details of SeCoT prompting.

3.1 Semantic Chain-of-Thought

As large language models (LLMs) continue to grow in size and complexity, empowering them to perform accurate and effective code reasoning has become a pivotal challenge in contemporary research. Chain-of-Thought (CoT) technology is a key strategy to enhance the model's reasoning capabilities. CoT has shown impressive results in multiple areas, such as common sense reasoning, mathematical problem solving, and scientific question answering. The core idea of CoT is to derive the final answer through a series of intermediate natural language reasoning steps, thereby guiding the model to describe how to solve the problem step by step. Figure 1(a) shows an example of our application of CoT in code generation tasks. Structured Chain-of-Thought (SCoT) adopts a reasoning step closer to pseudo-code, and uses the program execution sequence described in natural language to guide the model to generate code that is more consistent with the program structure. Figure 1(b) shows the SCoT in the code generation task. However, one limitation is that these CoT methods all ignore the semantic information inherent in the program, which describes more precise program execution behavior, intention, and functionality.

A program is a collection of instructions with rich semantic information. These instructions are more than simple commands; they carry rich semantic information, describing the movement and transformation of data in the program as well as the order and decisions of program execution. For example, given a requirement - *read text from a given file*, we can interpret it in terms of data flow and control flow.

Control flow:

- **Decision Making:** Initially, the program decides whether to read the file. This is a conditional decision based on the existence of the file.
- **Branch Execution:** If the file exists, the program will follow one execution path (reading the file). If the file doesn't exist, it will follow another execution path (raising an error).
- **Error Handling:** In certain scenarios, even if the file exists, the reading operation might fail (for instance, due to permission issues). The program might have another control flow branch to handle these exceptions.

Data flow:

- **File Path:** The program needs to know which file to read, typically specified by the file path or name.
- **File Content:** Once the file is opened, its content is read into the program's memory. This represents the data flow from the storage medium to the program's memory.
- **Error Information:** If an error occurs, the program might generate an error message. This represents another form of data flow within the program, as the error message might be passed to logging systems, user interfaces, or other error-handling mechanisms.

From this perspective, the semantics of a program involve how data (data flow) is moved and processed based on decisions in the control flow. These two aspects together determine the overall behavior and functionality of the program.

Therefore, to achieve a more fine-grained understanding and representation of program code by LLMs, **we propose the Semantic Chain of Thought (SeCoT)**. Figure 3 shows some examples of SeCoT. Compared with CoT and SCoT, our SeCoT explicitly introduces the semantic information of the program. The specific description is as follows.

Figure 2: SeCoT prompting example.

3.1.1 Control flow. In programming, control flow mainly consists of the following basic structures:

- **Sequence control structure.** This is the most basic structure, the code is executed in the order they are in the program.
- **Loop control structure.** These include statements such as *if*, *if_else*, and *switch_case*, which allow the program to choose different execution paths based on certain conditions.
- **Conditional control structure.** These include loop structures such as *for*, *while*, and *do_while*, which allow a program to repeatedly execute a section of code.
- **Jump control structure.** This includes *break*, *continue*, *return* and other statements. These statements allow the program to skip certain parts or exit a loop or function early.

Prior research [4, 15] has demonstrated that all programs, whether simple or complex, can be constructed using these fundamental structures. Therefore, SeCoT can be generalized to different real-world programs and allows LLMs to design more complex SeCoT for some difficult requirements. As shown in Figure 3, SeCoT flexibly utilizes various control structures to construct control flow information.

3.1.2 Data flow. Data flow describes how data moves and transforms within a program. It focuses on how data is transferred from one part to another and how data is processed and stored. Here are some key components of data flow:

- **Data Storage and Manipulation.** This involves variables, constants, and data structures like arrays, lists, and stacks, which provide ways to organize, store, and manipulate data.
- **Input and Output.** Data can be input from external sources into the program and output from the program to external destinations, including user input, file operations, and network communications.
- **Data Transformation and Processing.** Data often undergoes transformation, processing, or computation within the program, such as converting from one type to another or transitioning from one data structure to another.
- **External Interactions.** This encompasses interactions with databases, network communications, and interactions with other software or services through APIs.

These points briefly outline the main ways in which data flows and is processed within a program. Understanding data flow and control flow is key to understanding how a program works. Control flow describes the execution sequence of a program, while data flow describes how data moves and transforms within a program.

```
1. Control Flow:
   - Start -> Call merge(lst)
   - Use the `zip` function combined with the unpacking
     operator `*` to merge the first elements from each
     list together and the last elements from each list
     together                                  Sequence control
   - Convert the merged elements into lists    structure
   - Return the merged lists -> End   Jump control structure
2. Data Flow:
   - The input list of lists `lst` is taken as input
   - The lists within `lst` are unpacked using the `*`
     operator                                  Data operation
   - The `zip` function merges the first elements from
     each list together and the last elements from each
     list together
   - The merged elements are then converted back into
     lists
   - The final merged lists are returned
3. Finally: This combined representation provides a clear,
   structured context for understanding the function's
   logic and behavior.                          Summary
```

(a) Example 1

```
1. Control Flow:
   - Start -> Call prime_num(num)
   - Check if the input integer `num` is greater than
     or equal to 1
     - If true, iterate over each integer `i` from 2
       to num/2                          Loop control structure
       - For each `i`, check if `num` is divisible by `i`
       - If `num` is divisible by any `i`, return False
         indicating it's not a prime number
       - If `num` is not divisible by any `i`, return True
         indicating it's a prime number   Conditional control
                                          structure
   -If the input integer `num` is less than 1, return False
    indicating it's not a prime number -> End
2. Data Flow:
   - The input integer `num` determines the range of
     iteration and divisibility checks
   - The divisibility of `num` by integers in the range
     determines the return value (True/False)  Data operation
3. Finally: This combined representation provides a clear,
   structured context for understanding the function's
   logic and behavior.                          Summary
```

(b) Example 2

Figure 3: Examples of SeCoT in code generation, described how a program operates on data (data flow) and how a program determines the order in which operations are executed (control flow).

3.2 SeCoT prompting

Based on SeCoT, we propose a new code generation prompting technology called SeCoT prompting. SeCoT directly uses examples of <user requirement, code, SeCoT> to prompt pre-trained LLMs,

```
Below is an instruction that describes a task. Write a
response that appropriately completes the request.

### Instruction:
Complete the requirement code based on examples.
```

```
Example0:
def merge(lst):
    """
    Write a python function to merge the first and last
    elements separately in a list of lists.
    >>> merge([['x', 'y'], ['a', 'b'], ['m', 'n']])
    [['x', 'a', 'm'], ['y', 'b', 'n']]
    """
    return [list(ele) for ele in list(zip(*lst))]
    """
    1. Control Flow:
       - Start -> Call merge(lst)
       - ...
       - Return the merged lists -> End
    2. Data Flow:
       - The input list of lists `lst` is taken as input
       - ...
       - The final merged lists are returned
    3. Finally: This combined representation provides a
    clear, structured context for understanding the
    function's logic and behavior.
    """
```

```
.....
```

```
.....
```

```
Requirement:
def sum_Range_list(nums, m, n):
    """
    Write a python function to calculate the sum of the
    numbers in a list between the indices of a specified
    range.
    >>> sum_Range_list([1,0,1,2,5,6],4,5)
    11
    """
   # Please check the requirement and write a code based
   on it. Then you should output the data flow and control
   flow information of the code step by step.
```

Figure 4: SeCoT prompting example.

let the model generate final code for new user requirements and perform semantic analysis, which covers the analysis results of data flow and control flow of the final code.

We implement SeCoT prompting based on in-context learning. In-context learning directly uses pre-trained LLMs without modifying any parameters of the model. As shown in Figure 4, we show an example of SeCoT prompting. First of all, in the prompt, the instruction stipulates the overall task of LLMs [14, 15], that is, "Complete the requirement code based on examples". Then, we give several

demonstration examples of <requirement, code, semantic information> format as in-context learning examples. The first two parts of semantic information include data flow and control flow analysis results. In addition, in order for the model to understand the role of semantic information, we give a concise summary in the third part to remind the model to use semantic information to better understand program logic and behavior. The purpose of these examples is two-fold. First, they aim to help LLMs learn the mapping format between input and output (*i.e.*,, output the generated code and semantic information according to the input requirements). Second, they enable the model to learn how to understand and generate code at a more fine-grained semantic level by observing the mapping between source code and semantic information in examples.

Let M be the LLM that outputs the probability of generating a sequence. Let $E_k = < r_1, c_1, se_1 >, ..., < r_k, c_k, se_k >$ be the concatenation of K examples consisting of tuples of example user requirement r_i, generated code c_i, and SeCoT se_i. Let r_{new} be the new user requirement. The probability of generating the output code c_{new} using in-context learning can be formalized as this conditional probability:

$$M(c_{new}, se_{new}) = M(c_{new}|E_K, r_{new}) \times M(se_{new}|E_k, r_{new}, c_{new}) \tag{1}$$

3.3 Implementation Details

SeCoT prompting are a prompting technology for code generation. For the construction of semantic chain-of-thought data, we select a subset of <requirement, code> pairs from real-world benchmarks (*i.e.*, training data) as example seeds. Subsequently, we manually craft the SeCoT for each seed and obtain the sample <requirements, code, SeCoT> triples, which are used to make the prompt in Figure 4. The prompt contains three randomly sampled examples by default, and the maximum generated length is 512. We follow Code Llama [29] and use kernel sampling with p=0.95 and temperature 0.8 for generation. Examples and prompt templates are available in our open-source repository.

4 STUDY DESIGN

To evaluate the efficacy of SeCoT prompting, we embark on an extensive study addressing four pivotal research questions. In this section, we delineate the details of our investigation, encompassing the datasets employed, evaluation metrics, comparative baselines, and implementation details. Our experiments are structured to probe the following research questions:

RQ1: How does SeCoT prompting compared against existing methods? This RQ aims to ascertain whether SeCoT prompting offers superior accuracy in code generation compared to existing baseline approaches. We deploy four prevalent prompting strategies alongside SeCoT prompting across two LLMs. Subsequently, we employ unit tests to measure the correctness of programs generated by these diverse methods and present our findings in terms of Pass@k metrics.

RQ2: Is SeCoT prompting effective in actual engineering project development? This RQ aims to evaluate whether SeCoT prompts are useful in assisting actual project development. We used baseline ChatGPT to test on Redis, which is the open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker. We randomly select functions in Redis for replacement, and let LLMs generate the replaced code given the context, and use Redis' official test suite for testing, which requires LLMs to have better context understanding capabilities. We then use the Pass1 test pass rate to verify the effectiveness of our method.

RQ3: What contribution does different semantic information contribute in SeCoT prompting? As mentioned in Section 3.1, SeCoT prompting introduce data flow and control flow information to assist code generation. This RQ aims to analyze the contribution of different components. We choose LLMs as the base model. We then report the contributions of different components separately.

RQ4: Is SeCoT prompting robust to examples? Prompting techniques for large language models may be sensitive to in-context examples [43]. In this RQ, we evaluate the robustness of SeCoT prompt examples. Specifically, we follow existing work [15] in measuring the performance of SeCoT prompts using different example seeds. In addition, in order to lower the threshold for example annotation, we try to use examples generated by ChatGPT as in-context examples for evaluation.

4.1 Benchmarks

Following the previous work, we adopt two public mainstream benchmarks, MBPP and HumanEval, along with their extended test case versions, to evaluate the code generation ability of our SeCoT prompting and various baselines. The details of the benchmarks are described as follows.

- **HumanEval** comprises a collection of 164 meticulously handwritten programming challenges, introduced by OpenAI. Each challenge encompasses a function signature, a natural language (NL) description, a function body, and an array of unit tests, averaging 7.7 tests per challenge. For HumanEval, the function signature, NL description, and public test cases collectively serve as the input.
- **MBPP-sanitized**, a rigorously curated subset of MBPP (Mostly Basic Programming Problems), encompasses 427 crowd-sourced Python programming challenges. These challenges span a range of topics, from programming fundamentals to functionalities within the standard library and beyond. Each challenge is characterized by a natural language (NL) description, a corresponding code solution, and three automated test cases. For the MBPP-sanitized benchmark, only the NL description is provided as input.
- **HumanEval-ET** is a public expanded version of HumanEval, expanded iteration of HumanEval, enriched with over 100 supplementary test cases for each task. This augmented version incorporates edge test cases, thereby bolstering the robustness of code evaluation relative to the original benchmark.

Explanation: we also noticed a similar problem to SCoT [15], where existing LLMs are trained on a large number of code files from the open source community. Their training data may contain experimental benchmarks, leading to data leakage. But we believe

this does not affect the fairness of our experiment. In this paper, consistent with the existing approaches [9, 15], we select two specific models *i.e.*, ChatGPT and WizardCoder, and apply different prompting techniques to it. Therefore, the relative improvement between the reported baseline and our approach is credible.

4.2 Evaluation Metrics

In alignment with prior studies on code generation, we employ an execution-based metric, *Pass@k*, as our primary evaluation criterion. The *Pass@k* metric gauges the functional correctness of the generated code by executing the provided test cases. Specifically, for a given requirement, a code generation model is permitted to produce k distinct programs. The requirement is deemed satisfied if any of the generated programs successfully pass all test cases. We then calculate the percentage of requirements met out of the total, denoting this as *Pass@k*. Naturally, a higher *Pass@k* value indicates superior performance. In our experiments, we set k to 1, 3, and 5, based on our think that developers predominantly rely on the Top-5 outputs in real-world applications.

Consistent with the existing approach [15], we abstain from utilizing text-similarity-based metrics, such as BLEU [3]. These metrics, originally conceived for natural language generation tasks, fall short in accurately assessing the correctness of programs [**?**]. Consequently, we exclude these metrics from our experimental evaluation.

4.3 Comparison Baselines

We conduct various experiments, comparing multiple baselines to evaluate distinct aspects. Among these, four baselines–Direct prompting, Few-shot prompting, CoT prompting, and SCoT prompting, serve as basic baselines in all experiments, highlighting the efficacy of our approach.

- **Zero-shot prompting** [**?**] requires LLMs to generate code directly based on user requirements, that is, there are only user intentions in the prompt and no examples to refer to.
- **Few-shot prompting** [**?**] is designed to allow LLMs to learn the relationship between requirements and code from several randomly selected <user requirements, code> examples. Then given a new requirement, LLMs can generate a new program that implements the requirement.
- **CoT prompting** [30] is designed to generate a Chain-of-Thought for each question, and then guide LLMs to generate corresponding code step by step according to the Chain-of-Thought. Similar to Few-shot prompting, CoT prompting learns the relationship between requirements and CoT, code by studying several <user reqiurement, CoT, code> examples.
- **SCoT prompting** [15] improves CoT by guiding LLM to generate a program structure (*i.e.*, SCoT) similar to pseudocode and then output the code. Similar to Few-shot and CoT prompting, SCoT prompting learns the relationship between requirements and SCoT, code by studying several <user reqiurement, SCoT, code> examples.

To ensure the fairness of the experiment, consistent with methods such as Zero-shot prompting, Few-shot prompting and CoT prompting, for SCoT and SeCoT prompting, we uniformly use one step to generate intermediate reasoning steps and codes, as shown in formula 1. Although SCoT proves that the one-step pipeline and the two-step pipeline are equivalent in theory, considering the timeliness and convenience of users when using LLMs, prompts are usually not carried out in multiple steps.

4.4 Base Large Language Models

There are many code LLMs available. Our motivation is that existing LLMs can be divided into two categories: the closed-source API model represented by ChatGPT and the open-source model represented by Llama. For each category, we select a representative model as the base model.

(1) **We choose the most advanced closed source API model - ChatGPT [23] as the baseline model.** ChatGPT is the most advanced code generation LLM. ChatGPT is trained on large amounts of natural language text and code files. It is then trained with reinforcement learning and learns to follow human instructions. We use OpenAI's API to access ChatGPT, namely gpt-3.5-turbo-0301. Our method does not depend on specific LLMs and can be applied to different LLMs in a plug-and-play manner. In the future, we will explore its use on more powerful LLMs.

(2) **We choose the most advanced open source model - WizardCoder [21] as the baseline model.** WizardCoder is based on CodeLlama by introducing the Evol-Instruct method, which involves evolving existing code generation instruction data to generate more complex and diverse data sets. That is, based on the specific characteristics of the code domain, two evolutionary instructions are added: code debugging and code time-space complexity constraints. Improve CodeLlama [29] through supervised fine-tuning. In this article, we use the Huggingface [35] checkpoint (*i.e.*, WizardCoder-Python-13B-V1.0) open source by the WizardLM team to access WizardCoder.

5 RESULTS AND ANALYSIS

5.1 RQ1: How does SeCoT compared against existing methods?

Evaluation. In addressing our first research question, we apply SeCoT prompting and baselines to three benchmarks (*i.e.*, HumanEval, HumanEval-ET and MBPP-sanitized). To assess the functional correctness of the generated code, we employ unit tests as our evaluation metric. We apply baseline and SeCoT prompting to two LLMs (*i.e.*, ChatGPT and WizardCoder). Subsequently, we evaluate the performance of these various approaches on the aforementioned code generation benchmarks using the Pass@k metric, with k set to 1, 3, and 5.

Results and Analyses. The Pass@k results of different methods are summarized in Tables 1 and 2. **(1) SeCoT achieves the best performance among all baselines.** Tables 1 and 2 show that across the three benchmarks, SeCoT prompting achieve the best performance and can generate more correct programs than baselines. Obtaining code and fine-grained semantic information from intents can provide significant advantages in code generation compared to directly generating code from intents. In terms of Pass@1, SeCoT prompting outperforms it by 6.64% in HumanEval, 6.71% in HumanEval-ET, and 5.71% in MBPP. In terms of Pass@3, SeCoT prompting is better than 9.45% in HumanEval, 9.90% in HumanEval-ET, and 5.44% in MBPP. In terms of Pass@5, SeCoT prompting is

Table 1: The Pass@k (%) of SeCoT prompting and baselines on three code generation benchmarks. Bold values indicate the best performance.

ChatGPT	HumanEval			HumanEval-ET			MBPP		
Prompting Technique	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot	64.15	74.94	78.05	58.29	68.41	71.34	61.45	70.72	73.30
Few-shot	66.95	78.90	82.32	60.48	72.07	75.61	65.76	74.16	77.05
CoT	66.71	80.55	83.54	60.00	72.13	74.39	60.70	70.56	73.54
SCoT	65.61	77.62	81.10	60.12	71.40	75.00	63.65	72.65	74.47
SeCot_data	**68.90**	80.55	83.54	**63.41**	75.61	79.87	65.71	74.45	77.05
SeCot_control	68.05	80.73	84.76	61.70	73.66	78.05	65.76	74.45	77.05
SeCot	68.78	**82.68**	**86.59**	62.80	**75.79**	**79.88**	**65.85**	**74.57**	**77.28**

Table 2: The Pass@k (%) of SeCoT prompting and baselines on three code generation benchmarks. Bold values indicate the best performance.

WizardCoder	HumanEval			HumanEval-ET			MBPP		
Prompting Technique	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot	53.66	61.89	67.68	48.78	56.28	61.59	46.42	61.66	66.74
Few-shot	54.88	63.23	68.29	48.78	57.50	62.20	44.87	61.64	66.74
CoT	54.88	65.49	71.34	48.78	59.57	65.24	43.37	59.65	64.40
SCoT	53.04	61.71	68.90	46.95	54.94	62.20	42.58	59.74	65.10
SeCot_data	53.05	64.88	70.12	48.17	58.11	62.80	44.50	61.08	66.04
SeCot_control	54.27	65.98	71.34	48.78	59.45	64.63	45.95	62.53	64.40
SeCot	**55.48**	**67.74**	**75.61**	**49.39**	**61.10**	**68.90**	**46.89**	**62.93**	**67.92**

better than 11.72% in HumanEval, 11.97% in HumanEval-ET, and 5.43% in MBPP. The results show that SeCoT prompting can significantly improve the accuracy of LLMs in code generation and is more promising than existing prompting techniques. Furthermore, consistent with the existing research results [14], we find that SeCoT prompting shows more significant improvements on HumanEval and HumanEval-ET compared to MBPP-sanitized. We hypothesize that this is because in some MBPP-sanitized problems, the information provided about the intent is insufficient for the model to perform an efficient solution, making it almost impossible for to solve these problems even humans. **(2) SeCoT prompting can be applied to different open source or closed source LLMs.** In terms of Pass@1, SeCoT prompting further improve ChatGPT by 6.71% and WizardCoder by 3.39%. In terms of Pass@3, SeCoT prompting further improve ChatGPT by 9.90% and WizardCoder by 9.45%. In terms of Pass@5, SeCoT prompting further improve ChatGPT by 11.97% and WizardCoder by 11.87%. **(3) SeCoT has better stability.** From the table we can see that these baseline prompting techniques exhibit inconsistent performance for different test sets and settings. For example, in ChatGPT and HumanEval, Few-shot prompting is significantly better than Direct prompting, CoT prompting and SCoT prompting in Pass@1. However, in Pass@3, CoT prompting is significantly better than the other three baseline prompting technologies. However, our proposed SeCoT is significantly better than other prompting technologies in most configurations, which shows the stability of SeCoT prompting.

5.2 RQ2: Is SeCoT prompting effective in actual engineering project development?

Evaluation. In addressing our second research question, we apply SeCoT prompts and baselines to code implementations on real projects. To evaluate the functional correctness of the generated code, we employ unit tests as an evaluation metric. We tested using baseline ChatGPT on Redis, an open source in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.

We extract functions from different source files of Redis and sample according to the scale of the functions to build the test set. Specifically, we divided four intervals, that is, the number of lines of code is 0-4, 5-9, 10-14 and above or equal to 15. We randomly extract 5 functions from each of the four intervals for replacement, and let LLM generate the replacement code given the context (*i.e.*, requirements, function names and other contexts under the same file involved in the official code), And tested using Redis' official test suite. We then use the Pass1 test pass rate to verify the effectiveness of our method. Figure 5 shows our overall testing process. We carefully constructed this test data manually and will release it in an open source version. We believe that testing on Redis requires LLM to have better context understanding and coding implementation capabilities, so that it can better verify the actual effectiveness of the method. Figure 6 shows the examples of the codes we generated using the CoT, SCoT and SeCoT methods.

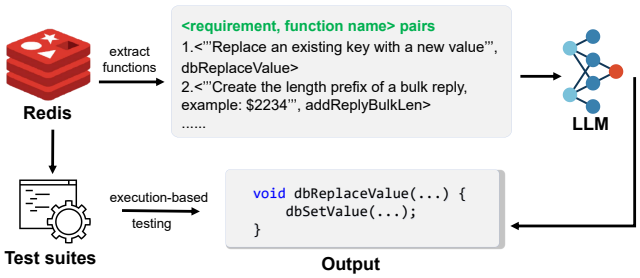


Figure 5: Experimental evaluation process on Redis.

Results and Analyses. The results of the different methods are summarized in Table 3. **(1) SeCoT achieves the best performance among all baselines.** In the Redis benchmark, SeCoT achieved the best performance across functions of different sizes. Obtaining code and fine-grained semantic information from intents can provide significant advantages in code generation compared to other prompting methods. SeCoT achieved the highest accuracy rates of 60%, 60%, 60% and 20% respectively in the range of 0-4, 5-9, 10-14, and 15+, with an average accuracy of 50%. The results show that SeCoT prompting can significantly improve the accuracy of LLM in real-world code generation and are more promising than existing prompting techniques.**(2) SeCoT prompting can capture nuances and generate more precise code.** For example, in case 1 of Figure 6, SeCoT built the correct *isValidPasswordHashChar* function, while both CoT and SCoT called the c language built-in function *isxdigit*. Although *isxdigit* function can determine the hexadecimal value, it ignores the requirement of lowercase characters 'a' 'f'. The *isValidPasswordHashChar* function correctly captures and implements these two requirements, which shows that SeCoT can capture more subtle semantic differences. During the cryptographic hash function value update process in case 2 of Figure 6, only the code generated by SeCoT prompting correctly updated the bitlen and datalen values during the loop. Bitlen is used to track the total length of processed data, and datalen is used to track the length of the current data block. However, the other two methods did not update or incorrectly updated these two values, causing Redis to run incorrectly. It is worth noting that in actual encryption applications, even minor errors may lead to serious security problems. Therefore, it is crucial to ensure that these values are calculated and managed correctly. SeCoT prompting alleviates this problem to a certain extent by capturing precise semantic information. **(3) All prompting methods (including SeCoT) still have considerable room for improvement in code generation of more than 15 lines.** As can be seen from Table 3, when the number of generated lines of code is less than 15 lines, all methods have higher accuracy. Especially for SeCoT prompting, the accuracy can reach 60%. However, when the number of lines of code exceeds 15 lines, the performance of all methods including SeCoT decreases to a great extent. This illustrates that long code generation remains challenging. Since the focus of this paper is not on solving the long code generation problem, we raise this challenge and leave it to future work to address.

Table 3: The Pass@1 (%) of SeCoT prompting and baselines on actual engineering project benchmarks. Bold values indicate the best performance.

ChatGPT/Code lines	< 5	5 – 9	10 – 14	15+	avg
Zero-shot	20.00	60.00	40.00	20.00	35.00
Few-shot	20.00	20.00	60.00	20.00	30.00
CoT	40.00	20.00	40.00	20.00	30.00
SCoT	40.00	20.00	40.00	20.00	30.00
SeCoT	60.00	60.00	60.00	20.00	**50.00**

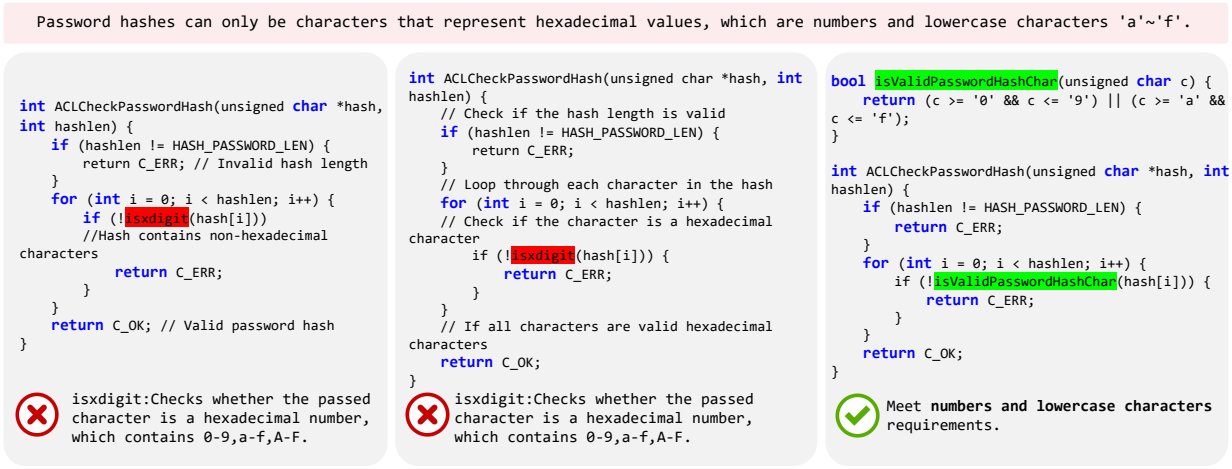
5.3 RQ3: What contribution does different semantic information contribute in SeCoT prompting?

Evaluation. As mentioned in Section 3.1, SeCoT (Semantic Chain-of-Thought) prompting introduces semantic information *i.e.*, data flow and control flow, to assist code generation. This research question (RQ) aims to analyze the contribution of different semantic information. We choose ChatGPT as the base model. We then conduct an ablation study by including data flow information and control flow independently.

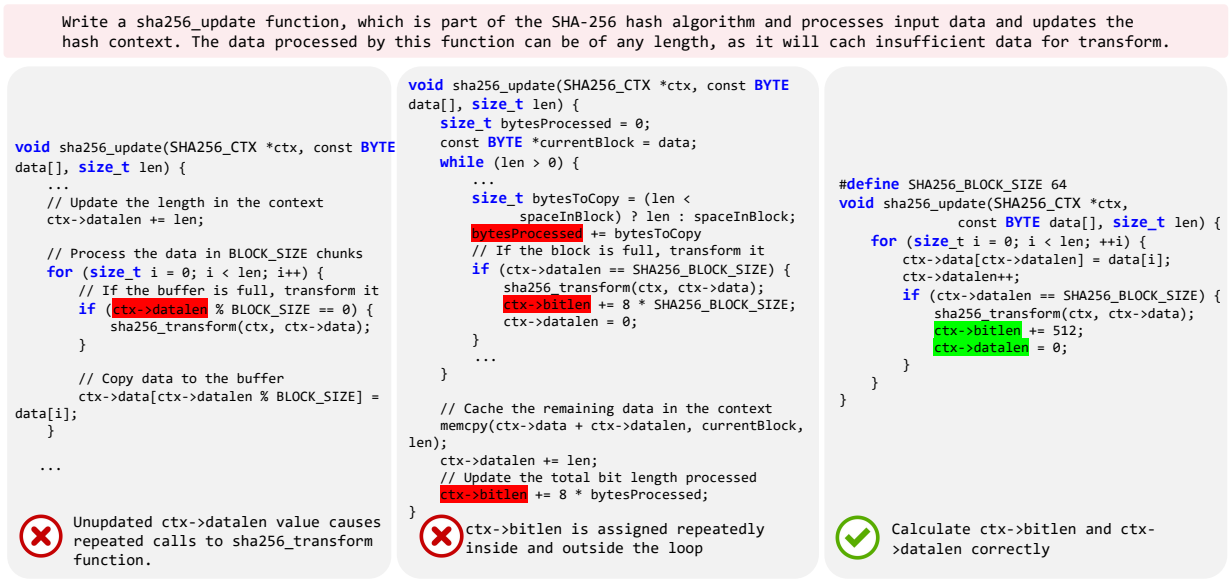
Results and Analyses. The results of the different methods are summarized in Table 1 and Table 2. Among them, SeCot_data is a semantic prompting that only contains data flow, SeCoT_control is a semantic prompting that only contains control flow, and SeCoT contains two types of semantic information: data flow and control flow. As can be seen from the table 1 and table 2, **(1) in most cases, SeCoT prompting containing two kinds of semantic information achieves the optimal performance**, which shows that the combination of two kinds of semantic information, data flow and control flow, helps LLMs understand better accurate code semantics and generate more correct codes. **(2) Even if it only contains a single semantic information *i.e.*, SeCoT_data and SeCoT_control, it is better than other prompting methods in most cases.** This further illustrates the importance of data flow and control flow information. That is, data flow interprets the operations performed by the program on data, and control flow determining the execution order of these operations, further enhances the code reasoning capabilities of LLMs.

Table 4: The results of the different in-context examples.

	HumanEval			HumanEval-ET		
Seed	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Seed A	68.78	82.68	86.59	62.80	75.79	79.88
Seed B	68.29	81.71	86.59	62.20	75.61	79.27
Seed C	68.41	81.59	85.98	62.20	75.18	79.88
ChatGPT	67.20	82.38	86.59	60.85	75.30	79.27



(a) Case1-ACLCheckPasswordHash implementations (CoT, SCoT and SeCoT)



(b) Case2-sha256_update implementations (CoT, SCoT and SeCoT)

Figure 6: Case Study on Redis evaluation.

5.4 RQ4: Is SeCoT prompting robust to examples?

Evaluation. In Section 3.1 we mentioned that the SeCoT (Semantic Chain-of-Thought) prompting relies on manually written examples to build prompts. In real applications, the performance of SCoT prompting may fluctuate as people may write different examples. Therefore, in this research question, we focus on exploring the robustness of SeCoT prompting to examples. Specifically, we select some <requirement, code> pairs as example seeds and manually write SeCoT for them to obtain examples in prompts. In this research question, we follow existing work [15] in measuring the robustness of SeCoT cues to examples in terms of seed selection. We choose ChatGPT as the base model and evaluate it in HumanEval and HumanEval-ET.

Seed Selection: This section aims to verify whether the SeCoT prompt depends on a specific seed. We select three sets of <requirements, code> pairs as seeds and ask an annotator to write SCoT for them. Next, we get three sets of examples. We measure the performance of SeCoT prompting using different sets of examples. In addition, to lower the threshold for example annotation, we try to use examples generated by ChatGPT as in-context examples for evaluation. Specifically, we use the same example as seed A, and then give two manually annotated examples to assist prompt ChatGPT to generate the corresponding data flow and control flow information in the seed A examples.

Results and Analyses. The results of the different methods are summarized in Table 4. SeCoT prompting are robust to examples. As shown in Tables 4, there are slight differences in the performance of SeCoT prompting when using different example seeds. This is expected for prompting techniques using examples. In addition, we found that the results of examples written by ChatGPT are similar to the results of manually written examples, which shows that our method can be combined with ChatGPT for further simple applications. We will implement the automatic SeCoT writing function in subsequent plug-ins.

6 THREATS TO VALIDITY

Generalizability of experimental results. To mitigate this threat, we carefully select benchmarks, metrics, and baselines . Following previous studies [2, 6, 14, 15], we select three representative code generation benchmarks. They are handwritten or collected from real-world programming communities. Secondly, we pick Redis [1], an actual engineering project, to evaluate the effectiveness of SeCoT in actual development projects. In terms of evaluation metrics, we choose a widely used metric - Pass@k [6], which utilizes test cases to check the correctness of the program. For comparison baselines, we select state-of-the-art prompting techniques and evaluate them on optimal open-source and closed-source code models [21, 23] respectively.

Prompts crafting. Because the sample selection and manual construction of different prompting methods may affect the degree of improvement that our method can achieve. This problem stems from the inherent sensitivity of LLMs to prompts, and to solve this problem requires fundamental improvements in LLMs [14]. But in our current approach, a considerable degree of improvement is ensured by random sampling of examples and the principle of controlled variables. We guarantee that SCoT prompting and baselines have the same example seeds and maximum generation lengths, and we think the relative improvements between hint methods are credible. In addition, to further improve the convenience of the method, we verified in Chapter 5.4 that ChatGPT can generate prompts comparable to humans.

7 RELATED WORK

7.1 Training of large language models

Large language models (LLMs) are usually based on the Transformer architecture [8]. Some of the noteworthy models include BERT [10], GPT-2 [27], and T5 [28]; After the emergence of GPT-3 [5] with 175B parameters, a number of larger models emerged, including PaLM [7], OPT [42], PanGu-α [41], Llama[31], Code Llama [29], WizardCoder [21] and GPT-4 [24], etc. These models have achieved remarkable results on various natural language processing and code intelligence tasks [13, 16, 18, 39]. These models are first pretrained on unsupervised large-scale data, *i.e.*, predicting the next word based on the first k words in the sentence, so that LLMs can realize the understanding and modeling of natural language and programming language. The model at this stage is called the base model. To enable the basic LLMs to follow human instruction output, instruction tuning [25] plays an important role. The instruction tuning method makes the model responsive to human instructions by fine-tuning the model using human-written <instruction, answer> pairs or supervised data from public benchmarks. In addition, self-instruction tuning [25, 33] is a simple and effective method to fine-tune other LLMs by using state-of-the-art LLMs as teachers to generate instruction data.

[1]https://redis.io/

7.2 Chain-of-Thought and Reasoning Ability

The emergence of the reasoning capabilities of large language models (LLMs) is the most important milestone on its road to general artificial intelligence (AGI). To achieve strong reasoning capabilities, artificial intelligence needs to show human-like behaviors or thinking patterns [12, 26, 36], and academia and industry have invested a lot of energy in this [32, 34]. The emergence of "Chain-of-Thought(CoT)" [30, 34] technology has brought LLMs a step closer to AGI. CoT first allows the LLM to generate a series of reasoning steps and then generates the final answer, which significantly improves the reasoning capabilities of LLM. Most efforts on CoT have focused on: i) the structure of the "chain", e.g., tree [37], graph [38]; and ii) the reliability of the chain, e.g. , self-consistency [33], retrieval-based prompts [14]. To further improve code-related reasoning capabilities, Structured Chain-of-Thought (SCoT) explicitly introduces program structure and requires LLMs to generate intermediate reasoning steps with program structure, and then generates the final code. However, existing CoT and SCoT methods ignore the rich semantic features in the source code, such as data flow and control flow information. This semantic information expresses the program's more precise execution behavior, intent, and functionality.

8 CONCLUSION AND FUTURE WORK

As LLMs continue to grow in size and complexity, how to effectively stimulate their code reasoning capabilities has become a key challenge. This paper proposes Semantic Chain-of-Thought (SeCoT) prompting, which greatly improves the reasoning capabilities of LLMs by guiding LLMs to analyze and understand the semantic information of the source code. A large-scale study on three benchmarks shows that SeCoT prompting significantly outperform the state-of-the-art methods in terms of Pass@k for code generation. In addition, the effectiveness of SeCoT in actual development has been proven through evaluation in actual engineering project Redis.

In the future, we will explore new prompting techniques that enhance code reasoning capabilities. For example, combining SeCoT with the recent Graph-of-Thought (GoT) further improves the accuracy of code generation. In addition, we will develop code generation plug-ins based on SeCoT prompting to improve actual development efficiency.

REFERENCES

[1] Anthropic. 2023. claude. https://claude.ai/.
[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. arXiv preprint arXiv:2108.07732 (2021).
[3] Debarag Banerjee, Pooja Singh, Arjun Avadhanam, and Saksham Srivastava. 2023. Benchmarking LLM powered Chatbots: Methods and Metrics. arXiv preprint arXiv:2308.04624 (2023).

[4] Corrado Böhm and Giuseppe Jacopini. 1966. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Commun. ACM 9, 5 (may 1966), 366–371. https://doi.org/10.1145/355592.365646

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021).

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311 (2022).

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. arXiv preprint arXiv:1901.02860 (2019).

[9] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. arXiv preprint arXiv:2304.02014 (2023).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[11] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. arXiv preprint arXiv:2301.09043 (2023).

[12] Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. arXiv preprint arXiv:2212.10403 (2022).

[13] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. Knod: Domain knowledge distilled tree decoder for automated program repair. arXiv preprint arXiv:2302.01857 (2023).

[14] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. arXiv preprint arXiv:2303.06689 (2023).

[15] Jia Li, Ge Li, yongmin Li, and Zhi Jin. 2023. Structured Chain-of-Thought Prompting for Code Generation. https://arxiv.org/pdf/2305.06599.pdf (2023).

[16] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. Towards Enhancing In-Context Learning for Code Generation. arXiv preprint arXiv:2303.17780 (2023).

[17] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! arXiv preprint arXiv:2305.06161 (2023).

[18] Mingwei Liu, Yanjun Yang, Yiling Lou, Xin Peng, Zhong Zhou, Xueying Du, and Tianyong Yang. 2023. Recommending Analogical APIs via Knowledge Graph Embedding. arXiv preprint arXiv:2308.11422 (2023).

[19] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. Comput. Surveys 55, 9 (2023), 1–35.

[20] Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. IEEE Transactions on Software Engineering (2023).

[21] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv preprint arXiv:2306.08568 (2023).

[22] Yingwei Ma, Yue Yu, Shanshan Li, Zhouyang Jia, Jun Ma, Rulin Xu, Wei Dong, and Xiangke Liao. 2023. MulCS: Towards a Unified Deep Representation for Multilingual Code Search. In 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 120–131.

[23] OpenAI. 2023. ChatGPT. https://openai.com/blog/chatgpt.

[24] OpenAI. 2023. GPT-4. https://openai.com/gpt-4.

[25] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4. arXiv preprint arXiv:2304.03277 (2023).

[26] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2022. Reasoning with language model prompting: A survey. arXiv preprint arXiv:2212.09597 (2022).

[27] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.

[28] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research 21, 1 (2020), 5485–5551.

[29] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023).

[30] Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Synthetic prompting: Generating chain-of-thought demonstrations for large language models. arXiv preprint arXiv:2302.00618 (2023).

[31] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023).

[32] Miles Turpin, Julian Michael, Ethan Perez, and Samuel R Bowman. 2023. Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting. arXiv preprint arXiv:2305.04388 (2023).

[33] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-Instruct: Aligning Language Model with Self Generated Instructions. arXiv preprint arXiv:2212.10560 (2022).

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903 (2022).

[35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019).

[36] Xiaohan Xu, Chongyang Tao, Tao Shen, Can Xu, Hongbo Xu, Guodong Long, and Jian-guang Lou. 2023. Re-Reading Improves Reasoning in Language Models. https://arxiv.org/pdf/2309.06275.pdf (2023).

[37] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. arXiv preprint arXiv:2305.10601 (2023).

[38] Yao Yao, Zuchao Li, and Hai Zhao. 2023. Beyond Chain-of-Thought, Effective Graph-of-Thought Reasoning in Large Language Models. arXiv preprint arXiv:2305.16582 (2023).

[39] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. arXiv preprint arXiv:2308.01240 (2023).

[40] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2023. degraphcs: Embedding variable-based flow graph for neural code search. ACM Transactions on Software Engineering and Methodology 32, 2 (2023), 1–27.

[41] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, et al. 2021. PanGu-α: Large-scale Autoregressive Pretrained Chinese Language Models with Auto-parallel Computation. arXiv preprint arXiv:2104.12369 (2021).

[42] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068 (2022).

[43] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In International Conference on Machine Learning. PMLR, 12697–12706.