Structured Chain-of-Thought Prompting for Code Generation

JIA LI ♂, GE LI*, YONGMIN LI, and ZHI JIN, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China

Large Language Models (LLMs) have shown impressive abilities in code generation. Chain-of-Thought (CoT) prompting is the state-of-the-art approach to utilizing LLMs. CoT prompting asks LLMs first to generate CoTs (*i.e.,* intermediate natural language reasoning steps) and then output the code. However, the accuracy of CoT prompting still can not satisfy practical applications. For example, gpt-3.5-turbo with CoT prompting only achieves 53.29% Pass@1 in HumanEval.

In this paper, we propose Structured CoTs (SCoTs) and present a novel prompting technique for code generation named SCoT prompting. Our motivation is that human developers follow structured programming. Developers use three programming structures (*i.e.,* sequential, branch, and loop) to design and implement structured programs. Thus, we ask LLMs to use three programming structures to generate SCoTs (structured reasoning steps) before outputting the final code. Compared to CoT prompting, SCoT prompting explicitly introduces programming structures and unlocks the structured programming thinking of LLMs. We apply SCoT prompting to two LLMs (*i.e.,* gpt-4-turbo, gpt-3.5-turbo, and DeepSeek Coder-Instruct-{1.3B, 6.7B, 33B}) and evaluate it on three benchmarks (*i.e.,* HumanEval, MBPP, and MBCPP). SCoT prompting outperforms CoT prompting by up to 13.79% in Pass@1. SCoT prompting is robust to examples and achieves substantial improvements. The human evaluation also shows human developers prefer programs from SCoT prompting.

CCS Concepts: • **Computing methodologies** → *Neural networks*; *Natural language processing*; • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Code Generation, Large Language Models, Prompting Engineering

1 INTRODUCTION

Code generation aims to automatically generate a program that satisfies a given natural language requirement [16, 19, 41]. Large Language Models (LLMs) have recently shown impressive performance in code generation, such as gpt-4 [26] and DeepSeek Coder [12]. During the inference, LLMs take a prompt as input that consists of several demonstration examples (*e.g.,* <requirement, code> pairs) and a new requirement. LLMs learn code generation from examples and analogously generate a new program. The performance of LLMs heavily relies on the prompt [11, 20, 42]. Nowadays, how to make an effective prompt (*i.e., Prompting technique*) for code generation is still an open question.

Chain-of-Thought (CoT) prompting [39] is the state-of-the-art (SOTA) prompting technique. CoT Prompting asks LLMs to generate a CoT and then output the code. A CoT is several intermediate natural language reasoning steps that describe how to write code step by step. Figure 1 (left) shows a CoT on code generation. However, CoT

*Corresponding author

Authors' address: Jia Li ♂, lijia@stu.pku.edu.cn; Ge Li, lige@pku.edu.cn; Yongmin Li, liyongmin@pku.edu.cn; Zhi Jin, zhijin@pku.edu.cn, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing, China.

ACM 1557-7392/2024/8-ART
https://doi.org/10.1145/3690635

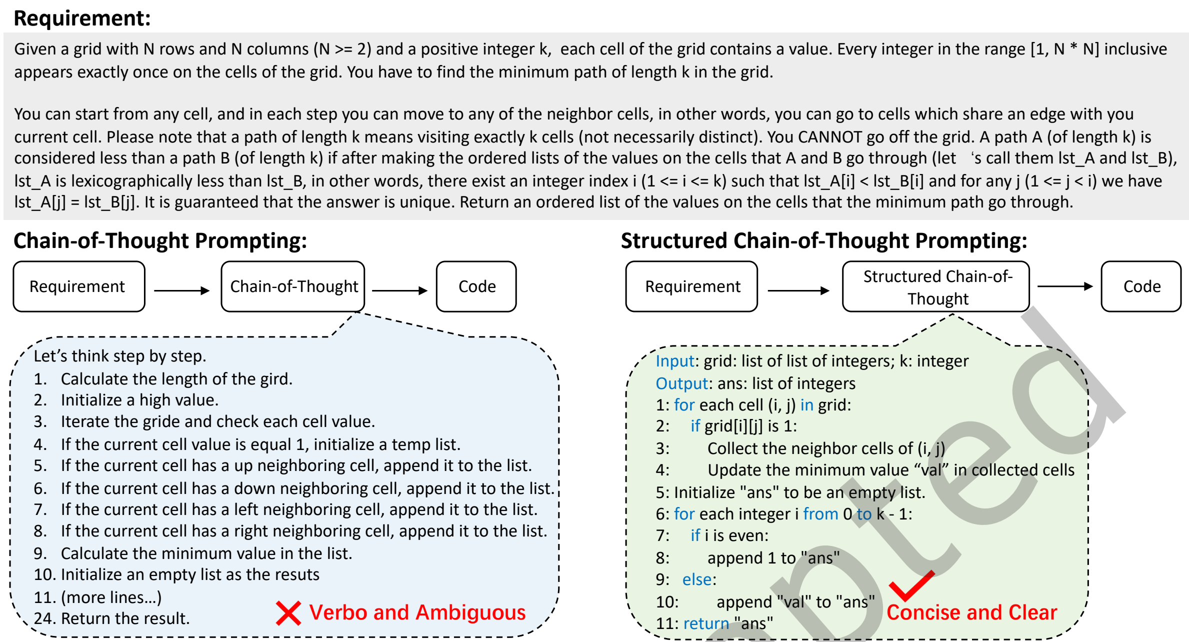

Fig. 1. The comparison of a Chain-of-Thought (CoT) and our Structured Chain-of-Thought (SCoT).

prompting brings slight improvements in code generation. For example, it only improves gpt-3.5-turbo by 0.82 points in Pass@1 upon a real-world benchmark [7].

Human developers typically follow *structured programming* to write high-quality programs. Specifically, developers leverage three programming structures (*i.e.*, sequential, branch, and loop structures) to decompose complex requirements and think about how to solve them. Intuitively, structured reasoning steps conduce to structured programs. A similar phenomenon is also found in fields such as programming education and is known as *structured programming thinking* [8]. However, CoT prompting can only represent the sequential structures in the code and is naturally unsuitable for branch and loop structures.

To alleviate the above knowledge gap, we propose a Structured CoT (SCoT) for code generation. A SCoT is a series of intermediate reasoning steps built with three programming structures (*i.e.,* sequential, branch, and loop structures). Figure 1 (right) shows a SCoT. Compared to the CoT, our SCoT has two advantages. ❶ **Our SCoT comprises three programming structures.** By explicitly generating programming structures, LLMs' programming abilities are unlocked. We steer LLMs to think about how to solve requirements using programming logic. ❷ **Our SCoT is a suitable midpoint between natural languages and the code.** As shown in Figure 1, the CoT is verbose and aggravates the burden on models during generation. In contrast, our SCoT is very concise. It uses programming structures to organize the reasoning process and leverages natural languages to describe specific operations. It can be viewed as a springboard in code generation. Trained on a large amount of natural language text and code data, LLMs can generate such SCoTs.

Specifically, a SCoT consists of two parts. The first part is an Input-Output (IO) structure. By generating an IO structure, LLMs define the entry and exit of the code, which clarifies requirements and facilitates the following implementation. The second part is a rough problem-solving process. Any code or algorithm can be composed of three basic structures, *i.e.,* sequence, branch, and loop structures [3]. We teach LLMs to generate the solving process based on three basic programming structures. It ensures that our SCoT can show problem-solving processes for wide-ranging programs. Because LLMs' training data contains lots of code data, we believe they can generate the above programming structures.

Based on the SCoT, we present SCoT prompting. By prompting several demonstration examples (*i.e.,* <requirement, SCoT, code>), it teaches LLMs to generate an SCoT and then implement the code. We apply SCoT prompting to five popular LLMs (*i.e.,* gpt-4-turbo [26], gpt-3.5-turbo [24], and DeepSeek Coder-Instruct-{1.3B, 6.7B, 33B}). We compare SCoT prompting to CoT prompting on three representative benchmarks (*i.e.,* HumanEval [7], MBPP [2], and MBCPP [1]). We use test cases to measure the correctness of generated programs and report the Pass@k ($k \in [1, 3, 5]$). In terms of Pass@1, SCoT prompting outperforms CoT prompting by up to 13.79% in HumanEval, 12.31% in MBPP, and 13.59% in MBCPP. The improvements are stable in different LLMs and programming languages. The human evaluation also shows that human developers prefer programs generated by SCoT prompting. We also discuss the robustness of SCoT prompting to demonstration examples. Results show that SCoT prompting does not depend on specific examples, writing styles, and example orderings.

Our contributions are as follows.

- We propose a Structured Chain-of-Thought (SCoT), which uses programming structures to build intermediate reasoning steps toward the structured code.
- We propose SCoT prompting for code generation. It prompts large language models to generate an SCoT and then implement the code.
- Qualitative and quantitative experiments show the superiority of SCoT prompting. We also discuss the robustness of SCoT prompting.

Paper Organization. Section 2 presents our proposed SCoT prompting. Section 3 and Section 4 show the design and results of our study, respectively. Section 5 and Section 6 discuss some results and describe the related work, respectively. Section 7 concludes the paper and points out future directions.

2 METHODOLOGY

In this section, we propose a Structured Chain-of-Thought (SCoT). A SCoT denotes several intermediate reasoning steps constructed by programming structures. Then, we present a novel prompting technique for code generation named SCoT prompting. SCoT prompting asks LLMs first to generate an SCoT and then output the final code. In the subsections, we first describe the design of our SCoT and further show the details of SCoT prompting.

2.1 Structured Chain-of-Thought

A standard Chain-of-Thought (CoT) is several intermediate natural language reasoning steps that lead to the final answer [39]. The CoT is initially designed for natural language generation (*e.g.,* commonsense reasoning [31]). Thus, the CoT only uses natural language to describe how to solve a problem step by step sequentially. Figure 1 (a) shows a CoT on code generation. However, the CoT brings slight improvements in code generation. For example, CoT prompting only improves gpt-3.5-turbo by 0.82 points in Pass@1 on HumanEval.

In this paper, we propose a Structured CoT. Our motivation is that human developers benefit from structured programming in coding. In other words, developers rely on three programming structures (*i.e.,* sequential, branch, and loop structures) to design and implement high-quality programs. Given a requirement - `reading text from a given file`, imagine a developer's thought process. The developer will use programming structures to design an initial idea: ``` ``if the given file exists: read text from the file; else: raise an error;'' ```. The programming structures clearly show the solving process and benefit the following code implementation. Thus, we leverage programming structures to build intermediate reasoning steps, obtaining the SCoT.

```
Input: paren_string: str
Output: list_of_int: List[int]
Initialize list_of_int to an empty list
for each string in paren_string do
  Initialize depth to 0
  for each character in string do
    if character is '(' then
      increment depth by 1
    elif character is ')' then
      decrease depth by 1
  append depth to list_of_int
return list_of_int
```

Loop Structure

Branch Structure

(a)

```
Input: string: str, substring: str
Output: count: int
Initialize count to 0
while substring is not found in string do
  if string is empty then
    return 0
  increment count by 1
  remove the first character of string
return count
```

Loop Structure

Sequential Structure

(b)

Fig. 2. Examples of the proposed SCoT.

Figure 2 shows some SCoTs. Compared to the CoT, our SCoT explicitly introduces three programming structures. Existing work [3] proved that any simple or complex program can be composed of three basic structures, *i.e.,* sequence structure, branch structure, and loop structure Thus, we introduce three basic structures, the details of which are as follows.

- **Sequential Structure.** The intermediate steps are sequentially placed, and all steps are at the same level.
- **Branch Structure.** It starts with a condition and places different intermediate steps for different results of the condition. In this paper, branch structures contain three formats, *i.e.,* `if ...`, `if ... else`, and `if ... elif ... else`. The `elif` is the shorthand for `else if` and creates a nested branch structure.
- **Loop Structure.** A set of intermediate steps are repeatedly conducted until conditions are unmet. In this paper, loop structures contain two basic formats: the `for loop` and the `while loop`.

We provide a few guidelines for writing SCoTs. ❶ Users should use the above programming structures (*e.g.,* `if ... else`) to build the SCoT. We allow the nesting between different programming structures. It allows LLMs to design more complex SCoT for some difficult requirements. As shown in Figure 2, the SCoT flexibly uses various programming structures to build a solving process. ❷ We recommend that users use natural language to express specific operations in SCoTs, such as `increase depth by 1` in Figure 2. Meanwhile, users can use common formal symbols (*e.g.,* +, =, and `!=`) in SCoTs. Experiments in Section 4.3 show that SCoT prompting is robust to symbols and natural language text.

In addition to three basic structures, we add the input-output structure, which contains input-output parameters and their types. Our motivation is that an input-output structure is required for a program, which indicates entry

Natural language instructions

```
Your task is to complete the following code. You should
first write a rough problem-solving process using three
programming structures (i.e., sequential, branch, and
loop structures) and then output the final code.

Here are some demonstration examples:
```

Demonstration examples

```
def sum_Of_Primes(n):
    """ Write a python function to find sum of prime
numbers between 1 to n. """
    # Let's think step by step
    # Input: n, an integer
    # Output: sum, an integer
    # 1. Initialize a list "prime" with True values.
    # 2. Initialize a variable "p" with 2.
    # 3. While p * p is less than or equal to n:
    # 4.   If prime[p] is True:
    # 5.     Set all the multiples of p to False.
    # 6.   Increment the variable "p" by 1.
    # 7. Compute the sum of the prime numbers.
    # 8. Return the sum.
    # Write your code here
    prime = [True] * (n + 1)
    p = 2
    while p * p <= n:
     (more lines…)
(more examples…)
```

Testing requirement

```
Input code:
def text_lowercase_underscore(text):
    """
    Write a function to find sequences of lowercase
letters joined with an underscore.
    """
    # Let's think step by step
```

(a)

```
Your task is to complete the following code. You should
first write a rough problem-solving process using three
programming structures (i.e., sequential, branch, and
loop structures) and then output the final code.

Here are some demonstration examples:
```

```
/**
* Write a c++ function to find sum of prime numbers
between 1 to n.
*/
int sumOfPrimes(int n) {
    // Let's think step by step
    // Input: n, an integer
    // Output: sum, an integer
    // 1. Initialize a list "prime" with True values.
    // 2. Initialize a variable "p" with 2.
    // 3. While p * p is less than or equal to n:
    // 4.   If prime[p] is True:
    // 5.     Set all the multiples of p to False.
    // 6.   Increment the variable "p" by 1.
    // 7. Compute the sum of the prime numbers.
    // 8. Return the sum.
    // Write your code here
    vector<bool> prime(n + 1, true);
    (more lines…)
(more examples…)
```

```
Input code:
/**
* Write a function to find sequences of lowercase
letters joined with an underscore.
*/
string textLowercaseUnderscore(string text) {
    // Let's think step by step
```

(b)

Fig. 3. Examples of prompts in SCoT prompting. (a) A prompt for Python, (b) A prompt for C++.

and exit. Generating the input-output structure is beneficial to clarify requirements and generate the following solving process.

2.2 SCoT prompting

Based on the SCoT, we propose a new prompting technique for code generation named SCoT prompting. It asks LLMs to generate a SCoT first and then output the final code. To implement SCoT prompting, we design a special prompt. Figure 3 shows two examples of our prompts for Python and C++. The designs of prompts are shown as follows.

❶ **The components in prompts.** Following previous approaches (*e.g.,* few-shot and CoT prompting), our prompts comprise three components, *i.e.,* natural language instructions, demonstration examples, and a testing requirement. The natural language instructions are written by authors and tell the goal of LLMs and related constraints. Demonstration examples are a few <requirement, SCoT, code> tuples. The instructions and demonstration examples aim to tell LLMs how to generate the code with SCoTs. Finally, the prompt ends with a testing requirement.

❷ **The format of prompts.** The prompts format combines the above components into a whole input sequence. The key challenge is how to represent requirements and SCoTs. Our motivation is that existing LLMs are trained on many code files from open-source repositories. These code files typically consist of many functions with comments. Therefore, we represent requirements and SCoT in a similar format. Specifically, as shown in Figure 3, the requirement is represented as a Python signature and a docstring. The SCoT is encoded as line comments. This prompt format is also consistent with previous studies [1, 7]. Following previous studies [1, 39], we insert two

Table 1. Statistics of the datasets in our experiments.

Statistics	HumanEval	MBPP	MBCPP
Language	Python	Python	C++
# Train	–	474	413
# Test	164	500	435
Avg. tests per sample	7.7	3	3

natural language hints (*i.e.,* `Let's think step by step`, `Write your code here`) into prompts. These hints are empirical tricks and benefit the reasoning abilities of LLMs.

2.3 Implementation Details

SCoT prompting is a prompting technique for code generation, which does not rely on specific LLMs. Users can flexibly apply SCoT prompting to more powerful LLMs in a plug-and-play fashion. We select a few (*e.g.,* three) <requirement, code> pairs from real-world benchmarks (*i.e.,* training data) as example seeds. Then, we manually write the SCoT for seeds and obtain examples - <requirement, SCoT, code> triples, which are used to make prompts in Figure 3. We recommend users use examples that cover all three programming structures.

3 STUDY DESIGN

To assess SCoT prompting, we conduct a large-scale study to answer four research questions. In this section, we present the details of our study, including datasets, evaluation metrics, comparison baselines, and implementation details.

3.1 Research Questions

Our study aims to answer the following Research Questions (RQ).

RQ1: How does SCoT prompting perform in terms of accuracy compared to baselines? This RQ aims to verify that SCoT prompting has a higher accuracy than existing prompting techniques on code generation. We apply three existing prompting techniques and SCoT prompting to five LLMs. Then, we use test cases to measure the correctness of programs generated by different approaches and report the Pass@k.

RQ2: Do developers prefer programs generated by SCoT prompting? The ultimate goal of code generation is to assist human developers in writing code. In this RQ, we hire 10 developers (including industry employees and academic researchers) to review the programs generated by SCoT prompting and baselines manually. We measure the quality of programs in three aspects: correctness and bad smells.

RQ3: Is SCoT prompting robust to examples? Prompting techniques may be sensitive to demonstration examples [11, 42]. In this RQ, we measure the robustness of SCoT prompting to examples in four aspects, including example seeds, writing styles of examples, example orderings, and the number of examples.

RQ4: What are the contributions of different programming structures in SCoT prompting? As stated in Section 2.1, SCoT prompting introduces three basic structures and the input-output structure. This RQ is designed to analyze the contributions of different structures. We select an LLM as the base model. Then, we individually remove a programming structure and report the fluctuations in performance.

3.2 Datasets

Following previous studies [6, 7, 23, 43], we conduct experiments on three representative code generation benchmarks, including the HumanEval in Python, MBPP in Python, and MBCPP in C++. The details of the benchmarks are described as follows.

- **HumanEval** [7] is a Python function-level code generation benchmark, which contains 164 hand-written programming problems. Each programming problem consists of an English requirement, a function signature, and several test cases, averaging 7.7 test cases per problem.
- **MBPP** [2] is a Python function-level code generation benchmark. It contains 974 programming problems that involve simple numeric manipulations or basic usage of standard libraries. Each problem contains an English requirement, a function signature, and three manually written test cases for checking functions.
- **MBCPP** [1] is a C++ function-level code generation benchmark. It consists of 848 programming problems that are collected by crowd-sourcing. Each problem contains an English description, a function signature, and three test cases for checking the correctness of functions.

We follow the original splits of three datasets. The statistics of the benchmarks are shown in Table 1. We randomly pick several samples from training data to make examples in prompts (Section 2.3). Then, we measure the performance of different approaches on test data. Because HumanEval does not contain train data, we reuse examples from MBPP in HumanEval. We notice that researchers have recently proposed repository-level code generation benchmarks [17, 18], which we leave as future work.

3.3 Evaluation Metrics

Following previous code generation studies [6, 7, 23, 43], we use Pass@k as our evaluation metrics. Specifically, given a requirement, a code generation model is allowed to generate k programs. The requirement is solved if any generated programs pass all test cases. We compute the percentage of solved requirements in total requirements as Pass@k. For Pass@k, a higher value is better. In our experiments, k is set to 1, 3, and 5, because we think that developers mainly use Top-5 outputs in real-world scenarios.

Previous work [1, 6, 7] found that standard Pass@k has high variance and proposed an unbiased Pass@k. We follow previous work and employ the unbiased Pass@k. Specifically, we generate $n \geq k$ programs per requirement (in this paper, we use $n = 20$, $k \in [1, 3, 5]$), count the number of solved requirements c, and calculate the unbiased Pass@k:

$$\text{Pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{1}$$

We also notice that previous code generation studies use text-similarity-based metrics (*e.g.,* BLEU [28]). These metrics are initially designed for natural language generation and are poor in measuring the correctness of programs [7]. Thus, we omit these metrics in our experiments.

3.4 Comparison Baselines

This paper proposes a new prompting technique for code generation. To assess the effectiveness of our approach, we select three mainstream prompting techniques as baselines.

- **Zero-shot prompting** [7] directly feeds the requirement into LLMs without examples. Then, it extracts a generated program from LLMs' outputs.

- **Few-shot prompting** [7] randomly selects several < requirement, code> pairs as examples. Given a requirement, it concatenates examples and the requirement together, making a prompt. Then, the prompt is fed into LLMs, and LLMs predict a new program.
- **Chain-of-Thought (CoT) prompting** [39] is a variant of few-shot prompting. CoT prompting produces a special prompt consisting of <requirement, CoT, code> triples as examples. A CoT is several intermediate natural language reasoning steps.

To ensure the fairness of comparison, all baselines and SCoT prompting have the same number of examples (*i.e.,* three examples) and example seeds.

The criteria for selecting baselines are three-fold. ❶ SCoT prompting is a prompting technique for code generation. Thus, we directly compare it to existing prompting techniques for code generation. We also notice some emerging prompting techniques in other fields, such as Self-Consistency [35] and Least-to-Most [44]. But these approaches are designed for specific tasks (*e.g.,* Arithmetic reasoning) and can not be directly applied to code generation. Thus, we omit them in this paper. ❷ Our approach is to augment LLMs and can be flexibly applied to different LLMs. Thus, we do not directly compare LLMs to our approach. ❸ We also omit some rank techniques for code generation [6]. They first use LLMs to generate many candidates and then leverage test cases or neural networks to rerank candidates. We think our work and these rank techniques are complementary. Users can use our approach to generate programs and then use post-processing techniques to select the final output. We further discuss the complementarity through experiments in Section 5.2.

3.5 Base Large Language Models

In this paper, we conduct experiments on five popular LLMs, including code and general LLMs. Code LLMs are designed for the source code and are mainly trained with lots of code data. We select an open-source code LLM - DeepSeek Coder [12] as the base model. General LLMs are proposed for general artificial intelligence and are trained with lots of natural language text and code. We select two powerful general LLMs as base models, including gpt-4-turbo [26] and gpt-3.5-turbo [24].

DeepSeek Coder [12] is a large language model for programming tasks released by DeepSeek-AI[1] in November 2, 2023. DeepSeek Coder consists of a series of code language models, each trained from scratch on 2T tokens, containing 87% code and 13% natural language. DeepSeek Coder provides code models with 1.3B, 6.7B and 33B parameter sizes. In terms of model architecture, each model integrates a decoder-only Transformer, incorporating Rotary Position Embedding and FlashAttention v2. This paper evaluates DeepSeek Coder-Instruct-{1.3B, 6.7B, 33B}.

gpt-4-turbo [26] released by OpenAI on March 14, 2023, marks another milestone in the field of deep learning. gpt-4 demonstrates superior performance compared to previous gpt models [4, 29, 30]. In our experiments, we use the version - gpt-4-turbo-1106. Its training data up to April 2023. It continues the auto-regressive prediction of the next token training objective inherited from the GPT series models. It incorporates reinforcement learning with human feedback (RLHF) [27] and red-teaming techniques. However, the pre-training data scope and scale, model size, and parameters remain closed-source at present.

gpt-3.5-turbo [24] is an improved gpt-3 model enhanced by a three-stage reinforcement learning with human feedback (RLHF) algorithm. Apart from improving instruction-following capabilities, the RLHF algorithm proves highly effective in mitigating the generation of harmful or toxic content, which is crucial for the practical deployment of LLMs in security-sensitive contexts. we utilized the released versions of gpt-3.5, namely gpt-3.5-turbo-1106, with training data up to September 2021. However, similar to gpt-4, the training details, training data, and model weights are currently closed-source.

[1]https://www.deepseek.com/

Our approach does not rely on specific LLMs and can be applied to different LLMs in a plus-and-play fashion. In the future, we will explore its usage on more powerful LLMs.

3.6 Sampling Settings

Following previous studies [7, 23, 43], we use nucleus sampling [14] to decode programs from LLMs. To ensure the fairness of experiments, all baselines and SCoT prompting generate 20 programs per requirement. By default, all prompts of SCoT prompting and baselines employ three fixed demonstration examples written by the same annotator. The sampling settings follow previous work [6]. Specifically, the temperature is 0.8, and the top-p is 0.95.

4 RESULTS AND ANALYSIS

4.1 RQ1: How does SCoT prompting perform in terms of accuracy compared to baselines?

In the first research question, we apply SCoT prompting and baselines to three benchmarks and use unit tests to measure the correctness of generated programs.

Setup. We apply baselines and SCoT prompting to five LLMs (Section 3.5). Then, we measure the performance of different approaches on three benchmarks (Section 3.2) using the Pass@k (Section 3.3).

Results. The Pass@k ($k \in [1, 3, 5]$) of different approaches are shown in Table 2. The numbers in red denote SCoT prompting's relative improvements compared to the SOTA baseline - CoT prompting. "DS Coder-Ins" is the abbreviation of "DeepSeek Coder-Instruct".

SCoT prompting substantially outperforms baselines in three benchmarks and five LLMs. Compared to the SOTA baseline - CoT prompting, in terms of Pass@1, SCoT prompting outperforms it by up to 13.79% in HumanEval, 12.31% in MBPP, and 13.59% in MBCPP. Pass@1 is a strict metric and is difficult to improve. The improvements show that SCoT prompting can significantly improve the accuracy of LLMs on code generation and is more promising than existing prompting techniques.

SCoT prompting is effective in different LLMs and programming languages. SCoT prompting is effective in different LLMs. Compared to CoT prompting, in terms of Pass@1, SCoT prompting improves gpt-4-turbo by up to 6.49%, gpt-3.5-turbo by up to 13.79%, and DeepSeek Coder-Instruct by up to 13.59%. Besides, SCoT prompting brings substantial improvements in Python (*i.e.,* HumanEval and MBPP) and C++ (*i.e.,* MBCPP).

Answer to RQ1: SCoT prompting substantially outperforms baselines in three benchmarks and five LLMs. In terms of Pass@1, SCoT prompting outperforms the SOTA baseline by up to 13.79% in HumanEval, 12.31% in MBPP, and 13.59% in MBCPP.

4.2 RQ2: Do developers prefer programs generated by SCoT prompting?

The ultimate goal of code generation is to assist developers in writing programs. In this RQ, we hire 10 developers (including industry employees and academic researchers) to manually assess the programs generated by SCoT prompting and baselines.

Setup. To ensure the fairness of evaluation, we follow settings of human evaluation in previous studies [13, 19]. The evaluation metrics contain *correctness* and *bad smell*. The correctness is to evaluate whether the generated programs satisfy the requirements. Different from the binary Pass@k, the correctness is a more fine-grained metric that assigns different scores to programs. We also check whether the generated programs contain bad code smells. The definitions of the two metrics are shown as follows:

- **Correctness.** 0 point: the program is totally inconsistent with the requirement. 1 point: the program is implemented but misses some details or contains minor mistakes. 2 points: the program is correctly implemented.

Table 2. The Pass@k (%) of prompting approaches on three benchmarks. Numbers in red denote SCoT prompting's relative improvements compared to the SOTA baseline - CoT prompting. DS Coder-Ins: DeepSeek Coder-Instruct.

LLMs	Prompting	HumanEval			MBPP			MBCPP		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
gpt-4-turbo	Zero-shot prompting	75.45	83.23	84.94	52.27	59.47	61.82	56.48	68.17	72.17
	Few-shot prompting	76.22	86.18	88.26	52.51	61.71	64.88	57.11	69.13	72.84
	CoT prompting	78.29	87.11	89.33	53.65	62.43	65.32	57.89	70.07	73.16
	SCoT prompting	**82.67**	**89.75**	**92.43**	**57.13**	**66.15**	**70.43**	**61.44**	**73.58**	**77.52**
Relative Improvement		**5.59%**	**3.03%**	**3.47%**	**6.49%**	**5.96%**	**7.82%**	**6.13%**	**5.01%**	**5.96%**
gpt-3.5-turbo	Zero-shot prompting	49.73	66.07	71.54	37.07	43.54	48.58	47.53	60.09	64.22
	Few-shot prompting	52.47	69.32	74.1	40	49.82	53.13	52.58	63.03	66.11
	CoT prompting	53.29	69.76	75.52	41.83	51.04	54.57	53.51	63.84	67.03
	SCoT prompting	**60.64**	**73.53**	**77.32**	**46.98**	**55.31**	**58.36**	**57.06**	**65.70**	**68.70**
Relative Improvement		**13.79%**	**5.40%**	**2.38%**	**12.31%**	**8.37%**	**6.95%**	**6.63%**	**2.91%**	**2.49%**
DS Coder-Ins-33B	Zero-shot prompting	73.66	85.88	88.74	48.22	57.48	60.91	49.22	63.45	67.92
	Few-shot prompting	73.93	86.04	88.75	48.37	58.15	61.59	50.13	64.11	68.19
	CoT prompting	74.97	87.05	89.87	48.85	58.17	61.65	51.12	65.77	69.20
	SCoT prompting	**79.50**	**89.12**	**91.24**	**52.79**	**61.67**	**65.44**	**55.27**	**69.81**	**72.44**
Relative Improvement		**6.04%**	**2.38%**	**1.52%**	**8.07%**	**6.02%**	**6.15%**	**8.12%**	**6.14%**	**4.68%**
DS Coder-Ins-6.7B	Zero-shot prompting	67.16	82.2	85.84	43.14	52.28	55.76	37.22	57.40	63.25
	Few-shot prompting	67.29	83.08	87.03	43.18	54.04	57.85	38.51	58.62	63.96
	CoT prompting	67.71	83.49	87.31	43.90	54.31	58.10	39.71	59.66	64.15
	SCoT prompting	**71.06**	**87.81**	**90.33**	**47.69**	**58.71**	**62.11**	**43.58**	**63.70**	**67.40**
Relative Improvement		**4.95%**	**5.17%**	**3.46%**	**8.63%**	**8.10%**	**6.90%**	**9.75%**	**6.77%**	**5.07%**
DS Coder-Ins-1.3B	Zero-shot prompting	56.62	72.03	77.17	34.5	46.18	50.79	28.02	43.30	50.44
	Few-shot prompting	57.53	75.83	81.17	35.54	46.60	50.35	29.10	44.33	51.37
	CoT prompting	59.81	76.39	81.12	36.22	47.35	51.86	30.16	45.12	51.79
	SCoT prompting	**64.05**	**81.68**	**85.08**	**40.74**	**50.19**	**53.97**	**34.26**	**48.19**	**55.73**
Relative Improvement		**7.09%**	**6.92%**	**4.88%**	**12.48%**	**7.70%**	**7.19%**	**13.59%**	**6.80%**	**7.61%**

Table 3. The 22 common code smells [9] used in human evaluation.

22 Common Code Smells			
Mysterious Name	Divergent Change	Lazy Element	Large Class
Duplicated Code	Shotgun Surgery	Speculative Generality	Alternative Classes with Different Interfaces
Long Function	Feature Envy	Temporary Field	Data Class
Long Parameter List	Data Clumps	Message Chains	Refused Bequest
Global Data	Primitive Obsession	Middle Man	
Mutable Data	Repeated Switches	Insider Trading	

- **Bad Smells.** Previous work [9] summarized 22 common bad smells, as shown in Table 3. We ask evaluators to read the related book [9] and understand these bad smells. Then, we manually count the number of bad smells in the generated programs.

We invite 10 developers with 3-5 years of development experience as evaluators. The evaluators include industry employees and academic researchers who are not co-authors of this paper. We explain the above aspects

Table 4. The results of human evaluation. The numbers in red denote SCoT prompting's relative improvements compared to the SOTA baseline - CoT prompting. All the p-values are substantially smaller than 0.05.

Approach	Correctness ↑	Code Smell ↓
Zero-shot prompting	1.012	1.041
Few-shot prompting	1.119	0.902
CoT prompting	1.225	0.743
SCoT prompting	**1.412**	**0.546**
Relative Improvement	**15.27%**	**36.08%**

to evaluators through some examples. We also discuss with evaluators and set the score of each aspect to an integer, ranging from 0 to 2 (from bad to good). We select a fixed LLM as the base model (*i.e.,* gpt-3.5-turbo) and collect 200 generated programs (*i.e.,* HumanEval: 50, MBPP: 50, and MBCPP: 100) per prompting approach, totalling 800 programs. We remove CoTs or SCoTs from generated programs before evaluations. The 800 programs are divided into 5 groups, with each questionnaire containing one group. We take three measures to ensure the questionnaires are unbiased. First, *unbiased distributions*. Each questionnaire contains 160 programs, of which each prompting approach accounts for 25% (*i.e.,* 40 programs). Second, *anonymity*. All programs in questionnaires are anonymous. Developers do not know the sources of the programs under evaluation. Third, *random orders*. The order of programs within a questionnaire is determined randomly. Each group is evaluated by two evaluators, and the final score is the average of two evaluators' scores. Evaluators are allowed to search the Internet for unfamiliar concepts.

Before the formal evaluation, we collected 200 generated programs on MBPP (train set) and conducted a preliminary evaluation. We answered developers' questions during the preliminary evaluation and ensured they understood the evaluation metrics and questionnaires correctly.

Results. The human evaluation results are shown in Table 4. The numbers in red denote SCoT prompting's relative improvements compared to the SOTA baseline - CoT prompting. We compute the t-values and p-values between SCoT prompting and baselines. All p-values are substantially smaller than 0.05.

SCoT prompting substantially outperforms baselines in the correctness and bad smells. Particularly, SCoT prompting outperforms the SOTA baseline - CoT prompting by 15.27% in correctness and 36.08% in bad smells. We attribute the improvements to our proposed SCoT. The SCoT constrains LLMs to use programming structures to generate intermediate reasoning steps. It allows LLMs to explore diverse solutions with three basic structures, improving the correctness of the code. Then, serving as a clear outline, the SCoT steers LLMs to generate high-quality programs with fewer bad smells.

Figure 4 shows two programs generated by SCoT prompting and few-shot prompting, respectively. Both programs pass unit tests. But the program from few-shot prompting contains a very complex statement highlighted in Figure 4). Developers have to put a lot of effort into understanding and maintaining this program. In contrast, the program from SCoT prompting has good readability, and the SCoT clearly explains the code's behaviour. Developers can further use the SCoT as comments of the program for future maintenance.

Answer to RQ2: Human developers prefer programs generated by SCoT prompting. Specifically, SCoT prompting outperforms the SOTA baseline by 15.27% in correctness and 36.08% in bad smells. A case study also shows the program from SCoT prompting is easy to read and maintain.

Requirement:

```
# Return True is list elements are monotonically increasing or decreasing.
```

Few-shot prompting:

```
def monotonic(l: list):
    if all(l[i] <= l[i+1] for i in range(len(l)-1)) or all(l[i] >= l[i+1] for i in range(len(l)-1)):
        return True
    else:
        return False
```

?

SCoT prompting (SCoT & Source Code):

```
Input: l: list
Output: True or False
for each element in l do
   if the element is greater than its previous element then
      l increases
   if the element is less than its previous element then
      l decreases
if both increase and decrease then
   return False
else
   return True
```

```
def monotonic(l: list):
    increasing = False
    decreasing = False
    for i in range(1, len(l)):
        if l[i] > l[i-1]:
            increasing = True
        if l[i] < l[i-1]:
            decreasing = True
    if increasing and decreasing:
        return False
    else:
        return True
```

Fig. 4. Two programs generated by few-shot prompting and SCoT prompting, respectively.

4.3 RQ3: Is SCoT prompting robust to examples?

As stated in Section 2.3, SCoT prompting requires demonstration examples to make prompts. In practice, people may write different examples, which makes the performance of SCoT prompting varies. Thus, in this RQ, we explore the robustness of SCoT prompting to examples.

Setup. As stated in Section 2.3, we select a few <requirement, code> pairs as example seeds and manually write SCoTs for them, obtaining demonstration examples. In this RQ, we measure the robustness of SCoT prompting to examples in the following four aspects:

❶ Example seed. This setting aims to validate SCoT prompting does not rely on specific seeds. We randomly select three example seeds from the training data, and each seed consists of three <requirement, code> pairs. Then, we hire an annotator to write SCoTs for seeds, obtaining three groups of examples. The ordering of examples in each group is randomly determined. We measure the performance of SCoT prompting with different groups of examples.

❷ Writing style. People have different writing styles. This setting aims to validate that SCoT prompting does not rely on specific writing styles. We hire three annotators to independently write SCoTs for the same example seed and obtain three groups of examples. The ordering of examples in all groups is the same. The three annotators have different background knowledge and working scenarios. We observe the SCoTs written by three annotators and summarize their writing styles. Figure 5 illustrates SCoTs written by three annotators. Annotator A is a PhD student in software engineering. He flexibly uses code keywords and natural language text to write SCoTs. Annotator B is an industry product manager. She prefers to write SCoTs using colloquial natural language text. Annotator C is an industry developer. He often uses some formal notations (*e.g.,* variables and operators) in SCoTs.

❸ Example ordering. Existing works [42] have found that LLMs are sensitive to the ordering of examples. In this setting, we make three demonstration examples and randomly change the ordering of examples. In this way,

```
Input: s, a string
Output: result, a string
1. Set up an alphabet.
2. Initialize a numerical bias.
3. for each char in s:
4.     find the index of character in alphabet.
5.     add the bias to the index.
6.     if the index is larger than 25, then:
7.         subtract 26 from the index.
8.     add the character to the result.
9. return the result.
```

(a) Annotator A

```
Input: A string, input string
Output: A string, processed string
1. Initialize an alphabet and a bias.
2. for each char in the input string:
3.     Add the char's index in alphabet and bias.
4.     If the index is greater than 25:
5.         Subtract 26 from the index.
6.     Concatenate the char and processed string.
7. Return the processed string
```

(b) Annotator B

```
Input: in_str, String
Output: out_str, String
1. Initialize an alphabet and a bias.
2. Initialize out_str as "".
3. for char in in_str:
4.     char.index += bias.
5.     if char.index > 25:
6.         char.index -= 26.
7.     Append the char into out_str.
8. Return out_str
```

(c) Annotator C

Fig. 5. Three SCoTs written by three annotators. They show different writing styles.

we obtain three groups of examples. The numbers and writing styles of these examples are the same. Then, we apply different groups of examples to LLMs and measure their performance.

❹ The number of examples. It is well known that more examples can improve the performance of LLMs in downstream tasks [42]. In this setting, we gradually increase the number of examples (*i.e.,* from 1 to 5) and observe the performance of SCoT prompting. All examples are annotated by the same annotator, and the ordering is determined randomly.

Metrics. In the first three settings ❶-❸, we compute the Pass@1 of prompting approaches with different groups of examples. For ease of analysis, we report the variances of Pass@1. The lower the variance, the more robust the approach is to the examples. For the fourth setting ❹, we show the Pass@1 scores of prompting

Table 5. The variances ↓ of Pass@1 of different prompting approaches under different example seeds.

LLM	HumanEval			MBPP			MBCPP		
	Few-shot	CoT	SCoT	Few-shot	CoT	SCoT	Few-shot	CoT	SCoT
gpt-4-turbo	0.77	0.46	**0.16**	0.92	0.56	**0.26**	1.12	0.62	**0.27**
gpt-3.5-turbo	1.02	0.67	**0.29**	1.31	0.74	**0.32**	1.37	0.81	**0.39**
DS-Coder-Ins-33B	1.24	0.82	**0.31**	1.53	0.84	**0.36**	1.61	0.98	**0.46**
DS-Coder-Ins-6.7B	1.43	0.88	**0.34**	1.68	0.89	**0.41**	1.72	1.15	**0.55**
DS-Coder-Ins-1.3B	1.67	0.93	**0.39**	1.84	0.93	**0.44**	1.87	1.31	**0.63**

Table 6. The variances ↓ of Pass@1 of different prompting approaches under different writing styles.

LLM	HumanEval		MBPP		MBCPP	
	CoT	SCoT	CoT	SCoT	CoT	SCoT
gpt-4-turbo	0.39	**0.13**	0.43	**0.17**	0.28	**0.11**
gpt-3.5-turbo	0.58	**0.22**	0.61	**0.24**	0.54	**0.22**
DS-Coder-Ins-33B	0.64	**0.27**	0.68	**0.27**	0.57	**0.25**
DS-Coder-Ins-6.7B	0.67	**0.31**	0.70	**0.29**	0.59	**0.25**
DS-Coder-Ins-1.3B	0.67	**0.32**	0.71	**0.30**	0.59	**0.26**

Table 7. The variances ↓ of Pass@1 of different prompting approaches under different example ordering.

LLM	HumanEval			MBPP			MBCPP		
	Few-shot	CoT	SCoT	Few-shot	CoT	SCoT	Few-shot	CoT	SCoT
gpt-4-turbo	0.13	0.10	**0.06**	0.14	0.12	**0.07**	0.15	0.14	**0.11**
gpt-3.5-turbo	0.16	0.14	**0.08**	0.16	0.15	**0.10**	0.20	0.19	**0.15**
DS-Coder-Ins-33B	0.22	0.15	**0.10**	0.18	0.17	**0.10**	0.23	0.21	**0.16**
DS-Coder-Ins-6.7B	0.23	0.17	**0.12**	0.20	0.19	**0.13**	0.25	0.23	**0.17**
DS-Coder-Ins-1.3B	0.26	0.17	**0.12**	0.23	0.20	**0.14**	0.25	0.23	**0.19**

approaches with different numbers of examples. Because zero-shot prompting does not require demonstration examples, we omit it in this RQ.

Results. Table 5, 6, 7 show the variances of Pass@1 of prompting approaches in different settings. SCoT prompting substantially outperforms CoT prompting in four settings. It shows that SCoT prompting is more robust to samples compared with baselines. We also notice slight variances in the performance of SCoT prompting under different settings. It is expected for prompting techniques using examples. Similar variances can be found in existing approaches, and SCoT prompting still outperforms CoT prompting in different settings.

Figure 6 shows the average Pass@1 of prompting approaches with the different numbers of examples. **SCoT prompting outperforms baselines with different numbers of examples.** The results show the superiority of SCoT prompting. Besides, when the number of examples exceeds three, the improvements of Pass@1 are very

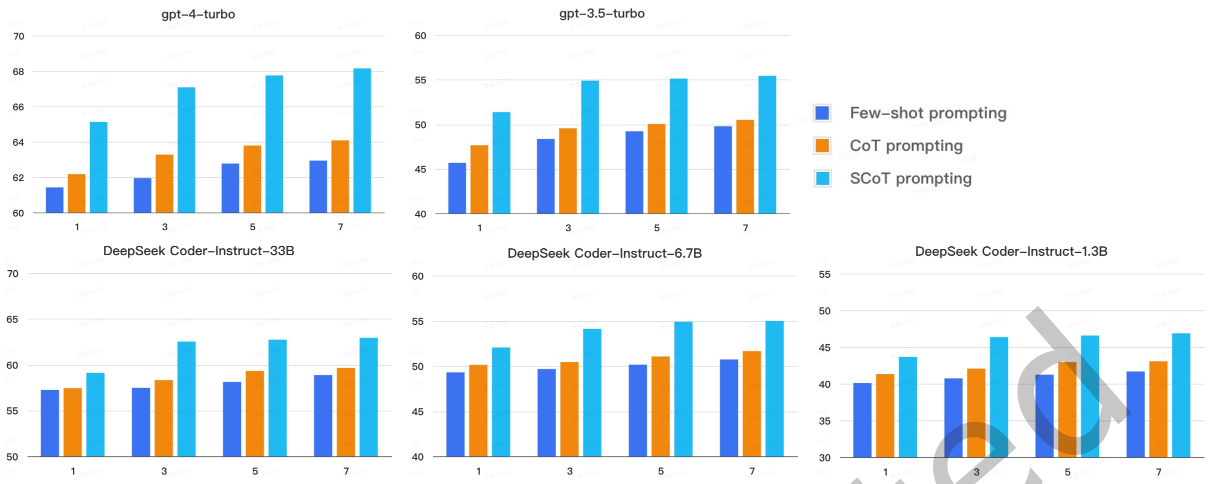

Fig. 6. The average Pass@1 of prompting approaches with the different numbers of examples.

slight. Considering that more examples will decrease inference efficiency, this paper sets the number of examples to 3 by default.

Answer to RQ3: Compared to baselines, SCoT prompting is more robust to examples, including example seeds, writing styles, example orderings, and the number of examples.

4.4 RQ4: What are the contributions of different programming structures in SCoT prompting?

As stated in Section 2.1, SCoT prompting introduces basic structures (*i.e.,* sequential, branch, and loop structures) and the input-output structure. This RQ is designed to analyze the contributions of different programming structures.

Setup. We conduct an ablation study by independently removing basic structures and the input-output (IO) structure. ❶ Removing branch and loop structures. In this setting, we use a CoT with an IO structure as the intermediate steps. Because the intermediate steps (*e.g.,* CoTs) naturally contain sequence structures that can not be removed, this setting mainly removes branch and loop structures. ❷ Removing IO structures. The SCoT contains a problem-solving process with basic structures without input-output parameters.

Results. The results are shown in Table 8. "w/o" is the abbreviation of "without".

Basic structures are beneficial to design a feasible solving process. In Table 8, after removing branch and loop structures, the performance of SCoT prompting drops obviously. We carefully inspect failed cases and find that LLMs benefit from using basic structures to clearly write a solving process. Figure 7 shows the intermediate steps of SCoT prompting and SCoT prompting without basic structures. SCoT prompting without basic structures uses CoTs, which sequentially describe how to write the code line by line and contain many ambiguities. For example, the scopes of two iterations on lines 2 and 4 are unclear. LLMs are likely to misunderstand the CoT and generate incorrect code. In contrast, SCoT prompting uses three basic structures to describe the solving process. The SCoT is clear and similar to the code, which benefits the following code implementation.

IO structures benefit the requirement understanding. In Table 8, after deleting the IO structure, the performance of SCoT prompting has a slight decrease. We analyze failed cases and think the IO structure benefits the understanding of requirements. Figure 8 shows two programs from SCoT prompting and SCoT prompting

Table 8. The results of ablation study. DS-Coder-Ins: DeepSeek Coder-Instruct. w/o: without

LLMs	Prompting	HumanEval			MBPP			MBCPP		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
gpt-4-turbo	CoT prompting	78.29	87.11	89.33	53.65	62.43	65.32	57.89	70.07	73.16
	SCoT prompting	**82.67**	**89.75**	**92.43**	**57.13**	**66.15**	**70.43**	**61.44**	**73.58**	**77.52**
	w/o Branch+Loop	79.44	87.39	89.84	54.46	63.79	66.14	59.04	71.38	74.33
	w/o IO	81.79	89.24	91.72	56.73	65.76	69.48	60.52	72.23	76.43
gpt-3.5-turbo	CoT prompting	53.29	69.76	75.52	41.83	51.04	54.57	53.51	63.84	67.03
	SCoT prompting	**60.64**	**73.53**	**77.32**	**46.98**	**55.31**	**58.36**	**57.06**	**65.70**	**68.70**
	w/o Branch+Loop	55.67	70.94	76.13	43.36	53.64	56.57	54.79	64.32	67.77
	w/o IO	59.65	72.79	77.12	46.13	54.76	57.88	56.61	65.01	68.42
DS-Coder-Ins-33B	CoT prompting	74.97	87.05	89.87	48.85	58.17	61.65	51.12	65.77	69.20
	SCoT prompting	**79.50**	**89.12**	**91.24**	**52.79**	**61.67**	**65.44**	**55.27**	**69.81**	**72.44**
	w/o Branch+Loop	75.65	87.97	90.33	49.39	59.36	63.03	52.19	66.73	70.44
	w/o IO	78.46	88.58	90.12	51.63	60.69	64.73	54.55	69.12	71.74
DS-Coder-Ins-6.7B	CoT prompting	67.71	83.49	87.31	43.90	54.31	58.10	39.71	59.66	64.15
	SCoT prompting	**71.06**	**87.81**	**90.33**	**47.69**	**58.71**	**62.11**	**43.58**	**63.70**	**67.40**
	w/o Branch+Loop	68.39	84.55	88.11	44.63	55.67	58.93	40.64	60.35	65.67
	w/o IO	70.32	87.15	89.83	46.56	57.87	61.80	42.79	63.04	66.59
DS-Coder-Ins-1.3B	CoT prompting	59.81	76.39	81.12	36.22	47.35	51.86	30.16	45.12	51.79
	SCoT prompting	**64.05**	**81.68**	**85.08**	**40.74**	**50.19**	**53.97**	**34.26**	**48.19**	**55.73**
	w/o Branch+Loop	60.37	77.86	82.33	37.54	48.66	52.75	31.65	46.33	52.12
	w/o IO	63.44	80.92	84.64	40.10	49.88	53.21	33.67	47.72	55.17

without the IO structure. We can see that SCoT prompting without the IO structure wrongly understands the output format and generates an incorrect program. After adding the IO structure, LLMs first reason about the input-output format and correctly return a boolean value.

Answer to RQ4: Basic and input-output structures contribute to the performance of SCoT prompting. After removing basic structures, the Pass@1 of SCoT prompting decreases by up to 8.2%. After removing input-output structures, the Pass@1 of SCoT prompting decreases by up to 2.37%.

5 DISCUSSION

5.1 SCoT *vs.* Pseudocode

We notice that the SCoT is similar to the pseudocode. The SCoT and pseudocode both contain an input-output structure and a problem-solving process. We randomly select 100 generated SCoTs and manually review them. We find that 26% of SCoTs are very close to the pseudocode. On one hand, we think the similarity enhances the usability of our approach. For example, users can quickly determine a program's behavior based on its SCoT. The SCoT can also be inserted into the comment and benefits future maintenance. On the other hand, the majority of SCoTs (74%) differ from the pseudocode because they are more abstract. Specifically, SCoTs tend to use natural languages to summarize an operation, *e.g.*, `calaluate the sum of list1`. But the pseudocode contains more implementation details, *e.g.*, `sum ← 0; for i in list1: sum ← sum + i;`.

Compared to the pseudocode, we think the SCoT is a better choice for intermediate steps. Because a SCoT naturally decomposes code generation into two steps. LLMs first focus on exploring diverse solutions and then implement a program in a standardized way. To validate this point, we design a variant of SCoT prompting,

SCoT prompting without basic structures:

```
Input: arry: list[list]
Output: result: int or float
1. Initialize a result with -999999
2. Iterate through the list of lists
3. Calculate the sum of the list
4. Update the result with the maximum of sum
and result
5. Return the result
```

SCoT prompting:

```
Input: arry: list[list]
Output: result: int or float
1: Initialize a result with -999999
2: for _list in the list of lists:
3:   Calculate the sum of the _list
4:   Update the result with the maximum of
sum and result
5: return the result
```

Fig. 7. The comparison of SCoT prompting and SCoT prompting without basic structures.

Table 9. The comparison of SCoT-P prompting and SCoT prompting. The numbers in red denote SCoT prompting's relative improvements compared to SCoT-P prompting.

Approach	HumanEval			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
CoT prompting	53.29	69.76	75.52	41.83	51.04	54.57	53.51	63.84	67.03
SCoT-P prompting	55.23	70.33	75.94	43.28	52.16	55.77	54.25	64.09	67.78
SCoT prompting	**60.64**	**73.53**	**77.32**	**46.98**	**55.31**	**58.36**	**57.06**	**65.70**	**68.70**
Relative Improvement	**9.80%**	**4.55%**	**1.82%**	**8.55%**	**6.04%**	**4.64%**	**5.18%**	**2.51%**	**1.36%**

named SCoT-P prompting. It deletes programming structure-related descriptions from instructions and considers three human-written pseudocode snippets as intermediate steps. Figure 9 shows a prompt of SCoT-P prompting. We apply SCoT-P prompting and SCoT prompting to gpt-3.5-turbo and measure their accuracy. The results are shown in Table 9. SCoT prompting substantially outperforms SCoT-P prompting on three benchmarks. The improvements show the superiority of our SCoT.

5.2 SCoT prompting *vs.* Post-processing Techniques

Some recent studies [6, 15, 41] propose *post-processing techniques* to improve the performance of LLMs on code generation. Given a requirement, they first sample many programs from LLMs and then use test cases or neural networks to post-process sampled programs. For example, CodeT [6] is a popular post-processing technique. CodeT does large-scale sampling and executes sampled programs on auto-generated test cases. Based on execution

SCoT prompting without IO structure:

```
def test_duplicate(arraynums):
  num_set = set(arraynums)
  if len(num_set) < len(arraynums):
    print('Find duplicate elements‘)
  else:
    print('No duplicate elements')
```

SCoT prompting:

```
def test_duplicate(arraynums):
  # Input: arraynums, a list of integers
  # Output: True if exist duplicate element,
False otherwise
  num_set = set(arraynums)
  if len(num_set) < len(arraynums):
    return True
  else:
    return False
```

Fig. 8. The comparison of SCoT prompting and SCoT prompting without the IO structure.

results, the programs are reranked. In this paper, we do not directly compare our approach to rank techniques due to two reasons.

SCoT prompting and post-processing techniques have different focuses, and they are complementary. Our work aims to design a new prompting technique and improve the accuracy of LLMs in code generation. Post-processing techniques do not care about LLMs and aim to refine the outputs of LLMs. In practice, users can use SCoT prompting to generate many programs and then leverage post-processing techniques to get the final code.

To verify the complementarity between SCoT prompting and post-processing techniques, we conduct an exploratory experiment. We select gpt-3.5-turbo as a base model and progressively introduce CodeT and SCoT prompting. The results on MBPP are shown in Figure 10. We can see that the performance of gpt-3.5-turbo is continually improved by adding CodeT and SCoT prompting.

Post-processing techniques rely on execution environments. Post-processing techniques require executing programs on test cases and using execution results to rerank programs. In many realistic programming scenarios, users want to get code suggestions for an unfinished project. It is infeasible to execute auto-generated programs. Thus, we think rank techniques have limited application scenarios and make additional use of the execution results. Our approach works in a general scenario and does not use execution results. Thus, it is unfair to directly compare SCoT prompting to rank techniques.

5.3 Threats to Validity

There are three main threats to the validity of our work.

❶ **The generalizability of experimental results.** To mitigate this threat, we carefully select the benchmarks, metrics, and baselines. Following previous studies [1, 2, 7], we pick three representative code generation benchmarks. They are hand-written or collected from real-world programming communities, and cover two popular

```
Your task is to complete the following code. You should first write a
rough problem-solving process then output the final code.

Here are some demonstration examples:
```

```
def sum_Of_Primes(n):
    """
    Write a python function to find sum of prime numbers between 1 to n.
    """
    # Let's think step by step
    # 1. prime <- [True] * (n + 1)
    # 2. p <- 2
    # 3. while p * p <= n:
    # 4.   if prime[p] == True:
    # 5.     for each i in range(p * 2, n + 1, p):
    # 6.       prime[i] <- False
    # 7.   p += 1
    # 8. sum <- 0
    # 9. for each i in range(2, n + 1):
    # 10.   if prime[i]:
    # 11.     sum += i
    # 12. return sum
    prime = [True] * (n + 1)
     (more lines…)
(more examples…)
```

```
Input code:
def text_lowercase_underscore(text):
    """
    Write a function to find sequences of lowercase letters joined with
an underscore.
    """
    # Let's think step by step
```

Fig. 9. The prompt of SCoT-P prompting.

languages (*i.e.*, Python and C++). For evaluation metrics, we select a widely used metric - Pass@k, which utilizes test cases to check the correctness of programs. We use the unbiased Pass@k which is more reliable [7]. For comparison baselines, we select the SOTA prompting techniques and conduct a comprehensive comparison in Section 4. SCoT prompting and baselines have the same example seeds and maximum generation lengths.

❷ **The design of prompts.** Existing work [42] found that LLMs are sensitive to the design of prompts (*e.g.*, natural language instructions and demonstration examples). In our prompts, we focus on exploring the SCoT and set other factors constant. Therefore, there may be more effective prompts to implement SCoT prompting, *e.g.*, clearer natural language instructions, and better examples. These investigations are beyond the scope of this paper and we leave them to future work.

❸ **Data leakage.** Existing LLMs are trained with extensive code files from open-source communities. Their training data may contain the experimental benchmarks, leading to data leakage. However, we think that it does not affect the fairness of our experiments. In this paper, we select a specific LLM (*e.g.*, gpt-3.5-turbo) as the base model and apply different prompting techniques to it. Thus, the reported relative improvements between baselines and our approach are credible. In the future, we will add the latest benchmarks to alleviate this threat.

6 RELATED WORK

Large language models (LLMs) for Source Code are large-scale neural networks that are pre-trained with a large corpus consisting of natural language text and source code. Nowadays, LLMs for source code have been expanding and can be divided into two categories: foundation models and instruction-tuned models.

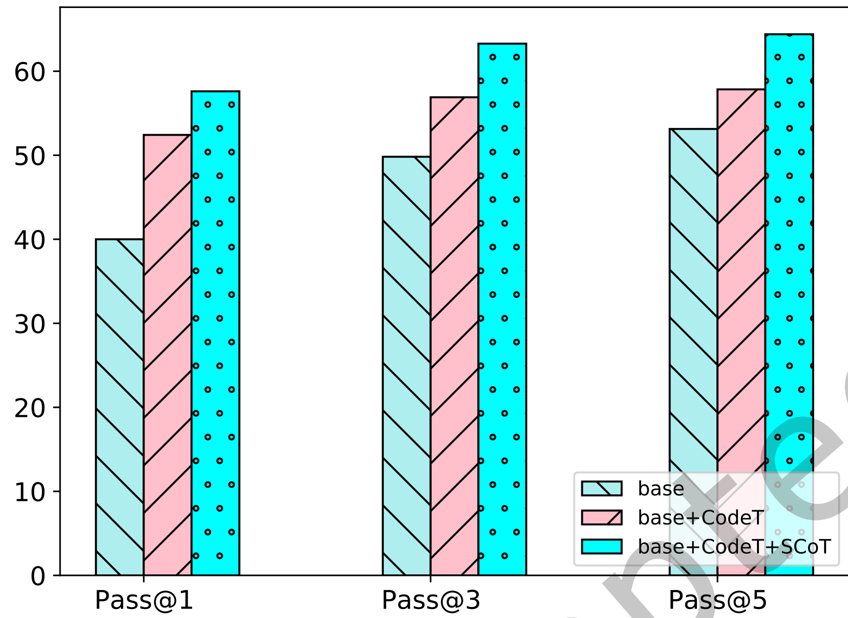

Fig. 10. The complementarity between CodeT and SCoT prompting.

Foundation models are pre-trained on a large-scale corpus with the next-token prediction objective. They are mainly used to continually complete the given context, such as code completion. After the success of GPT series [4, 29, 30] in NLP, OpenAI fine-tunes GPT models on code to produce closed-source Codex (*i.e.,* code-davinci-002) [7]. There follow many open-source replication attempts, *e.g.,* CodeParrot [33], CodeGen [23], CodeGeeX [43], InCoder [10], StarCoder [21] and CodeT5+ [37].

Instruction-tuned models are models after instruction tuning [38]. Instruction tuning trains models to understand human users' instructions and perform tasks by following instructions. gpt-3.5-turbo [25] is trained with human feedback [27], powerful on both natural language tasks and programming tasks. Many attempts to train an "open-source gpt-3.5-turbo". Alpaca [32] is LLaMA [34] tuned using self-instruct [36] and gpt-3.5-turbo's feedback. Code Alpaca [5] is LLaMA tuned using self-instruct and gpt-3.5-turbo's feedback, with instructions focusing on programming tasks. WizardCoder [22] is StarCoder [21] tuned using Evol-Instruct [40] and gpt-3.5-turbo's feedback with Code Alpaca's dataset as seed dataset. InstructCodeT5+ [37] is CodeT5+ [37] tuned on Code Alpaca's dataset.

Prompting Techniques. With the enormous number of parameters (*e.g.,* code-davinci-002: 175 billion parameters), it is hard to directly fine-tune LLMs on code generation. *Prompting techniques* are a popular approach, which leverages LLMs to generate code by inputting a special prompt.

Early, researchers proposed zero-shot prompting and few-shot prompting. Zero-shot prompting concatenates a task instruction (*e.g.,* `please generate a program based on the requirement`) and a requirement together, making a prompt. Based on the zero-shot prompting, few-shot prompting further adds several ⟨requirement, code⟩ pairs to the prompts, so that LLMs can learn code generation from given examples.

The Chain-of-Thought (CoT) prompting [39] is a recently proposed prompting technique. CoT prompting asks LLMs first to generate CoTs (*i.e.,* intermediate natural language reasoning steps) and then output the final code. It allows LLMs to first design a solving process that leads to the code. CoT prompting has achieved the SOTA results in natural language generation and sparked lots of follow-up research, such as self-consistency prompting [35], least-to-most prompting [44]. But these prompting techniques are designed for natural language generation and bring slight improvements in code generation.

In this paper, we propose a novel prompting technique named Structured Chain-of-Thought (SCoT) prompting. Different from standard CoT prompting, SCoT prompting explicitly introduces programming structures and asks LLMs to generate intermediate reasoning steps with programming structures. We compare CoT prompting and SCoT prompting in Section 4. The results show that SCoT prompting significantly outperforms CoT prompting in three benchmarks.

7 CONCLUSION AND FUTURE WORK

Large Language Models (LLMs) with Chain-of-Thought (CoT) prompting is the state-of-the-art (SOTA) approach to generating code. It first generates a CoT and then outputs the code. A CoT is several intermediate natural language reasoning steps. However, CoT prompting still has low accuracy in code generation. This paper proposes a Structured CoT (SCoT) and presents a new prompting technique for code generation, named SCoT prompting. SCoT prompting asks LLMs to generate a SCoT using programming structures (*i.e.,* sequential, branch, and loop structures). Then, LLMs generate the code based on the SCoT. A large-scale study on three benchmarks shows that SCoT prompting significantly outperforms CoT prompting in Pass@k and human evaluation. Besides, SCoT prompting is robust to examples and obtains stable improvements.

In the future, we will explore new prompting techniques for code generation. For example, source code can be represented by a tree (*e.g.,* abstract syntax tree). We can design a tree-based prompting technique, which uses LLMs to generate a tree.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China (Nos. 62192731, 62152730), the National Key R&D Program (No. 2023YFB4503801), the National Natural Science Foundation of China (Nos. 62072007, 62192733, 61832009, 62192730), and the Major Program (JD) of Hubei Province (No.2023BAA024). Ge Li is the corresponding author.

REFERENCES

[1] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net. https://openreview.net/forum?id=Bo7eeXm6An8

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[3] Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (1966), 366–371. https://doi.org/10.1145/355592.365646

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[5] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

[6] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net. https://openreview.net/forum?id=ktrw68Cmu9c

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[8] Anna Eckerdal, Michael Thuné, and Anders Berglund. 2005. What does it take to learn 'programming thinking'?. In *International Computing Education Research Workshop 2005, ICER '05, Seattle, WA, USA, October 1-2, 2005*, Richard J. Anderson, Sally Fincher, and Mark Guzdial (Eds.). ACM, 135–142. https://doi.org/10.1145/1089786.1089799

[9] Martin Fowler. 2002. Refactoring: Improving the Design of Existing Code. In *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2418)*, Don Wells and Laurie A. Williams (Eds.). Springer, 256. https://doi.org/10.1007/3-540-45672-4_31

[10] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net. https://openreview.net/forum?id=hQwb-lbM6EL

[11] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023.* IEEE, 761–773. https://doi.org/10.1109/ASE56229.2023.00109

[12] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024). https://doi.org/10.48550/ARXIV.2401.14196 arXiv:2401.14196

[13] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. *CoRR* abs/2206.13179 (2022). https://doi.org/10.48550/arXiv.2206.13179 arXiv:2206.13179

[14] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net. https://openreview.net/forum?id=rygGQyrFvH

[15] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems* 35 (2022), 13419–13432.

[16] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. CodeEditor: Learning to Edit Source Code with Pre-trained Models. *ACM Trans. Softw. Eng. Methodol.* 32, 6 (2023), 143:1–143:22. https://doi.org/10.1145/3597207

[17] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *CoRR* abs/2404.00599 (2024). https://doi.org/10.48550/ARXIV.2404.00599 arXiv:2404.00599

[18] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. In *Findings of the Association for Computational Linguistics ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand and virtual meeting, 3603–3614. https://aclanthology.org/2024.findings-acl.214

[19] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 2124–2135. https://doi.org/10.1109/ICSE48619.2023.00179

[20] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. AceCoder: An Effective Prompting Technique Specialized in Code Generation. *ACM Trans. Softw. Eng. Methodol.* (jul 2024). https://doi.org/10.1145/3675395 Just Accepted.

[21] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun

Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=KoFOg41haE

[22] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=UnUwSIgK5W

[23] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=iaYcJKpY2B_

[24] OpenAI. 2023. gpt-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5.

[25] OpenAI. 2023. gpt-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.

[26] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[27] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[28] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[29] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018). https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

[30] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019). https://insightcivic.s3.us-east-1.amazonaws.com/language-models.pdf

[31] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. CommonsenseQA: A Question Answering Challenge Targeting Commonsense Knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4149–4158. https://doi.org/10.18653/v1/n19-1421

[32] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[33] CodeParrot Team. 2022. CodeParrot. https://huggingface.co/codeparrot/codeparrot.

[34] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). https://doi.org/10.48550/ARXIV.2302.13971 arXiv:2302.13971

[35] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=1PL1NIMMrw

[36] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 13484–13508. https://aclanthology.org/2023.acl-long.754

[37] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 1069–1088. https://doi.org/10.18653/V1/2023.EMNLP-MAIN.68

[38] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=gEZrGCozdqR

[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=_VjQlMeSB_J

[40] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. WizardLM: Empowering Large Pre-Trained Language Models to Follow Complex Instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=CfXh93NDgH

[41] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 769–787.

https://doi.org/10.18653/v1/2023.acl-long.45

[42] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 12697–12706. http://proceedings.mlr.press/v139/zhao21c.html

[43] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *CoRR* abs/2303.17568 (2023). https://doi.org/10.48550/ARXIV.2303.17568 arXiv:2303.17568

[44] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=WZH7099tgfM