Chain-of-Thought in Neural Code Generation: From and For Lightweight Language Models

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, Taolue Chen

Abstract—Large Language Models (LLMs) have demonstrated remarkable potential in code generation. The integration of Chain of Thought (CoT) reasoning can further boost their performance. However, current CoT methods often require manual writing or LLMs with over 100 billion parameters to generate, impeding their applicability in resource-constrained scenarios. In this study, we investigate lightweight Language Models (ℓLMs), which are defined to have fewer than 10 billion parameters. Empirically, we find that most ℓLMs cannot generate high-quality CoTs when prompted by the few-shot method, but can take advantage of high-quality CoTs generated elsewhere to improve their performance in code generation. Based on these findings, we design a novel approach COTTON which can leverage ℓLMs to automatically generate CoTs for code generation. We synthesize new datasets and conduct extensive experiments on various benchmarks. The results show that the CoTs generated by COTTON outperform the baselines in terms of automated and human evaluation metrics. In particular, the CoTs generated by COTTON boost various ℓLMs to achieve higher performance gains than those generated by LLMs such as ChatGLM (130B), and are competitive with those generated by Gemini and gpt-3.5-turbo. The results also reveal that COTTON not only improves the performance of ℓLMs, but also enhances the performance of LLMs. Our study showcases the potential of ℓLMs in software engineering applications.

Index Terms—Code Generation, Chain-of-Thought, Large Language Model, Lightweight Language Model, Program Language Processing

✦

1 INTRODUCTION

Neural code generation, which can automatically generate programs from natural language requirements based on deep learning, has become a promising approach to meet the challenges of the ever-increasing complexity of software and alleviate the burden on programmers [1], [2]. Recently, large language models (LLMs), such as GPT4 [3], have demonstrated impressive performance in code generation tasks [4]. The state-of-the-art LLMs normally have over 100 billion parameters, making even their deployment highly non-trivial. These LLMs pose challenges in terms of time, computational, and financial costs when applied to code generation, rendering them impractical for most individual users, or in resource-constrained scenarios, such as restricted access to LLM APIs or constrained GPU availability. [5], [6]. For software engineering applications, it is imperative to develop *lightweight* language-model-based techniques that are more friendly for users (e.g., individual end users). Fu et

- *Guang Yang is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. E-mail: novelyg@outlook.com*
- *Yu Zhou (Corresponding author) is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. E-mail: zhouyu@nuaa.edu.cn*
- *Xiang Chen is with the School of Information Science and Technology, Nantong University, China. E-mail: xchencs@ntu.edu.cn*
- *Xiangyu Zhang is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. E-mail: zhangx1angyu@nuaa.edu.cn*
- *Terry Yue Zhuo is with Monash University and CSIRO's Data61. E-mail: terryzhuo25@gmail.com*
- *Taolue Chen (Corresponding author) is with School of Computing and Mathematical Sciences, Birkbeck, University of London, UK. E-mail: t.chen@bbk.ac.uk*

al. [7] defined models with parameters greater than 100B as large models and those with parameters less than 10B as small models. Admittedly, the precise definition of large and small models is debatable and may evolve with the advance of technology. In this study, we define (pre-trained) language models (LMs) with less than 10 billion parameters as *lightweight Language Models* (ℓLM), the rationale of which is that these models can be deployed on a single user graphics card (e.g., RTX 3090 or RTX 4090) based on the current technology. The general aim is to develop techniques to tackle software engineering challenges based on ℓLMs but with competitive performance as state-of-the-art LLMs, which would enable efficient, yet more accessible, software engineering applications.

Recent studies [8]–[11] have highlighted the importance of enhancing LLM performance by providing adequate information in the prompts. To improve LLMs without retraining or fine-tuning, researchers have resorted to Chain of Thought (CoT) techniques [12]. A CoT is, in a nutshell, a series of intermediate natural language reasoning steps that lead to the final output, which enables LLMs to provide more reliable answers through thoughtful consideration and explanation. CoT techniques have shown effectiveness in logical reasoning tasks by breaking them down into understandable intermediate steps, enabling LLMs to handle each step individually. This process not only enhances model performance but also offers the potential for model interpretability.

Inspired by the success of CoT techniques in logical reasoning, researchers have explored their application in the code generation task. For example, Jiang et al. [13] proposed a self-planning approach. Li et al. [14] introduced a structured CoT approach to assist models in understanding

(a) Evaluation on ℓLMs without chain-of-thought
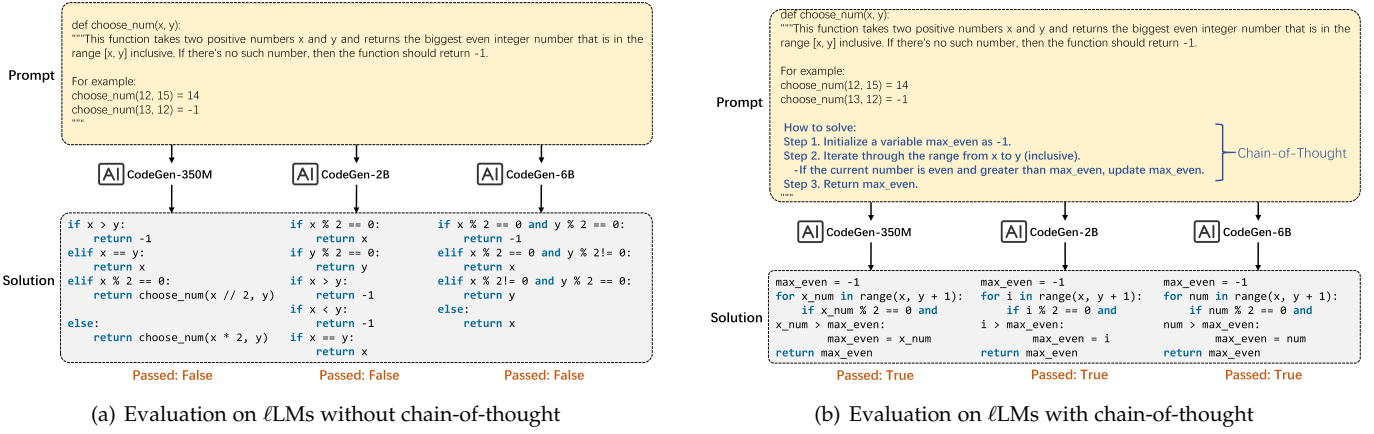
(b) Evaluation on ℓLMs with chain-of-thought

Fig. 1. The motivating examples illustrating the potential of using chain-of-thought for ℓLMs in code generation

complex intentions and reducing problem-solving difficulties. Zhuo [15] introduced an evaluation metric for code generation based on LLMs and demonstrated that CoT can enhance evaluation reliability.

The previous research has primarily focused on investigating the impact of CoT on LLMs, leaving questions regarding whether ℓLMs can also benefit from the guidance of CoT. In Fig. 1, we present a motivation example to demonstrate the potential of CoT for ℓLMs in code generation. Specifically, in Fig.1(a), the programming task is `choose_num`, which takes two positive numbers x and y and returns the largest even integer that falls into the interval $[x, y]$. The example highlights that the original prompts for the ℓLMs (CodeGen-350M, CodeGen-2B, and CodeGen-6B) fail to generate correct code solutions. However, by leveraging CoT in Fig.1(b), we modify the original prompt by using "How to solve:" and breaking down the problem into multiple steps, where the natural language explanations guide the model's understanding of the task, including instructions on branching and looping structures. With the new CoT, these ℓLMs can generate correct code solutions.

Furthermore, previous studies possess certain limitations as the current methods for CoT generation heavily rely on manual writing of CoTs or the utilization of LLMs [16], [17], leading to high costs. These limitations motivate us to investigate the following two main questions. (1) Can ℓLMs independently generate high-quality CoTs to guide code generation, and (2) can ℓLMs benefit from generated CoTs? Here, "independently" means no model training or model parameter updating.

Empirical observations. To address the first question, we conduct empirical studies on the CoT generation capabilities of 11 different ℓLMs and two LLMs. We adopt a zero-shot approach [18] and some few-shot approaches (such as Self-planning [13], SCoT [14], and the self-cot we propose), which provide ℓLMs with a set of examples to generate the corresponding CoT. Our finding shows that most ℓLMs with parameter scales ranging from 0.3 to 7 billion, unfortunately, did *not* demonstrate the ability to generate high-quality CoTs independently (cf. Section 5.1 for details). To address the second question, we compare the performance of ℓLMs in code generation *with* and *without* CoTs. Our findings suggest that all ℓLMs obtain performance improvement with

the CoTs. As an example, the performance of the CodeT5 + 6B model on the HumanEval-plus dataset [4] can be improved from 26.83% to 43.90% with the CoTs generated by our methods (cf. Section 5.3 for details).

Fig. 1 provides a motivation example, where Code-Gen [19] is used as a case study. We evaluate its performance by considering varying parameter sizes 350M, 2B, and 6B. Without CoT, these models did not generate the correct code (cf. Fig 1(a)). However, with CoT, we decompose user requirements into three intermediate steps. In the first step, we initialize a variable `max_even` as -1; in the second step, we define the details of loop conditions and judgment conditions; in the third step, we return the value. As such, we can effectively instruct the models on the necessary actions at each step, and they eventually generate semantically correct code (though these variables have different names).

Technical contributions. Based on the empirical observations, a natural question is how to enable ℓLMs to generate meaningful CoTs for code generation. To this end, we design a novel approach COTTON (**C**hain **O**f **T**hough**T** c**O**de ge**N**eration). Specifically, COTTON contains data collection, model training, and model inference steps. To build the corpus, we first mine shared open source datasets (such as TheVault [20]) to gather pairs of natural language and programming language. Then, we improve the dataset quality by using carefully designed heuristic cleaning rules. To ensure the quality of CoTs in the corpus, we use ChatGPT as the base agent and propose a multi-agent alignment method to construct high-quality CoTs (details in Section 3.1). Finally, our collected CodeCoT-9k consists of 9,264 data pairs.

For model training, we employ CodeLlama-7b[1] as the base model to generate CoTs automatically based on the given prompt. CodeLlama-7b incorporates advanced techniques (such as RMSNorm [21] and Group Query Attention [22]), which enhance its performance beyond the Transformer [23]. By applying these techniques, we can further improve the performance of COTTON. To reduce the training cost, we adopt instruction-tuning and LoRA techniques [24] to fine-tune the model parameters. This approach allows COTTON to be trained efficiently on a single consumer graphics card while maintaining its performance.

1. https://github.com/facebookresearch/codellama

Evaluation. We conduct a comprehensive evaluation of the quality of the CoTs generated by COTTON on the HumanEval benchmark [25]. To ensure the generalizability of COTTON, we further collected a new code generation dataset OpenEval, and evaluated COTTON on the OpenEval benchmark as well. Specifically, we selected nine other commonly used models as base models and compared the results with the same training process. The quality of CoTs generated by COTTON is superior to others in both automated and human evaluation metrics. Furthermore, we evaluate the performance on ℓLMs when adopting the generated CoTs on code generation benchmarks (such as HumanEval, HumanEval-plus, and OpenEval). The results show that, for various ℓLMs, the CoTs generated by COTTON achieve higher performance gains than those generated by LLMs such as ChatGLM (130B) [26], and are competitive with those generated by Gemini and gpt-3.5-turbo.

Taking the CodeT5+ 6B model as an example, we observed significant performance improvements on the HumanEval, HumanEval-plus, and OpenEval benchmarks by incorporating COTTON. For the pass@1 metric on HumanEval and HumanEval-plus, COTTON enhances the performance of the CodeT5+ 6B model from 26.22% and 26.83% to 42.68% and 43.90% respectively. In comparison, the LLM ChatGLM 130B only achieves an improvement of 36.59%. Similarly, on the OpenEval benchmark, COTTON boosts the pass@1 metric of the CodeT5+ 6B model from 20.22% to 35.39%. Meanwhile, the LLM ChatGLM 130B achieves an improvement of only about 32.02%.

In addition, to further evaluate the capabilities of COTTON, we conducted experiments to assess its impact on LLMs (such as gpt-3.5-turbo). The results show that gpt-3.5-turbo could show a significant performance improvement in code generation when guided by the CoTs generated by COTTON where the performance even exceeds that of the GPT-4 zero-shot scenario on the HumanEval dataset.

Finally, to demonstrate the effectiveness of COTTON compared to fine-tuning, we utilize the StarCoder-series models as examples. The results indicate that the combination of StarCoder-7B with COTTON has already exceeded the performance of StarCoder-16B in zero-shot scenarios and can even achieve comparable results to a fine-tuned StarCoder-16B model. The efficacy of COTTON in enhancing performance across multiple models without the necessity of fine-tuning individual model is noteworthy, as it not only saves time and computation resources for model construction but also opens an inviting avenue to avoid traditional fine-tuning in adapting language models.

In summary, the main contributions of our study can be summarized as follows:

- We empirically show that most existing ℓLMs lack the capability to generate high-quality CoT independently.
- We design a novel approach COTTON to generate high-quality CoTs for guiding code generation, the efficacy of which has been confirmed by extensive experiments on a comprehensive set of benchmarks.
- We construct a new dataset of high-quality CoTs, i.e., CodeCoT-9k, by mining existing open-source

datasets. Moreover, we also construct OpenEval[2], another dataset to benchmark the code generation performance. These datasets could be reused in similar software engineering tasks.

To facilitate the replication of COTTON, we make our source code, trained models, and datasets publicly available on GitHub.[3]

Structure of the paper. The rest of the paper is organized as follows. Section 2 provides preliminary knowledge related to our study. Section 3 describes the framework of COTTON and its key components. Section 4 and 5 present the experimental design and the result analysis respectively. Section 6 further discusses our approach followed by the related work review in Section 7. Section 8 concludes our study and outlines possible future directions.

2 PRELIMINARIES

In this section, we first formulate the code generation task. Then we provide the fundamental concepts of our used base model CodeLlama.

2.1 Code Generation Task Formulation

Let $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{|\mathcal{D}|}$ denote a code generation dataset, comprising $|\mathcal{D}|$ pairs (X_i, Y_i). Here, X_i represents the functional description, and Y_i represents the corresponding code snippet. Neural code generation model M_{code} aims to generate Y_i conditioned on X_i. This autoregressive generation process is parameterized by θ_{code}, and can be expressed as

$$P_{\theta_{code}}(Y_i|X_i) = \prod_{k=1}^{n} P_{\theta_{code}}(Y_{i,k}|X_i, Y_{i,1} : Y_{i,k-1})$$

where $Y_{i,1} : Y_{i,k-1}$ represents the previous sequence before the k-th token of Y_i, and n denotes the number of tokens in the target sequence Y_i.

To improve the code generation performance, we utilize a CoT generation model, denoted as M_{cot}, which generates high-quality CoT C_i based on X_i. Then the original input sequence X_i will be augmented by concatenating with the generated CoT C_i, resulting in a new input sequence $\hat{X}_i = X_i \oplus C_i$, where $\oplus$ denotes the concatenation operation. Subsequently, we approximate the probability of generating the code snippet Y_i given the input sequence X_i as

$$P(Y_i|X_i) \propto \underbrace{P_{\theta_{cot}}(C_i|X_i)}_{M_{cot}} \underbrace{P_{\theta_{code}}(Y_i|X_i, C_i)}_{M_{code}}$$

In our study, we treat the existing neural code generation model M_{code} as a black box. We intend to train a model M_{cot} to generate CoTs, which can be used to guide code generation and further improve the performance of this task.

---

2. https://github.com/NTDXYG/open-eval
3. https://github.com/NTDXYG/COTTON

2.2 Code Language Model CodeLlama

CodeLlama [27] is a code language model built on Llama-2 [28], which stands out for its exceptional performance in open models, padding capabilities, support for large input contexts, and zero-sample instruction tracking for programming tasks. In our study, we use CodeLlama-7B as the base model for COTTON due to its remarkable code understanding and generation capabilities.

Embedding Layer. CodeLlama tokenizes the given functional description X into sub token sequences $\{w_i\}_{i=1}^{N}$ using byte-pair encoding (BPE) and the SentencePiece algorithm [29]. Each sub-word w_i is then transformed into an embedding (row) vector $x_i \in \mathbb{R}^d$, where d represents the dimension of the embedding vector. These embedding vectors are combined into a matrix $X = \{x_i\}_{i=1}^{N}$, which represents the meaningful relationship between the tokens in the input sequence. By using this embedding matrix, COTTON captures the semantic information of the functional description and prepares it for further processing in the subsequent layers of the model.

RMSNorm. CodeLlama utilizes Root Mean Square Layer Normalization (RMSNorm) instead of LayerNorm for normalization purposes [21]. RMSNorm operates by normalizing each embedding vector x_i by dividing it through the root mean square. This normalization process helps reduce the impact of noise and improves computational efficiency. For each embedding vector x_i, the calculation formula of RMSNorm is defined as follows.

$$x_i = \frac{x_i}{RMS(X)} \cdot g_i$$

where $RMS(X) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$ represents the root mean square of the embedding matrix X, and g_i denotes the rescale factor.

Group Query Attention (GQA). CodeLlama introduces GQA [22] as a modification to the standard multi-head attention mechanism. This modification optimizes the model's performance by dividing the Query heads into groups, with each group sharing the same Key and Value matrix. Moreover, the model incorporates Rotary Position Embedding (RoPE) [30] and FlashAttention [31] for further improvement.

Specifically, for the given matrix X, the model computes the Query, Key, and Value matrix as follows.

$$q_i = f_q(x_i, i)$$

$$k_j = group(f_k(x_j, j))$$

$$v_j = group(f_v(x_j, j))$$

where q_i represents the Query vector of the embedding vector x_i, incorporating the position information. k_j and v_j denote the Key and Value vectors of the embedding vector x_j, respectively, incorporating the position information j. The grouping operation is applied to ensure that the Query heads within each group share the same Key and Value matrix. To compute the self-attention output corresponding to the i-th embedding vector x_i, an attention score is computed between q_i and the other k_j vectors. This attention

score is then multiplied by the corresponding v_j vectors and summed to obtain the output vector.

$$a_{i,j} = \frac{\exp\left(\frac{q_i^T k_i}{\sqrt{d}}\right)}{\sum_{m=1}^{N}\exp\left(\frac{q_i^T k_m}{\sqrt{d}}\right)} \qquad o_i = \sum_{j=1}^{N} a_{i,j} \cdot v_j$$

where $a_{i,j}$ represents the attention score and o_i denotes the output vector for the i-th embedding vector.

FFN. The Feed Forward Network (FFN) in CodeLlama consists of linear layers and an activation function. It operates on the matrix X to calculate the output using a specific formula.

$$\mathbf{FFN}(X) = f_{down}(f_{up}(X) \times SiLU(f_{gate}(X)))$$

where $SiLU$ represents the activation function, defined as the element-wise product of the Sigmoid function and the input. This activation function introduces non-linearity to the network.

During the overall process, CodeLlama generates the output probability P for a given input X through a series of operations, including Group Query Attention, RMSNorm, and FFN. Initially, the input X is normalized using RMSNorm. Then, GQA is applied to the normalized input and added to the original input, resulting in X_{hidden}. Next, X_{hidden} is again normalized using RMSNorm, and FFN is applied to it, obtaining X_{final}. Finally, X_{final} is normalized using RMSNorm and passed through the function f_{vocab} to map it to the output probability space, resulting in the final output probability P.

$$P = f_{vocab}(\mathbf{RMSNorm}(X_{final})))$$

3 OUR APPROACH

The workflow of our proposed COTTON is shown in Fig. 2. There are three major steps, i.e., data collection, model training, and model inference. In the rest of this section, we show the details of these three steps.

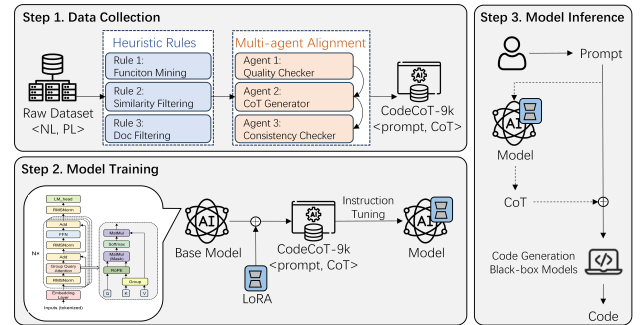Fig. 2. The workflow of the proposed approach COTTON

3.1 Data Collection

The construction of the CoT dataset CodeCoT-9k follows the process shown in Step 1 in Fig. 2. We begin by selecting TheVault[4], MBPP[5] and LeetCode[6] as our raw datasets.

4. https://github.com/FSoft-AI4Code/TheVault
5. https://huggingface.co/datasets/mbpp
6. https://huggingface.co/datasets/mhhmm/leetcode-solutions-python

These datasets consist of natural language descriptions of functional requirements paired with corresponding implementation code snippets. They have been widely used in the literature [32].

However, after our manual analysis, we find that a significant portion of the code snippets in these datasets are not self-contained, requiring external modules or files for program comprehension [33], which poses challenges for CoT generation. In addition, we notice that some code snippets consist of trivial or template code (for defining constants, setting parameters, configuring GUI elements, etc), which is not useful for CoT generation. As a result, to improve the quality of our dataset, we design two data cleaning methods, i.e., heuristic rule-based cleaning and multi-agent alignment-based cleaning.

Heuristic rule-based cleaning. These rules are designed to filter data that contains syntactically incorrect code and documents that are inconsistent with the code, in addition to avoiding data leakage problem. We define three heuristic rules as below.

- **R1** Code Filtering. We utilize the AST parser tool to extract method-level code and corresponding functional comments, by which syntactically incorrect code can be filtered.
- **R2** Doc Filtering. We utilize DocChecker [34] to determine the consistency between the documentation and the code, which can maintain an accurate alignment between the comments and the code effectively. We then remove code snippets with inconsistent documentation.
- **R3** Similarity Filtering. To prevent data leakage in the training set, We utilize the codet5p embedding model[7] for code representation learning. We then remove the code snippets that exceed the semantic similarity threshold by considering the cosine similarity.

Multi-agent alignment-based cleaning. These agents are designed to filter low-quality data, including snippets that are not educationally meaningful and CoTs that are inconsistent with the semantics of the code. We leverage the power of multiagents to align and clean the data, where they are based on gpt-3.5-turbo.[8] Specifically, we implement the agent definition by constructing the specific prompt [35]:

- **A1** **Quality Checker**. This agent assesses the educational value of the data and removes low-quality items, ensuring that the dataset comprises high-quality code snippets.
- **A2** **CoT Generator.** We first transform code snippets and functional comments into a standardized prompt and solution format similar to HumanEval [25]. This agent employs a one-shot approach, providing an example to aid the agent in learning the desired output style and generating CoT based on user input. Importantly, we intentionally do not disclose the specific implementation details of the code to the agent, which encourages the agent to generate diverse CoTs, as code implementation can vary widely.

7. https://huggingface.co/Salesforce/codet5p-110m-embedding
8. https://platform.openai.com/docs/models/gpt-3-5

- **A3** **Consistency Checker**. This agent examines the consistency between the CoT instructions generated by Agent 2 and the code snippets. It removes code snippets with inconsistencies, ensuring that the CoT instructions accurately reflect the code's behavior.

Quality Checker

Give you a code snippet, determine its educational value for a student whose goal is to learn basic coding concepts.
If it has educational value, return only "Yes", else return "No".

Consistency Checker

Given a piece of code and a chain of thought, determine whether they express exactly the same functional semantics.
If consistent, return only "Yes", else return "No".

CoT Generator

Given a piece of code, output the corresponding implementation idea.
Example:
Input:
from typing import List
def below_zero(operations: List[int]) ->bool:
""" You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account falls below zero, and at that point function should return True. Otherwise it should return False.
"""
Output:
How to solve:
Step 1. Initialize account balance as 0.
Step 2. Iterate through operations.
-add value to account balance.
-If account balance <0, return True.
Step 3. Return False.
Input: $[X]$
Output: $[Y]$

3.2 Model Training

To enable the training of ℓLMs with limited resources, researchers have explored parameter-efficient fine-tuning methods since full parameter fine-tuning is impractical in this scenario. As demonstrated in the previous study [36], these methods have been proven to achieve high performance by providing sufficient instructions to the language models. In contrast to continuous-based soft prompt methods [37], [38] , our approach employs a set of discrete tokens as instruction prompts, which are meaningful and easily interpretable. For our CoT generation task, we design a template that incorporates task-specific instructions, which is illustrated as follows.

> **Instruction Template**
>
> ### Given a piece of code, output the corresponding implementation idea.
> ### Input: $[X]$
> ### Output: $[Y]$

To address the challenge of excessive parameters in the base model, we employ the LoRA method [24] to facilitate efficient fine-tuning with limited resources. Unlike traditional fine-tuning methods that update all weights of the model, LoRA introduces trainable low-rank matrices to approximate weight adjustments. This approach leverages the observation that the adaptation process inherently exhibits a low "intrinsic rank." Let $\boldsymbol{W_0} \in \mathbb{R}^{d \times k}$ denote the pre-trained matrix. The weight adjustment approximation from $\boldsymbol{W_0}$ to $\boldsymbol{W_0} + \Delta \boldsymbol{W}$ using LoRA can be expressed as:

$$\boldsymbol{W_0} + \Delta \boldsymbol{W} = \boldsymbol{W_0} + \boldsymbol{BA}$$

Here, $\boldsymbol{B} \in \mathbb{R}^{d \times r}$ and $\boldsymbol{A} \in \mathbb{R}^{r \times k}$, where $r \ll \min(d, k)$ represents the rank. During the fine-tuning process, $\boldsymbol{W_0}$ remains unchanged, while $\boldsymbol{B}$ and $\boldsymbol{A}$ become the trainable parameters. Given an input $\boldsymbol{X}$ and its associated original output $\boldsymbol{H}$, the adjusted output $\bar{\boldsymbol{H}}$ is computed as:

$$\bar{\boldsymbol{H}} = \boldsymbol{W_0} \boldsymbol{X} + \Delta \boldsymbol{W} \boldsymbol{X} = \boldsymbol{H} + \boldsymbol{BAX}$$

To initialize the matrices, Matrix $\boldsymbol{A}$ is initialized by random Gaussian values, while $\boldsymbol{B}$ is initialized by zeros. This ensures that the initial value of $\Delta \boldsymbol{W} = \boldsymbol{BA}$ is zero at the start of training. To increase the number of trainable parameters and improve the capabilities, we apply LoRA to adapt all linear layers simultaneously.

3.3　Model Inference

In the inference phase, the model trained by COTTON can be efficiently deployed on a single consumer graphics card to generate CoT. Note that COTTON is a stand-alone tool and its deployment does not require the use of LLM. To accelerate the decoding process, COTTON utilizes the Greedy Search algorithm, which, in a nutshell, selects the token with the highest probability at each decoding step, resulting in a more deterministic output during inference.

The generated CoT by COTTON serves as an additional piece of information that is added to the original prompt in code generation tasks. In practical scenarios, to improve the user experience, we recommend generating CoT only when the code generation model fails to generate the correct code. This approach ensures that the CoT is used when necessary to avoid unnecessary overhead.

4　EXPERIMENTAL SETUP

To evaluate the effectiveness and benefits of our proposed approach, we mainly design the following three research questions (RQs):

RQ1: Can ℓLMs generate high-quality CoT independently?

In this RQ, we want to investigate whether ℓLMs have the capability to generate high-quality CoTs independently (i.e., the first question mentioned in Section 1). A negative finding of this RQ can constitute the motivation for designing our approach COTTON.

RQ2: Can COTTON generate higher-quality CoTs?

In this RQ, we want to evaluate the effectiveness of our approach COTTON. Specifically, we aim to compare its performance against state-of-the-art base models. Since we are the first to study automated CoT generation for code generation, we select relevant base models from similar research topics. We employ automatic evaluation metrics to assess the quality of the generated CoTs from various perspectives. We also conduct a human evaluation to assess the effectiveness of our approach, since automatic metrics may not fully capture the semantic similarity and educational value of the generated CoTs.

RQ3: Can ℓLMs effectively benefit from CoT?

While ℓLMs may not be able to generate high-quality CoTs independently, they may benefit from the provided CoTs to improve code generation performance (i.e., the second question mentioned in Section 1.). In this RQ, we want to validate the effectiveness of leveraging CoT for ℓLMs.

4.1　Dataset

4.1.1　Code Generation

To evaluate the performance of ℓLMs with and without CoT in the zero-shot scenario, we conduct experiments on three code generation datasets.

HumanEval/HumanEval-plus. The HumanEval dataset [25] was developed and published by OpenAI, consisting of 164 Python programming problems. Each problem includes an average of 7.8 test cases. which can provide a comprehensive evaluation of code generation capabilities. The HumanEval-plus dataset [4] aims to alleviate the test case coverage limitation in HumanEval, which may result in false positive rates. Note that HumanEval and HumanEval-plus are only different in terms of test cases and not in terms of CoT generation tasks.

OpenEval. To ensure fairness and generalizability, we collect a new code generation dataset OpenEval. This dataset includes 178 problems selected from the competition-level code translation dataset AVATAR [39]. For each problem, we designed additional test cases in a manual way that can effectively evaluate the quality of the generated code and minimize bias and leakage. Particularly, we hired two software engineers with 2∼3 years of development experience each to construct 5 test cases for each code segment to ensure the diversity of test cases.

4.1.2　CoT Generation

Following the method in Section 3.1, we collect a total of 9,264 CoT-generated samples. These samples are randomly split into a training set of 9,000 samples and a validation set of 264 samples. To evaluate the performance of COTTON, we generate CoTs on the HumanEval and OpenEval datasets, using the same methodology described in Section 3.1. The derived datasets are HumanEval-CoT and OpenEval-CoT respectively. Furthermore, we utilize Agent 2 (cf. Section 3.1) as the Teacher Model. An example in our used datasets is shown in Fig. 3. Finally, Table 1 provides statistical information about the datasets used for our evaluation.
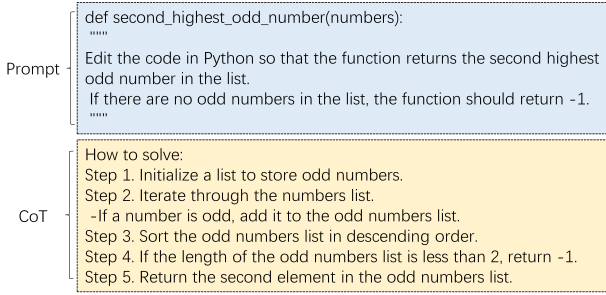
Prompt
```
def second_highest_odd_number(numbers):
    """
    Edit the code in Python so that the function returns the second highest
    odd number in the list.
    If there are no odd numbers in the list, the function should return -1.
    """
```

CoT
```
How to solve:
Step 1. Initialize a list to store odd numbers.
Step 2. Iterate through the numbers list.
  -If a number is odd, add it to the odd numbers list.
Step 3. Sort the odd numbers list in descending order.
Step 4. If the length of the odd numbers list is less than 2, return -1.
Step 5. Return the second element in the odd numbers list.
```

Fig. 3. An example in our used datasets

TABLE 1
Statistical information of our datasets

Type	Train	Valid	HumanEval-CoT	OpenEval-CoT
Count	9,000	264	164	178
Avg in Prompt	63.76	61.32	80.09	79.39
Median in Prompt	41.00	39.50	64.50	64.50
$\leq$ 256 in Prompt	98.03%	98.11%	98.78%	99.44%
Avg in CoT	85.40	82.41	94.54	93.99
Median in CoT	74.00	74.50	86.00	85.00
$\leq$ 256 in CoT	99.07%	99.24%	99.39%	99.44%

4.2 Code Generation Models

Based on the performance evaluations conducted by Gunasekar et al. [33], we select the following state-of-the-art ℓLMs for our experiments: CodeGen [19], StarCoder [40], and CodeT5+ [41]. These models have demonstrated promising performance in various code generation tasks [42].

CodeGen. CodeGen [19] is an open-source large language model specifically designed for code generation, with a particular focus on multi-turn program synthesis. It enhances program synthesis by breaking down complex user intents into multiple steps, facilitating the model's understanding. For our experiments, we select three models of different parameter sizes: 350M, 2B, and 6B.

StarCoder. StarCoder [40] is a language model trained on a diverse corpus of source code and natural language text. Its training data covers over 80 programming languages and includes text extracted from GitHub issues, commits, and notebooks. We select three models of different parameter sizes: 1B, 3B, and 7B.

CodeT5+. CodeT5+ [41] is a code large language model with an encoder-decoder architecture, capable of performing various code understanding and generation tasks. It offers flexibility by supporting different modes of operation. In our experiments, we select four models of different parameter sizes: 220M, 770M, 2B, and 6B.

4.3 Evaluation Metrics

4.3.1 Code Generation

To evaluate the performances of code generation models, we employ the Pass@1 metric and CoT-Pass@1 metric.

Pass@1. The Pass@1 metric measures the percentage of generated code snippets that pass the corresponding test cases without considering the CoT. This metric evaluates the ability of the code generation model to generate functionally correct code.

CoT-Pass@1. When the code generated by the model without the guidance of the CoT fails to pass the corresponding test cases, the model is provided with the CoT guidance to generate code. The CoT-Pass@1 metric is used to measure the percentage of generated code snippets that successfully pass the test cases after considering the CoT. This metric specifically evaluates the model's capability to generate code that passes the test cases when guided by the CoT. It focuses on assessing the model's ability to improve its code generation performance by incorporating the guidance provided by the CoT in cases where the initial code generation without CoT fails to pass the test cases.

4.3.2 CoT Generation

To evaluate the performance of the CoTs generated by COTTON, we first employ four automatic evaluation metrics commonly used in similar generation tasks (such as code generation [43], [44], code summarization [45], [46], and code translation [47] tasks).

BLEU (Bilingual Evaluation Understudy) [48] is a machine translation metric that measures the lexical similarity between two texts by computing the overlap of n-grams. In our evaluation, we utilize BLEU-1, -2, -3, -4 to evaluate the quality of the generated CoTs.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) [49] is an enhanced automatic evaluation metric based on BLEU. It incorporates an alignment algorithm based on dictionaries and linguistic knowledge, placing greater emphasis on word order and grammatical structure matching.

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation)-L [50] measures the similarity between the generated CoT and the ground-truth CoT by comparing their longest common subsequence. This metric is advantageous for handling long sequences and is not limited to single vocabulary items.

Consistency To further automatically assess the semantic correctness of the generated CoT, we use Agent 3 (cf. Section 3.1) to examine the consistency between the CoT and the code snippets.

The values of these performance metrics range from 0 to 1 and are displayed as percentages. A higher value indicates a closer match between the generated CoT and the ground-truth CoT. To compute BLEU, METEOR, and ROUGE-L, we utilize the nlg-eval library.[9]

4.4 Implementation Details and Running Platform

The hyper-parameters are tuned according to actual performance. We show the values of these hyper-parameters in Table 2. For the implementation of COTTON and other base models, we utilize the PyTorch[10] and Transformers[11] libraries. Our implementation is based on PyTorch 1.8, and the experiments are conducted on a machine with an Intel(R) Xeon(R) Silver 4210 CPU and the GeForce RTX 3090 GPU. It took about 6 hours to complete the model training of COTTON.

TABLE 2
Hyper-parameters and their values

Hyper-parameter	Value	Hyper-parameter	Value
Optimizer	AdamW	Random Seed	42
Learning Rate	1e-4	Training batch size	1
Lora R	8	Lora alpha	16
Max input length	256	Max output length	256
Epoch	20	Early Stop	5

5 EXPERIMENTAL RESULT ANALYSIS

5.1 RQ1: Can ℓLMs generate high-quality CoTs independently?

To investigate whether ℓLMs can independently generate high-quality CoTs for code generation, we conduct experiments based on few-shot prompt learning techniques as described in Section 3.1. We evaluate the performance of ℓLMs with model sizes ranging from 0.35B to 7B, the base model CodeLlama, and several representative LLMs (such as InternLM 123B [51], ChatGLM 130B [26], Gemini [52], gpt-3.5-turbo, and gpt-4 [53]). Among these LLMs, Gemini, gpt-3.5-turbo, and gpt-4 are currently considered to be the most promising LLMs in code generation tasks and have shown promising performance.

Table 3 presents the performance of ℓLMs and LLMs across all evaluation metrics on the HumanEval-CoT and OpenEval-CoT datasets. In terms of lexical similarity, ℓLMs generally performed worse than LLMs. For example, using the **METEOR** metric, LLMs like gpt-4 can achieve scores around 0.35 on the HumanEval-CoT dataset and around 0.37 on the OpenEval-CoT dataset. In contrast, the majority of ℓLMs scored below 0.3 on both datasets. In terms of semantic perspective, most ℓLMs are difficult to generate high-quality CoTs, while LLMs demonstrated better performance. For example, based on the **Consistency** metric, gpt-4 can achieve scores above 0.96 on the HumanEval-CoT dataset and above 0.87 on the OpenEval-CoT dataset. In contrast, the majority of ℓLMs scored below 0.6 in both data sets. In addition, in terms of the Consistency metric, gpt-3.5-turbo and gpt-4 achieved the best results on the OpenEval and HumanEval datasets, respectively, yet the cost of calling gpt-4 is 20 times higher than that of gpt-3.5-turbo. Therefore, we choose gpt-3.5-turbo as the Teacher Model in our study.

By further analysis based on varying models and parameter sizes, we can achieve interesting insights. Among CodeGen, StarCoder, CodeT5+, and CodeLlama models, StarCoder and CodeLlama showed potential in generating high-quality CoTs for guiding code generation. This could be attributed to the pre-training dataset of StarCoder and CodeLlama. The pre-training dataset of StarCoder includes information from GitHub commits and issues while the pre-training dataset of CodeLlama includes information from contains 8% of code-related natural language samples and 7% of general-purpose natural language data. The additional sources of natural language information may contribute to a better code understanding, resulting in

9. https://github.com/Maluuba/nlg-eval
10. https://pytorch.org/
11. https://github.com/huggingface/transformers

higher-quality CoTs. In contrast, the encoder-decoder model CodeT5+ exhibited the lowest performance compared to the decoder-only models (i.e., CodeGen and StarCoder). This performance difference may be attributed to the model's architecture. In the few-shot scenario, the encoder-decoder model may perform worse than the decoder model on the CoT generation task.

Regarding parameter scale, only the StarCoder model exhibited a positive correlation between larger parameter scale and better performance. The performance of the Code-Gen and CodeT5+ models did not strictly align with the scale of parameters. Only the 2B version of these models demonstrated improved performance.

> **Summary of RQ1**
>
> The majority of ℓLMs face challenges in generating high-quality CoTs for guiding code generation independently.

5.2 RQ2: Can COTTON generate higher-quality CoTs

We treat the CoT generation as a text generation problem (i.e., convert a user's functional requirements into CoT). To provide a comprehensive evaluation, we compare nine commonly used baselines, including CodeBERT [54], Graph-CodeBERT [55], CodeGPT [56], CodeGPT-adapter [56], PLBART [57], CodeT5 [58], NatGen [59], Llama2 [28], and CodeGeeX2 [60]. To ensure a fair comparison, we perform full parameter fine-tuning for models with less than 1B parameters. For models with more than 1B parameters, we employ LoRA [24] for parameter-efficient fine-tuning.

- **CodeBERT**. CodeBERT [54] is a pre-trained encoder-only model that can handle multi-modal inputs, such as natural language descriptions of code or code comments. It combines these inputs with code snippets to generate more accurate and complete representations of code.
- **GraphCodeBERT**. GraphCodeBERT [55] is an extension of CodeBERT specifically designed to capture more fine-grained information about code structures and dependencies. It enhances the capabilities of CodeBERT by incorporating graph-based representations.
- **CodeGPT**. CodeGPT [56] is a pre-trained decoder-only model employing a 12-layer Transformer Decoder architecture. It is trained with the same structure as GPT-2.
- **CodeGPT-adapter.** CodeGPT-adapter [56] is an extension of CodeGPT that utilizes domain-adaptive learning. It is initialized with a pre-trained GPT-2 model and then continues pretraining on the code dataset.
- **PLBART.** PLBART [57] is a sequence-to-sequence model pre-trained on a large collection of Java and Python functions and natural language descriptions via denoising autoencoding.
- **CodeT5.** CodeT5 [58] is a unified pre-trained encoder-decoder Transformer model that considers

TABLE 3
Performance comparison of ℓLMs and LLMs to independently generate CoTs on the HumanEval-CoT and OpenEval-CoT datasets.

Corpus	Type	Model	Model Size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Meteor	Rouge-L	Consistency
HumanEval-CoT	ℓLM	CodeGen	350M	16.39	11.68	8.79	6.86	13.08	23.56	0.61
		CodeGen	2B	40.34	30.90	24.16	19.25	23.60	37.95	39.02
		CodeGen	6B	27.48	20.33	15.54	12.19	18.95	31.37	28.66
		StarCoder	1B	38.80	29.79	23.18	18.33	24.46	38.83	31.71
		StarCoder	3B	42.13	32.16	25.04	19.97	25.11	40.53	60.96
		StarCoder	7B	43.53	34.20	27.16	22.02	26.58	43.76	78.66
		CodeT5+	220M	9.37	5.88	4.18	3.13	5.79	8.33	0.61
		CodeT5+	770M	22.25	16.28	12.33	9.51	17.76	24.73	0.61
		CodeT5+	2B	35.77	27.89	22.09	17.77	22.55	35.17	27.44
		CodeT5+	6B	19.37	13.18	9.56	7.28	14.05	22.99	7.32
		CodeLlama	7B	44.88	36.13	29.10	23.80	27.34	43.49	82.32
	LLM	InternLM	123B	52.08	42.23	34.50	28.67	32.65	43.09	89.63
		ChatGLM	130B	53.28	43.28	35.52	29.64	32.61	43.03	86.59
		Gemini	Not-Available	52.50	43.56	36.43	30.92	32.14	53.28	91.46
		gpt-4	Not-Available	60.14	50.49	43.03	37.24	34.55	54.32	96.95
		Teacher(gpt-3.5-turbo)	Not-Available	-	-	-	-	-	-	93.29
OpenEval-CoT	ℓLM	CodeGen	350M	31.74	25.32	21.11	17.96	17.62	29.29	0.56
		CodeGen	2B	39.22	31.21	25.22	20.73	24.27	37.43	29.78
		CodeGen	6B	32.54	25.84	20.92	17.21	20.79	33.15	27.53
		StarCoder	1B	41.33	32.09	25.32	20.44	24.67	35.98	21.35
		StarCoder	3B	44.85	35.90	29.17	24.21	26.45	38.74	43.26
		StarCoder	7B	46.23	36.98	30.07	24.88	28.35	43.33	57.87
		CodeT5+	220M	14.70	9.98	7.88	6.59	9.13	10.56	0.00
		CodeT5+	770M	29.35	22.95	18.11	14.52	19.96	28.59	0.00
		CodeT5+	2B	29.96	24.07	19.61	16.27	21.31	32.27	21.91
		CodeT5+	6B	26.58	21.00	17.20	14.42	17.13	28.12	5.06
		CodeLlama	7B	52.09	43.89	37.27	32.13	30.52	48.96	71.91
	LLM	InternLM	123B	55.59	46.77	40.01	34.80	35.28	46.48	84.27
		ChatGLM	130B	55.28	46.47	39.68	34.56	35.14	45.64	76.40
		Gemini	Not-Available	57.49	49.04	42.43	37.25	34.13	54.88	81.46
		gpt-4	Not-Available	62.54	54.34	47.88	42.81	36.84	58.02	87.64
		Teacher(gpt-3.5-turbo)	Not-Available	-	-	-	-	-	-	89.33

token type information in code and better leverages the code semantics conveyed from developer-assigned identifiers.

- **NatGen.** NatGen [59] is an extension of CodeT5 that exploits the bimodal and dual-channel nature of code information to learn the naturalizing of source code.
- **CodeGeeX2.** CodeGeeX2 [60] is a multilingual code generation model, which is based on the ChatGLM2 architecture.
- **Llama2.** Llama2 [28] is an extension of Llama. Building on Llama, Llama2 increases the size of the pre-trained corpus by 40%, doubles the context length of the model, and employs a grouped query attention mechanism.

5.2.1 Automatic Evaluation.

We first evaluate different base models on the HumanEval-CoT and OpenEval-CoT datasets. We consider different evaluation metrics, including BLEU-1, BLEU-2, BLEU-3, BLEU-4, Meteor, Rouge-L, and Consistency. The results are shown in Table 4.

For the HumanEval-CoT dataset, COTTON consistently outperformed the other base models across various evaluation metrics. This indicates that COTTON generates CoTs with higher lexical similarity compared to the base models. For example, based on the **METEOR** metric, COTTON achieved a score of 0.38, while the other base models scored between 0.27 and 0.37. In terms of semantic similarity, COTTON also outperformed the other base models on the HumanEval-CoT dataset. With a score of 0.93, COTTON demonstrated a higher level of semantic similarity to the

actual code compared to the scores ranging from 0.29 to 0.92 achieved by the other base models. Importantly, COTTON even outperforms larger LLMs (i.e., InternLM 123B, ChatGLM 130B, and Gemini), indicating its effectiveness in generating high-quality CoTs. We can find similar trends on the OpenEval-CoT dataset in Table 4, where COTTON achieves better performance compared to the other base models.

Furthermore, we observe a correlation between the parameter scale of the models and the performance of the generated CoTs. Specifically, larger models (such as CodeGeeX2, Llama2, and CodeCoT) showed significant performance improvement in CoT generation compared to the other models. The larger parameter scales of these models enabled them to capture more complex patterns and dependencies in the data, leading to better CoT generation ability.

5.2.2 Human Evaluation.

Although automatic evaluation metrics can offer valuable insights into the quality of generated CoTs, these metrics mainly concentrate on overlap or semantic similarity with the CoTs generated by the Teacher Model, possibly overlooking whether the generated CoT is indeed inspiring or offers meaningful guidance to developers. Furthermore, the enhancements in Table 4 brought about by COTTON over CodeGeeX2 and LLama2 may not be immediately apparent.

To further evaluate the quality of the generated CoTs, we conducted a human study using four groups of CoTs generated by different base models (CodeGeeX2 and LLama2) that performed similarly based on the results in Table 3 and Table 4. By involving human evaluators, we aimed to obtain

TABLE 4
Performance comparison of different base models to generate CoTs on the HumanEval-CoT and OpenEval-CoT datasets (Here COTTON is based on CodeLlama-7b).

Corpus	Base Model	Model Size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Meteor	Rouge-L	Consistency
HumanEval-CoT	CodeBERT	173M	46.35	38.79	33.10	28.81	27.52	50.66	29.27
	GraphCodeBERT	173M	47.32	39.75	34.17	30.08	27.91	50.68	33.54
	CodeGPT	124M	26.91	47.96	41.14	36.13	31.91	52.95	57.32
	CodeGPT-adapter	124M	54.10	45.19	38.42	33.40	30.56	51.04	52.44
	PLBART	139M	42.95	35.07	29.13	24.81	24.25	33.85	21.34
	CodeT5	223M	61.03	53.16	46.85	42.00	34.89	58.93	79.88
	NatGen	223M	62.76	54.84	48.50	43.59	35.92	59.91	82.32
	CodeGeeX2	6B	62.57	54.18	47.54	42.33	35.77	59.72	92.68
	LLama2	7B	**66.56**	58.00	51.05	45.62	37.65	61.39	89.63
	COTTON	7B	65.97	**58.21**	**51.89**	**46.87**	**38.22**	**63.38**	**93.29**
OpenEval-CoT	CodeBERT	173M	34.19	27.18	21.98	18.14	23.02	41.20	8.99
	GraphCodeBERT	173M	37.87	30.09	24.42	20.27	23.73	42.23	11.24
	CodeGPT	124M	48.70	40.20	33.58	28.62	28.23	46.19	35.96
	CodeGPT-adapter	124M	49.91	41.41	34.76	29.64	28.99	46.90	42.13
	PLBART	139M	45.54	37.19	30.90	26.23	25.48	34.73	17.98
	CodeT5	223M	57.65	50.18	44.29	39.63	33.81	55.84	65.17
	NatGen	223M	60.25	52.75	46.87	42.19	35.19	57.86	58.99
	CodeGeeX2	6B	64.86	56.83	50.54	45.62	37.05	61.31	79.21
	LLama2	7B	64.89	56.98	50.67	45.54	37.24	60.40	71.91
	COTTON	7B	**67.04**	**59.56**	**53.60**	**48.80**	**38.80**	**62.92**	**83.71**

a more comprehensive assessment of the CoTs and gain a deeper understanding of their practical implications.


```
Read the following CoT and answer the questions:

Prompt:
def generate_integers(a, b):
    """
    Given two positive integers a and b, return the even digits between a and b, in
ascending order.
    """

Reference:
How to solve:
Step 1. Initialize an empty list to store the even digits.
Step 2. Iterate through the range from a to b (inclusive).
    - Convert each number to a string.
    - Iterate through each digit in the string.
        - If the digit is even, add it to the list of even digits.
Step 3. Sort the list of even digits in ascending order.
Step 4. Return the sorted list of even digits.

Candidate 1:
How to solve:
Step 1. Initialize an empty list to store the even digits.
Step 2. Iterate through the range from a to b (inclusive).
    -If the current number is even and not already in the list, append it to the list.
Step 3. Return the list of even digits.

Candidate 2:
How to solve:
Step 1. Initialize an empty list to store the even digits.
Step 2. Iterate through the range from a to b (inclusive).
    -Convert each number to a string.
    -Iterate through each character in the string.
        -If the character is even, append it to the list.
Step 3. Sort the list in ascending order.
Step 4. Return the sorted list.

Candidate 3:
How to solve:
Step 1. Initialize an empty list to store the even digits.
Step 2. Iterate through the range from a to b (inclusive).
    -Convert each number to a string.
    -Iterate through each character in the string.
        -If the character is a digit and it is a digit that is divisible by 2, add it
to the list.
Step 3. Return the list of even digits.
```

Evaluate (Score 0 to 4, 4 is the best)	Candidate 1	Candidate 2	Candidate 3
Similarity			
Naturalness			
Educational Value			

Fig. 4. A sample questionnaire used in human study

In our human study, we adopted the methodology used in previous studies [61]–[63]. The quality of the generated CoTs was evaluated based on three aspects, as illustrated in Fig. 4:

- **Similarity**. This aspect measures the semantic similarity between the generated CoT and the reference.

- **Naturalness**. This aspect assessed the grammaticality and fluency of the generated CoT.
- **Educational Value**. This aspect evaluated whether volunteers could gain inspiration from the generated CoTs to solve problems, thereby measuring their educational value.

TABLE 5
The average score and standard deviation (in parentheses) of human study.

	Aspect	LLama2	CodeGeeX2	**COTTON**
HE	Similarity	2.847 (0.875)	2.762 (0.918)	**3.038** (0.743)
	Naturalness	3.418 (0.842)	3.375 (0.592)	**3.442** (0.713)
	Educational Value	2.985 (0.788)	2.973 (0.812)	**3.102** (0.779)
OE	Similarity	3.024 (0.645)	2.976 (0.615)	**3.128** (0.734)
	Naturalness	3.521 (0.746)	3.567 (0.789)	**3.691** (0.657)
	Educational Value	2.955 (0.858)	3.002 (0.876)	**3.209** (0.891)

TABLE 6
The p-values between three aspects of the human study, and all the p-values are substantially smaller than 0.005.

	Model	Similarity	Similarity	Educational Value
HE	Llama2	2.14e-5	1.85e-5	4.23e-5
	CodeGeeX2	5.68e-5	3.62e-5	1.45e-5
OE	Llama2	1.53e-4	4.28e-4	8.85e-4
	CodeGeeX2	1.68e-4	4.29e-4	3.67e-4

For the human evaluation, we recruited six assessors (six postgraduate students) who were familiar with Python programming language. We selected all samples from the HumanEval-CoT dataset (164 samples) and the OpenEval-CoT dataset (178 samples) as the evaluation subjects. For each sample, we collected the ground truth CoT and three generated CoTs.

To ensure a thorough evaluation, we divided all the samples into three groups, with each group containing 114

TABLE 7
The performance of code generation models with or without the CoTs generated by different self sot prompt methods on the HumanEval(HE),
HumanEval-plus(HE-p), and OpenEval(OE) datasets

Corpus	Model	Pass@1	CoT-Pass@1 (different CoT generation methods)				
			Self-CoT	Think Step by Step	Self-planning	SCoT	COTTON
HE	CodeGen 350M	14.63	12.80	15.85 (↑ 8.34%)	18.29 (↑ 25.02%)	15.85 (↑ 8.34%)	**20.73** (↑ 41.70%)
	CodeGen 2B	25.61	25.00	27.44 (↑ 7.15%)	26.22 (↑ 2.38%)	26.83 (↑ 4.76%)	**34.76** (↑ 35.73%)
	CodeGen 6B	27.44	31.10 (↑ 13.34%)	29.88 (↑ 8.89%)	28.66 (↑ 4.45%)	30.49 (↑ 11.12%)	**39.63** (↑ 44.42%)
	StarCoder 1B	12.80	12.80	15.85 (↑ 23.83%)	17.68 (↑ 38.13%)	14.63 (↑ 14.30%)	**25.00** (↑ 95.31%)
	StarCoder 3B	17.07	21.34 (↑ 25.01%)	17.07	20.12 (↑ 17.87%)	20.12 (↑ 17.87%)	**30.49** (↑ 78.62%)
	StarCoder 7B	21.95	29.88 (↑ 36.13%)	28.05 (↑ 27.79%)	25.61 (↑ 16.67%)	29.88 (↑ 36.13%)	**37.20** (↑ 69.48%)
	CodeT5+ 220M	12.20	14.02 (↑ 14.92%)	14.02 (↑ 14.92%)	12.20	11.59	**18.90** (↑ 54.92%)
	CodeT5+ 770M	17.07	17.68 (↑ 3.57%)	17.68 (↑ 3.57%)	17.68 (↑ 3.57%)	18.90 (↑ 10.72%)	**26.83** (↑ 57.18%)
	CodeT5+ 2B	23.78	25.00 (↑ 5.13%)	27.44 (↑ 15.39%)	26.22 (↑ 10.26%)	26.22 (↑ 10.26%)	**30.49** (↑ 28.22%)
	CodeT5+ 6B	26.22	32.32 (↑ 23.26%)	33.54 (↑ 27.92%)	34.15 (↑ 30.24%)	32.93 (↑ 25.59%)	**42.68** (↑ 62.78%)
OE	CodeGen 350M	7.30	8.99 (↑ 23.15%)	8.43 (↑ 15.48%)	7.87 (↑ 7.81%)	10.67 (↑ 46.16%)	**12.92** (↑ 76.99%)
	CodeGen 2B	16.85	19.10 (↑ 13.35%)	16.85	20.22 (↑ 20.00%)	20.22 (↑ 20.00%)	**26.97** (↑ 60.06%)
	CodeGen 6B	21.91	23.60 (↑ 7.71%)	23.60 (↑ 7.71%)	28.09 (↑ 28.21%)	29.78 (↑ 35.92%)	**33.71** (↑ 53.86%)
	StarCoder 1B	8.99	10.67 (↑ 18.69%)	8.99	11.24 (↑ 25.03%)	11.80 (↑ 31.26%)	**17.42** (↑ 93.77%)
	StarCoder 3B	11.24	16.85 (↑ 49.91%)	14.61 (↑ 29.98%)	19.10 (↑ 69.93%)	14.61 (↑ 29.98%)	**19.10** (↑ 69.93%)
	StarCoder 7B	23.03	29.21 (↑ 26.83%)	28.65 (↑ 24.40%)	25.84 (↑ 12.20%)	28.09 (↑ 21.97%)	**33.15** (↑ 43.94%)
	CodeT5+ 220M	7.87	10.67 (↑ 35.58%)	12.36 (↑ 57.05%)	10.67 (↑ 35.58%)	8.43 (↑ 7.12%)	**13.48** (↑ 71.28%)
	CodeT5+ 770M	9.55	11.24 (↑ 17.70%)	13.48 (↑ 41.15%)	13.48 (↑ 41.15%)	13.48 (↑ 41.15%)	**16.29** (↑ 70.58%)
	CodeT5+ 2B	15.17	19.66 (↑ 29.60%)	16.85 (↑ 11.07%)	19.66 (↑ 29.60%)	17.42 (↑ 14.83%)	**28.65** (↑ 88.86%)
	CodeT5+ 6B	20.22	21.35 (↑ 5.59%)	30.90 (↑ 52.82%)	24.72 (↑ 22.26%)	31.46 (↑ 55.59%)	**35.39** (↑ 75.02%)
HE-p	CodeGen 350M	15.24	12.80	16.46 (↑ 8.01%)	18.29 (↑ 20.01%)	15.85 (↑ 4.00%)	**20.73** (↑ 36.02%)
	CodeGen 2B	26.22	25.00	27.44 (↑ 4.65%)	26.22	26.83 (↑ 2.33%)	**35.37** (↑ 34.90%)
	CodeGen 6B	27.44	32.32 (↑ 17.78%)	30.49 (↑ 11.12%)	29.27 (↑ 6.67%)	30.49 (↑ 11.12%)	**40.85** (↑ 48.87%)
	StarCoder 1B	13.41	13.41	15.85 (↑ 18.20%)	17.68 (↑ 31.84%)	14.63 (↑ 9.10%)	**26.22** (↑ 95.53%)
	StarCoder 3B	17.07	21.95 (↑ 28.59%)	17.07	20.12 (↑ 17.87%)	20.73 (↑ 21.44%)	**31.71** (↑ 85.76%)
	StarCoder 7B	22.56	30.49 (↑ 35.15%)	28.66 (↑ 27.04%)	25.61 (↑ 13.52%)	31.10 (↑ 37.85%)	**38.41** (↑ 70.26%)
	CodeT5+ 220M	12.20	14.02 (↑ 14.92%)	14.02 (↑ 14.92%)	12.20	11.59	**19.51** (↑ 59.92%)
	CodeT5+ 770M	17.68	17.68	17.68	17.68	19.51 (↑ 10.35%)	**27.44** (↑ 55.20%)
	CodeT5+ 2B	25.00	25.00	28.05 (↑ 12.20%)	27.44 (↑ 9.76%)	26.22 (↑ 4.88%)	**31.71** (↑ 26.84%)
	CodeT5+ 6B	26.83	32.93 (↑ 22.74%)	34.15 (↑ 27.28%)	34.76 (↑ 26.56%)	32.93 (↑ 22.74%)	**43.90** (↑ 63.62%)

samples. Each group was evaluated anonymously by two assessors in terms of similarity, naturalness, and educational value. The score for each aspect ranges from 0 to 4, with higher scores indicating higher quality and the final score is the average of two assessors' scores.

A sample questionnaire is shown in Fig. 4. To guarantee human study quality, the generated CoTs were presented in a random order, ensuring that the assessors had no knowledge of which approach generated the CoT. Moreover, the assessors were allowed to use the internet to look up any related concepts they were unfamiliar with. Finally, we limited each assessor to evaluate only 20 samples in half a day. This was done to prevent fatigue and maintain a high level of concentration during the evaluation process.

The results of the human study are summarized in Table 5, which shows the average scores and standard deviations of all evaluated samples by the assessors. We can observe that the proposed approach COTTON outperforms Llama2 and CodeGeeX2 in terms of similarity, naturalness, and educational value. To determine the statistical significance of these differences, we conducted Fisher's exact test [64] and observed a statistically significant disparity in Table 6 (i.e., p-value < 0.05). Furthermore, we utilized Fleiss Kappa [65] to assess the agreement among the six assessors. The overall Kappa value based on the comparison results is 0.7, signifying substantial agreement among the assessors.

These findings provide further evidence of the effectiveness of COTTON in generating high-quality CoTs for

guiding code generation. The higher average score values obtained by COTTON indicate that it is capable of generating CoTs that are more grammatically correct, fluent, and valuable in terms of providing guidance and inspiration to developers. These results highlight the competitiveness of our proposed approach in generating CoTs that are not only grammatically sound but also valuable from an educational perspective.

> **Summary of RQ2**
>
> COTTON consistently outperforms the state-of-the-art base models in generating CoTs in terms of lexical and semantic similarity, as well as its ability to provide valuable guidance and inspiration to developers.

5.3 RQ3: Can ℓLMs benefit from the generated CoT?

5.3.1 Effect of self-generated CoTs on ℓLMs.

To assess the impact of generated CoTs on improving the code generation performance of LLMs (such as CodeGen, StarCoder, and CodeT5+ as discussed in Section 4.2), we initially compare different models in the Self-CoT scenario (i.e., the CoTs generated by the model itself using the few-shot method in **RQ1**). Furthermore, since Self-planning [13] and SCoT [14] have demonstrated that well-designed self-CoT prompts can effectively improve the performance of

TABLE 8
The performance of code generation models with or without the CoTs generated by different methods on the HumanEval(HE),
HumanEval-plus(HE-p), and OpenEval(OE) datasets (Here COTTON is based on CodeLlama-7b).

Corpus	Model	Pass@1	CoT-Pass@1 (different CoT generation methods)				
			Gemini	CodeLlama	ChatGLM	Teacher	COTTON
HE	CodeGen 350M	14.63	**24.39** (↑ 66.71%)	18.90 (↑ 29.19%)	18.29 (↑ 25.02%)	**24.39** (↑ 66.71%)	20.73 (↑ 41.70%)
	CodeGen 2B	25.61	37.20 (↑ 45.26%)	31.10 (↑ 21.44%)	31.71 (↑ 23.82%)	**41.46** (↑ 61.89%)	34.76 (↑ 35.73%)
	CodeGen 6B	27.44	**43.29** (↑ 57.76%)	36.59 (↑ 33.35%)	31.71 (↑ 15.56%)	**43.29** (↑ 57.76%)	39.63 (↑ 44.42%)
	StarCoder 1B	12.80	21.95 (↑ 71.48%)	17.68 (↑ 38.13%)	17.68 (↑ 38.13%)	**28.66** (↑ 123.91%)	25.00 (↑ 95.31%)
	StarCoder 3B	17.07	34.15 (↑ 100.06%)	25.61 (↑ 50.03%)	25.61 (↑ 50.03%)	**39.63** (↑ 132.16%)	30.49 (↑ 78.62%)
	StarCoder 7B	21.95	38.41 (↑ 74.99%)	33.54 (↑ 52.80%)	34.15 (↑ 35.72%)	**41.46** (↑ 88.88%)	37.20 (↑ 69.48%)
	CodeT5+ 220M	12.20	19.51 (↑ 59.92%)	18.29 (↑ 49.92%)	16.46 (↑ 34.92%)	**23.17** (↑ 89.92%)	18.90 (↑ 54.92%)
	CodeT5+ 770M	17.07	26.83 (↑ 57.18%)	23.78 (↑ 39.31%)	23.78 (↑ 39.31%)	**31.71** (↑ 85.76%)	26.83 (↑ 57.18%)
	CodeT5+ 2B	23.78	**38.41** (↑ 61.52%)	28.05 (↑ 17.96%)	29.27 (↑ 23.09%)	**38.41** (↑ 61.52%)	30.49 (↑ 28.22%)
	CodeT5+ 6B	26.22	45.73 (↑ 74.41%)	38.41 (↑ 46.69%)	36.59 (↑ 39.55%)	**47.56** (↑ 81.39%)	42.68 (↑ 62.78%)
OE	CodeGen 350M	7.30	**16.85** (↑ 13.08%)	14.04 (↑ 92.33%)	12.92 (↑ 76.99%)	15.17 (↑ 107.81%)	12.92 (↑ 76.99%)
	CodeGen 2B	16.85	**30.34** (↑ 80.06%)	21.91 (↑ 30.03%)	25.28 (↑ 50.03%)	29.21 (↑ 73.35%)	26.97 (↑ 60.06%)
	CodeGen 6B	21.91	34.27 (↑ 56.41%)	29.21 (↑ 33.32%)	32.02 (↑ 46.14%)	**37.64** (↑ 71.79%)	33.71 (↑ 53.86%)
	StarCoder 1B	8.99	17.42 (↑ 93.77%)	14.61 (↑ 62.51%)	16.85 (↑ 87.43%)	**19.66** (↑ 118.69%)	17.42 (↑ 93.77%)
	StarCoder 3B	11.24	18.54 (↑ 64.95%)	15.17 (↑ 34.96%)	14.61 (↑ 29.98%)	17.98 (↑ 59.96%)	**19.10** (↑ 69.93%)
	StarCoder 7B	23.03	34.83 (↑ 51.24%)	28.65 (↑ 24.40%)	29.78 (↑ 29.31%)	**38.20** (↑ 65.87%)	33.15 (↑ 43.94%)
	CodeT5+ 220M	7.87	**15.17** (↑ 92.76%)	10.67 (↑ 35.58%)	12.92 (↑ 64.17%)	14.04 (↑ 78.40%)	13.48 (↑ 71.28%)
	CodeT5+ 770M	9.55	**21.35** (↑ 123.56%)	16.85 (↑ 76.44%)	17.98 (↑ 88.27%)	20.22 (↑ 111.73%)	16.29 (↑ 70.58%)
	CodeT5+ 2B	15.17	28.65 (↑ 88.86%)	24.72 (↑ 62.95%)	25.28 (↑ 66.64%)	**30.90** (↑ 103.69%)	28.65 (↑ 88.86%)
	CodeT5+ 6B	20.22	33.15 (↑ 63.95%)	28.09 (↑ 38.92%)	32.02 (↑ 58.36%)	**36.52** (↑ 80.61%)	35.39 (↑ 75.02%)
HE-p	CodeGen 350M	15.24	24.39 (↑ 60.04%)	19.51 (↑ 28.02%)	18.29 (↑ 20.01%)	**25.00** (↑ 64.04%)	20.73 (↑ 36.02%)
	CodeGen 2B	26.22	37.80 (↑ 44.16%)	31.71 (↑ 20.94%)	32.32 (↑ 23.26%)	**42.68** (↑ 62.78%)	35.37 (↑ 34.90%)
	CodeGen 6B	27.44	43.90 (↑ 59.99%)	37.20 (↑ 35.57%)	32.32 (↑ 17.78%)	**44.51** (↑ 62.21%)	40.85 (↑ 48.87%)
	StarCoder 1B	13.41	22.56 (↑ 68.23%)	18.29 (↑ 36.39%)	18.29 (↑ 36.39%)	**29.27** (↑ 118.27%)	26.22 (↑ 95.53%)
	StarCoder 3B	17.07	34.76 (↑ 103.63%)	25.61 (↑ 50.03%)	26.22 (↑ 53.60%)	**40.85** (↑ 139.31%)	31.71 (↑ 85.76%)
	StarCoder 7B	22.56	39.02 (↑ 72.96%)	34.15 (↑ 51.37%)	34.76 (↑ 54.08%)	**43.29** (↑ 91.89%)	38.41 (↑ 70.26%)
	CodeT5+ 220M	12.20	20.12 (↑ 64.90%)	18.90 (↑ 54.92%)	17.07 (↑ 39.92%)	**23.78** (↑ 94.92%)	19.51 (↑ 59.92%)
	CodeT5+ 770M	17.68	27.44 (↑ 55.20%)	24.39 (↑ 37.95%)	24.39 (↑ 37.95%)	**32.32** (↑ 82.81%)	27.44 (↑ 55.20%)
	CodeT5+ 2B	25.00	39.02 (↑ 56.08%)	28.66 (↑ 14.64%)	29.88 (↑ 19.52%)	**39.63** (↑ 58.52%)	31.71 (↑ 26.84%)
	CodeT5+ 6B	26.83	46.34 (↑ 72.72%)	39.02 (↑ 45.43%)	37.20 (↑ 38.65%)	**48.78** (↑ 81.81%)	43.90 (↑ 63.62%)

large models in code generation, we also include Think step-by-step [18], Self-planning [13], and SCoT [14] as baselines in our experiments.

We use the Pass@1 metric to evaluate the effectiveness of using CoTs in improving the performance of ℓLMs, which can help understand whether ℓLMs can effectively benefit from the instructions provided by CoT. The results are presented in Table 7, with the best result highlighted in boldface and the second-best result underscored. We can observe that, under the guidance of the CoT generated by the Self-CoT method, the performance improvement of all ℓLMs is very limited, and even the performance of some ℓLMs may decrease. This finding supports the conclusion of **RQ1**, i.e., ℓLMs cannot independently generate high-quality CoTs.

5.3.2 Effect of CoTs generated by different LMs on ℓLMs.

We analyze the performance of ℓLMs when utilizing CoTs generated by CodeLlama, ChatGLM 130B, Gemini, and the Teacher model. Specifically, we compare the utility of CoTs generated by CodeLlama and the selected LLMs with those generated by COTTON to determine whether COTTON-generated CoTs are more beneficial for ℓLMs. The results are presented in Table 8, with the best result highlighted in boldface and the second-best result underscored.

Our findings indicate that all ℓLMs can effectively leverage the guidance provided by CoTs to enhance the quality of generated code, assuming the CoTs' quality is ensured.

This emphasizes the potential of utilizing CoTs to improve the performance of ℓLMs.

Furthermore, we examine the disparity between CoTs generated by COTTON and those generated by the Teacher model. This comparison sheds light on how closely COTTON can approximate the more advanced Teacher model in CoT generation. The experimental results reveal that COTTON outperforms ChatGLM 130B and closely rivals Gemini and GPT-3.5-turbo in the CoT generation task for code generation.

5.3.3 Effect of CoTs generated by different base models on ℓLMs.

In our work, essentially we fine-tune a base model CodeLlama to obtain COTTON. A legitimate question is why we choose CodeLlama, or whether other base models could yield better results. To further investigate this question, we take different base ℓLM models to tune them for CoT generation purposes. We then conduct experiments using the generated CoTs, where the results are presented in Table 9, with the best result in boldface and the second-best result underscored.

Based on the results, it is observed that using CodeLlama as the base model leads to more significant performance improvements in code generation for the majority of ℓLMs, which justifies our choice and highlights the importance of choosing an appropriate base model when generating CoTs to enhance the code generation performance of ℓLMs.

TABLE 9
The performance of code generation models with or without the CoTs generated by COTTON using different base models on the HumanEval(HE), HumanEval-plus(HE-p), and OpenEval(OE) datasets (Here COTTON is based on CodeLlama-7b).

Corpus	Model	Pass@1	CoT-Pass@1 (different base models)				
			GraphCodeBERT	CodeGPT-adapter	NatGen	CodeGeeX2	COTTON
HE	CodeGen 350M	14.63	15.24 (↑ 4.17%)	15.85 (↑ 8.34%)	18.90 (↑ 29.19%)	18.90 (↑ 29.19%)	**20.73** (↑ 41.70%)
	CodeGen 2B	25.61	25.61	24.39	32.32 (↑ 26.20%)	33.54 (↑ 30.96%)	**34.76** (↑ 35.73%)
	CodeGen 6B	27.44	31.71 (↑ 15.56%)	31.71 (↑ 15.56%)	34.15 (↑ 24.45%)	35.96 (↑ 19.51%)	**39.63** (↑ 44.42%)
	StarCoder 1B	12.80	17.07 (↑ 33.36%)	15.24 (↑ 19.06%)	15.85 (↑ 23.83%)	19.51 (↑ 52.42%)	**25.00** (↑ 95.31%)
	StarCoder 3B	17.07	20.12 (↑ 17.87%)	19.51 (↑ 14.29%)	26.22 (↑ 53.60%)	**31.10** (↑ 82.19%)	30.49 (↑ 78.62%)
	StarCoder 7B	21.95	28.05 (↑ 27.79%)	24.39 (↑ 11.12%)	30.49 (↑ 38.91%)	34.15 (↑ 55.58%)	**37.20** (↑ 69.48%)
	CodeT5+ 220M	12.20	12.80 (↑ 4.92%)	14.63 (↑ 19.92%)	17.07 (↑ 39.92%)	**18.90** (↑ 54.92%)	**18.90** (↑ 54.92%)
	CodeT5+ 770M	17.07	18.90 (↑ 10.72%)	18.29 (↑ 7.15%)	21.95 (↑ 28.59%)	25.00 (↑ 46.46%)	**26.83** (↑ 57.18%)
	CodeT5+ 2B	23.78	27.44 (↑ 15.39%)	25.00 (↑ 5.13%)	28.05 (↑ 17.96%)	30.49 (↑ 28.22%)	**30.49** (↑ 28.22%)
	CodeT5+ 6B	26.22	31.71 (↑ 20.94%)	32.32 (↑ 23.26%)	37.80 (↑ 44.16%)	38.41 (↑ 46.49%)	**42.68** (↑ 62.78%)
OE	CodeGen 350M	7.30	11.80 (↑ 61.64%)	12.36 (↑ 69.32%)	10.67 (↑ 46.16%)	**14.61** (↑ 100.14%)	12.92 (↑ 76.99%)
	CodeGen 2B	16.85	17.42 (↑ 3.38%)	22.47 (↑ 33.35%)	25.28 (↑ 50.03%)	24.72 (↑ 46.71%)	**26.97** (↑ 60.06%)
	CodeGen 6B	21.91	25.84 (↑ 17.94%)	24.72 (↑ 12.83%)	26.97 (↑ 23.09%)	28.09 (↑ 28.21%)	**33.71** (↑ 53.86%)
	StarCoder 1B	8.99	11.24 (↑ 25.03%)	10.67 (↑ 18.69%)	11.80 (↑ 31.26%)	13.48 (↑ 49.94%)	**17.42** (↑ 93.77%)
	StarCoder 3B	11.24	11.80 (↑ 4.98%)	14.04 (↑ 24.91%)	16.29 (↑ 44.93%)	**20.22** (↑ 19.10%)	19.10 (↑ 69.93%)
	StarCoder 7B	23.03	22.47	25.28 (↑ 9.77%)	25.28 (↑ 9.77%)	31.46 (↑ 36.60%)	**33.15** (↑ 43.94%)
	CodeT5+ 220M	7.87	9.55 (↑ 21.35%)	11.80 (↑ 49.94%)	**15.73** (↑ 99.87%)	15.17 (↑ 92.76%)	13.48 (↑ 71.28%)
	CodeT5+ 770M	9.55	15.17 (↑ 58.85%)	13.48 (↑ 41.15%)	13.48 (↑ 41.15%)	**16.85** (↑ 76.44%)	16.29 (↑ 70.58%)
	CodeT5+ 2B	15.17	17.42 (↑ 14.83%)	20.79 (↑ 37.05%)	23.03 (↑ 51.81%)	24.72 (↑ 62.95%)	**28.65** (↑ 88.86%)
	CodeT5+ 6B	20.22	23.60 (↑ 16.72%)	24.72 (↑ 22.26%)	24.16 (↑ 19.49%)	29.78 (↑ 47.28%)	**35.39** (↑ 75.02%)
HE-p	CodeGen 350M	15.24	15.24	15.85 (↑ 4.00%)	18.90 (↑ 24.02%)	19.51 (↑ 28.02%)	**20.73** (↑ 36.02%)
	CodeGen 2B	26.22	25.61	24.39	32.32 (↑ 23.26%)	35.37 (↑ 34.90%)	**35.37** (↑ 34.90%)
	CodeGen 6B	27.44	31.71 (↑ 15.56%)	31.71 (↑ 15.56%)	34.15 (↑ 24.45%)	36.59 (↑ 33.35%)	**40.85** (↑ 48.87%)
	StarCoder 1B	13.41	17.07 (↑ 27.29%)	15.24 (↑ 13.65%)	15.85 (↑ 18.20%)	20.12 (↑ 50.04%)	**26.22** (↑ 95.53%)
	StarCoder 3B	17.07	20.12 (↑ 17.87%)	19.51 (↑ 14.29%)	26.22(↑ 53.60%)	31.71 (↑ 85.76%)	**31.71** (↑ 85.76%)
	StarCoder 7B	22.56	28.05 (↑ 24.34%)	24.39 (↑ 8.11%)	30.49 (↑ 35.15%)	34.76 (↑ 54.08%)	**38.41** (↑ 70.26%)
	CodeT5+ 220M	12.20	12.80 (↑ 4.92%)	14.63 (↑ 19.92%)	17.07 (↑ 39.92%)	19.51 (↑ 59.92%)	**19.51** (↑ 59.92%)
	CodeT5+ 770M	17.68	18.90 (↑ 6.90%)	18.29 (↑ 3.45%)	21.95 (↑ 24.15%)	25.61 (↑ 44.85%)	**27.44** (↑ 55.20%)
	CodeT5+ 2B	25.00	27.44 (↑ 9.76%)	25.00	28.05 (↑ 12.20%)	31.10 (↑ 24.40%)	**31.71** (↑ 26.84%)
	CodeT5+ 6B	26.83	31.71 (↑ 18.19%)	32.32 (↑ 20.46%)	37.80 (↑ 40.89%)	39.02 (↑ 45.43%)	**43.90** (↑ 63.62%)

By selecting the most effective base model, COTTON can optimize the performance and effectiveness of ℓLMs in generating high-quality code.

> **Summary of RQ3**
>
> ℓLMs can effectively leverage the guidance provided by CoTs. This emphasizes the potential of utilizing CoTs to enhance the performance of ℓLMs in code generation tasks.

6 DISCUSSIONS

6.1 Evaluating COTTON on LLMs

To further evaluate the capabilities of COTTON, we conduct experiments to assess its impact on LLMs, specifically focusing on gpt-3.5-turbo. Our goal is to investigate whether COTTON can enhance the performance of LLMs and to what extent this enhancement is achieved.

Gpt-3.5-turbo serves as a representative model in code generation and is one of the most state-of-the-art LLMs available. We consider its performance for code generation in zero-shot scenarios and then under the guidance of various COTs, including the CoTs generated by COTTON, the CoTs self-generated by Self-planning [13], SCoT [14], and the Self-CoT method proposed in our study.

Table 10 shows the impact of COTTON and other self-generated CoT methods on gpt-3.5-turbo. The experimental

TABLE 10
Evaluating COTTON and other CoT prompt methods on gpt-3.5-turbo

Corpus	Pass@1	CoT-Pass@1			
		COTTON	Self-Planning	SCoT	Self-CoT
HE	56.10	74.39	**77.44**	76.83	**77.44**
OE	26.97	43.26	42.70	44.38	**45.51**
HE-p	57.32	76.22	78.66	78.66	**79.27**

results show that the gpt-3.5-turbo model demonstrates a significant improvement in code generation performance when guided by various CoTs. When comparing the zero-shot scenarios with the guided code generation, we observe a substantial increase in Pass@1, indicating that the incorporation of CoTs, including those generated by COTTON, positively influences the model's ability to generate more accurate code. Notably, it even exceeds the performance of the GPT4 zero-shot scenario on the HumanEval dataset (67.0). Moreover, the Self-CoT method proposed in our study can perform better than Self-Planning and SCoT on all datasets.

6.2 Ablation study

The data collection methodology for CodeCoT-9k relies on heuristic rules and multi-agent alignment. The heuristic rules (including Code Filtering, Document Filtering, and Similarity Filtering) have been widely used in training various LLMs (such as DeepSeek-Coder [66], StarCoder [40],

TABLE 11
The performance comparison of zero-shot, fine-tuning and COTTON with StarCoder series models.

| | StarCoder | Zero-shot | Pass@1 | | COTTON | CoT-Pass@1 | |
			Fine-tune	Instruction-tune		COTTON w. Fine-tune	COTTON w. Instruction-tune
HE	1B	12.80	14.63 (↑ 14.30%)	16.46 (↑ 28.59%)	25.00 (↑ 95.31%)	25.00 (↑ 95.31%)	**28.05** (↑ 119.14%)
	3B	17.07	22.56 (↑ 32.16%)	24.36 (↑ 42.71%)	30.49 (↑ 78.62%)	31.71 (↑ 85.76%)	**35.37** (↑ 107.21%)
	7B	21.95	25.61 (↑ 16.67%)	26.83 (↑ 22.23%)	37.20 (↑ 69.48%)	38.41 (↑ 74.99%)	**40.24** (↑ 83.33%)
	16B	34.10	37.80 (↑ 10.85%)	39.02 (↑ 14.43%)	43.90 (↑ 28.74%)	44.51 (↑ 30.53%)	**46.95** (↑ 37.68%)
OE	1B	8.99	8.43	11.24 (↑ 25.03%)	17.42 (↑ 93.77%)	17.98 (↑ 100%)	**20.22** (↑ 124.92%)
	3B	11.24	18.54 (↑ 64.95%)	19.66 (↑ 74.91%)	19.10 (↑ 69.93%)	20.79 (↑ 84.96%)	**22.47** (↑ 99.91%)
	7B	23.03	24.16 (↑ 4.91%)	26.40 (↑ 14.63%)	33.15 (↑ 43.94%)	33.71 (↑ 46.37%)	**35.96** (↑ 56.14%)
	16B	27.53	29.78 (↑ 8.17%)	31.46 (↑ 14.28%)	39.33 (↑ 42.86%)	40.45 (↑ 46.93%)	**42.70** (↑ 55.10%)
HE-p	1B	13.41	14.63 (↑ 9.10%)	16.46 (↑ 22.74%)	26.22 (↑ 95.53%)	26.83 (↑ 100.07%)	**28.66** (↑ 113.72%)
	3B	17.07	22.56 (↑ 32.16%)	24.36 (↑ 42.71%)	31.71 (↑ 85.76%)	31.71 (↑ 85.76%)	**36.59** (↑ 114.35%)
	7B	22.56	26.22 (↑ 16.22%)	26.83 (↑ 22.23%)	38.41 (↑ 70.26%)	38.41 (↑ 74.99%)	**40.24** (↑ 83.33%)
	16B	34.10	38.41 (↑ 12.64%)	39.02 (↑ 14.43%)	44.51 (↑ 30.53%)	44.51 (↑ 30.53%)	**46.95** (↑ 37.68%)
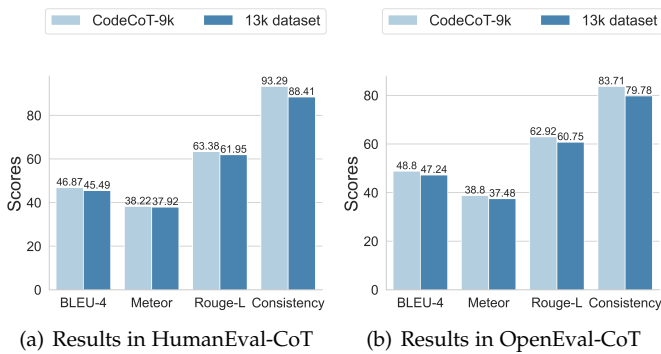
Fig. 5. The performance of COTTON by whether using the Consistency Checker

CodeT5+ [41], and WizardCoder [67]), and their importance has been demonstrated. The multi-agent alignment, particularly the Quality Checker, has been proven by Gunasekar et al. [33] to be beneficial for training models with educational value. However, the specific contribution of the Consistency Checker to the overall quality of the dataset remains unclear. To evaluate the impact of the Consistency Checker, we conduct an ablation experiment. When the Consistency Checker is not applied, the dataset consists of 13k examples. In this experiment, we train COTTON separately on CodeCoT-9k and a 13k dataset without Consistency Checker filtering.

The ablation results of the Consistency Checker are shown in Fig. 5 where we can observe that the use of the Consistency Checker has a significant impact on the performance of COTTON. Specifically, in terms of the Consistency metric, without the Consistency Checker for filtering, COTTON shows a relative performance decrease of 5.23% on HumanEval-CoT and 4.69% on OpenEval-CoT.

6.3 Comparing COTTON with other tune methods

In this subsection, we first compare COTTON with traditional fine-tuning method using the StarCoder-series models as examples. The empirical study [36] shows that LoRA usually makes the most favorable trade-off between cost and performance in code generation tasks. Hence we mainly compare COTTON with LoRA in our discussion. During the traditional fine-tuning process, natural language is used as input, and code is generated as output.

Given that existing state-of-the-art instruction-tuning methods usually involve additional data, such as human feedback [68] and compiler feedback [69], [70], which can significantly improve the performance of downstream tasks of the model, we follow the method of Zheng et al. [71] and add CoT in the form of multi-turn dialogues during instruction-tuning to explore the impact of instruction-tuning on the performance of the model.

The results shown in Table 11 demonstrate that the combination of StarCoder-7B with COTTON can exceed the performance of StarCoder-16B in zero-shot scenarios and can even achieve results comparable to a fine-tuned/instruction-tuned StarCoder-16B model. In addition, we find that multi-tune instruction-tuning method outperform traditional fine-tuning method, which is a finding similar to that of Zheng et al. [71]. We also find that COTTON, as an independent CoT generation model, can be combined with the fine-tuned/instruction-tuned model to further improve the performance of the model in code generation tasks.

Furthermore, when considering GPU across model deployment, inference, and training with float16 precision (as shown in Table 12), we carefully set parameters including a maximum input and output length of 256 and a batch size of 1. It is important to highlight that fine-tuning a 7B model with LoRA already pushes the limits on a single consumer-grade GPU (GeForce RTX series). For individual developers, fine-tuning a 16B model would require multiple professional-grade GPUs (Quadro series and Tesla series), leading to significantly higher hardware costs compared to fine-tuning a 7B model. In contrast, COTTON enables performance enhancement across multiple models *without* the need of fine-tuning individual model, which is highly desirable.

6.4 The impact of different decoding strategies

In the field of natural language processing [72], [73], the performance of downstream tasks depends not only on the quality of the model itself but also on the decoding strategy in the prediction stage. In this subsection, we evaluate the impact of different decoding strategies on the performance

TABLE 12
The comparison of GPU memory usage with StarCoder series models.

StarCoder	Deploy	Inference	Training (Lora)
1B	2.34 GB	∼3.50 GB	∼12.00 GB
3B	6.15 GB	∼7.27 GB	∼16.00 GB
7B	14.72 GB	∼15.90 GB	∼23.00 GB
16B	31.88 GB	∼33.00 GB	∼40.00 GB

of COTTON. We consider four classical decoding strategies [74] commonly used in text generation:

- **Greedy Search.** This strategy selects the token with the highest probability at each decoding step, resulting in a deterministic output.
- **Multinomial Sampling.** This strategy samples tokens from the probability distribution at each decoding step, introducing randomness into the output.
- **Beam Search.** This strategy maintains a beam of the top-k partial sequences and selects the most probable complete sequence based on the joint probability.
- **Constrastive Search.** This strategy [75] aims to optimize a trade-off between exploration and exploitation during decoding by considering both the model's predicted probability and the contrastive loss.

(a) The experimental results in HumanEval-CoT

(b) The experimental results in OpenEval-CoT

Fig. 6. The performance of COTTON by using different decoding strategies

We show the evaluation results of using different decoding strategies in Fig. 6 where we can find that the choice of decoding strategy does not have a significant impact on the performance of COTTON. Specifically, we only observe the minimal difference (i.e., variations of no more than 0.01) in terms of BLEU-4, METEOR, and Rouge-L performance metrics. This suggests that COTTON consistently performs well across different decoding strategies, generating high-quality COTs that are comparable in terms of these performance metrics. Therefore, the effectiveness of COTTON in generating high-quality code does not heavily rely on the specific decoding strategy.

6.5 The impact of different prompt methods

In this subsection, we will evaluate the impact of different prompt methods on the performance of COTTON. We consider four classical prompt methods [76] and one method without prompt:

- **None:** This method refers to not using any prompt and letting the model generate CoTs without any specific guidance.
- **Prefix Tuning:** This method [77] involves adding a task-specific sequence of vectors before the model input as a prefix.
- **Prompt Tuning:** This method [37] learns soft prompts to condition frozen language models for specific downstream tasks.
- **P-Tuning:** This method [78] utilizes trainable continuous prompt embeddings in combination with discrete prompts.
- **Alpaca Prompt.** This method [79] designs a specific template for the prompts used to finetune LoRA models.
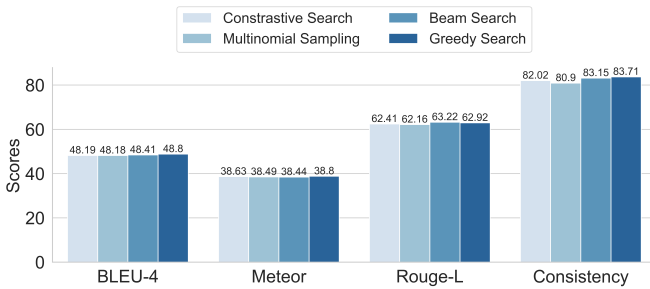
Table 13 shows the performance of COTTON under different prompt methods on the HumanEval-CoT and OpenEval-CoT datasets, measured by BLEU-4 and Meteor scores. Among the prompt methods, Prefix Tuning consistently achieves the highest scores on both datasets. It demonstrates the effectiveness of adding task-specific prefixes to guide the model's responses. Prompt Tuning and Alpaca Prompt also show competitive performance, indicating the benefits of using learned soft prompts or specific prompt templates.

However, our proposed prompt template outperforms all the other prompt methods. It achieves the highest scores on both datasets, surpassing the baseline and other prompt-based approaches. This suggests that our method successfully leverages the strengths of the prompt methods while introducing additional improvements to enhance the model's performance.

This discussion shows that considering a variety of different prompt methods during model training can have unexpected benefits in improving model performance. Therefore, when targeting different downstream tasks, it is necessary to carefully try different prompt methods to maximize the potential of the model.

6.6 The impact of different hyper-parameter setting

Recall that R and $Alpha$ are the most important hyper-parameters for LoRA, where R represents the LoRA attention dimension and $Alpha$ represents the scaling parameter for LoRA. In this subsection, we analyze the effect of different hyper-parameter settings on the performance of COTTON.

The results are shown in Table 14, where we compare COTTON with other different hyper-parameter settings, and

TABLE 13
The performance of COTTON under different prompt methods

Method	HumanEval-CoT		OpenEval-CoT	
	BLEU-4	Meteor	BLEU-4	Meteor
None	44.98	36.12	44.89	37.18
Prefix Tuning	46.25	37.48	47.72	38.23
Prompt Tuning	45.72	36.35	46.81	37.52
P-Tuning	45.43	36.87	46.23	37.67
Alpaca Prompt	45.56	37.05	47.36	37.90
Ours	46.87	38.22	48.80	38.80

TABLE 14
The performance of COTTON under different hyper-parameter settings

R	$Alpha$	HumanEval-CoT		OpenEval-CoT	
		BLEU-4	Meteor	BLEU-4	Meteor
4	8	45.74	37.33	46.75	37.68
	16	46.29	37.87	47.20	38.12
	32	46.83	38.08	48.10	38.23
8	8	46.06	37.55	47.65	38.39
	16	46.87	**38.22**	**48.80**	**38.80**
	32	**46.95**	38.15	48.60	38.54
16	8	45.52	37.12	46.88	37.25
	16	46.41	37.91	47.95	37.41
	32	46.68	38.02	48.45	38.12

the performance is evaluated in terms of the BLEU-4 and METEOR metrics for both HumanEval-CoT and OpenEval-CoT. Based on the results, we find that a higher value of $Alpha$ tends to yield better performance. On the other hand, the impact of R on COTTON performance is relatively less significant: varying its value does not lead to substantial performance improvement in terms of these evaluation metrics.

6.7 Threats to Validity

In this subsection, we analyze potential threats to the validity of our empirical study.

Threats to Internal Validity. The first internal threat is the possibility of implementation faults in COTTON. To mitigate this threat, we conduct a careful code inspection of the implementation and utilize well-established third-party libraries (such as PyTorch and Transformers). The second internal threat is the implementation correctness of the considered baselines. To alleviate this threat, we implemented all baselines based on their shared models on platforms such as Hugging Face[12].

Threats to External Validity. The main external threat lies in the datasets used in our study. To mitigate this threat, we start by selecting widely used open datasets as the raw data for CoT generation. We then apply three heuristic rule-based cleaning methods to preprocess these datasets. Moreover, we propose a multi-agent alignment-based method that leverages multiple agents to align and clean the data. For the code generation dataset, we select popular HumanEval and HumanEval-plus datasets. To ensure the generalization ability of COTTON on another dataset, we also construct a new code generation dataset OpenEval, which provides a

12. https://huggingface.co/models

diverse and challenging set of programming tasks that could evaluate the model's ability to generate high-quality code.

Threats to Construct Validity. The main construct threat is related to the metrics used in our automated evaluation. By treating CoT generation as a text generation problem, we utilize metrics based on term overlap (such as **BLEU**, **METEOR**, and **ROUGE-L**), which have been commonly used in similar studies on programming language processing [80], [81]. Moreover, we introduce the **Consistency** metric to assess alignment based on the nature of the code generation task under investigation. Furthermore, to ensure the generalizability of COTTON, we collect a new code generation dataset OpenEval. We manually design five test cases for each problem to ensure as much quality as possible for OpenEval. Finally, we employ **Pass@1** and **CoT-Pass@1** to evaluate the performance of code generation models. It is noticed that the Pass@k metric would significantly increase the computational cost of the evaluation, especially when k is large, in our study we decided to focus only on the Pass@1 metric in the evaluation, which reflects the actual ability of the model to generate the correct code for a given input in one go, which is the most critical aspect of functional correctness. To complement automated evaluation, we also conducted a human study to validate the effectiveness of our proposed approach further. To guarantee the quality of our human study, we follow the human study methodology used in previous studies of similar software engineering tasks [61], [82].

7 RELATED WORK

In this section, we summarize related studies on neural code generation and chain of thought generation.

7.1 Code Generation

Earlier research on neural code generation predominantly relied on heuristic rules and expert systems, such as probabilistic grammar-based approaches [83], [84] and domain-specific language techniques [85], [86]. However, these methods exhibited inflexibility and lacked scalability [86]. Other studies attempted to utilize static language models such as n-gram [87], [88] and Hidden Markov models [89], but they struggled with sparse vector representations and failed to effectively capture long-term dependencies. Consequently, researchers turned their attention to neural networks, specifically CNN [90], [91], RNN [92], [93], and LSTM [94], [95], to model the relationship between natural language and code. In 2017, the Transformer model [23], initially designed for machine translation, was introduced and later applied to the task of neural code generation [96], [97]. However, these deep learning models require a substantial amount of (labeled) natural language and code pairs for training and have inherent limitations in their capabilities.

With the development of language models, researchers have seen a diagram shift to pre-training and fine-tuning in neural code generation. At this stage, models with less than 1 billion parameters are commonly used. For instance, CodeBERT [54] has been utilized for automatic generation of exploit code [43], [98], while CodeGPT [56] has been applied to automatic generation of Java and Python code.

PLBART [57] and CodeT5 [58] are pre-trained on multiple programming languages, making them suitable as base models for multi-language code generation tasks. Built upon the CodeT5 model, code of mixed programming styles (turducken) was generated where multi-task learning was used to enforce syntactic constraints [44]. In addition, models such as JuPyT5 [99] and PyMT5 [99] focus on the Python language specifically. They construct dedicated datasets and further enhance code generation performance through fine-tuning.

More recently, there has been a remarkable advancement in the development of LLMs with over 10 billion parameters. These models have demonstrated the ability to generate code in a zero-shot manner. A notable milestone is Codex [25], which boasts an impressive 12 billion parameters. Codex has showcased its capabilities by solving 72.31% of challenging Python programming problems created by humans. This model has also been successfully integrated into the commercial product Copilot.[13] Following the success of Codex, several other LLMs designed specifically for code generation tasks have emerged. For example, AlphaCode [2] focuses on solving competitive-level programming problems, while InCoder [100] supports code completion in arbitrary positions using bidirectional contexts. Additional models include CodeGen [19], [101], StarCoder [40], WizardCoder [67], OctoCoder [102] and CodeLlama [27], which have demonstrated their potential to solve complex programming problems and assist developers in various settings.

However, fine-tuning these LLMs can be computationally expensive and resource-intensive. The focus of the current paper is the ℓLMs which we intend to use for code generation without updating parameters. In particular, we leverage the newly introduced CoT technology, which, as shown in the current paper, can be an effective means to improve the quality and accuracy of the generated code by ℓLM. Our study provides a cost-effective alternative to ℓLMs directly for code generation, making them more accessible for a wider range of applications and users.

7.2 Chain of Thought Generation

As the number of model parameters and volume of training data increase, LLMs have demonstrated impressive reasoning capabilities [103]. Recently, there has been a growing interest in enhancing the performance of LLMs in downstream tasks without the need to update model parameters. One way to achieve this is to harness the inferential reasoning abilities of LLMs, a notable approach of which is the CoT prompting method [12]. This method enables LLMs to provide reliable answers through thoughtful consideration and explanation. Various approaches have been studied aiming to generate more accurate and reliable CoT possibly using LLMs themselves. For instance, He et al. [104] incorporate external knowledge as supporting information to generate more faithful CoT. Wang et al. [105] utilize self-consistency by generating multiple inference paths and answers, selecting the most frequently occurring answer as the final output, thereby improving the quality of CoT. Creswell et

al. [106] propose a selection-inference framework that employs LLMs as general processing modules. This framework alternates between selection and inference steps, generating a series of interpretable, causal reasoning steps leading to the final answer. Zhou et al. [107] introduce the least-to-most prompting method, which breaks down complex problems into simpler subproblems and solves them sequentially.

The methods have limitations in relying on LLMs with more than 100 billion parameters. Researchers have developed smaller language models via knowledge distillation. Ho et al. [108] introduced Fine-tune-CoT, which leverages GPT3(175B) as a reasoning teacher to enable complex reasoning in smaller models, thereby significantly reducing the model size requirements. Li et al. [109] proposed Symbolic Chain-of-Thought Distillation (SCoTD), a method for training smaller student models using rationalizations sampled from a much larger teacher model. This approach distills the reasoning capabilities of the larger model into smaller models. Shridhar et al. [110] utilized the step-by-step Chain-of-Thought (CoT) reasoning capabilities of larger models and distilled these abilities into smaller models.

Our primary objective is to generate high-quality CoTs for code generation at a manageable cost. Apart from the domain-specific features in constructing CoT, we design a stand-alone model of relatively small sizes dedicated to CoT generation which can improve the performance code generation.

7.3 Multi-agent Collaboration

Multi-agent collaboration refers to a framework where multiple autonomous agents interact with each other in a shared environment. These agents can be program scripts, software bots, or robots, each with their capabilities, goals, and perceptions [111]. They can communicate, cooperate, compete, or negotiate with each other to achieve complex goals or solve problems. LLMs within multi-agent collaboration systems is an emerging area of research in the deep learning community [112].

For instance, Zhang et al. [113] proposed ProAgent, a system designed for robotic tasks that analyzes the current context, anticipates teammates' intentions, and formulates strategies based on this reasoning. Chen et al. [114] developed VisualGPT, which leverages vision-based Pretrained Language Models for image captioning tasks. In terms of code generation, Huang et al. [115] proposed AgentCoder, a novel solution comprising a multi-agent framework with the programmer agent, the test designer agent, and the test executor agent.

Our proposed multi-agent alignment-based cleaning method contains Quality Checker, CoT Generator, and Consistency Checker. Quality Checker serves the purpose of evaluating and filtering the educational significance of the code, while the CoT Generator focuses on generating the corresponding CoTs. Additionally, the Consistency Checker plays a crucial role in assessing and filtering the semantic consistency of the generated CoTs with the code. These three agents possess unique capabilities and task goals, enabling them to communicate and collaborate effectively to produce high-quality CoTs.

---

13. https://github.com/features/copilot

8 CONCLUSION

In this paper, we have introduced CodeCoT-9k and COT-TON, which leverage lightweight language models (with parameters less than 10B) to generate high-quality CoT for code generation. We have demonstrated the effectiveness and efficiency of COTTON in generating high-quality code CoTs. When equipped with these CoTs, existing ℓLMs have demonstrated significant performance improvements in code generation tasks. Our study enables ℓLMs to perform the code generation task better without additional updates to model parameters, making them more accessible to individual users.

Potential future research includes further improving the performance of COTTON. We plan to extend COTTON to other programming languages and explore various potentially promising techniques, such as retrieval augmented generation, adversarial training and contrastive learning. Moreover, based on the results of our study, it is important and promising to explore the lightweight language models for other software engineering tasks.

ACKNOWLEDGEMENTS

REFERENCES

[1] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[3] R. A. Poldrack, T. Lu, and G. Beguš, "Ai-assisted coding: Experiments with gpt-4," *arXiv preprint arXiv:2304.13187*, 2023.

[4] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *arXiv preprint arXiv:2305.01210*, 2023.

[5] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," *arXiv preprint arXiv:2307.10169*, 2023.

[6] A. Nazir and Z. Wang, "A comprehensive survey of chatgpt: Advancements, applications, prospects, and challenges," *Meta-Radiology*, p. 100022, 2023.

[7] Y. Fu, H. Peng, L. Ou, A. Sabharwal, and T. Khot, "Specializing smaller language models towards multi-step reasoning," *arXiv preprint arXiv:2301.12726*, 2023.

[8] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan, "Improving chatgpt prompt for code generation," *arXiv preprint arXiv:2305.08360*, 2023.

[9] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[10] J. Cao, M. Li, M. Wen, and S.-c. Cheung, "A study on prompt design, advantages and limitations of chatgpt for deep learning program repair," *arXiv preprint arXiv:2304.08191*, 2023.

[11] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[13] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," *arXiv preprint arXiv:2303.06689*, 2023.

[14] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *arXiv preprint arXiv*, vol. 2305, 2023.

[15] T. Y. Zhuo, "Large language models are state-of-the-art evaluators of code generation," *arXiv preprint arXiv:2304.14317*, 2023.

[16] J. Huang and K. C.-C. Chang, "Towards reasoning in large language models: A survey," *arXiv preprint arXiv:2212.10403*, 2022.

[17] S. Qiao, Y. Ou, N. Zhang, X. Chen, Y. Yao, S. Deng, C. Tan, F. Huang, and H. Chen, "Reasoning with language model prompting: A survey," *arXiv preprint arXiv:2212.09597*, 2022.

[18] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[19] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[20] D. N. Manh, N. L. Hai, A. T. Dau, A. M. Nguyen, K. Nghiem, J. Guo, and N. D. Bui, "The vault: A comprehensive multilingual dataset for advancing code understanding and generation," *arXiv preprint arXiv:2305.06156*, 2023.

[21] B. Zhang and R. Sennrich, "Root mean square layer normalization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[22] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[24] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2021.

[25] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[26] A. Zeng, X. Liu, Z. Du, Z. Wang, H. Lai, M. Ding, Z. Yang, Y. Xu, W. Zheng, X. Xia *et al.*, "Glm-130b: An open bilingual pre-trained model," in *The Eleventh International Conference on Learning Representations*, 2022.

[27] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[28] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[29] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–73, 2022.

[30] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *arXiv preprint arXiv:2104.09864*, 2021.

[31] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[32] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[33] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.

[34] N. D. Q. B. Anh T. V. Dau, Jin L.C. Guo, "Bootstrapping code-text pretrained language model to detect inconsistency between code and comment," *EACL 2024 - Demonstration track*, 2024.

[35] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *arXiv preprint arXiv:2308.11432*, 2023.

[36] T. Y. Zhuo, A. Zebaze, N. Suppattarachai, L. von Werra, H. de Vries, Q. Liu, and N. Muennighoff, "Astraios: Parameter-efficient instruction tuning code large language models," *arXiv preprint arXiv:2401.00788*, 2024.

[37] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3045–3059.

[38] Y. Gu, X. Han, Z. Liu, and M. Huang, "Ppt: Pre-trained prompt tuning for few-shot learning," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 8410–8423.

[39] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," *arXiv preprint arXiv:2108.11590*, 2021.

[40] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[41] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[42] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of nlp and software engineering: A survey on language models for code," 2023.

[43] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "Exploitgen: Template-augmented exploit code generation based on codebert," *Journal of Systems and Software*, vol. 197, p. 111577, 2023.

[44] G. Yang, Y. Zhou, X. Chen, X. Zhang, Y. Xu, T. Han, and T. Chen, "A syntax-guided multi-task learning approach for turducken-style code generation," *arXiv preprint arXiv:2303.05061*, 2023.

[45] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, and H. Lin, "Ccgir: Information retrieval-based code comment generation method for smart contracts," *Knowledge-Based Systems*, vol. 237, p. 107858, 2022.

[46] G. Yang, X. Chen, Y. Zhou, and C. Yu, "Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 361–372.

[47] G. Yang, Y. Zhou, X. Zhang, X. Chen, T. Han, and T. Chen, "Assessing and improving syntactic adversarial robustness of pre-trained models for code translation," 2023.

[48] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[49] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[50] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[51] I. Team, "Internlm: A multilingual language model with progressively enhanced capabilities," https://github.com/InternLM/InternLM, 2023.

[52] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[53] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[54] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[55] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2020.

[56] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[57] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668.

[58] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[59] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30.

[60] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[61] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 349–360.

[62] K. Liu, X. Chen, C. Chen, X. Xie, and Z. Cui, "Automated question title reformulation by mining modification logs from stack overflow," *IEEE Transactions on Software Engineering*, 2023.

[63] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "Skcoder: A sketch-based approach for automatic code generation," *arXiv preprint arXiv:2302.06144*, 2023.

[64] R. A. Fisher, "On the interpretation of $\chi 2$ from contingency tables, and the calculation of p," *Journal of the royal statistical society*, vol. 85, no. 1, pp. 87–94, 1922.

[65] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[66] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[67] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[68] A. Chen, J. Scheurer, T. Korbak, J. A. Campos, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez, "Improving code generation by training with natural language feedback," *arXiv preprint arXiv:2303.16749*, 2023.

[69] S. Dou, Y. Liu, H. Jia, L. Xiong, E. Zhou, J. Shan, C. Huang, W. Shen, X. Fan, Z. Xi *et al.*, "Stepcoder: Improve code generation with reinforcement learning from compiler feedback," *arXiv preprint arXiv:2402.01391*, 2024.

[70] J. Yang, A. Prabhakar, K. Narasimhan, and S. Yao, "Intercode: Standardizing and benchmarking interactive coding with execution feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[71] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and

X. Yue, "Opencodeinterpreter: Integrating code generation with execution and refinement," *arXiv preprint arXiv:2402.14658*, 2024.

[72] L. Massarelli, F. Petroni, A. Piktus, M. Ott, T. Rocktäschel, V. Plachouras, F. Silvestri, and S. Riedel, "How decoding strategies affect the verifiability of generated text," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 223–235.

[73] G. Wiher, C. Meister, and R. Cotterell, "On decoding strategies for neural text generators," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 997–1012, 2022.

[74] H. Face, "Text generation strategies," 2023, https://huggingface.co/docs/transformers/generation_strategies#decoding-strategies.

[75] Y. Su, T. Lan, Y. Wang, D. Yogatama, L. Kong, and N. Collier, "A contrastive framework for neural text generation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 548–21 561, 2022.

[76] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "Peft: State-of-the-art parameter-efficient fine-tuning methods," https://github.com/huggingface/peft, 2022.

[77] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 4582–4597.

[78] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang, "Gpt understands, too," *AI Open*, 2023.

[79] "alpaca-lora: Instruct-tune llama on consumer hardware," https://github.com/tloen/alpaca-lora, 2023.

[80] Z. Gao, X. Xia, J. Grundy, D. Lo, and Y.-F. Li, "Generating question titles for stack overflow from mined code snippets," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–37, 2020.

[81] K. Liu, G. Yang, X. Chen, and C. Yu, "Sotitle: A transformer-based post title generation approach for stack overflow," in *Proceedings of The 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)*, 2022.

[82] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 135–146.

[83] T. Cohn, P. Blunsom, and S. Goldwater, "Inducing tree-substitution grammars," *The Journal of Machine Learning Research*, vol. 11, pp. 3053–3096, 2010.

[84] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 2014, pp. 472–483.

[85] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 2010, pp. 13–24.

[86] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7443–7464.

[87] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 532–542.

[88] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 419–428.

[89] I. Sutskever, G. E. Hinton, and G. W. Taylor, "The recurrent temporal restricted boltzmann machine," *Advances in neural information processing systems*, vol. 21, 2008.

[90] Z. Liu, Y. Dou, J. Jiang, and J. Xu, "Automatic code generation of convolutional neural networks in fpga implementation," in *2016 International conference on field-programmable technology (FPT)*. IEEE, 2016, pp. 61–68.

[91] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062.

[92] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2016, pp. 2073–2083.

[93] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.

[94] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka, "Tree-to-sequence attentional neural machine translation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 823–833.

[95] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 440–450.

[96] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[97] M. Shah, R. Shenoy, and R. Shankarmani, "Natural language to python source code using transformers," in *2021 International Conference on Intelligent Technologies (CONIT)*. IEEE, 2021, pp. 1–4.

[98] P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic, "Evil: exploiting software via natural language," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 321–332.

[99] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan, "Training and evaluating a jupyter notebook data science assistant," *arXiv preprint arXiv:2201.12901*, 2022.

[100] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[101] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[102] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," *arXiv preprint arXiv:2308.07124*, 2023.

[103] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[104] H. He, H. Zhang, and D. Roth, "Rethinking with retrieval: Faithful large language model inference," *arXiv preprint arXiv:2301.00303*, 2022.

[105] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations*, 2022.

[106] A. Creswell, M. Shanahan, and I. Higgins, "Selection-inference: Exploiting large language models for interpretable logical reasoning," in *The Eleventh International Conference on Learning Representations*, 2022.

[107] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. V. Le *et al.*, "Least-to-most prompting enables complex reasoning in large language models," in *The Eleventh International Conference on Learning Representations*, 2022.

[108] N. Ho, L. Schmid, and S.-Y. Yun, "Large language models are reasoning teachers," *arXiv preprint arXiv:2212.10071*, 2022.

[109] L. H. Li, J. Hessel, Y. Yu, X. Ren, K.-W. Chang, and Y. Choi, "Symbolic chain-of-thought distillation: Small models can also" think" step-by-step," *arXiv preprint arXiv:2306.14050*, 2023.

[110] K. Shridhar, A. Stolfo, and M. Sachan, "Distilling multi-step reasoning capabilities of large language models into smaller models via semantic decompositions," *arXiv preprint arXiv:2212.00193*, 2022.

[111] W. Du and S. Ding, "A survey on multi-agent deep reinforcement learning: from the perspective of challenges and applications," *Artificial Intelligence Review*, vol. 54, pp. 3215–3238, 2021.

[112] S. Agashe, Y. Fan, and X. E. Wang, "Evaluating multi-agent coordination abilities in large language models," *arXiv preprint arXiv:2310.03903*, 2023.

[113] C. Zhang, K. Yang, S. Hu, Z. Wang, G. Li, Y. Sun, C. Zhang, Z. Zhang, A. Liu, S.-C. Zhu *et al.*, "Proagent: Building proac-

tive cooperative ai with large language models," *arXiv preprint arXiv:2308.11339*, 2023.

[114] J. Chen, H. Guo, K. Yi, B. Li, and M. Elhoseiny, "Visualgpt: Data-efficient adaptation of pretrained language models for image captioning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 18 030–18 040.

[115] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.

Xiangyu Zhang is currently pursuing a Master's degree at the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation and model interpretability.

Guang Yang received the M.D. degree in computer technology from Nantong University, Nantong, in 2022. Then he is currently pursuing the Ph.D degree at Nanjing University of Aeronautics and Astronautics, Nanjing. His research interest is AI4SE and he has authored or co-authored more than 20 papers in refereed journals or conferences, such as ACM Transactions on Software Engineering and Methodology (TOSEM), Empirical Software Engineering, Journal of Systems and Software, International Conference on Software Maintenance and Evolution (ICSME), and International Conference on Software Analysis, Evolution and Reengineering (SANER). More information about him can be found at: https://ntdxyg.github.io/

Terry Yue Zhuo received the B.Sc. degree from the University of New South Wales, Sydney in 2021. Then he received the B.Sc. (Honours) degree from Monash University, Australia, in 2022. He is currently pursuing a Ph.D. degree at Monash University and CSIRO's Data61, Australia. He is also working at CSIRO's Data61 as a research engineer and visiting the SOAR group at Singapore Management University. He has authored more than 15 papers in refereed journals or conferences, such as The Web Conference (WWW), ACM Transactions on Software Engineering and Methodology (TOSEM), Annual Meeting of the Association for Computational Linguistics (ACL) and Conference on Empirical Methods in Natural Language Processing (EMNLP). He received the best paper award in the Deep Learning for Code (DL4C) workshop in ICRL 2023. His research interests include empirical software engineering, code intelligence and responsible AI. More information about him can be found at https://terryyz.github.io/.

Yu Zhou is a full professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his BSc degree in 2004 and PhD degree in 2009, both in Computer Science from Nanjing University China. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politechnico di Milano, Italy. From 2015-2016, he visited the SEAL lab at University of Zurich Switzerland, where he is also an adjunct researcher. His current research interests are mainly generative models for software engineering, software evolution analysis, mining software repositories, and reliability analysis. He has been supported by several national research programs in China. More information about him can be found at: https://csyuzhou.github.io/.

Xiang Chen received the B.Sc. degree in the School of Management from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the Department of Information Science and Technology, Nantong University, Nantong, China. He has authored or co-authored more than 120 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Information and Software Technology, Journal of Systems and Software, International Conference on Software Engineering (ICSE), The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), International Conference Automated Software Engineering (ASE). His research interests include software engineering, in particular software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology. More information about him can be found at: https://xchencs.github.io/index.html

Taolue Chen received the Bachelor and Master degrees from Nanjing University, China, both in Computer Science. He was a junior researcher (OiO) at the Centrum Wiskunde & Informatica (CWI) and acquired the PhD degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a lecturer at the School of Computing and Mathematical Sciences, Birkbeck, University of London. He had been a postdoctoral researcher at University of Oxford (UK). His research spans Software Engineering, Program Language, Verification and Machine Learning, and the present research focus is neuro-symbolic software engineering. He has published over 140 papers in journals and conferences such as POPL, LICS, CAV, OOPSLA, ICSE, ESEC/FSE, ASE, ETAPS (TACAS, FoSSaCS, ESOP, FASE), NeurIPS, ICLR, IJCAI, AAAI, EMNLP and IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), Empirical Software Engineering, ACM Transactions on Computational Logic (TOCL), Information and Computation, Logical Methods in Computer Science. He won the Best Paper Award of SETTA'20, the 1st Prize in the CCF Software Prototype Competition 2022, and the QF_Strings (Single Query Track) at the International Satisfiability Modulo Theories Competition 2023. He has served editorial board or program committee for various international journals and conferences. More information about him can be found at https://chentaolue.github.io/.