Beyond Code: Evaluate Thought Steps for Complex Code Generation

Liuwen Cao[1,2], **Yi Cai**[1,2], **Hongkui He**[1,2], **Hailin Huang**[1,2], **Jiexin Wang**[1,2,*]
[1]School of Software Engineering, South China University of Technology
[2]Key Laboratory of Big Data and Intelligent Robot
(South China University of Technology) Ministry of Education
caoliuwencc@163.com, ycai@scut.edu.cn, jiexinwang@scut.edu.cn

Abstract

Code generation aims to generate code in a general-purpose programming language, such as C++, based on natural language intents. Existing efforts primarily focus on relatively simple programming problems and fail to evaluate the thought process involved in complex programming scenarios. In this paper, we introduce "steps-guided code generation", a task that assesses the quality of both thought steps and code implementation to evaluate the overall management of handling a complex programming problem. To support this task, we construct CodeStepsEval, a real-world scenario dataset of complex programming problems in the C++ programming language with varying levels of difficulty. Comprehensive experiments on this dataset demonstrate the importance of high-quality steps in enhancing code generation performance and the challenges faced by the code LLMs in this task.

Keywords: code generation, real-world complex programming dataset

1. Introduction

Programming is a highly vital skill in modern society, offering the ability to automate tasks and enhance efficiency. However, mastering the art of translating natural language(NL) intents into executable code typically requires years of study and practice. Code generation, also known as program synthesis, addresses this challenge by automatically generating code based on NL intents. This not only makes programming more accessible and efficient for non-programmers but also provides significant benefits to experienced developers. One notable example is GitHub Copilot,[1] an in-IDE developer assistant that automatically generates code based on the user's context, greatly aiding efficient and effective code writing.

Various methods have been introduced for code generation. Early works (Ling et al., 2016; Xiao et al., 2016; Sun et al., 2019) typically approach code generation as a sequence-to-sequence problem and focus on developing neural architectures. However, these models often struggle to learn sophisticated programming patterns. More recently, Large Language Models (LLMs) pre-trained on numerous code data, such as GPT-3 (Brown et al., 2020) have opened up new opportunities for addressing these limitations. Initialization with GPT-3, OpenAI's Codex (Chen et al., 2021) demonstrates impressive performance by correctly solving 30-70% of novel Python problems. However, these LLMs primarily focus on relatively simple programming problems where implementing a function to achieve a specific small-scale functionality, such as "Write a function to get the n smallest items from a list". They face challenges in handling complex programming problems. For instance, in the APPS benchmark which contains programming competition problems, CodeRL proposed by (Le et al., 2022) surpasses the Codex model and achieves the best results across differ-

NL-intent:
There is a given sequence of integers $a_1, a_2, ..., a_n$, where every number is from 1 to 3 inclusively. You have to replace the minimum number of numbers in it so that all the numbers in the sequence are equal to each other. The first line contains an integer n $(1 \le n \le 10^6)$. The second line contains a sequence of integer $a_1, a_2, ..., a_n$ $(1 \le a_i \le 3)$. Print the minimum number of replacements needed to be performed to make all the numbers in the sequence equal.

Steps:
Read the input integer n
Loop n times, read integer x and count the occurrences of x
Output n minus the maximum occurrence of elements

Solution:
```
n = int(input())
s = [0] * 4
for i in range(1, n+1):
    x = int(input())
    s[x] += 1

print(n - max(s[1], max(s[2], s[3])))
```

Test cases:
Input >>
9
1 3 2 2 2 1 1 2 3
Output >>
5

Figure 1: An example of a programming problem adapted from CodeStepsEval dataset. Only one test case is shown in this example.

*Corresponding authors
[1]https://github.com/features/copilot/

LREC-COLING 2024, pages 2296–2306
20-25 May, 2024.

ent difficulty levels(Hendrycks et al., 2021a). However, CodeRL only achieves 4.48% and 2.36% for interview and competitive problems, respectively, significantly lower than its pass rate of 15.27% for introductory problems[2]. This highlights the significant challenge that current code generation models face in handling complex programming problems.

In a real-world programming context, handling complex programming problems necessitates two distinct capabilities. The first is to conceive high-level thoughts that outline the steps to address a given problem(Dijkstra et al., 1976; Jiang et al., 2023). The second is to implement code based on these thought steps and NL intents. These two abilities together constitute a comprehensive reflection of one's problem-solving capabilities in complex programming scenarios. For instance, during job interviews, interviewers assess a developer's programming proficiency by simultaneously evaluating their problem-solving conceptual thought and code implementation capabilities. Existing work concentrates on code-only evaluation while neglecting the evaluation of thought steps. As a result, it fails to depict programming proficiency in complex programming contexts.

To fill this gap, we propose a new code generation task: steps-guided code generation. This task focuses on real-world complex programming problems and evaluates not just the quality of implemented code but also the quality of generated thought steps. For steps evaluation, we design a steps-wise similarity(SWS) metric to evaluate the quality of steps at a sub-step level. Nevertheless, the lack of code generation datasets with annotated steps hinders the development of steps-guided code generation. Therefore, we devise a semi-automatic pipeline that aids in the creation of a code generation dataset containing thought steps.

Our constructed dataset, CodeStepsEval, consists of 10,479 programming problems in C++ language categorized into different difficulty levels: basic, challenging, and advanced. As illustrated in Figure 1, each problem in CodeStepsEval includes NL intents, thought steps, a solution with a reference C++ program, and multiple test cases to evaluate the correctness of the generated code. To demonstrate the effectiveness of CodeStepsEval in advancing research on code generation, we conduct extensive experiments using three baseline models with parameter sizes ranging from 350M to 2.7B. These experiments include assessing the model's performance with the guidance of the thought steps, investigating the factors contributing to the effects of the steps, and evaluating the models' performance across different levels of difficulty.

Our main contributions can be summarized as follows:

- We introduce a steps-guided code generation task that simulates the real-world scenario in solving complex programming problems.
- We create the CodeStepsEval for the introduced task in C++ programming language. Unlike prior work on code generation which mostly focuses on simple problems and code-only evaluation, we evaluate models on their ability to generate C++ code for complex problems with thought steps evaluation into consideration.
- We undertake a comprehensive analysis of the curated dataset, which not only shows the quality and utility of the resulting data but also substantiates the effectiveness of thought steps to tackle complex programming problems.

2. Related works

Code Generation (Ling et al., 2016) treats code generation as a sequence-to-sequence modeling problem and proposes a structured attention mechanism to generate the source code. To exploit syntactic and semantic constraints of code, many researchers turn to the Seq2Tree model for code generation. For instance, (Yin and Neubig, 2017, 2018) propose a Seq2Tree model powered by code grammar to capture the code syntax as prior knowledge. From a different direction, (Yin and Neubig, 2019) improves the code generation performance by reranking an N-best list of predicted codes. (Xu et al., 2020) enhances the code generation model by incorporating the extracted external knowledge such as API documentation. Furthermore, (Wei et al., 2019) boosts the performance based on dual learning with the help of the code summarization task.

Recently, Large Language Models (LLMs) such as GPT-3 (Brown et al., 2020) have opened up new opportunities for addressing code generation. OpenAI's Codex (Chen et al., 2021), trained on 54 million software repositories from GitHub, demonstrates impressive performance by correctly solving 30-70% of novel Python problems. Following that, various code LLMs like InCoder(Fried et al., 2022), CodeGen(Nijkamp et al., 2022), AlphaCode(Li et al., 2022), and CodeT5+(Wang et al., 2023) have been emerging, yielding impressive results. Generally, these code LLMs demonstrate a preference and more proficiency in solving relatively simple Python programming problems, and they face challenges in tackling complex programming tasks. This has sparked some studies with

[2]As mentioned in APPS, interview-level and competition-level problems are more complex than introductory ones. This arises from their inclusion of problems in programming competitions such as ACM

a concentration on addressing complex programming problems. For instance, (Jiang et al., 2023) introduces a non-training self-planning approach based on few-shot prompting to enhance the performance of the code-davinci-002(GPT-3.5 series) model on HumanEval-X. (Li et al., 2023) utilizes a brainstorming strategy to generate diverse thoughts and select them by a trained ranker model. In this work, we unify the stages of thought and code implementation into a comprehensive assessment of complex programming proficiency.

Code generation datasets Several datasets have been proposed for code generation. (Chen et al., 2021) introduces HumanEval, a dataset that encompasses 164 hand-written programming problems in Python and evaluates code by designed test cases. HumanEval-X (Zheng et al., 2023) serves as a multilingual version of the HumanEval dataset, incorporating various programming languages such as C++, Java, and Python. MBXP(Athiwaratkun et al., 2022) is a manually curated multilingual dataset for code generation, it contains 848 C++ programming problems. The dataset mentioned above focuses on relatively simple programming tasks, such as implementing a function to achieve a specific small-scale functionality, for instance, "Write a function to get the n smallest items from a dataset". Differ from the datasets mentioned above, APPS(Hendrycks et al., 2021a) comprises code competition programming problems that involve real-world scenarios with multiple complex requirements. The APPS dataset (Hendrycks et al., 2021a) consists of a total of 10,000 code competition problems(5,000 for training, 5,000 for testing) and their associated Python code. Unlike existing datasets, the goal of our dataset focuses on solving complex programming problems in C++ and additionally serves as a resource to evaluate both thought and code.

3. The CodeStepsEval Dataset

In this section, we provide a comprehensive overview of the creation process for the CodeStepsEval. We begin by introducing the steps-guided code generation task. Subsequently, we explore the creation of CodeStepsEval. Finally, we present the essential statistics of CodeStepsEval and compare it with other code generation datasets.

3.1. Task definition

The steps-guided code generation task initially takes the NL intents as input and produces the thought steps as output. Subsequently, it proceeds to implement the code based on the NL intents and the generated thought steps. Specifically, this task can be formalized as the following:

$$p(c|i) = p(s|i) \cdot p(c|i, s), \tag{1}$$

where i, s, and c denote the NL intents text along with the prompt, the thought steps, and the code.

3.2. Dataset Creation

NL-code collection We focus on algorithmic programming competitions as they encompass complex programming problems that require programmers to have a deep understanding of the problem, devise effective strategies, and apply various algorithms. We collect programming problems from Luogu[3] website, which hosts competitions and offers problems spanning different levels of difficulty. Each problem collected from the website consists of the following four components: (1) NL intents, which describe the problem to be solved; (2) Problem difficulty, indicating the level of difficulty for the problem; (3) Multiple candidate C++ codes, the solutions that successfully pass platform testing. (4) A few input-output pairs that serve as test cases.

Next, we carefully filter out problems that could potentially lead to security vulnerabilities, such as the C++ code "int result = system("rm -rf *");"-this code becomes vulnerable since it could remove all files in our local host by calling the system function. This precaution is taken to prevent the model from learning malicious behaviors. Lastly, we rely on the Google benchmark toolkit[4] and execute all candidate codes to select the optimal C++ code (with the minimum run time) as the reference code in candidate codes, ensuring the inclusion of high-quality solutions[5].

Steps annotation Next, we annotate the collected problems with steps using a semi-automatic approach. We choose the semi-automatic way to obtain the thought steps for two reasons: First, manual annotation of steps requires annotators with high problem-solving skills for complex problems, and thus it is too costly. Second, considering the advancements in large language models (LLMs), these models already demonstrate surprising performance in understanding the problem intents. To facilitate manual annotation and boost efficiency, we begin by employing LLM as a tool to generate preliminary steps, followed by manual refinement of these steps.

Specifically, GPT-3.5-turbo, a powerful large language model, is employed to generate the preliminary steps for each programming problem. The prompt for constructing preliminary steps is as

[3] https://www.luogu.com.cn/

[4] https://github.com/google/benchmark

[5] Note that CodeStepsEval comprises multiple C++ codes for each problem, which can be employed for reinforcement learning or contrastive learning.

follows:
Instruction:*"Your goal is to act as a human being to generate accurate and easy-understand thought steps to solve a programming problem, according to the given description and reference program of the programming problem. The requirement is as follows:*
1. Provide steps in JSON list format.
2. Ensure the generated steps effectively capture the essence of the given code.
Prolem description:
<fill the programming problem description here>
Reference C++ program:
<fill the reference c++ program here>

For manual refinement of steps, we recruited 15 annotators who are proficient in C++ programming language and have actively participated in algorithmic programming competitions. The objective of 10 annotators is to refine the steps for all problems, including ensuring the correctness of all steps, rectifying erroneous steps, removing redundant steps, and incorporating essential steps that might be overlooked. 5 annotators verify the correctness of all modified steps, and those with unanimous agreement are retained. The entire refinement takes two months to complete.

3.3. Dataset Statistics

After applying all of the aforementioned filtering steps, we successfully obtained the CodeStepsEval dataset[6], which consists of 10,479 data instances for steps-guided code generation. Additionally, We randomly chose 1,000 instances as the test set with 300, 350, and 350 instances for basic level, challenging level, and advanced level, respectively. In addition, as test cases play a crucial role in verifying a code's correctness and can be used for reinforcement learning (Le et al., 2022), we manually augment test cases in the test set based on the NL intents and verify the correctness of test cases with the reference code. On average, we include 1.6 test cases in the training set and 17.6 test cases in the test set. Detailed statistics of CodeStepsEval is displayed in Table 1

Table 2 compares CodeStepsEval to existing code generation datasets. Compared with C++ datasets MBXP and HumanEval-X, codeStepsEval exhibits a longer intent and code. This implies that CodeStepsEval is more challenging and capable of assessing a model's proficiency in solving complex programming problems. Additionally, CodeStepsEval has more test cases. This can substantially reduce the number of "false passed" codes. Moreover, compared with competition-related datasets such as APPS, CodeStepsEval is focused on the C++ programming language and enables the measurement of the quality of thought steps, thereby reflecting the real-world programming scenario in solving complex programming problems.

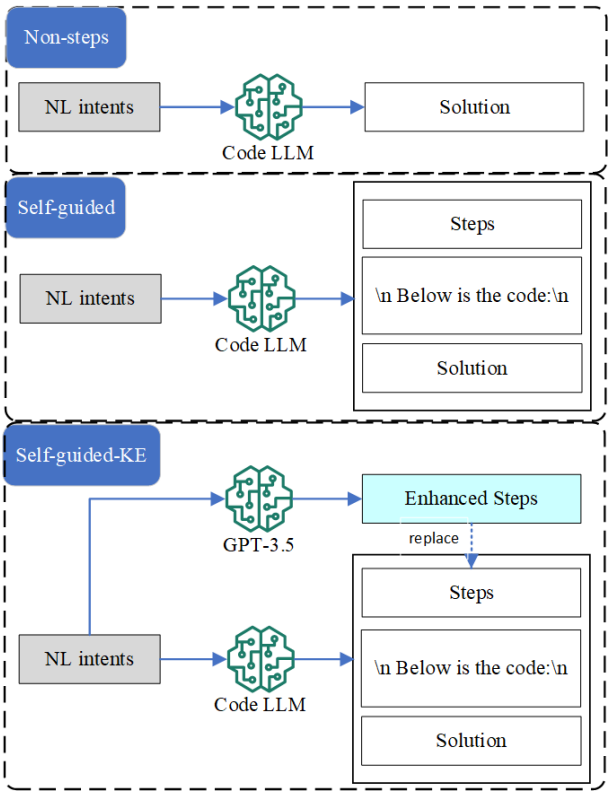

Figure 2: Three baselines.

4. Baselines

As shown in Figure 2, we adopt three approaches to train and evaluate across multiple code LLMs on the CodeStepsEval dataset, serving as baselines for code generation tasks: no-steps, self-guided, and self-guided-KE.

Non-steps: This baseline can be regarded as classic code generation in view of models taking NL intents as input and directly generating code without the phase of generating steps. For this baseline, we prompt models using the following input: "NL intents:`\n` + p_str + `\n` + Answer:`\n`", where p_str represents the raw text of NL intents.

Self-guided: In this baseline, as illustrated in Equation 1, the code LLM initially generates steps and subsequently generates code based on both the steps and the NL intents. It is noteworthy that we can employ casual language modeling to generate steps and code at the implementation level. Specifically, we use the same prompt in non-steps but the label in this baseline is the concatenation of steps and code, with "\n Below is the code`\n`" as a separator. As shown in Figure 2, the separator can

[6] https://github.com/galbya/CodeStepsEval

		Basic level	Challenge level	Advanced level	Total/Avg
Train	# Problems	468	2,795	6,216	9,479
	Avg NL Length	223	262	298	284
	Avg Code Length	44	76	174	138
	Avg Solutions	5.3	5.0	4.3	4.5
	Avg Steps	5.8	5.9	5.7	5.7
	Avg Steps Length	58	67	69	68
	Avg Test Cases	1.4	1.6	1.7	1.6
Test	# Problems	300	350	350	1,000
	Avg NL Length	199	256	270	237
	Avg Code Length	41	72	164	82
	Avg Solutions	4.5	4.9	4.3	4.6
	Avg Steps	5.5	5.6	5.6	5.6
	Avg Steps Length	56	66	65	62
	Avg Test Cases	21.2	17.3	12.2	17.6

Table 1: Detailed statistics of CodeStepsEval.

	MBXP	HumanEval	HumanEval-X	APPS	CodeStepsEval
#Problems	848*	164	164*	10,000	1,0479*
Source	HW	HW	HW	Competitions	Competitions
Intent Length	39.2	61.7	61.3	319.4	236.8
Code Length	29.2	24.3	21.1	62.2	82.6
Test Cases	3.1	7.7	7.7	20.0	17.6
Thought Steps	No	No	No	No	Yes
Programming Language	C++	Python	C++	Python	C++

Table 2: Comparison of existing code generation datasets. Source: HW(Hand-written) or Competition(Programming competition website). * denotes the number of instances in C++ programming language

function as a specific indicator when we need to extract the steps. This means that models must learn to generate the separator between steps and code during the training process. In our experiments, we find that fine-tuned models are able to do this without difficulty.

Self-guided-KE: Given that code LLMs are dedicated to code understanding and generation, their capability for step generation may be limited. We propose a knowledge-enhanced self-guided baseline, denoted as "self-guided-KE". At this baseline, we use the GPT-3.5-turbo model for generating the enhanced steps. During testing, we integrate the enhanced steps into the self-guided prompt for code generation.

4.1. Models

For each baseline, we conducted the experimental evaluation on various existing code LLMs:

CodeGen(Nijkamp et al., 2022): a family of code LLMs available in different parameter sizes (350M, 2.7B, 6.1B, and 16.1B). Models are trained using a combination of NL and code data collected from THEPILE (Gao et al., 2020)). We use the multilingual version with parameter sizes of 350M and 2.7B as baseline models, both of them are additionally trained on the BIGQUERY[7], which contains code under an open-source license in multiple programming languages.

Starcoderbase (Fried et al., 2022): an ensemble of models with parameter sizes ranging from 1B to 7B. It is trained on 80+ programming languages from The Stack(Kocetkov et al., 2022), utilizing the fill-in-the-middle objective. For our experiments, we utilize the Starcoderbase model with 1B parameters.

5. Experiments

5.1. Fine-tuning

We perform fine-tuning for each baseline on the CodeStepsEval training set with both problem text and problem format(described in Section 4). During fine-tuning, We adopt the cross-entropy loss as the training loss. Note that the prompt text is excluded from the training loss. For CodeGen-350M, we fine-tuned all parameters in the model. For Starcoderbase-1B and CodeGen-2.7B, due to the limitation of computation resources, we resorted to parameter-efficient fine-tuning approaches that only fine-tune a small number of model parameters while freezing most parameters of the pre-trained LMs. Specifically, we adopt the LoRA approach (Hu et al., 2021) on the query, key, and value matrices with a rank of 8. This results in fine-tuning the Starcoderbase-1B model with 3.47M param-

[7]https://cloud.google.com/bigquery/public-data

Algorithm 1 The computation procedure of SWS and F1

Input: Ground truth steps $s = \{s_1, s_2, ..., s_M\}$
predicted steps $s^{'} = \{s_1^{'}, s_2^{'}, ..., s_N^{'}\}$, threshold α
Define j = 1, sim_sum = 0, $correct$ = 0
for i=1,2,...,M **do**
  While j <= N **do**
    Calculate the similarity sim between s_i and $s_j^{'}$
    If sim >=α
      j = j+1
      sim_sum += sim
      $correct$ += 1; break
    j = j+1
  end While
end for
// SWS computation
SWS = sim_sum/M
// F1 computation
$precision = correct$ / N; $recall = correct/M$
$F_1 = 2 * \frac{precision*recall}{precision+recall}$

eters and the CodeGen-2.7B model with 1.64M parameters, respectively.

5.2. Metrics

We evaluate steps from two dimensions: the overall perspective and the local perspective. For the overall perspective evaluation, we employ automated metrics BLEU@k, Rouge-L@k, and Similarity@k. These metrics consider steps as a whole and measure the character or semantic similarity between reference and predicted steps. BLEU@k is defined as the highest BLEU-4 score among k sampled steps, while Rouge-L@k and Similarity@k are similarly defined as the highest Rouge-L scores and Similarity scores among k sampled steps. We compute the Similarity score using sentence-transformers[8]. For the assessment from a local perspective, we involve the designed stepwise Similarity (SWS) metric and the F1 metric. Given the reference steps $s = \{s_1, s_2, ..., s_M\}$, and the generated steps $s^{'} = \{s_1^{'}, s_2^{'}, ..., s_N^{'}\}$, SWS calculates the similarity between each reference substep and each predicted substep. Sub-steps are deemed identical if their similarity surpasses a predefined threshold α. We compute the sum of similarities for all identical sub-steps and utilize the average Similarity as the value of SWS. In Algorithm 1, we summarize the computation procedure for SWS and the F1 metric.

To evaluate the code, we rely on CodeBLEU@k and Pass@k(Kulal et al., 2019). CodeBLEU(Ren et al., 2020) is an automatic match-based metric specifically designed for code generation tasks. It is based on BLEU(Papineni et al., 2002) but considers syntactic and semantic matches based on the code structure and n-gram match. Pass@k is an execution-based metric for measuring the exact functional correctness of generated code, where k code samples are generated for each problem. A problem is considered solved if any sample passes all the unit tests. Since this computation of Pass@k can have high variance, we follow (Chen et al., 2021) and use the unbiased version of Pass@k as the estimator:

$$Pass@k = E_{problems}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}] \quad (2)$$

where $k \leq n$ is the number of samples and $c \leq n$ is the number of codes that pass all test cases. $1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$ is the estimated Pass@k for a single problem. E is the expectation of Pass@k over all problems.

5.3. Settings

For training, we truncate the problem(the concatenation of prompt and program) up to 2048 tokens. We adopt AdamW(Loshchilov and Hutter, 2019) with a learning rate of 2e-5 and weight decay of 0.01 to update the model parameters for 5 epochs. The batch size is set to 4, 2, and 2 for CodeGen-350M, Starcoderbase-1B, and CodeGen-2.7B, respectively. For testing, the α in the SWS metric is set to 0.8. We compute the average pass@k with n=20 and k=1, 5, 10. Following (Chen et al., 2021), we use a temperature of 0.2 for pass@1 and pass@5, and a temperature of 0.8 for pass@10. We adopt nucleus sampling (Holtzman et al., 2019) with a top-p value of 0.95 for all program evaluations in this work. We use the Nvidia 3090 GPU and Nvidia Tesla P100 GPU for fine-tuning and inference, respectively.

5.4. Results

Main Results We present the main results for all baselines in Table 3. We find that the performance of the self-guided models is inferior to that of the non-step models. In the case of the CodeGen-2.7B model, the self-guided approach yields lower results on the pass@1 metric compared to the non-step approach. We hypothesize that these results may arise because code LLMs like CodeGen are primarily tailored for code generation, potentially resulting in limited proficiency in generating steps. Consequently, the poor quality of generated steps may lead the model to produce erroneous code.

In addition, we observe that the performance of self-guided-KE models surpasses the self-guided models by a large margin. When compared to self-guided CodeGen-350M, the self-guided-KE CodeGen-350M shows over ten times improvements in SWS@10 and F1@10 scores, respectively. Regarding the code evaluation metrics, Pass@5 and CodeBLEU@5, the self-guided-KE Starcoderbase-1B model outperforms the self-guided Starcoderbase-1B model by 141% and

[8] https://www.sbert.net/

Setting	Model	Steps evaluation					Code evaluation					
		BL@10	RG@10	Sim@10	SWS@10	F1@10	Pass@1	Pass@5	Pass@10	CB@1	CB@5	CB@10
Non-Steps	CodeGen-350M	-	-	-	-	-	1.81	5.56	8.08	22.89	26.98	28.53
	Starcoderbase-1B	-	-	-	-	-	4.73	11.53	14.88	28.21	33.15	34.86
	CodeGen-2.7B	-	-	-	-	-	2.12	5.79	8.59	22.96	26.96	28.35
Self-guided	CodeGen-350M	8.37	31.83	0.64	5.16	5.77	1.01	4.11	6.87	22.81	28.91	30.94
	Starcoderbase-1B	8.25	31.83	0.65	7.21	8.32	1.32	5.15	8.33	24.44	30.51	32.58
	CodeGen-2.7B	6.57	29.77	0.61	5.58	5.85	1.07	4.17	6.88	23.70	29.90	32.03
self-guided -KE	CodeGen-350M	57.42	71.29	0.91	58.76	63.41	2.79	7.88	11.29	27.06	33.81	35.98
	Starcoderbase-1B	57.75	71.33	0.91	59.40	63.75	5.30	12.40	17.30	28.97	35.08	37.44
	CodeGen-2.7B	56.81	70.63	0.89	58.13	62.96	2.38	7.70	11.41	26.68	32.82	35.06

Table 3: Main results on the test set. BL, RG, Sim, and CB denote BLEU-4, Rouge-L, Similarity, and CodeBLEU, respectively

Setting	Model	Failed Type(%)			
		SynE	SemE	Timeout	RE
Non-Steps	CodeGen-350M	13.09	72.55	0.55	1.07
	Starcoderbase-1B	17.31	63.59	0.89	2.00
	CodeGen-2.7B	15.87	70.74	0.45	1.07
Self-guided	CodeGen-350M	32.22	54.77	2.70	3.22
	Starcoderbase-1B	27.30	57.88	2.04	3.57
	CodeGen-2.7B	31.75	54.53	3.81	3.44
Self-guided-KE	CodeGen-350M	39.34	44.92	2.81	3.30
	Starcoderbase-1B	31.57	49.34	2.36	3.76
	CodeGen-2.7B	36.32	47.69	3.11	4.04

Table 4: Failed type percentage on failed programs in the test set. SynE(Syntax Error): generated code fails to compile (e.g., missing semicolons or lacking braces). SemE(Semantic Error): the code generated by the model can compile but does not align with the expected output of the test cases. Timeout: the program's run time exceeds the predefined limit. RE(Runtime Error): the program is able to compile without errors but encounters errors during execution, such as out of bounds of an array.

14.9%. Moreover, we find that with the substantial enhancement in step quality, self-guided-KE models consistently outperform non-steps models in terms of code evaluation. Considering the Pass@10 and CodeBLEU@10 scores, the self-guided-KE Starcoderbase-1B model exhibits a 16.2% and 7.6% improvement over the self-guided Starcoderbase-1B model. This indicates that high-quality steps play a pivotal role in enhancing the model's ability to address programming problems. Furthermore, this implies that the primary reason for the diminished effectiveness of self-guided models lies in the low quality of their generation steps as poor-quality steps predicted by models can lead to misguided guidance. It also highlights the importance of the model's capability to predict correct steps in the steps-guided code generation task.

Failed Program Analysis To explore how steps improve the model's performance, we conducted a qualitative failed type analysis on generated codes that failed to pass test cases. The results are given in Table 4. We observe that with the guidance of steps, the steps-related models(self-guided and self-guided-KE) show a significant reduction in semantic error, while leading to an increase in syntax errors(non-steps CodeGen-350M vs self-guided CodeGen-350M, non-steps Starcoderbase-1B vs self-guided-KE Starcoderbase-1B) and a slight rise in timeout and runtime errors. Furthermore, we also find that with higher-quality steps provided, self-guided-KE models exhibit a further reduction in semantic errors(Self-guided CodeGen-2.7B vs Self-guided-KE CodeGen-2.7B). As a result, we can conclude that steps contribute to enhancing the semantic correctness of syntactically correct code while having a negative impact on generating syntactically correct codes. We hypothesize that this might be due to the model's overemphasis on the guidance of steps while leading to a neglect of syntax in the code implementation.

Comparing different difficulty To explore the performance of the model at different difficulty levels, we report the steps and code evaluation results of all baselines across three difficulty levels. As shown in Table 5, with the increase in difficulty, the model's performance across three baselines gradually decreases, highlighting the substantial challenges that remain in current models for tackling complex programming problems. Additionally, We observe that the self-guided-KE Starcoderbase model exhibited a 196% improvement in Pas@5 performance compared to the self-guided Starcoderbase model at the basic level of difficulty. However, the improvement was only 94% and 3.5% at the challenging level and advanced level, respectively. This suggests that the enhancement gains by the self-guided-KE model are evident in addressing basic-level programming problems. As the complexity of the problems increases, the importance of correct code implementation becomes increasingly prominent.

Effect of the α As shown in Algorithm 1, the α in SWS metric is an important hyper-parameter that controls the threshold for step similarity. we vary α from 0.6 to 0.9 with an increment of 0.1 and inspect the performance of the self-guided Starcoderbase-1B model across various SWS@5 intervals. The larger α indicates that we are stricter in determining whether two steps are aligned. Tabel 7 shows the results. we can find that at higher α, SWS@5 at higher intervals is more robust to imply the higher performance of code. For instance, at alpha=0.6,

Setting	Models	Basic level				Challenging level				Advanced level			
		SWS@5	F1@5	P@5	CB@5	SWS@5	F1@5	P@5	CB@5	SWS@5	F1@5	P@5	CB@5
Non-Steps	CodeGen-350M	-	-	8.03	33.51	-	-	3.73	26.05	-	-	4.14	17.88
	Starcoderbase-1B	-	-	21.73	41.14	-	-	6.40	31.73	-	-	2.41	22.37
	CodeGen-2.7B	-	-	7.15	34.92	-	-	5.93	25.88	-	-	3.44	15.79
Self-guided	CodeGen-350M	4.65	5.77	5.12	34.80	5.10	6.38	3.29	28.12	4.96	5.64	3.63	20.59
	Starcoderbase-1B	5.79	6.60	7.18	36.97	5.61	6.97	3.86	29.86	4.84	5.39	3.70	21.09
	CodeGen-2.7B	4.74	6.11	5.90	36.83	3.12	3.61	3.06	29.03	3.64	4.02	2.96	20.08
Self-guided-KE	CodeGen-350M	59.17	62.34	12.66	40.96	57.41	61.09	4.11	32.79	54.89	59.42	4.30	23.83
	Starcoderbase-1B	60.18	63.10	21.27	42.71	57.95	61.54	7.53	34.21	54.73	58.73	3.83	24.13
	CodeGen-2.7B	58.79	61.83	12.71	39.90	56.70	60.84	3.98	32.05	53.95	58.54	3.72	22.62

Table 5: Comparing different difficulty.

Setting	Basic level			Challenging level			Advanced level		
	Alignment	Correctness	False Positive	Alignment	Correctness	False Positive	Alignment	Correctness	False Positive
Self-guided	1.5	3.62	0.2	1.42	3.62	0	0.8	4.4	0
Self-guided-KE	3.25	3.92	0.5	2.45	3.95	0	3.75	4.6	0

Table 6: Human evaluation results from sampled 30 problems with different difficulty levels. Alignment and Correctness denote the average score across problems.

Setting	SWS@5	Basic level		Challenging level		Advanced level	
		P@5	CB@5	P@5	CB@5	P@5	CB@5
$\alpha = 0.6$	0.0-0.2	7.65	37.10	3.56	29.05	3.59	20.65
	0.2-0.4	5.97	36.65	3.75	32.01	4.62	22.93
	0.4-0.6	10.44	35.96	13.42	33.16	0.0	25.05
	0.6-0.8	0.0	37.72	0.0	33.16	-	-
	0.8-1.0	-	-	-	-	-	-
$\alpha = 0.7$	0.0-0.2	7.19	37.19	3.61	29.31	3.91	20.73
	0.2-0.4	8.26	35.27	4.68	32.72	2.40	23.68
	0.4-0.6	0.0	35.31	14.91	38.52	0.0	24.10
	0.6-0.8	0.0	47.18	-	-	-	-
	0.8-1.0	-	-	-	-	-	-
$\alpha = 0.8$	0.0-0.2	7.15	36.90	3.71	29.49	3.75	20.89
	0.2-0.4	8.05	37.50	4.12	34.45	2.98	24.40
	0.4-0.6	0.0	51.68	0.0	44.56	-	-
	0.6-0.8	-	-	-	-	-	-
	0.8-1.0	-	-	-	-	-	-
$\alpha = 0.9$	0.0-0.2	7.01	36.85	3.71	29.69	3.67	20.93
	0.2-0.4	12.25	40.77	7.89	34.55	4.47	25.12
	0.4-0.6	-	-	-	-	-	-
	0.6-0.8	-	-	-	-	-	-
	0.8-1.0	-	-	-	-	-	-

Table 7: Effect of α. - signifies the lack of samples falling within this interval, and as a consequence, there are no associated results.

the model's Pass@5 within the 0.2-0.4 range is inferior to that in the 0.0-0.2 range. In contrast, at alpha=0.9, SWS@5 demonstrates a strong correlation with the quality of the generated code. Opting for a higher alpha value enables a more robust assessment of the correlation between steps quality and code quality.

6. Human evaluation

In this section, we conduct a human evaluation to assess the generated steps and code on the Starcoderbase-1B model of self-guided and self-guided-KE baselines. Specifically, for each baseline, we select a total of 30 problems from the test set, covering basic level (10 problems), challenging level (10 problems), and advanced level (10 problems). We introduce two criteria to assess the generated steps and codes:

- Alignment: Steps comprise precise and abundant information to guide the generation of correct code.
- Correctness: The generated code is accurate to pass all annotated test cases and handling boundary test cases properly.

Then, four undergraduate students majoring in software engineering with C++ programming competition experience are recruited to score generated steps and codes from two criteria. Each criterion is scored from a minimum of 0 to a maximum of 5 points. For each programming problem, we calculate the Alignment and Correctness score based on the average of all student's scores.

The evaluation results are shown in Table 6. First, we observe that, compared to the self-guided baseline, the self-guide-KE baseline showed a significant improvement in the Alignment criteria but a slight improvement in the Correctness criteria. This indicates that the self-guided-KE baseline generated partially correct steps but lacked comprehensive guidance for generating correct code. Second, we find that as the difficulty level increased, both baselines enhanced in the Correct criteria. This suggests that simple problems are more prone to generating false positive codes.

7. Conclusion

In this paper, we introduced the steps-guided code generation task to model the behavior of constructing thought steps and implementing code in solving complex programming problems and have curated CodeStepsEval, a code generation dataset with thought steps. Experimental results and in-depth analysis demonstrate the effectiveness of thought steps and the challenge of this task. We hope that the CodeStepsEval dataset can serve as an important resource for assessing a model's comprehensive programming ability (Thought steps construction and code implementation) in complex programming problems.

8. Acknowledgements

This work was supported by the National Natural Science Foundation of China (62076100), Fundamental Research Funds for the Central Universities, SCUT (x2rjD2230080), the Science and Technology Planning Project of Guangdong Province (2020B0101100002), Guangdong Provincial Fund for Basic and Applied Basic Research - Regional Joint Fund Project (Key Project) (2320191025000031 8,308155351064), CAAI-Huawei MindSpore Open Fund, CCF-Zhipu AI Large Model Fund.

9. References

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 5436–5446. Association for Computational Linguistics.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. In *Proceedings of BigScience Episode# 5–Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136.

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

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

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. 1976. *A discipline of programming*. prentice-hall Englewood Cliffs.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics*, pages 1536–1547. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021a. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on*

Neural Information Processing Systems Datasets and Benchmarks Track.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021b. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2021. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

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

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2022. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023. Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 599–609, Berlin, Germany. Association for Computational Linguistics.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 574–584.

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

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified

text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7055–7062.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H Chi, Quoc V Le, Denny Zhou, et al. 2023. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1341–1350, Berlin, Germany. Association for Computational Linguistics.

Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 476–486.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559, Florence, Italy. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.