Self-Planning Code Generation with Large Language Models

XUE JIANG, YIHONG DONG, LECHENG WANG, ZHENG FANG, QIWEI SHANG, GE LI, ZHI JIN, and WENPIN JIAO, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China

Although large language models (LLMs) have demonstrated impressive ability in code generation, they are still struggling to address the complicated intent provided by humans. It is widely acknowledged that humans typically employ planning to decompose complex problems and schedule solution steps prior to implementation. To this end, we introduce planning into code generation to help the model understand complex intent and reduce the difficulty of problem-solving. This paper proposes a self-planning code generation approach with large language models, which consists of two phases, namely planning phase and implementation phase. Specifically, in the planning phase, LLM plans out concise solution steps from the intent combined with few-shot prompting. Subsequently, in the implementation phase, the model generates code step by step, guided by the preceding solution steps. We conduct extensive experiments on various code-generation benchmarks across multiple programming languages. Experimental results show that self-planning code generation achieves a relative improvement of up to 25.4% in Pass@1 compared to direct code generation, and up to 11.9% compared to Chain-of-Thought of code generation. Moreover, our self-planning approach also enhances the quality of the generated code with respect to correctness, readability, and robustness, as assessed by humans.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Computing methodologies → Artificial intelligence**;

Additional Key Words and Phrases: Code Generation, Large language models, Planning

This research is supported by the National Key R & D Program under Grant No. 2023YFB4503801, the National Natural Science Foundation of China under Grant Nos. 62072007, 62192733, 61832009, and 62192730, and the Key Program of Hubei under Grant JD2023008.

Authors' Contact Information: Xue Jiang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: jiangxue@stu.pku.edu.cn; Yihong Dong, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: dongyh@stu.pku.edu.cn; Lecheng Wang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: wanglecheng@stu.pku.edu.cn; Zheng Fang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: fangz@pku.edu.cn; Qiwei Shang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: shangqiwei@stu.pku.edu.cn; Ge Li (Corresponding author), Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: lige@pku.edu.cn; Zhi Jin, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: zhijin@pku.edu.cn; Wenpin Jiao, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: jwp@sei.pku.edu.cn.

ACM 1557-7392/2024/9-ART182
https://doi.org/10.1145/3672456

ACM Reference format:
Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-Planning Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 182 (September 2024), 30 pages.
https://doi.org/10.1145/3672456

"The art of programming is the art of organizing complexity."

— Edsger W. Dijkstra

1 Introduction

Programming is a pervasive and powerful tool for problem-solving. As one of the most central problems in programming theory, code generation allows machines to program automatically to satisfy human intent expressed in the form of some specification. In recent years, code generation has achieved great progress in both academia and industry [4, 11, 25, 38, 45]. In particular, **large language models (LLMs)** [3, 5] demonstrate impressive code generation abilities, attracting attention from various fields such as artificial intelligence, natural language processing, and software engineering.

In code generation, the human-provided intent is usually a natural language description of "what to do" problem, while the model solves the problem by generating "how to do" code. When the intent is straightforward, it is easy to map to the code, which can be well handled by state-of-the-art code generation models [5, 26]. However, as the problem becomes complicated and scaled, directly generating complex code satisfying intent is challenging for both people and models (even LLMs). In practice, software development is to give software solutions for real-world problems, and the generation of these solutions requires a planning process to guarantee the quality of coding [1, 36, 43]. Accordingly, programmers outline a plan in advance and then complete the entire code step by step following the plan. For complex code generation tasks, such planning is not just beneficial, it is imperative. Therefore, we desire to incorporate planning into code generation. Plan-aided code generation has the following two benefits. (1) It breaks down the complex problem into several easy-to-solve subproblems, which reduces the difficulty of problem-solving. (2) It abstracts the problem and provides instructions for solving it, which helps the model understand how to generate code. Therefore, planning in advance can facilitate the generation of correct codes for complex problems.

Generally, plan-aided code generation presupposes the existence of an approach for converting intent into plan. However, if we build such an approach from scratch, it requires a large amount of resources to label intent-plan pairs for training a model. Few-shot prompting provides an important way of using LLMs without training. A successful technique of few-shot prompting is **chain of thought (CoT)** [57], which enables LLMs to perform step-by-step reasoning to solve reasoning tasks, such as mathematical [9], commonsense [51], and symbolic reasoning [60]. The ability to generate CoTs, as demonstrated by the LLMs, helps us to achieve planning. Furthermore, we can employ few-shot prompting techniques to implement planning without the need for fine-tuning.

Nonetheless, directly applying CoT in the process of planning for code generation remains challenging. Fundamentally, CoT and code[1] are both descriptions of solutions to achieve the final goal, just in different forms: one in natural language and the other in **programming language**

[1]The program is defined as a collection of commands written in a computer language to achieve a specific goal or solve a specific problem.

Intent

```
prime_fib returns 5-th number that is a Fibonacci number and it's also prime.
```

Code & Code CoT

```
def prime_fib(n: int):
    Define a function to check if a number is prime.
    def is_prime(n: int):
    Check if the number is divisible by any number between 2 and n-1.
        if n < 2:
            return False
    Check if the number is divisible by any number between 2 and n-1.
        for i in range(2, n):
            if n % i == 0:
                return False
    If the number is not divisible by any number between 2 and n-1, it's prime.
        return True

    Initialize a list "fib_seq" with the first two Fibonacci numbers 1, 1.
    fib_seq = [1, 1]
    Initialize a counter "counter" with the value of n.
    counter = n
    While the counter is greater than 0, extend the list "fib_seq" with the
    next Fibonacci number.
    while counter > 0:
        fib_seq.append(fib_seq[-1] + fib_seq[-2])
    Check if the last number in the list "fib_seq" is prime.
    If it's prime, decrement the counter.
        if is_prime(fib_seq[-1]):
            counter -= 1
    Return the last number in the list "fib_seq".
    return fib_seq[-1]
```

Result `prime_fib(5) → 89`

Fig. 1. An example of code and CoT for code generation.

(PL).[2] Figure 1 shows an example of code and CoT for code generation. Generating solutions to problems, whether through CoT or code, presents similar challenges. Thus, using CoT directly does not reduce the difficulty of code generation. We should implement planning based on the principle of problem decomposition.

In this article, we propose a self-planning code generation approach with LLMs that exploits the planning capabilities of LLMs themselves to facilitate code generation. Self-planning code generation consists of two phases during inference: (1) Planning phase, LLM generates plans for problems by providing only a few intent-to-plan demonstrations as examples in prompting; (2) Implementation phase, the LLM generates code that adheres to the intent step by step, guided by the plans. Self-planning code generation leverages few-shot prompting to generate plans autonomously without annotating plan corpus and extra training.

Empirical evaluations have provided evidence that self-planning approach can substantially improve the performance of LLMs on code generation. (1) Self-planning approach showed a relative improvement of up to 25.4% in Pass@1 over the direct generation approach and up to 11.9% over CoT approach of code generation. (2) We show that self-planning is an emergent ability that appears on large enough LLMs, but planning can benefit most LLMs. (3) We explore several variants of the self-planning approach in depth and demonstrate that our designed self-planning approach is the optimal choice in these variants. (4) We validate the effectiveness of self-planning approach across multiple PLs including Python, Java, Go, and JavaScript. (5) We analyze the quality (i.e., correctness, readability, and robustness) of the code generated by self-planning approach through human evaluation.

[2]The authors of CoT also discussed the relationship between CoT and program synthesis and execution in extended related work [57], considering their work as a generalization of program synthesis and execution in the natural language domain.

2 Self-Planning

In self-planning code generation, we propose conducting planning prior to the actual code generation by LLMs. This process can be divided into two phases.

Planning Phase. In the planning phase, we employ an LLM to abstract and decompose the intent to obtain a plan for guiding code generation. We take advantage of the ability of LLM to perform planning through few-shot prompting. In few-shot prompting, we require only a few labeled examples to demonstrate the task at hand as a prompt. Subsequently, this prompt is incorporated into the model input during inference, enabling the model to adhere to the prompt in order to accomplish the given task.

In our approach, the prompt C is specifically designed as k examples concatenated together, i.e., $C \triangleq \langle x_1^e \cdot y_1^e \rangle \parallel \langle x_2^e \cdot y_2^e \rangle \parallel \ldots \parallel \langle x_k^e \cdot y_k^e \rangle$, where each example $\langle x_i^e \cdot y_i^e \rangle$ consists of the example intent x_i^e and its associated plan y_i^e to demonstrate the planning task. The plan is a scheduling of subproblems that abstract and decompose from intent, which is set to $y_i^e = \{q_{i,j}\}_{j=1}^n$, where $q_{i,j}$ is jth step in plan y_i^e. During inference, the test-time intent x will be concatenated after the prompt, and $C \parallel x$ will be fed into the LLM $\mathcal{M}$, which will attempt to do planning for the test-time intent. Thus, we can obtain the test-time plan y.

Note that k of the prompt is a fairly low number, meaning we can achieve self-planning by labeling only a few examples demonstrating planning.

Implementation Phase. In the implementation phase, we use the plan obtained during the planning phase to guide LLM in generating code. We append the plan y to the intent x as input for the LLM $\mathcal{M}$. The LLM generates the final code z by way of predicting the next token.

The above two phases can be formalized as the following equation.

$$\begin{aligned} \mathcal{P}(z|x, C) &= \sum_{\hat{y}} \mathcal{P}(z|\hat{y}, x, C) \cdot \mathcal{P}(\hat{y}|x, C), \\ &\propto \mathcal{P}(z|y, x, C) \cdot \mathcal{P}(y|x, C), \end{aligned} \tag{1}$$

where $\hat{y}$ is any of all possible plans, and y denotes one of the plans generated by $\mathcal{M}$. In this article, we adopt the plan with the highest probability as y. We further simplify $\mathcal{P}(z|y, x, C) = \mathcal{P}(z|y, x)$ via conditional independence assumptions, thus

$$\mathcal{P}(z|x, C) \triangleq \underbrace{\mathcal{P}(z|y, x)}_{\text{Implementation phase}} \cdot \underbrace{\mathcal{P}(y|x, C)}_{\text{Planning phase}}. \tag{2}$$

Crafting Prompts for Self-Planning. According to the methodology of few-shot prompting, we need to construct some examples as prompts to instruct the model for planning. Therefore, we prepare k example intents and write plans for each intent following the subsequent principles.

(1) The plan is organized in the form of a numbered list, where each item in the list represents a step.
(2) Every step represents a single, easily implementable sub-task. These sub-tasks are formulated as imperative sentences that start with verbs, focusing on the action needed in each step.
(3) The steps should be written concisely and at a high level, avoiding overly detailed implementation specifics. A step like "Check if a number is prime" is more appropriate than a detailed implementation such as "If the number is less than 2, it's not prime. Check if the number is divisible by any number between 2 and $n - 1$. If the number is not divisible by any number between 2 and $n - 1$, it's prime."

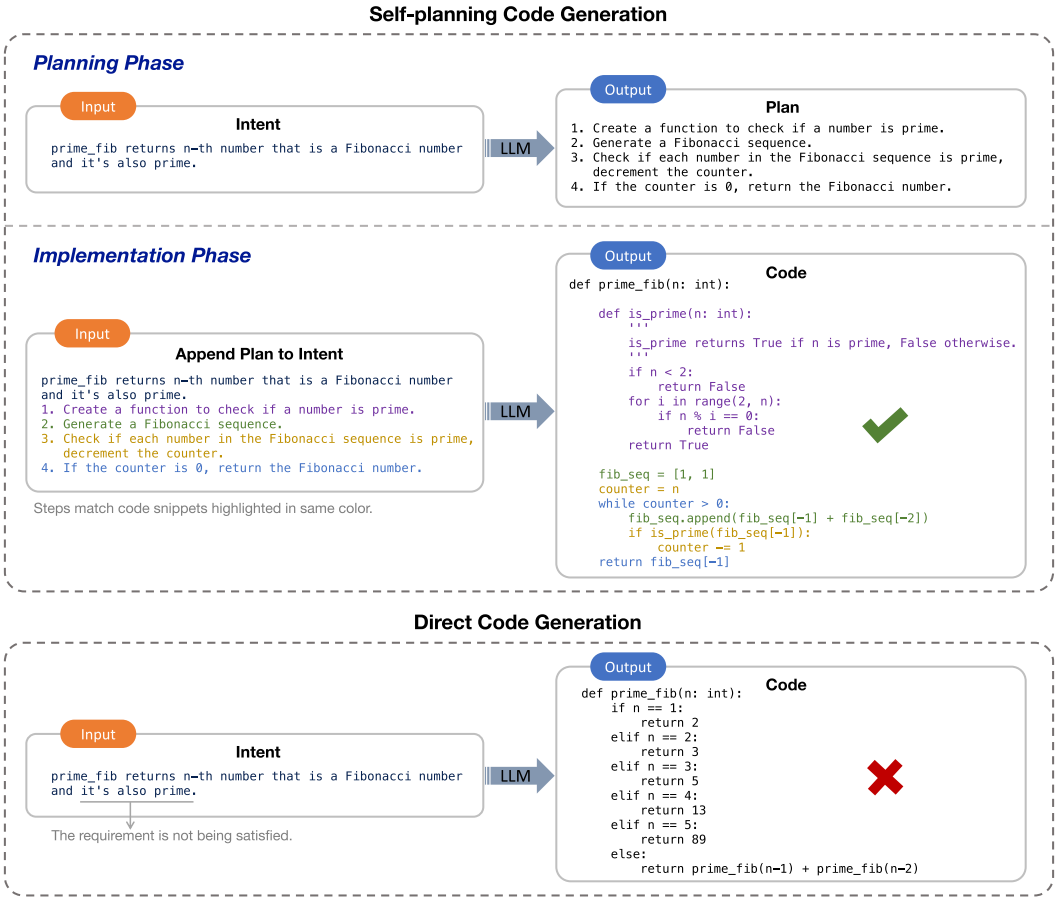

Fig. 2. Self-planning code generation is carried out in two phases (i.e., planning phase and implementation phase): (1) In planning phase, LLM decomposes an intent into a set of easy-to-solve sub-problems and devises a plan for executing the solution steps; (2) In implementation phase, LLM generates code following the intent and plan, which assists self-planning code generation to be capable of handling more difficult problems than direct code generation with LLM. Direct code generation uses the intent as input to the LLM and the LLM generates the code directly. The LLM used here to demonstrate the example of code generation is code-davinci-002.

(4) The execution of the plan happens sequentially, but the plan can incorporate conditional (if) and looping (loop) keywords for more complex structures. This allows for branching paths and loops as necessary while still maintaining the logical progression of the plan.

Self-planning prompts can be freely written within these simple principles, so the crafting of the prompts is relatively straightforward and efficient.

Example. An example of self-planning code generation derived from the real benchmark HumanEval is shown in Figure 2. In the planning phase, human provides an intent to *find the nth number that is a Fibonacci number and it's also prime.*[3] LLM abstracts two subproblems from the intent, i.e., *generating a Fibonacci sequence* and *determining if a number is prime*, and plans four

[3]The self-planning prompt is appended before the intent, guiding LLM to perform planning, which we've omitted in Figure 2 for presentation purposes.

steps to solve the subproblems combinatorial. Then entering the implementation phase, we append the plan to the intent and feed it to LLM. LLM generates code under the navigation of the steps, and surprisingly, it wraps "*determine if a number is prime*" into a subfunction and calls it. At the same time, the previously generated plan significantly augments the readability of the code, which is an important aspect of measuring code quality.

In contrast, LLM (e.g. code-davinci-002) cannot understand that the intent is a combination of multiple problems in direct code generation. LLM knows to write something about "prime" and "Fibonacci," but actually, it generates a confusing code, i.e., it enumerates the first five correct samples[4] and then calculating the Fibonacci numbers, completely losing the requirement to determine *whether it is a prime number*.

In short, when code generation tasks become complex, incorporating planning to handle the complexity becomes necessary.

3 Evaluation

We evaluate our self-planning approach by addressing the following **research questions (RQs)**:

- *RQ1:* How does self-planning approach perform in code generation compared to baseline approaches?
- *RQ2:* How does the self-planning approach perform based on different LLMs?
- *RQ3:* What is the optimal design for the self-planning approach?
- *RQ4:* How does self-planning approach perform in multilingual code generation?
- *RQ5:* How does the complexity of the problem affect self-planning?

3.1 Experiment Setup

3.1.1 Benchmarks. Following the previous work [13, 33, 35, 66], we adopt two public mainstream benchmarks, **Mostly Basic Programming Problems (MBPP)** and HumanEval, along with their multilingual versions and extended test case versions, to evaluate the code generation ability of our self-planning approach and various baselines.

MBPP-Sanitized [2] benchmark is a manually verified subset of MBPP, contains 427 crowd-sourced Python programming problems, covering programming fundamentals, standard library functionality, and more. Each problem consists of a **natural language (NL)** description, a code solution, and 3 automated test cases. For MBPP-sanitized, the NL description is provided as input.

HumanEval [5] is a set of 164 handwritten programming problems, proposed by OpenAI. Each problem includes a function signature, NL description, function body, and several unit tests, with an average of 7.7 tests per problem. For HumanEval, function signature, NL description, and public test cases are provided as input.

HumanEval-X [66] is constructed based on HumanEval to better evaluate the multilingual capabilities of code generation models. HumanEval-X consists of 820 high-quality human-crafted data samples (each with test cases) in Java, JavaScript, Go, and so forth.

MBPP-ET and *HumanEval-ET* [10] are two public expanded versions of MBPP and HumanEval, each including over 100 additional test cases per task. This updated version includes edge test cases that enhance the soundness of code evaluation in comparison to the original benchmark.

3.1.2 Metrics. To assess the accuracy of the generated code, we employ two types of metrics: an execution-based metric, i.e., *Pass@k* and *AvgPassRatio*, and a match-based metric, i.e., *CodeBLEU* (Details of metrics can be found in Appendix A). The execution-based metrics measure the functional

[4]This is related to the fact that the benchmark HumanEval provided five public test cases as additional input, and the model copied them.

correctness of the generated code through executing the given test cases, and the match-based metrics measure the similarity between the generated code and the given reference code.

3.1.3 Basic Baselines. We conduct various experiments, comparing multiple baselines to evaluate distinct aspects. Among these, three baselines—Direct, Code CoT, and Ground-truth Planning, serve as basic baselines in all experiments, highlighting the efficacy of our approach.

Direct generates code using LLMs in a zero-shot setting, implying only intent and no examples are available in the prompt.

Code CoT generates a CoT for each question by using Code CoT prompt (described in Crafting Prompt) and then generates the corresponding code. This approach aligns with self-planning in that both adopt a two-stage generation approach.

Ground-truth Planning is set to investigate the maximum potential of the planning approach in code generation, we directly supply the model with ground-truth plans to perform the implementation phase, skipping the planning phase.

3.1.4 Implementation Details. The implementation details of our experiment are as follows.

Crafting Prompt. We use a different prompt for problems in MBPP and in HumanEval in a fixed way. We employ a simple method to select the questions used to construct the prompts, i.e., random sampling with fixing a seed. Specifically, for HumanEval, we randomly sample eight questions from HumanEval to create the prompt. For MBPP, given MBPP has an additional small training set, we identified four representative categories of problems: string processing, numerical calculations, number theory problems, data structure manipulations, and geometric computations, and eight problems are randomly sampled from these categories. Prompts for our approach and all baselines are constructed utilizing the same problems.

For the self-planning prompt, we manually crafted plans for the problems. Self-planning prompts for HumanEval and MBPP are listed in Appendix C. *For the baseline code CoT, it is implemented by ourselves according to the original paper of CoT, as there is currently no CoT prompt designed for code generation. The way to create a Code CoT prompt is by providing ground-truth code of the 8 problems and then using the instruction "Generate detailed comments for this code." to enable LLMs to generate comments as intermediate steps. To avoid bias caused by errors in LLMs generation and inconsistencies in style, the generated Code CoT prompts are manually reviewed and adapted to the same numbered list format as the self-planning prompts.* The instance of Code CoT prompt can be found in Appendix E. The examples selected from the dataset for prompting will be excluded from the evaluation.

Ground-Truth Plan Generation and Validation. Labeling plans for all datasets is labor-intensive. To mitigate this concern, we utilize the existing ground-truth code to inversely generate a plan, which we then adopt as the ground-truth plan in our experiments. We adopt the few-shot prompting approach to implement this strategy. By reusing self-planning prompts, we construct the corresponding prompts $C^p \triangleq \langle x_1^e \cdot c_1^e \cdot y_1^e \rangle \parallel \langle x_2^e \cdot c_2^e \cdot y_2^e \rangle \parallel \ldots \parallel \langle x_k^e \cdot c_k^e \cdot y_k^e \rangle$, where each example $\langle x_i^e \cdot c_i^e \cdot y_i^e \rangle$ consists of the example intent x_i^e, ground-truth code c_i^e from dataset, plan y_i^e to demonstrate the planning task. The instance of the prompt can be found in Appendix E.

We manually validated the generated ground-truth plans on HumanEval dataset. The experimental results show that most of the generated ground-truth plans follow the principles of self-planning and satisfy the requirements completely, with only a very small number (about 3%) having poorly described steps. Therefore, the generated ground-truth plans are relatively high-quality.

Model Configuration and Evaluation. All basic baselines adopt code-davinci-002 as the base model and set the max generation length to 300 by default. We obtain only one plan in the planning phase by greedy search. For the matrics Pass@1, AvgPassRatio, and CodeBLEU, we use the greedy search setting with temperature 0 and top p 1 to generate one code. For Pass@k ($k \geq 2$), we generate 10 samples for each problem in benchmarks and set temperature to 0.8 and top p to 0.95.

Table 1. Comparison of Self-Planning Approaches and Various Baselines, and the Number After ↑ Denotes the Performance Improvement Achieved in LLM Upon Incorporating the Corresponding Approach, i.e., the Relative Improvement Compared to Approach Direct

Approach	HumanEval			HumanEval-ET		MBPP-Sanitized			MBPP-ET	
	Pass@1	CodeBLEU	AvgPassRatio	Pass@1	AvgPassRatio	Pass@1	CodeBLEU	AvgPassRatio	Pass@1	AvgPassRatio
Code pre-trained models										
AlphaCode (1.1B)	17.1	-	-	-	-	-	-	-	-	-
Incoder (6.7B)	16.0	16.2	28.7	12.2	27.9	14.6	16.9	17.9	11.8	17.4
CodeGeeX (13B)	25.9	23.1	31.4	16.0	36.3	19.9	18.4	38.8	18.2	26.9
CodeGen(16.1B)	34.6	22.8	57.5	26.3	52.6	36.6	24.5	41.6	28.1	36.9
PaLM Coder (560B)	36.0	-	-	-	-	-	-	-	-	-
Direct	48.1	24.0	63.2	37.2	62.7	49.8	25.6	54.8	37.7	46.4
Few-shot	52.6	28.2	74.9	44.2	72.8	53.5	26.1	56.9	38.2	48.8
Code CoT	53.9 (↑ 12.1%)	30.4	75.6	45.5 (↑ 22.3%)	74.7	54.5 (↑ 9.4%)	26.4	58.7	39.6 (↑ 5.0%)	49.9
Self-planning	**60.3** (↑ 25.4%)	28.6	80.8	**46.2** (↑ 24.1%)	76.4	**55.7** (↑ 11.8%)	24.9	59.6	**41.9** (↑ 11.2%)	51.0
Ground-truth Planning	74.4 (↑ 54.7%)	41.0	88.1	57.7 (↑ 55.1%)	85.2	65.1 (↑ 30.7%)	33.7	69.0	50.7 (↑ 34.5%)	60.2

4 Experimental Results

4.1 Comparison With Baselines (RQ1)

Evaluation. We conduct a comparison between the self-planning approach and the following baselines, which comprise our main experimental result. First, we benchmark our approach against a range of widely recognized LLMs pre-trained on code, including AlphaCode (1.1B) [26], Incoder (6.7B) [13], CodeGeeX (13B) [66], CodeGen-Mono (16.1B) [33], and PaLM Coder (560B) [8]. The aim is to ascertain the performance level at which our approach operates relative to these recognized models. Second, we establish code-davinci-002 [5] as our base model and compare self-planning approach with Direct, Few-shot, and Code CoT to demonstrate the effectiveness of our approach, where the Few-shot approach uses the requirements and code pairs as prompt. Third, we investigate the impact of the ground-truth planning approach, which can be considered as an underestimated upper bound for the base model employing self-planning. Fourth, we sampled the code during LLM generation to investigate whether planning affected the diversity of the generated code. Note that sampling is limited to code rather than plan.

Results. The results are summarized in Table 1, which demonstrate a significant effect of self-planning code generation. The self-planning approach is based on a powerful base model, which far outperforms other models pre-trained with code, even PaLM Coder, which has three times the number of parameters. The experimental results suggest that obtaining the plan or Code CoT from the intent can provide a noteworthy advantage in code generation compared to the direct generation of code from the intent. However, the advantage of Code CoT over Few-shot is marginal. Self-planning outperforms both Code CoT and Few-shot across four public code generation datasets, showing a notable improvement in Pass@1 over Code CoT and Few-shot. While our approach demonstrates slightly lower performance on the CodeBLEU metric, it's worth noting that CodeBLEU assesses the similarity between the generated and reference code. This metric can be limiting, as the reference code may not represent the sole valid solution—a critique often associated with match-based metrics. Moreover, we evaluated the impact of utilizing the ground-truth plan in facilitating code generation. This approach simulates to some extent the ground-truth planning provided by developers and provides an understanding of the approximate upper bound (which is actually low) of the self-planning approach. The results in Table 1 indicate a substantial improvement in the use of ground-truth planning, as evidenced by a relative improvement of over 50% and 30% on HumanEval and MBPP-sanitized benchmarks respectively. Overall, the self-planning approach showed a more significant improvement on HumanEval compared to on MBPP-sanitized. We hypothesize that this is due to the fact that in some of the MBPP-sanitized problems, the information provided about

Table 2. Pass@k (%) of Self-Planning and Other Approaches on HumanEval Benchmarks

Approach	Pass@1	Pass@2	Pass@5	Pass@10
Direct	48.1	55.1	64.7	75.0
Code CoT	53.9	56.4	63.5	68.6
Self-planning	60.3	66.0	70.5	76.3
Ground-truth Planning	74.4	75.6	85.3	89.1

Table 3. Performance of Self-Planning Approach Across Various Base LLMs on HumanEval Benchmarks

Approach	Direct			Self-Planning			Planning		
	Pass@1	CodeBLEU	AvgPassRatio	Pass@1	CodeBLEU	AvgPassRatio	Pass@1	CodeBLEU	AvgPassRatio
text-davinci-003 (175B)	55.1	31.5	72.2	65.4	29.6	80.1	65.4	30.2	80.9
code-davinci-002 (175B)	48.1	24.0	63.2	59.0	29.3	77.8	-	-	-
text-davinci-002 (175B)	48.1	24.4	63.1	50.0	28.8	69.4	57.1	30.4	76.3
code-cushman-001 (13B)	34.0	20.8	53.2	30.1	23.1	50.5	44.9	26.7	67.1
text-curie-001 (6.7B)	0.0	4.3	3.2	0.0	12.4	0.0	0.0	12.4	0.0
text-babbage-001 (1B)	0.6	4.8	4.3	0.0	6.2	1.4	0.0	7.9	0.0
text-ada-001 (350M)	0.0	3.9	0.9	0.0	7.4	0.2	0.0	7.8	0.1

(a) Due to the maximum input length limitation, the evaluation of the self-planning is performed in the 4-shot setting.
(b) We use code-davinci-002 for planning and the corresponding LLM for implementation.

intentions is not sufficient to allow the model to perform an effective solution, and even humans are barely able to solve these problems.

Another result of Pass@k, with multiple samples, is shown in Table 2. The diversity and accuracy of the self-planning approach consistently outperform Direct as the sample size increases. In contrast, the diversity of Code CoT decreases rapidly, and its Pass@5 and pass@10 are both lower than Direct, indicating that detailed solution steps entail a loss of diversity. Pass@k for ground-truth planning has been maintained at a high level. It is worth noting that when sampling 10 codes, Ground-truth planning is able to solve close to 90% of tasks.

4.2 Performance on Different LLMs (RQ2)

Evaluation. In this evaluation, we investigate the performance of self-planning approaches on different LLMs. We conduct experiments on the OpenAI language model family, including ada, cabbage, curie, cushman, and davinci. We use three 175B models—text-davinci-002, code-davinci-002, and text-davinci-003, which differ in training strategy and data. Furthermore, we apply the plan generated by code-davinci-002 during the planning phase to the implementation phase of other models, aiming to investigate the impact of planning for models with varying scales. Since the input length limit of the small-size model is restrictive, we use the 4-shot setting for all prompting baselines in this experiment.

Results. The experimental results are presented in Table 3. When the model is small, the impact of self-planning is less pronounced, constrained by the model's inherent abilities. As the model size reaches 13B, the performance of LLMs in code generation begins to exhibit emerging ability, but self-planning ability remains relatively weak. At 175B, self-planning approach consistently outperforms the Direct approach across all models. For the same 175B model, code-davinci-002, fine-tuned on code, demonstrates a stronger self-planning ability than text-davinci-002. Furthermore, self-planning ability can be enhanced through **reinforcement learning with human feedback (RLHF)**. It is evident that the self-planning ability of text-davinci-003 is significantly improved

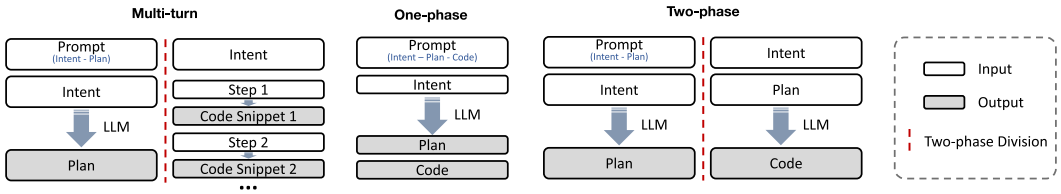

Fig. 3. Illustrations of the variants including two-phase, one-phase, and multi-turn.

compared to text-davinci-002. Therefore, we posit that besides increasing model size, incorporating code training data and RLHF can also enhance the model's self-planning capabilities.

Subsequently, our experiments revealed that employing the plan generated by code-davinci-002 for models with lower abilities significantly improves their performance, particularly in the cases of code-cushman-001 and text-davinci-002. Text-ada-001, text-babbage-001, and text-curie-001 do not exhibit such performance as their inherent code generation ability is almost non-existent. An interesting observation is that when we utilize the plan generated by code-davinci-002 for the text-davinci-003 model, which is an upgraded version of the former, the resulting performance is approximately on par with text-davinci-003 for self-planning. This shows that text-davinci-003 does not improve the planning ability compared to code-davinci-002, what is improved is the code generation ability.

In general, self-planning is an emergent ability that can only appear in large-enough language; however, planning proves to be effective for most of the models.

4.3 Variants of Self-Planning (RQ3)

Evaluation. We explore numerous variants in order to identify better choices for self-planning approach. First, we evaluate three planning and implementation schemes: multi-turn, one-phase, and two-phase. The illustrations of the variants including multi-turn, one-phase, and two-phase are shown in Figure 3. The *Multi-turn* approach involves the iterative use of solution steps of plan to generate the corresponding code snippets that eventually compose the entire code, which is introduced by CodeGen [33]. In contrast, one-phase and two-phase schemes, both single-turn methods, employ all steps (i.e., the plan) to generate the entire code in a single iteration. However, while the *One-phase* approach simultaneously generates the plan and code, the *Two-phase* approach delineates these into separate phases. Note that the one-phase approach requires labeling both the plan and the corresponding code in the prompt, as demonstrated in "Instance of Self-planning Prompt (One-phase)" in Appendix E. Second, we evaluate the effects of self-planning with various example numbers (i.e., n-shot). Third, we explore six intermediate step configurations: Code CoT, Zero-shot CoT, narrative Code CoT, narrative plan, extremely concise plan, and Plan2CoT. *Zero-shot CoT* uses the instruction "Let's think step by step" [22] to produce intermediate steps and code. To eliminate the effect of the generation method, we also establish the baseline *Zero-shot CoT (Two-phase)* for comparison, using the two-phase generation method consistent with our approach. The *Narrative Plan* is an ablation of our proposed plan, i.e., it removes the form of a numbered list and is presented as narrative text. The *Narrative Code CoT* is also an ablation, which makes the Code CoT the same in form as the original CoT. The *Extremely Concise Plan* is an extremely concise version of our proposed plan. It is composed of only a few phrases or verbs (keep only the keywords as much as possible), and an example of it is displayed in Appendix E. *Plan2CoT* means incorporate both the plan and the Code CoT during code generation, i.e., we first generate a plan and then generate a Code CoT, ultimately resulting in code. In this evaluation, in addition to evaluating the code generation quality using metrics Pass@1, CodeBLEU, and AvgPassRatio,

Table 4. Comparison of Self-Planning and its Variants on HumanEval Benchmark

Variant	Pass@1	CodeBLEU	AvgPassRatio	Avg. Num. of I/O/I+O Tokens
Schemes of Planning and Implementation				
Multi-turn	28.2	18.1	47.5	2,218.6/153.8/2,372.4
One-phase	62.8	28.9	78.5	2,354.5/105.9/2,460.4
Number of Few-shot Example				
1-shot	53.2	28.4	74.3	517.9/107.1/625.0
2-shot	54.8	26.7	74.4	700.4/111.1/811.5
4-shot	59.0	29.3	76.9	1,047.8/113.7/1,161.5
Configurations of Intermediate Step				
Zero-shot CoT	18.0	12.6	37.6	276.8/78.8/355.6
Zero-shot CoT (Two-phase)	50.6	23.8	70.9	505.5/381.8/887.3
Narrative Code CoT	50.6	29.0	72.2	2,630.5/118.1/2,748.6
Narrative Plan	55.8	27.7	73.6	1,779.1/115.8/1,894.9
Extremely Concise Plan	61.5	28.3	79.1	1,669.8/84.7/1,754.5
Plan2CoT	56.4	29.5	77.9	4,547.9/185.4/4,733.3
Direct	48.1	24.0	63.2	130.6/41.5/172.1
Code CoT	53.9	30.4	75.6	2,337.5/123.6/2,461.1
Self-Planning (Two-phase, 8-shot)	60.3	28.6	80.8	1,885.3/110.9/1,996.2

we also measure the number of input/output tokens for the various variants. This approach helps understand the possible performance improvement in relation to possible cost increases.

Results. The results of the different variants on HumanEval benchmark are shown in Table 4.

In the result of the group *Schemes of Planning and Implementation*, we find that the multi-turn usually fails to generate correct codes. This is attributed to the nature of LLMs. Since LLMs are trained on enormous concatenated texts and codes to predict the next token, LLMs may have truncation issues, i.e., they cannot precisely control the termination of their output. When using a plan to generate part of functions (usually several statements), it is difficult to define truncation rules. When implemented as a one-phase process, the self-planning approach has been shown to yield slightly improved performance compared to the two-phase way. However, this improvement is achieved at the cost of the increased complexity of crafting prompts. Specifically, the two-phase way only requires providing intent and plan examples in the prompt, whereas the one-phase way requires additional writing of the corresponding code examples.

In the result of the group *Number of Few-shot Example*, we can observe that the performance of self-planning with n-shot improves as the value of n increases. However, it is crucial to consider the input length limit of LLMs (typically 2,048 or 4,096). As a result, it is not feasible to indefinitely increase the value of n without exceeding the input length limit. Considering the limitation of input length and the saturation of model performance growth, we generally recommend using either 8-shot or 4-shot for self-planning in LLMs.

In the result of the group *Configurations of Intermediate Step*, the improvement of Code CoT over direct code generation is relatively small compared to the self-planning approach, as the challenge of generating an accurate and sufficiently detailed CoT is comparable to that of direct code generation. The performance with the Zero-shot CoT is worse than the few-shot prompt Code CoT we crafted. Thus CoT is not optimal for the code generation task, and the planning approach is more suitable. The degraded performance exhibited by narrative Code CoT and plan emphasizes the importance of clear, separated steps. For LLMs, consecutive and undifferentiated steps may lead to suboptimal understanding. Minor performance enhancement observed when self-planning approach employs an extremely concise plan reveals the powerful comprehension capabilities of LLMs, as well as the pivotal role that keywords play in the planning process. The performance

Table 5. Comparison of Self-Planning and Other Approaches on Multilingual Datasets

Approach	Python		Java	
	Pass@1	CodeBLEU	Pass@1	CodeBLEU
Direct	48.1	24.0	50.6	38.0
Code CoT	53.9 (↑ 12.1%)	30.4	56.4 (↑ 11.5%)	39.0
Self-planning	**60.3** (↑ 25.4%)	28.6	**61.5** (↑ 21.5%)	39.0
Ground-truth Planning	74.4 (↑ 54.7%)	41.0	66.7 (↑ 31.8%)	45.8
	JavaScript		Go	
Direct	53.2	26.7	42.9	22.2
Code CoT	52.6 (↑ −1.1%)	27.0	48.1 (↑ 12.1%)	27.1
Self-planning	**55.8** (↑ 4.9%)	25.6	**53.0** (↑ 23.5%)	26.5
Ground-truth Planning	60.3 (↑ 13.3%)	29.6	58.3 (↑ 35.9%)	32.0

of "Plan2CoT" outperformed Code CoT approach, suggesting that planning prior to generating Code CoT can enhance the accuracy of Code CoT. However, it is slightly less effective than self-planning approach. We hypothesize that one layer of abstraction is sufficient for function-level code generation tasks in HumanEval. Relatively, excessive levels of abstraction may increase the probability of errors.

In terms of cost, self-planning has a lower token usage compared to other few-shot prompting approaches. It is particularly noteworthy that the total number of tokens used in 1-shot self-planning not only is lower than in Zero-shot CoT (Two-phase), but it also achieves higher performance. When using self-planning, users can make a trade-off between the number of tokens and performance as needed, opting to use different numbers of shots.

Overall, all variants except one-stage and extremely concise plan underperform the self-planning approach in terms of performance. However, one-stage approach necessitates the provision of code within the prompt, presenting a barrier to entry for individuals lacking programming expertise, while extremely concise plan approach does not align with human writing conventions. Taking all these factors into account, our proposed planning method performs as the optimal choice among the variants we explored.

4.4 Performance on Multilingual Code Generation (RQ4)

Evaluation. We evaluate the generality of our self-planning approach on HumanEval-X benchmarks across multiple PLs, i.e., Python, Java, JavaScript, and Go. Rather than customizing plans for each specific PL, we utilize the same intent-plan pair example as Python across all PLs.

Results. As demonstrated in Table 5, our self-planning approach exhibits positive results across all PLs when compared to Direct and Code CoT. It is evident that our method yields the most significant improvement for Python. This may be due to the fact that we tend to solve in Python when writing plans, introducing some of the features that are common to Python, such as dict, list, and so forth. As a result, the improvements are more pronounced for PLs that have Python-like features, such as Go and Java. We believe that if plans are customized for other PLs, their effectiveness would be further enhanced. Moreover, if a plan is created independent of a specific PL, its generalization across different languages would be improved.

4.5 Effect of Problem Complexity on Self-Planning

Evaluation. In this section, we focus on evaluating the impact of problem complexity on the performance of self-planning approach. We split the questions in HumanEval according to their

Table 6. Pass@1 of Self-Planning on Problems of Varying Complexity

Approach	Complexity 1	Complexity 2	Complexity 3
Direct	65.2	36.2	31.1
Code CoT	68.1 (↑ 4.4%)	44.7 (↑ 23.4%)	37.3 (↑ 19.9%)
Self-planning	71.2 (↑ 9.3%)	55.3 (↑ 52.7%)	50.2 (↑ 61.5%)

difficulty. Specifically, we followed the LLMs evaluation methodology [37] by providing the intent and code examples of each problem for GPT-4 and using a specific instruction I to score the difficulty of the problem. Thereafter, we perform a full manual review of the scoring results. Finally, we categorize the HumanEval dataset into three difficulty levels based on a segmentation function f.

I = "Please rate the level of difficulty of this problem with an integer between 0 and 10, where 0 is the easiest, and 10 is the hardest. Return the response in a JSON format, with a variable "score" containing your score, and another variable "explanation" with the explanation for this score. Your explanation must have at least 20 words."

$$\text{difficulty} = f(score) = \begin{cases} 1 & \text{if } 0 < score \leq 2 \\ 2 & \text{if } score = 3 \\ 3 & \text{if } score \geq 4 \end{cases}.$$

The reason we choose to use this segmentation function is that by splitting the questions in this way, we are able to make the number of questions contained in each difficulty level more balanced. The categorization results are: difficulty level 1 contains 67 questions, difficulty level 2 contains 47 questions, and difficulty level 3 contains 43 questions. After splitting the dataset in this manner, we evaluated the Pass@1 metric for three approaches: Direct, Code CoT, and Self-planning.

Results. The results of the evaluation are shown in Table 6. The self-planning approach exhibits higher performance than all baseline approaches, Direct and Code CoT, when dealing with problems of varying difficulty. It is worth noting that the performance improvement of the self-planning approach is particularly significant when dealing with higher difficulty problems (Complexity 2, Complexity 3), exceeding the improvement on lower difficulty problems (Complexity 1). This observation underscores that the Self-planning approach excels in handling complex problems, while simultaneously maintaining efficient performance for simpler tasks.

5 Human Evaluation

In this section, we conduct a human evaluation to assess the quality of the self-planning and baseline approaches. This evaluation is designed to reflect the practical experience of human developers using these code generation approaches. The results of this evaluation will provide valuable insights into the usability and practicality of self-planning code generation.

Evaluation. We first establish a set of criteria to assess the generated code, as outlined below.

- —Correctness: High-quality code should be correct and produce the expected output or behavior. This means that the code should meet the requirements, and its functionality should be accurate and precise.
- —Readability: High-quality code should be easy to read and understand by developers, facilitating future maintenance. This can be achieved through clear naming conventions, consistent indentation and formatting, and using comments to explain complex or unclear sections.

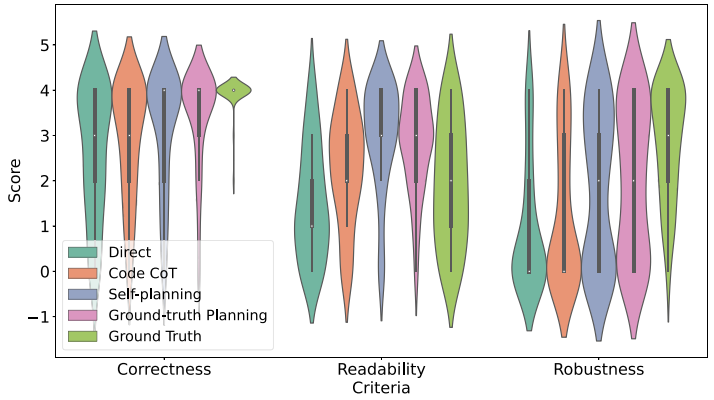

Fig. 4. Violin plot for human evaluation. The violin plot is a combination of a box line plot, which shows the location of the quartiles (the upper edge of the box represents the upper quartile, the middle point is the median, and the lower edge of the box is the lower quartile), and a kernel density plot, which shows the density at any location.

—Robustness: High-quality code should be robust and handle unexpected situations or edge cases gracefully.

Second, we sample 50 tasks from the HumanEval benchmark, and each task contains five codes: ground-truth code, direct-generated code, self-planning-generated code, Code CoT-generated code, and ground-truth planning-generated code. We asked developers to score each code on five aspects from the criteria. The scores are integers ranging from 0 to 4, where 0 is the worst and 4 is the best. Note that we show developers five codes for one task at a time, making it easy for developers to compare the five codes and score the gaps.

Finally, we assemble a team of evaluators, including 10 developers with 2–5 years of Python programming experience, and divide them into two evaluation groups (Group A and Group B). The evaluation is conducted in the form of an anonymous questionnaire, which is displayed in the Appendix F. Each evaluation team is required to evaluate all tasks, and each evaluator is randomly assigned 10 tasks (questionnaires), where the codes generated by the different methods corresponding to each task are randomly ordered.

Results. The evaluation results of the two groups are summarized in Figure 4. Correctness scoring closely aligns with the Pass@1 results, while also considering partial correctness. The self-planning approach outperforms both Direct and Code CoT but falls short of Ground-truth planning and Ground-truth.

In terms of readability, the self-planning approach excels as the most readable, distinguished by a blend of accuracy and a coherent outline of its planned steps. The Ground-truth planning approach's readability closely follows that of the self-planning approach. Conversely, the Code CoT exhibits subpar readability. Its solution steps provide excessive detail, potentially hindering the identification of crucial elements within the code and becoming outdated if code modifications occur. This can adversely affect code maintainability if the solution steps do not accurately represent the current state. The readability of both Direct and Ground-truth is deemed insufficient.

We find that the incorrect code usually receives a score of 0 for robustness item. Consequently, the robustness of violin plots displays a broader pattern at the lower end. Regarding robustness, the self-planning approach surpasses Code CoT and performs comparably to Ground-truth planning,

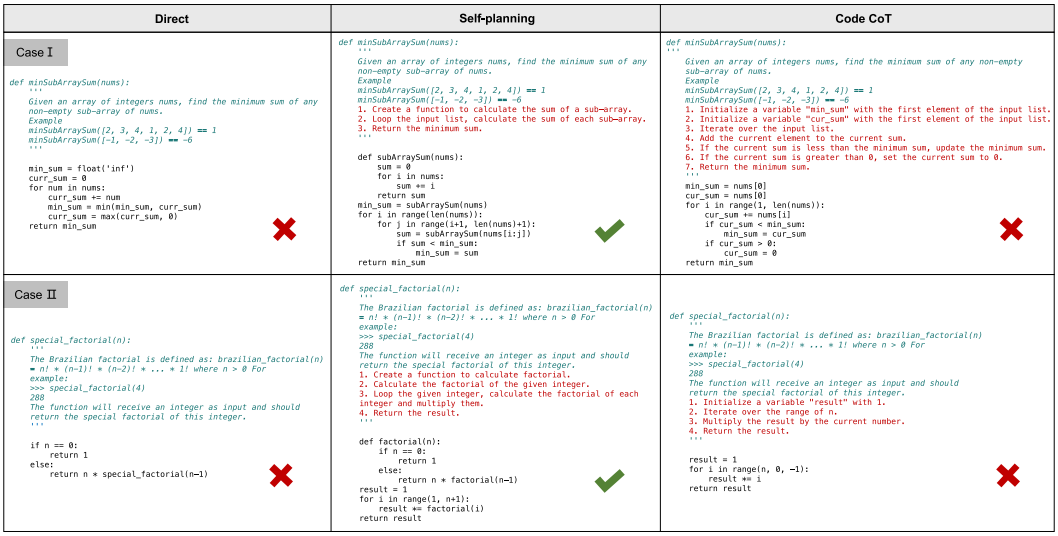

Fig. 5. Two real cases from HumanEval with self-planning, Code CoT (with self-planning format), and direct code generation. The input, generated plan, and code are highlighted in green, red, and black, respectively.

since the self-planning approach can thoroughly consider some edge cases and determines the legality of inputs, as evidenced by the qualitative examples 5.

In conclusion, through human evaluation, our self-planning approach exhibits the best readability among all approaches, and its correctness and robustness performance is on par with the ground-truth planning approach.

Qualitative Examples. To further understand the effectiveness of our approach, we examine cases where self-planning approach has contrasting performances to Direct approach. As depicted in Figure 5, we demonstrate the performance of both direct, self-planning, and Code CoT code generation through two cases. In these cases, the direct and Code CoT code generation approach only addresses a limited aspect of intent, which often results in incorrect code generation. In contrast, the self-planning code generation approach first converts the intent into plan, and then systematically resolves each solution step of plan. This approach effectively minimizes the risk of overlooking critical elements.

In case I, the task of LLMs is "*Given an array of integers nums, find the minimum sum of any non-empty sub-array of nums.*" The code generated directly by LLM only considers a subset of the sub-arrays, whereas our approach ensures that none of them are overlooked. In case II, the task of LLMs is "*Receive an integer as input and return the special factorial of this integer.*" The direct code generation simply implements the standard factorial in a recursive manner, neglecting the definition of the special factorial. In contrast, our approach implements the standard factorial through the use of sub-function and subsequently uses the sub-function to construct the special factorial. We can find that the semantics of the code generated by Code CoT and direct code generation are almost the same, only the expression form is different. This may confirm the point that the difficulty of generating Code CoT from intent and generating code is comparable.

Overall, our self-planning approach offers a more thorough and nuanced way for addressing complex tasks assigned to LLMs, in contrast to direct and Code CoT code generation, which provides a more straightforward and limited solution.

6 Threats to Validity

There are two primary threats to the validity of our study.

—Considering the inherent sensitivity of LLMs to prompts, the primary threat comes from crafting prompts, as the example selection and plan writing in this operation can affect the degree of improvement achievable by our approach. This issue necessitates fundamental improvements to the LLMs for resolution [34, 69]. In our current approach, the random sampling of examples and adherence to plan-writing principles ensure a considerable level of improvement. However, there is potential for optimization. Several research efforts have explored how automated techniques for selecting quality examples [42] and generating prompts [65, 68] can be used to maximize the performance of the prompting approach. These results can be introduced into our approach to further improve the performance.

—The second major threat to our study pertains to the generalizability of experimental results. To address this threat, we assessed self-planning on seven public benchmark datasets, in line with previous work [5, 10, 66]. These datasets span four mainstream PLs: Python, Java, JavaScript, and Go. To validate the quality of the generated code, we employed the widely accepted metric, Pass@k, which leverages test cases to gauge the functional correctness of code. Additionally, we utilized the unbiased version of Pass@k [5] to diminish evaluation errors that arise from sampling.

7 Related Work

7.1 Code Generation

Traditional code generation approaches are based on supervised learning, which initially treats code as equivalent to natural language [21, 27, 56] and then gradually incorporates more code-specific features, such as abstract syntax tree [39, 49, 50, 61, 62], API calls [16, 17, 40]. Furthermore, Mukherjee et al. [32] present a generative modeling approach for source code that uses a static analysis tool. Dong et al. [11] devise a PDA-based methodology to guarantee grammatical correctness for code generation.

With the rise of pre-training, CodeT5 [54], UniXcoder [18] applied pre-trained models to code generation task. The introduction of subsequent models like Codex [5], InCoder [13], CodeGen [33], AlphaCode [26], and CodeGeeX [66], continues to push the direction. A noteworthy trend among these models is the rapid increase in the number of parameters in the pre-trained models, which leads to a tremendous improvement in the performance of code generation. This has sparked a variety of studies, with the focus on utilizing LLMs as the backbone, enhancing their code generation performance through various techniques [4, 7, 63, 64], and achieving very promising results. These approaches can be summarized as post-processing strategies, i.e., operations such as reranking and modifying the code after the model generates it. In contrast, our approach is classified as pre-processing. Therefore, our approach and post-processing approaches are orthogonal and can be used concurrently.

Recently a related work **structured chain-of-thought (SCoT)** prompting [24] has been proposed. SCoT incorporates the sequential, branching, and looping structures into CoT for code generation, hoping to create a structured thought process. In terms of motivation, SCoT is different from our self-planning. Self-planning, inspired by requirements analysis in software engineering, is proposed to decompose complex problems so as to reduce the complexity of problem-solving.

7.2 Prompting Techniques

Few-shot prompting [28] is a technique that emerged as the number of model parameters exploded. Instead of fine-tuning a separate language model checkpoint for each new task, few-shot prompting

can be utilized by simply providing the model with a limited number of input-output examples that illustrate the task. A few-shot prompt technique known as CoT [57] achieves a significant improvement that transcends the scaling laws by generating intermediate reasoning steps before the answer to address language reasoning tasks. *CoT in its original paper focused on solving a range of reasoning tasks such as mathematical reasoning, symbolic reasoning, and commonsense reasoning, and did not implement CoT for code-generation tasks.* Inspired by CoT, a series of prompting works has been proposed. Least-to-most prompting [67] reduces a complex problem into a series of sub-problems and then solves the sub-problems in order, adding the answer to the previous sub-problems to the prompt each time solving begins. PAL [14] and PoT [6] are proposed to generate code as the intermediate reasoning steps, delegating solving to the compiler, thus improving solution accuracy. Nonetheless, the aforementioned approaches are adept at addressing relatively simple mathematical [23, 58], commonsense [31, 44], and symbolic reasoning [60] problems characterized by limited problem spaces and established solution patterns. Consequently, their applicability to code generation remains restricted. Beyond these, CodeGen proposes a multi-turn prompting approach. There are two major differences between this approach and self-planning: (1) It uses ground-truth intermediate steps (i.e., human-written annotations) to prompt LLMs, while our plan is generated by LLMs itself. (2) It iteratively uses the intermediate steps to generate the corresponding code snippets that eventually compose the entire code, while self-planning generates the entire code at once via plan.

7.3 Self-Improvement of LLMs

The use of LLMs to enhance the performance of LLMs themselves has become a current research hotspot. Huang et al. [20] demonstrate that an LLM can enhance its performance on reasoning datasets by training on the data it generated itself. Self-Instruct [53] improves the instruction-following capabilities of LLMs by bootstrapping off their own generations. Self-Refine [30] uses the LLMs to provide feedback for its own output and refine itself. Self-Evaluation [59] introduces a stepwise self-evaluation mechanism to guide and calibrate the reasoning process of LLMs. Self-validation [15] improves few-shot clinical information extraction by utilizing the LLM to provide provenance for its own extractions and checking its own output. Self-Criticism [52] achieves alignment with being helpful, honest, and harmless by utilizing LLMs to judge themselves. Beyond these, Promptbreeder [12] utilizes a self-referential self-improvement mechanism with LLMs that evolves and adapts prompts for a specific domain.

7.4 Planning with LLMs

The practice of utilizing LLMs for planning in various applications is increasingly garnering attention, particularly for their potential to comprehend complex problems and optimize decision-making processes. LLM-Planner [48] harnesses the power of LLMs to do few-shot planning for embodied agents. HuggingGPT [46] integrates LLMs to plan the invocation of various AI models from the machine learning community (e.g., Hugging Face) for handling a wide range of complex AI tasks spanning different paradigms and domains. DEPS [55] proposes an interactive planning approach based on LLMs capable of robustly completing more than 70 Minecraft tasks. ProgPrompt [47] leverages LLMs to generate situated robot task plans.

8 Discussion and Future Work

When discussing the limitations of self-planning code generation, a major limitation may be the manual crafting of prompts. However, we should also be aware that in previous approaches, a huge number of examples may be needed in order to train a model to understand planning. This makes data efficiency an important issue. However, we propose a self-planning approach that directly

teaches LLMs to understand planning with only a few examples and can be crafted by people without programming knowledge. This improvement in data efficiency and low barrier can make the self-planning code generation approach easier to apply in practice.

Additionally, our approach employs an approximate sequential executed list to represent plans, which is similar to the functional points delineated in requirements documents. Considering the powerful capabilities of LLMs, some rudimentary loops and branch structures in the plan may not be necessary. For instance, in Figures 2 and 5, the sub-functions encompass loops and branch structures. However, we merely need to specify the function of these sub-functions without describing the internal loop and branching structure. Some intricate plans can be simplified through iterative planning until they no longer pose challenges for LLM.

Finally, this paper attempts to reduce the difficulty of code generation by planning for human intent, which is consistent with the methodology of dealing with problem complexity in requirements engineering, i.e. abstraction and decomposition [29]. The current LLMs are capable of generating code that addresses simple human requirements, however, it is still a long way from producing a fully functional piece of software. The requirements of software development are significantly more complex and intricate. It may be worthwhile to explore beyond code writing to the realm of requirements analysis, incorporating the methodology of requirements engineering with LLMs.

9 Conclusion

In this article, we have explored plan-aided code generation and proposed a simple but effective approach to perform self-planning and generate code with LLMs. Self-planning code generation outperforms direct generation with LLMs on multiple code generation datasets by a large margin. Moreover, self-planning approach leads to enhancements in the correctness, readability, and robustness of the generated code, as evidenced by human evaluation. Empirical evidence indicates that although self-planning is an emergent ability, incorporating planning strategies can yield advantages for most models.

Appendices

A Details of Metrics

Pass@k. We use the unbiased version [26] of Pass@k, where $n >= k$ samples are generated for each problem, count the number of correct samples $c <= n$ which pass test cases, and calculate the following estimator,

$$\text{Pass@k} = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \tag{3}$$

AvgPassRatio. The average proportion of test cases [19] that generated codes $\mathbf{g}'_p s$ pass:

$$\frac{1}{|P|} \sum_{p \in P} \frac{1}{|C_p|} \sum_{c \in C_p} \mathbb{I}\left\{ \text{Eval}\left(\mathbf{g}_p, \mathcal{I}_{p,c}\right) = \mathcal{O}_{p,c} \right\}, \tag{4}$$

where $|\cdot|$ indicates the cardinality of a set, $\mathbb{I}(\cdot)$ is an indicator function, which outputs 1 if the condition is true and 0 otherwise, and $\text{Eval}\left(\mathbf{g}_p, \mathcal{I}_{p,c}\right)$ represents an evaluation function that obtains outputs of code $\mathbf{g}_p$ by way of executing it with $\mathcal{I}_{p,c}$ as input.

CodeBLEU. CodeBLEU [41] is a variant of BLEU that injects code features for code evaluation. CodeBLEU considers abstract syntax tree and dataflow matching in addition to n-gram

co-currency (BLEU),

$$\begin{aligned}\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{weight} \\ + \delta \cdot \text{Match}_{ast} + \zeta \cdot \text{Match}_{df}.\end{aligned}$$

B Experimental Verification of Subproblem Decomposition in Self-Planning

We conduct additional experiments to further verify the decomposition of subproblems by our approach. We count the number of generated sub-functions (denotes a successfully decomposed sub-problem) for Direct, Code CoT (i.e., the CoT implemented for code generation in this article), and Self-planning on the HumanEval dataset. The results show that Direct, Code CoT, and Self-planning generate 1, 9, and 37 sub-functions, respectively. Moreover, we pick the multi-objective problems (e.g., "prime_fib returns *n*th number that is a Fibonacci number and it's also prime.") that decomposition is skilled at solving. The correct number of multi-objective problems (total of 58) using Direct, Code CoT, and Self-planning are 11, 18, and 40, respectively. These experiments indicate that self-planning helps break down the problem into sub-problems, thereby facilitating code generation.

C Self-Planning Prompt for HumanEval Benchmarks

```
def encrypt(s):
    '''
    Create a function encrypt that takes a string as an argument and returns a string encrypted with the alphabet being rotated. The alphabet should be rotated in a manner such that the letters shift down by two multiplied to two places.
    For example:
    encrypt('hi') returns 'lm'
    encrypt('asdfghjkl') returns 'ewhjklnop'
    encrypt('gf') returns 'kj'
    encrypt('et') returns 'ix'
    1. Create a alphabet, bias two places multiplied by two.
    2. Loop the input, find the latter bias letter in alphabet.
    3. Return result.
    '''
```

```
def check_if_last_char_is_a_letter(txt):
    '''
    Create a function that returns True if the last character of a given string is an alphabetical character and is not a part of a word, and False otherwise. Note: "word" is a group of characters separated by space.
    Examples:
    check_if_last_char_is_a_letter("apple pie") → False
    check_if_last_char_is_a_letter("apple pi e") → True
    check_if_last_char_is_a_letter("apple pi e ") → False
    check_if_last_char_is_a_letter("") → False
    1. If the string is empty, return False.
    2. If the string does not end with a alphabetical character, return False.
    3. Split the given string into a list of words.
```

4. Check if the length of the last word is equal to 1.
”’

---

def file_name_check(file_name):
”’

Create a function which takes a string representing a file’s name, and returns ‘Yes’ if the the file’s name is valid, and returns ‘No’ otherwise. A file’s name is considered to be valid if and only if all the following conditions are met: - There should not be more than three digits (‘0’-‘9’) in the file’s name. - The file’s name contains exactly one dot ‘.’ - The substring before the dot should not be empty, and it starts with a letter from the latin alphapet (‘a’-‘z’ and ‘A’-‘Z’). - The substring after the dot should be one of these: [‘txt’, ‘exe’, ‘dll’]

Examples:

file_name_check(“example.txt”) => ‘Yes’

file_name_check(“1example.dll”) => ‘No’ (the name should start with a latin alphapet letter)

1. Check if the file name is valid according to the conditions.
2. Return “Yes” if valid, otherwise return “NO”.
”’

---

def fruit_distribution(s,n):
”’

In this task, you will be given a string that represents a number of apples and oranges that are distributed in a basket of fruit this basket contains apples, oranges, and mango fruits. Given the string that represents the total number of the oranges and apples and an integer that represent the total number of the fruits in the basket return the number of the mango fruits in the basket.

for examble:

fruit_distribution(“5 apples and 6 oranges”, 19) ->19 − 5 − 6 = 8

fruit_distribution(“0 apples and 1 oranges”, 3) -> 3 − 0 − 1 = 2

fruit_distribution(“2 apples and 3 oranges”, 100) -> 100 − 2 − 3 = 95

fruit_distribution(“100 apples and 1 oranges”, 120) -> 120 − 100 − 1 = 19

1. Extract the numbers of oranges and apples from given string.
2. Calculate the sum of oranges and apples.
3. Deduct the sum from the total number of fruits.
4. Return the number of mangoes.
”’

---

def prime_fib(n: int):
”’

prime_fib returns n-th number that is a Fibonacci number and it’s also prime.

Examples:

>>> prime_fib(1) 2

>>> prime_fib(2) 3

>>> prime_fib(3) 5

>>> prime_fib(4) 13

>>> prime_fib(5) 89
1. Create a function to check if a number is prime.
2. Generate a Fibonacci sequence.
3. Check if each number in the Fibonacci sequence is prime, decrement the counter.
4. If the counter is 0, return the Fibonacci number.
"'

---

def compare_one(a, b):
"
Create a function that takes integers, floats, or strings representing real numbers, and returns the larger variable in its given variable type. Return None if the values are equal. Note: If a real number is represented as a string, the floating point might be . or ,
Examples:
compare_one(1, 2.5) → 2.5
compare_one(1, "2,3") → "2,3"
compare_one("5,1", "6") → "6"
compare_one("1", 1) → None
1. Store the original inputs.
2. Check if inputs are strings and convert to floats.
3. Compare the two inputs and return the larger one in its original data type.
"'

---

def sort_even(l: list):
"'
This function takes a list l and returns a list l' such that l' is identical to l in the odd indicies, while its values at the even indicies are equal to the values of the even indicies of l, but sorted.
Examples:
>>> sort_even([1, 2, 3])
[1, 2, 3]
>>> sort_even([5, 6, 3, 4])
[3, 6, 5, 4]
1. Create a list of all the even indices of the given list.
2. Sort the list of even indices.
3. Return a new list that is identical to the original list in the odd indicies, and equal to the sorted even indices in the even indicies.
"'

---

def search(lst):
"'
You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a frequency greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it appears in the list. If no such a value exist, return −1.
Examples:
search([4, 1, 2, 2, 3, 1]) == 2

search([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
search([5, 5, 4, 4, 4]) == −1
1. Create a frequency dict.
2. Sort the input list.
3. Loop the input list, if frequency no lesser than the integer, set result.
4. Return the result.
”’

D Self-planning Prompt for MBPP Benchmarks

Write a function to sum the length of the names of a given list of names after removing the names that start with a lowercase letter.
1. Loop the input list.
2. If the name not start with lowercase letter, add the length of the name to result.
3. Return the result.

Write a function to increment the numeric values in the given strings by k.
1. Loop the input list.
2. If a string is a number, increment it.
3. Return modified list.

Write a python function to find sum of all prime divisors of a given number.
1. Create a inner function to check if a number is prime.
2. Loop all number less than the input that is prime.
3. Check if the input is divisible by that.
4. Return the result.

Write a function to find the lateral surface area of a cone.
1. Calculate the generatrix of the cone.
2. Return the result.
3. Please import inside the function.

Write a function to remove all tuples with all none values in the given tuple list.
1. Loop the given tuple list.
2. Check if all elements in the tuple are None.
3. If not, append the tuple to the result list.
4. Return the result.

Write a python function to find the last two digits in factorial of a given number.
1. Calculate the factorial of the input number.
2. Return the last two digits of it.

Write a python function to replace multiple occurence of character by single.
1. Create a pattern that the input character repeats mulitiple times.
2. Replace the pattern in input string with input character.
3. Please import inside the function.

Write a python function to move all zeroes to the end of the given list.
1. Count the number of zeros.
2. Remove the zeros from the list.
3. Append the zeros to the end of the list.
4. Return the list.

E Instances of Baseline Prompt

Instance of CoT Prompting with Self-Planning Format

```
def encrypt(s):
    '''
    Create a function encrypt that takes a string as an argument and returns a string encrypted with the alphabet being rotated. The alphabet should be rotated in a manner such that the letters shift down by two multiplied to two places.
    For example:
    encrypt('hi') returns 'lm'
    encrypt('asdfghjkl') returns 'ewhjklnop'
    encrypt('gf') returns 'kj'
    encrypt('et') returns 'ix'
    Let's think step by step.
    1. Create a string "alphabet" with all letters of the alphabet.
    2. Assign the number of places to shift the letters to a variable "bias".
    3. Initialize a string "result" with an empty string.
    4. Iterate over the characters of the string "s".
    5. Find the index of the character in the string "alphabet".
    6. Add the number of places to shift the letters to the index.
    7. If the index is larger than 25, subtract 26 from the index.
    8. Add the character at the index to the string "result".
    9. Return the string "result".
    '''
```

Instance of Extremely Concise Style Self-Planning Prompt

```
def encrypt(s):
    '''
    Create a function encrypt that takes a string as an argument and returns a string encrypted with the alphabet being rotated. The alphabet should be rotated in a manner such that the letters shift down by two multiplied to two places.
    For example:
    encrypt('hi') returns 'lm'
    encrypt('asdfghjkl') returns 'ewhjklnop'
    encrypt('gf') returns 'kj'
    encrypt('et') returns 'ix'
```

```
    1. Alphabet, bias 4.
    2. Latter bias, append.
    '''
```

Instance of Ground-Truth Planning Prompt

```
def encrypt(s):
    '''
    Create a function encrypt that takes a string as an argument and returns a string encrypted with the alphabet being rotated. The alphabet should be rotated in a manner such that the letters shift down by two multiplied to two places.
    '''
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    bias = 2 * 2
    result = ''
    for char in s:
        index = alphabet.find(char) + bias
        if index > 25:
            index = index - 26
        result += alphabet[index]
    return result
Write steps according to the code.
    1. Create a alphabet, bias two places multiplied by two.
    2. Loop the input, find the latter bias letter in alphabet.
    3. Return result.
```

Instance of Self-Planning Prompt (One-phase)

```
def encrypt(s):
    '''
    Create a function encrypt that takes a string as an argument and returns a string encrypted with the alphabet being rotated. The alphabet should be rotated in a manner such that the letters shift down by two multiplied to two places.
    For example:
    encrypt('hi') returns 'lm'
    encrypt('asdfghjkl') returns 'ewhjklnop'
    encrypt('gf') returns 'kj'
    encrypt('et') returns 'ix'
    1. Create a alphabet, bias two places multiplied by two.
    2. Loop the input, find the latter bias letter in alphabet.
    3. Return result.
    '''
    # Write your code here.
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    bias = 2 * 2
    result = ''
```

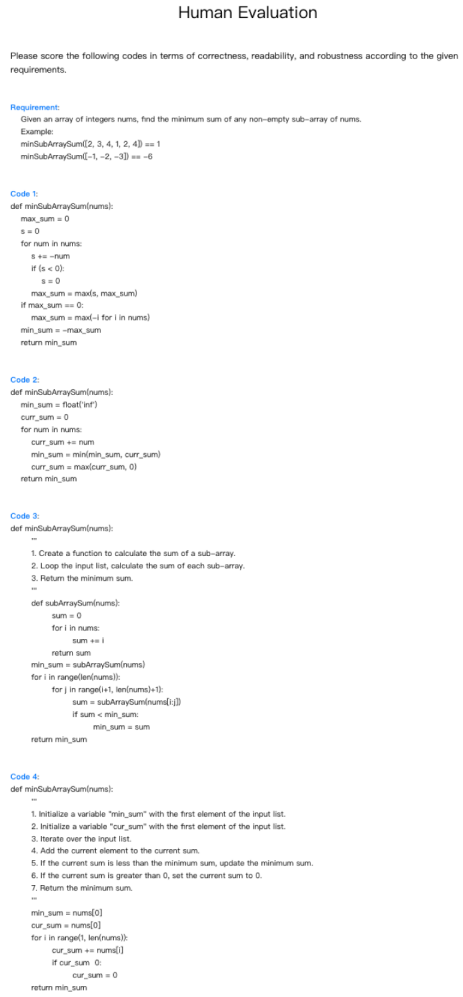

Fig. 6. Questionnaire Part 1.

```
    for char in s:
        index = alphabet.find(char) + bias
        if index > 25:
            index = index − 26
        result += alphabet[index]
    return result
```

F Human Evaluation Questionnaire

The questionnaires for the human evaluation of a task are shown in Figures 6 and 7. We distributed 50 questionnaires, which correspond to 50 tasks.

Fig. 7. Questionnaire Part 2.

References

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. arXiv:1709.08439. Retrieved from https://doi.org/10.48550/arXiv.1709.08439

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS '20)*. 1877–1901.

[4] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *Proceedings of the International Conference on Learning Representations (ICLR '23)*.

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

[6] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. arXiv:2211.12588. Retrieved from https://arxiv.org/abs/2211.12588

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. arXiv:2304.05128. Retrieved from https://arxiv.org/abs/2304.05128

[8] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2022. Scaling instruction-finetuned language models. arXiv:2210.11416. Retrieved from https://arxiv.org/abs/2210.11416

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. arXiv:2110.14168. Retrieved from https://arxiv.org/abs/2110.14168

[10] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. CodeScore: Evaluating code generation by learning code execution. arXiv:2301.09043. Retrieved from https://arxiv.org/abs/2301.09043

[11] Yihong Dong, Ge Li, and Zhi Jin. 2023. CODEP: Grammatical seq2seq model for general-purpose code generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. ACM, 188–198.

[12] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. arXiv:2309.16797. Retrieved from https://arxiv.org/abs/2309.16797

[13] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A generative model for code infilling and synthesis. arXiv:2204.05999. Retrieved from https://arxiv.org/abs/2204.05999

[14] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. PAL: Program-aided language models. arXiv:2211.10435. Retrieved from https://arxiv.org/abs/2211.10435

[15] Zelalem Gero, Chandan Singh, Hao Cheng, Tristan Naumann, Michel Galley, Jianfeng Gao, and Hoifung Poon. 2023. Self-verification improves few-shot clinical information extraction. arXiv:2306.00024. Retrieved from https://arxiv.org/abs/2306.00024

[16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE '16)*. ACM, 631–642.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI '17)*, 3675–3681.

[18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL '22)*, Vol. 1. Association for Computational Linguistics, 7212–7225.

[19] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS Datasets and Benchmarks '21)*.

[20] Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2023. Large language models can self-improve. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*. Association for Computational Linguistics, 1051–1068.

[21] Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL '16)*, Vol. 1. The Association for Computer Linguistics, 12–22.

[22] Takeshi Kojima, Shixiang S. Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS '22)*. 22199–22213.

[23] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS '22)*, Vol. 35. 3843–3857.

[24] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. arXiv:2305.06599. Retrieved from https://doi.org/10.48550/arXiv.2305.06599

[25] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SKCODER: A sketch-based approach for automatic code generation. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. 2124–2135.

[26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[27] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the Association for Computational Linguistics (ACL '16)*, Vol. 1. The Association for Computer Linguistics.

[28] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys* 55, 9 (2023), 1951–195:35.

[29] Linda A Macaulay. 2012. *Requirements Engineering*. Springer Science & Business Media.

[30] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa P. Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative refinement with self-feedback. arXiv:2303.17651. Retrieved from https://arxiv.org/abs/2303.17651

[31] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '22)*. Association for Computational Linguistics, 1384–1403.

[32] Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Christopher M. Jermaine. 2021. Neural program generation modulo static analysis. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS '21)*, Vol. 34. 18984–18996.

[33] 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. arXiv:2203.13474. Retrieved from https://doi.org/10.48550/arXiv.2203.13474

[34] Venkata P. S. Nookala, Gaurav Verma, Subhabrata Mukherjee, and Srijan Kumar. 2023. Adversarial robustness of prompt-based few-shot learning for natural language understanding. In *Proceedings of the Association for Computational Linguistics: Findings (ACL '23)*. Association for Computational Linguistics, 2196–2208.

[35] OpenAI. 2023. GPT-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[36] Kai Petersen, Claes Wohlin, and Dejan Baca. 2009. The waterfall model in large-scale development. In *Proceedings of the Product-Focused Software Process Improvement (PROFES '09)*. F. Bomarius, M. Oivo, P. Jaring, and P. Abrahamsson (Eds.), Lecture Notes in Business Information Processing, Vol. 32, Springer, 386–400.

[37] Gustavo Pinto, Isadora Cardoso-Pereira, Danilo Monteiro, Danilo Lucena, Alberto L. O. T. de Souza, and Kiev Gama. 2023. Large language models for education: Grading open-ended questions using ChatGPT. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES '23)*. ACM, 293–302.

[38] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *Proceedings of the International Conference on Learning Representations (ICLR '22)*.

[39] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL '17)*, Vol. 1. Association for Computational Linguistics, 1139–1149.

[40] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 419–428.

[41] 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:2009.10297. Retrieved from https://arxiv.org/abs/2009.10297

[42] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning to retrieve prompts for in-context learning. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '22)*. Association for Computational Linguistics, 2655–2671.

[43] Nayan B. Ruparelia. 2010. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes* 35, 3 (2010), 8–13.

[44] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. 2022. Multitask prompted training enables zero-shot task generalization. In *Proceedings of the 10th International Conference on Learning Representations (ICLR '22)*. OpenReview.net.

[45] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE '22)*. ACM, 1533–1543.

[46] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving AI tasks with chatgpt and its friends in huggingface. arXiv:2303.17580. Retrieved from https://doi.org/10.48550/arXiv.2303.17580

[47] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. ProgPrompt: Generating situated robot task plans using large language models. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '23)*. IEEE, 11523–11530.

[48] Chan Hee Song, Brian M. Sadler, Jiaman Wu, Wei-Lun Chao, Clayton Washington, and Yu Su. 2023. LLM-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '23)*. IEEE, 2986–2997.

[49] 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 (AAAI '19)*, Vol. 33. AAAI Press, 7055–7062.

[50] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '20)*, Vol. 34. AAAI Press, 8984–8991.

[51] Alon Talmor, Oyvind Tafjord, Peter Clark, Yoav Goldberg, and Jonathan Berant. 2020. Leap-of-thought: Teaching pre-trained models to systematically reason over implicit knowledge. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS '20)*. 20227–20237.

[52] Xiaoyu Tan, Shaojie Shi, Xihe Qiu, Chao Qu, Zhenting Qi, Yinghui Xu, and Yuan Qi. 2023. Self-criticism: Aligning large language models with their understanding of helpfulness, honesty, and harmlessness. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: Industry Track (EMNLP '23)*. Association for Computational Linguistics, 650–662.

[53] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL '23)*, Vol. 1. Association for Computational Linguistics, 13484–13508.

[54] Yue Wang, Weishi Wang, Shafiq R. Joty, and Teven 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 (EMNLP '21)*, Vol. 1. 8696–8708.

[55] Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. arXiv:2302.01560. Retrieved from https://arxiv.org/abs/2302.01560

[56] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS '19)*, Vol. 32. 6559–6569.

[57] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. arXiv:2201.11903. Retrieved from https://doi.org/10.48550/arXiv.2201.11903

[58] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS '22)*, Vol. 35. 32353–32368.

[59] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, Xu Zhao, Min-Yen Kan, Junxian He, and Qizhe Xie. 2023. Self-evaluation guided beam search for reasoning. In *Proceedings of the 37th Conference on Neural Information Processing Systems*.

[60] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR '23)*. OpenReview.net.

[61] 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 (ACL '17)*, Vol. 1. Association for Computational Linguistics, 440–450.

[62] Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP '18)*. Association for Computational Linguistics, 7–12.

[63] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. arXiv:2303.05510. Retrieved from https://arxiv.org/abs/2303.05510

[64] Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2022. Coder reviewer reranking for code generation. arXiv:2211.16490. Retrieved from https://arxiv.org/abs/2211.16490

[65] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2023. Automatic chain of thought prompting in large language models. In *Proceedings of the International Conference on Learning Representations (ICLR '23)*. OpenReview.net.

[66] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X. arXiv:2303.17568. Retrieved from https://arxiv.org/abs/2303.17568

[67] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed H. Chi. 2022. Least-to-most prompting enables complex reasoning in large language models. arXiv:2205.10625. Retrieved from https://arxiv.org/abs/2205.10625

[68] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. In *Proceedings of the International Conference on Learning Representations (ICLR '23)*. OpenReview.net.

[69] Kaijie Zhu, Jindong Wang, Jiaheng Zhou, Zichen Wang, Hao Chen, Yidong Wang, Linyi Yang, Wei Ye, Neil Zhenqiang Gong, Yue Zhang, and Xing Xie. 2023. PromptBench: Towards evaluating the robustness of large language models on adversarial prompts. arXiv:2306.04528. Retrieved from https://arxiv.org/abs/2306.04528

Received 16 October 2023; revised 9 May 2024; accepted 22 May 2024