Self-planning Code Generation with Large Language Models

Xue Jiang, **Yihong Dong**, **Lecheng Wang**, **Zheng Fang**, **Qiwei Shang**, **Ge Li** , **Zhi Jin**, **Wenpin Jiao**
Key Lab of High Confidence Software Technology (PKU), Ministry of Education
School of Computer Science, Peking University, China
{jiangxue, dongyh, wanglecheng, shangqiwei}@stu.pku.edu.cn,
{fangz, lige, zhijin}@pku.edu.cn, jwp@sei.pku.edu.cn

***Abstract*—Although large language models 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 model (LLM), 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. Moreover, our self-planning approach also enhances the quality of the generated code with respect to correctness, readability, and robustness, as assessed by humans.**

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

- Edsger W.Dijkstra

I. 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 [1]–[5]. In particular, LLMs [6], [7] demonstrates impressive code generation abilities, attracting attention from various fields such as artificial intelligence, natural language processing (NLP), 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 [7], [8]. 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 [9]–[11]. 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. Few-shot prompting provides an important way of using LLMs without training. A successful technique of few-shot prompting is Chain of Thought (CoT) [12], which enables LLMs to perform step-by-step reasoning to solve reasoning tasks, such as mathematical [13], commonsense [14], and symbolic reasoning [15]. The ability to generate CoTs, as demonstrated by the LLMs, helps us to achieve planning. Furthermore, we can employ CoT-like prompting techniques to implement planning without the need for fine-tuning.

Nonetheless, directly applying CoTs in the process of planning for code generation remains unfeasible. The primary goal of CoT is to deliver results by generating concrete and complete intermediate steps as a chain of reasoning, thereby minimizing the chances of reasoning errors [12], [16], [17]. In some work [18], [19], code is used to replace or simulate the intermediate steps of CoT with the aim to improve the accuracy of problem-solving. From this perspective, code is viewed as an equivalent representation of CoT in a programming language. As shown in Fig. 1, the CoT describes each step of how to do it explicitly by providing concrete operations, variable names, conditions, and processes, which closely resemble the

*Corresponding author

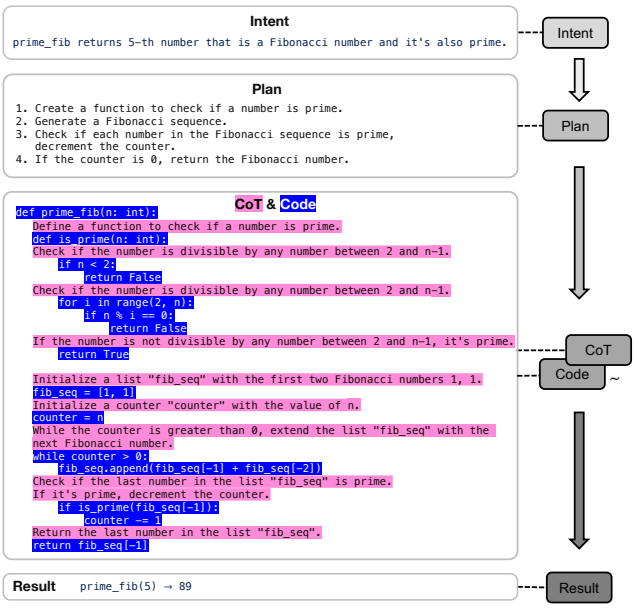

Fig. 1: An example of plan, CoT, and code and their abstraction hierarchy relationships.

code. However, coding is a process that progresses from rough to detailed, and from abstract to concrete. Given that the CoT is nearly as detailed and specific as code, the complexity of generating CoT from intent is virtually equivalent to creating code directly from intent, making standard CoT unsuitable for code generation [20]. We propose the "plan" for planning code generation, which is defined as a high-level representation that sits between the intent and CoT, to reduce the difficulty of direct coding. An example of a plan, CoT, and code and their hierarchical relationship of abstraction between intent and result is displayed in Fig. 1. Specifically, the plan is a rough checklist of steps outlining the entire program's structure. It succinctly describes what needs to be done, offering a more abstract view, without providing detailed guidance on implementation.

In this paper, 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, LLMs generate plans for problems by providing only a few intent-to-plan demonstrations as examples in prompting; 2) Implementation phase, LLMs generate code that adheres to the intent step by step, guided by the plan. 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. 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 investigate the reasons why self-planning approach work on LLMs for code generation, i.e., planning can enhance the model's confidence in predicting key tokens such as control, arithmetic, and punctuation. 5) We validate the effectiveness of self-planning approach across multiple programming languages (PLs) including Python, Java, Go, and JavaScript. 6) We analyze the quality (i.e., correctness, readability, and robustness) of the code generated by self-planning approach through human evaluation.

II. 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 gneration. 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 j-th 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 paper, 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:

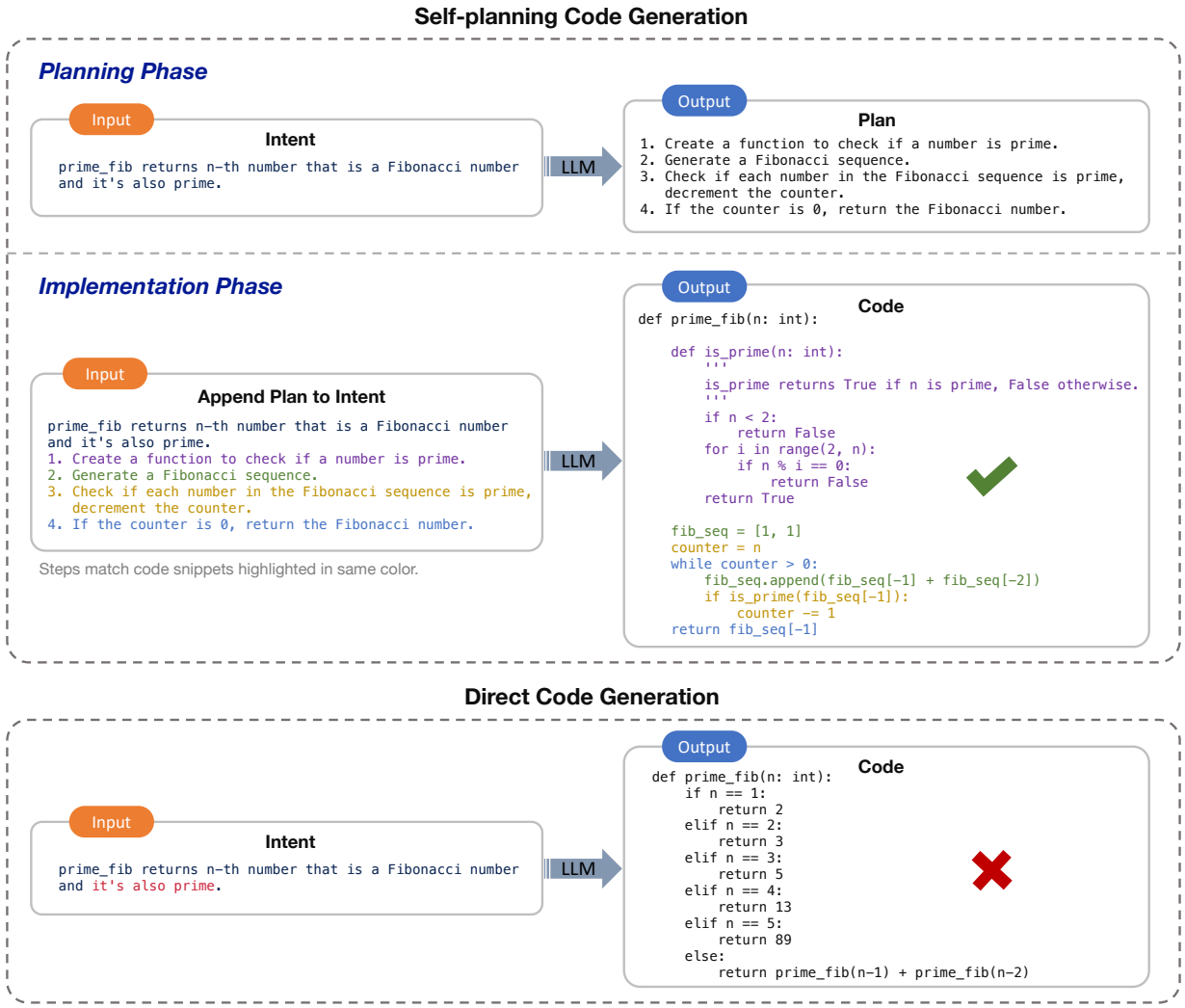

Fig. 2: Self-planning code generation is carried out in two phases (i.e., planning phase and implementation phase): 1) In planning phase, LLM decomposes an intent into a set of easy-to-solve sub-problems and devises a plan for executing the solution steps; 2) In implementation phase, LLM generates code 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.

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

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. Executing this plan involves carrying out these steps in sequence.
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". This ensures the plan is strategic and adaptable rather than becoming bogged down in details.
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 repetition as necessary while still maintaining the logical progression of the plan.
5. The plan should conclude with a return statement. This statement indicates the final result, tying up the entire

process in a definitive manner.

Self-planning prompts can be freely written within these 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 Fig. 2. In the planning phase, human provides an intent to *find the n-th number that is a Fibonacci number and it's also prime* [1]. LLM abstracts two subproblems from the intent, i.e., *generating a Fibonacci sequence* and *determining if a number is prime*, and plans four steps to solve the subproblems combinatorially. 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.

In contrast, LLM 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 [2] 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, it is inevitable that planning needs to be incorporated to handle the complexity.

III. 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**: Why does self-planning approach work for code generation from a statistical point of view?
- **RQ5**: How does self-planning approach perform in multilingual code generation?

A. Experiment Setup

1) Benchmarks: Following the previous work [21]–[24], we adopt two public mainstream benchmarks, 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. [25] This benchmark is a manually verified subset of MBPP (Mostly Basic Programming Problems), contains 427 crowd-sourced Python programming problems, covering programming fundamentals, standard library functionality, and more. Each problem consists of an NL description, a code solution, and 3 automated test cases. For MBPP-sanitized, the NL description is provided as input.

HumanEval. [7] 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. [24] 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, etc..

MBPP-ET. and **HumanEval-ET** [26] 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.

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

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

CoT generates a chain of thought for each question by using CoT prompt and then generates the corresponding code. This approach is consistent with self-planning which is both two-stage processes.

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.

4) Implementation Details: The implementation details of our experiment are as follows.

Crafting Prompt. We categorize all benchmarks into the HumanEval and MBPP series, each having a fixed prompt. For HumanEval series, we randomly sample 8 questions from HumanEval to create the prompt. For MBPP series, given MBPP has an additional small training set, we identified four representative categories of problems: string processing, numerical calculations, and number theory problems, data structure manipulations, and geometric computations, and 8 questions are randomly sampled from these categories. Prompts for all baselines are constructed utilizing the same questions. For the self-planning approach, we manually craft the self-planning prompts. Self-planning prompts for HumanEval series and

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

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

TABLE I: Comparison of self-planning approaches and various baselines.

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
CoT	53.9	30.4	75.6	45.5	74.7	54.5	26.4	58.7	39.6	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

MBPP series are listed in Appendix B. For baseline CoT, since its original paper does not include a prompt for code generation tasks, we need to create a prompt for it. The way to create a 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 CoT prompts are manually reviewed and adapted to the same numbered list format as the self-planning prompts. The instance of CoT prompt can be found in Appendix D. The examples selected from the dataset for prompting will be excluded from the evaluation.

Ground-truth Plan Generation Labeling plans for all datasets is labor-intensive. To mitigate this concern, we adopt an alternative strategy of generating a pseudo-ground-truth plan through a 'reverse planning' approach. This approach annotates plans for code in reverse by utilizing ground-truth planning prompt which comes from self-planning prompts and ground-truth code. The instance of ground-truth planning prompt can be found in Appendix D.

Model Configuration and Evaluation. All basic baselines adopt code-davinci-002 as 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.

IV. Experimental Results

A. 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) [8], Incoder (6.7B) [22], CodeGeeX (13B) [24], CodeGen-Mono (16.1B) [23], and PaLM Coder (560B) [27]. The aim is to ascertain the performance level at which our approach operates relative to these recognized models. Second, we establish code-davinci-002 [7] as our base model and compare self-planning approach with Direct and CoT to demonstrate the effectiveness of our approach. 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 I, 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 CoT from the intent can provide a noteworthy advantage in code generation compared to the direct generation of code from the intent. As anticipated, our proposed self-planning approach demonstrates the best performance, showing a significant improvement in Pass@1 over the CoT approach across all benchmarks. While our approach demonstrates slightly lower performance on the CodeBLEU metric, it's worth noting that CodeBLEU assesses 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 I 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 benchmark 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 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 II. The diversity and accuracy of the self-planning approach consistently outperform Direct as the sample size

increases. In contrast, the diversity of 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.

TABLE II: 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
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

B. 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 small size model is restrictive, we use 4-shot setting for all prompting baselines in this experiment.

Results. The experimental results are presented in Table III. When the model is small, the impact of self-planning is less pronounced, constrained by 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 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 compared to text-davinci-002. Therefore, we posit that besides increasing model size, incorporating code training data and RLHF can also enhance 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 upgrade version than 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.

C. 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, two-phase. 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. 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 method delineates these into separate phases. Second, we evaluate the effects of self-planning with various example numbers (i.e., n-shot). Third, we explore five intermediate step configurations: CoT, unnumbered narrative CoT, unnumbered narrative plan, extremely concise plan, and Plan2CoT. CoT represents the style of step writing that we mentioned while crafting prompts, which is not recommended due to its low degree of abstraction from code. The unnumbered narrative CoT and plan are ablations of CoT and our proposed plan, i.e., it removes the form of a numbered list and is presented as narrative text. The extremely concise plan is an extremely concise version of our proposed plan. It composes of only a few phrases or verbs (keep only the keywords as much as possible), and an example of it is displayed in Appendix D. Plan2CoT means to start from the intent, first generate a plan then generate a CoT, ultimately resulting in code, i.e., we incorporate both the plan and the CoT during code generation.

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

In the result of group **Schemes of Planning and Implementation**, we find that the multi-turn usually fails to generate correct codes. This can be ascribed to two possible causes: 1) there is a lack of one-to-one correspondence between the code snippets and steps, with the implementation of a solution step possibly interspersed with multiple code snippets; 2) LLM faces challenges in determining the termination point for code generation with a sub-plan. Since the final goal or intent is specified at the outset, this often results in the model persistently generating the entire code without considering the indicated end of a step, making it difficult to determine where to truncate the code. 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 increased complexity of crafting prompt. 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.

TABLE III: Self-planning performance scales as a function of model size.

Approach	Diect			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

[1] Due to the maximum input length limitation, the evaluation of the self-planning is performed in 4-shot setting.
[2] In "Planning", we apply the plan generated by code-davinci-002 during the planning phase to the implementation phase of other models.

TABLE IV: Comparison of self-planning and its variants on HumanEval benchmark.

Variant	Pass@1	CodeBLEU	AvgPassRatio
Direct	48.1	24.0	63.2
Schemes of Planning and Implementation			
Multi-turn	28.2	18.1	47.5
One-phase	**62.8**	28.9	78.5
Number of Few-shot Exemple			
1-shot	53.2	28.4	74.3
2-shot	54.8	26.7	74.4
4-shot	59.0	29.3	76.9
Configurations of Intermediate Step			
CoT	53.9	30.4	75.6
Unnumbered Narrative CoT	50.6	29.0	72.2
Unnumbered Narrative Plan	55.8	27.7	73.6
Extremely Concise Plan	61.5	28.3	79.1
Plan2CoT	56.4	29.5	77.9
Self-Planning (Two-phase, 8-shot)	60.3	28.6	**80.8**

In the result of group **Number of Few-shot Exemple**, 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 2048 or 4096). 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 group **Configurations of Intermediate Step**, the improvement of 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. Thus CoT is not optimal for the code generation task, and planning approach is more suitable. The degraded performance exhibited by unnumbered narrative 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 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 of "Plan2CoT" outperformed CoT approach, suggesting that planning prior to generating CoT can enhance the accuracy of 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.

Overall, all variants except one-stage and extremely concise plan underperform the self-planning approach in terms of performance. However, one-stage approach requires more information, 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.

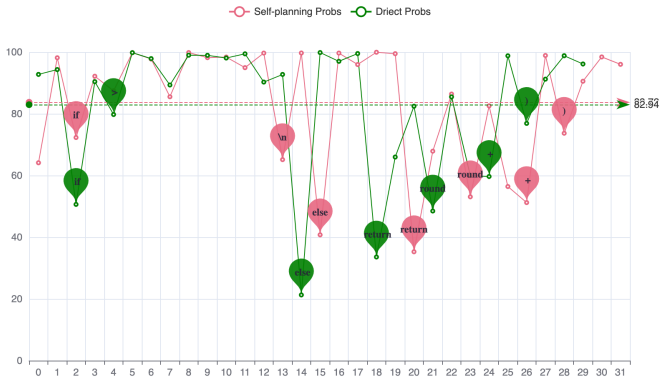

Fig. 3: An example of self-planning approach in improving probability (%) of key tokens generated by LLM. Each point on the line represents a generated token, where key tokens are marked, referring to those tokens with probabilities lower than the average.

D. Causes of the role of self-planning (RQ4)

To explore why self-planning works for code generation, we statistics the probability of code generated by LLM and find that the average probability of the code generated using the self-planning approach, as compared to the Direct approach, improved from 84.5% to 88.0%. To further understand the source of this improvement, we analyze the probability (i.e., confidence) of each token (or sub-token) generated by the LLM. Our analysis reveals that the confidence of some key tokens (such as control, arithmetic, and punctuation) is usually low in Direct approach, while our self-planning approach can improve the confidence of these key tokens. Fig. 3 illustrates an example where the self-planning approach has increased the probability of key tokens within the code generated by LLM. Intuitively, this proves that the self-planning approach reduces the complexity of code generation by providing guidance at key points of code generation.

Evaluation. We expand our analysis of key tokens on the HumanEval benchmark and assessing the impact of our self-planning approach on these key tokens.

First, we statistics the key tokens that is harder to predict on the benchmark. For each task of the benchmark, we calculate the average probability of all tokens generated by the Direct approach and then identify the tokens that fall below average probability. These tokens are regarded as the key tokens for the task as they present a greater challenge for LLM to predict accurately. We aggregated the probabilities and frequencies of the key tokens for all tasks, calculated the mean probability for each, and ranked the key tokens according to the mean probability to determine the top 20. Note that tokens appearing less than 10 times are omitted from our analysis as they are considered rare words with low frequency.

Second, we examine the improvement of our self-planning approach in the probability of key tokens compared to Direct. For each task, we calculate the difference in probability of key tokens of self-planning generated code relative to the corresponding key tokens of Direct generated code. On the whole benchmark, we calculate the average of the differences to get the improvement of self-planning approach on key tokens.

TABLE V: The improvement of the self-planning approach with respect to the top 20 tokens with the lowest prediction probability in HumanEval benchmark.

Key Token	Avg.prob↓	Freq	△Avg.prob
pass	**23.1**	**12**	**-**
count	24.4	11	1.2
#	28.4	33	3.4
max	41.7	13	1.2
return	**41.8**	**98**	**19.4**
x	42.3	35	1.3
number	42.9	13	3.1
if	**43.3**	**77**	**11.9**
a	47.6	14	11.6
else	**52.7**	**28**	**6.5**
sum	53.0	16	3.0
l	53.7	34	3.2
for	**53.8**	**52**	**12.2**
s	53.8	11	-0.1
’	55.1	19	8.1
==	55.7	34	13.9
el	56.1	22	3.2
>	57.8	19	4.5
int	58.6	11	4.0
[	59.0	33	11.7

[1] “Avg.prob” denotes the average probability of key tokens generated by Direct approach.
[2] “Freq” denotes the frequency of occurrence for key tokens.
[3] “△Avg.prob” denotes the average probability improvement of each key token through the self-planning approach.
[4] “↓” denotes in ascending order.

Results. Table V presents the top 20 most challenging tokens (or subtokens) to predict in the HumanEval benchmark. These 20 tokens can be classified into four categories: control key tokens, arithmetic key tokens, punctuation key tokens, and identifier key tokens.

Control key tokens, such as “pass”, “return”, “if”, “else”, “for”, and “el” (a subtoken that combines with “if” to form “elif”), are crucial for denoting the start of a sub-functional block of code. Among all token types, control key tokens exhibit the most significant improvement in the self-planning approach compared to the Direct approach. The Direct approach often incorrectly generates the “pass” token, which indicates that no action is taken, while this behavior is not observed in the self-planning approach. As a result, the improvement in the probability of the “pass” token is denoted as ’-’.

Arithmetic key tokens include “==”, “>”, “count”, “max”, “number”, “sum”, and “int”, relate to arithmetic operations. Punctuation key tokens, such as “#”, “’”, and “[” mark the beginning of comments, strings, and indices. The self-planning approach shows improvement in predicting both arithmetic and punctuation key tokens.

Identifier key tokens, comprising “x”, “a”, “l”, and “s”, are referred to as “meaningless identifiers” since they lack specific meaning or purpose within the code. Programmers often assign these arbitrary names for convenience during the coding process. The self-planning approach exhibits limited improvement in utilizing meaningless identifiers and, in some instances, demonstrates a decrease, indicating a modest tendency for the self-planning approach to employ such identifiers.

In conclusion, the language model displays low confidence for certain key tokens, indicating difficulties in decision-making. Incorporating a planning approach can assist the model in predicting these key tokens more confidently. The point of the planning approach is to obtain a high-quality plan that calibrates crucial points, enabling the model to generate correct code by following the plan.

E. Performance on Multilingual Code Generation (RQ5)

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 VI, our self-planning approach exhibits positive results across all PLs when compared to Direct and 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, etc. 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 further enhance. Moreover, if a plan is created independent of a specific PL, its generalization across different languages would be improved.

TABLE VI: 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
CoT	53.9	30.4	56.4	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
CoT	52.6	27.0	48.1	27.1
Self-planning	**55.8** (↑ 4.9%)	25.6	**5.0** (↑ 24.0%)	26.5
Ground-truth Planning	60.3 (↑ 13.3%)	29.6	58.3 (↑ 35.9%)	32.0

V. 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.
- 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, 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. 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.

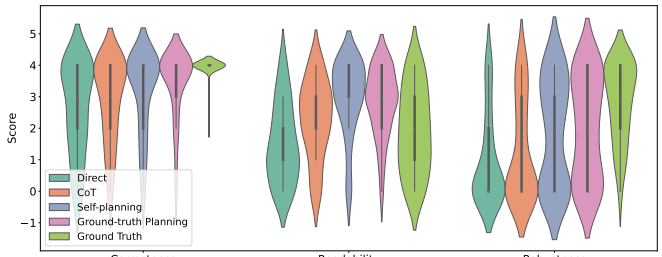

Fig. 4: Violin plot for human evaluation. The violin plot is a combination of a box line plot, which shows the location of the quartiles (the upper edge of the box represents the upper quartile, the middle point is the median, and the lower edge of the box is the lower quartile), and a kernel density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

Results. The evaluation results of the two groups are summarized in Fig. 4. Correctness scoring closely aligns with the Pass@1 results, while also considering partial correctness. The self-planning approach outperforms both Direct and 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 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 CoT and performs comparably to Ground-truth planning, 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 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 Fig. 5, we demonstrate the performance of both direct, self-planning, and CoT code generation through two cases. In these cases, the direct and 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 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 CoT from intent and generating code is comparable.

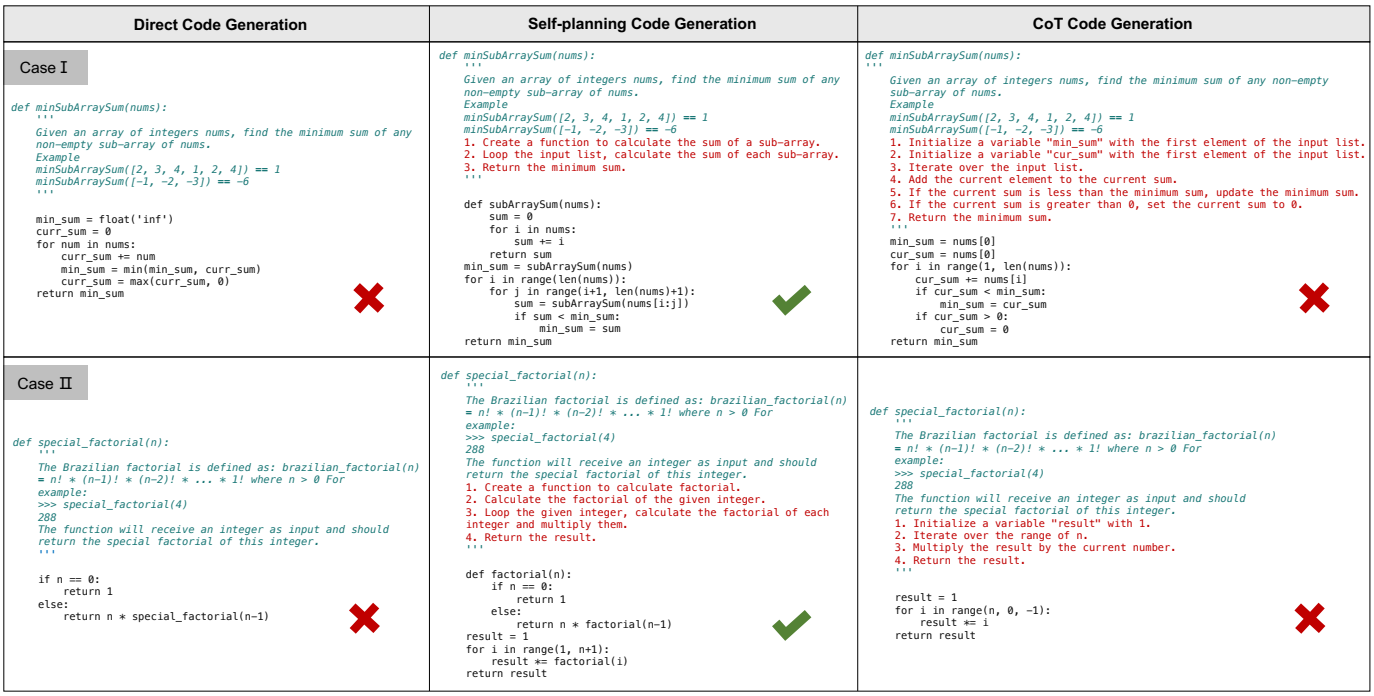

Fig. 5: Two real cases from HumanEval with Self-planning and direct code generation. The input, generated plan, and code are highlighted in green, red, and black, respectively.

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

VI. Threats to Validity

The primary threat comes from crafting prompt, as the example selection and plan writing in this operation can affect the degree of improvement achievable by our approach. This issue stems from the inherent sensitivity of LLMs to prompts, necessitating fundamental improvements to the LLMs for resolution [28], [29]. 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 [30] and generating prompts [31], [32] can be used to maximize the performance of the prompting approach. These results can be introduced into our approach to further improve the performance.

VII. Related Work

A. Code Generation

Traditional code generation approaches are based on supervised learning, which initially treats code as equivalent to natural language [33]–[35] and then gradually incorporates more code-specific features, such as abstract syntax tree [36]–[40], API calls [41]–[43]. Furthermore, Mukherjee et al. [44] present a generative modeling approach for source code that uses a static-analysis tool. Dong et al. [2] devise a PDA-based methodology to guarantee grammatical correctness for code generation.

With the rise of pre-training, CodeT5 [45], UniXcoder [46] applied pre-trained models to code generation task. The introduction of subsequent models like Codex [7], InCoder [22], CodeGen [23], AlphaCode [8], and CodeGeeX [24], 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 large language models as the backbone, enhancing their code generation performance through various techniques [5], [47]–[49], 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.

B. Prompting Techniques

Few-shot prompting [50] 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 Chain of thought (CoT) [12] achieves a significant improvement that transcends the scaling laws by generating intermediate reasoning steps before the answer to address language reasoning

tasks. Inspired by CoT, a series of prompting works has been proposed. Least-to-most prompting [51] 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 [18] and PoT [19] is proposed to generating 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 [52], [53], commonsense [54], [55], and symbolic reasoning [15] problems characterized by limited problem spaces and established solution patterns. Consequently, their applicability to code generation remains restricted.

VIII. Discussion and Future Work

When discussing the limitations of self-planning code generation, a major limitation may be manual crafting of prompt. however, we should also be aware that in previous approaches, thousands 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 8 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 uses a sequential list to represent plans with branch-and-loop structures. However, this representation is limited to one level of nesting, limiting its ability to express complex logical structures and multi-level subproblem decomposition. To further improve and enhance the approach, it can be considered to convert this sequential list into a hierarchical list. By introducing a hierarchical structure, the subtasks of the plan can be organized into higher-level modules, which will allow for clearer problem decomposition and better handling of complex problem domains.

Finally, this paper attempts to reduce the difficulty of code generation by planing for human intent, which is consistent with the methodology of dealing with problem complexity in requirements engineering, i.e. abstraction and decomposition [56]. 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.

IX. Conclusion

In this paper, 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.

References

[1] G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, "Synchromesh: Reliable code generation from pre-trained language models," in *ICLR*, 2022.

[2] Y. Dong, G. Li, and Z. Jin, "CODEP: grammatical seq2seq model for general-purpose code generation," in *ISSTA*. ACM, 2023, pp. 188–198.

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

[4] S. Shen, X. Zhu, Y. Dong, Q. Guo, Y. Zhen, and G. Li, "Incorporating domain knowledge through task augmentation for front-end javascript code generation," in *ESEC/SIGSOFT FSE*. ACM, 2022, pp. 1533–1543.

[5] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen, "CodeT: Code generation with generated tests," in *ICLR*, 2023.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS 2020*, 2020.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

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

[9] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," 2002.

[10] K. Petersen, C. Wohlin, and D. Baca, "The waterfall model in large-scale development," in *PROFES*, ser. Lecture Notes in Business Information Processing, vol. 32. Springer, 2009, pp. 386–400.

[11] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Softw. Eng. Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[13] K. Cobbe, V. Kosaraju, M. Bavarian, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training verifiers to solve math word problems," *CoRR*, vol. abs/2110.14168, 2021.

[14] A. Talmor, O. Tafjord, P. Clark, Y. Goldberg, and J. Berant, "Leap-of-thought: Teaching pre-trained models to systematically reason over implicit knowledge," in *NeurIPS*, 2020.

[15] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *ICLR*. OpenReview.net, 2023.

[16] M. Turpin, J. Michael, E. Perez, and S. R. Bowman, "Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting," *CoRR*, vol. abs/2305.04388, 2023.

[17] Q. Lyu, S. Havaldar, A. Stein, L. Zhang, D. Rao, E. Wong, M. Apidianaki, and C. Callison-Burch, "Faithful chain-of-thought reasoning," *CoRR*, vol. abs/2301.13379, 2023.

[18] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig, "PAL: program-aided language models," *CoRR*, vol. abs/2211.10435, 2022.

[19] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," *CoRR*, vol. abs/2211.12588, 2022.

[20] C. Tai, Z. Chen, T. Zhang, X. Deng, and H. Sun, "Exploring chain-of-thought style prompting for text-to-sql," *CoRR*, vol. abs/2305.14215, 2023.

[21] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.

[22] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *CoRR*, vol. abs/2204.05999, 2022.

[23] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[24] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x," *CoRR*, vol. abs/2303.17568, 2023.

[25] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021.

[26] Y. Dong, J. Ding, X. Jiang, Z. Li, G. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *CoRR*, vol. abs/2301.09043, 2023.

[27] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, E. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, S. Narang, G. Mishra, A. Yu, V. Y. Zhao, Y. Huang, A. M. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, "Scaling instruction-finetuned language models," *CoRR*, vol. abs/2210.11416, 2022.

[28] V. P. S. Nookala, G. Verma, S. Mukherjee, and S. Kumar, "Adversarial robustness of prompt-based few-shot learning for natural language understanding," in *ACL (Findings)*. Association for Computational Linguistics, 2023, pp. 2196–2208.

[29] K. Zhu, J. Wang, J. Zhou, Z. Wang, H. Chen, Y. Wang, L. Yang, W. Ye, N. Z. Gong, Y. Zhang, and X. Xie, "Promptbench: Towards evaluating the robustness of large language models on adversarial prompts," *CoRR*, vol. abs/2306.04528, 2023.

[30] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," in *NAACL-HLT*. Association for Computational Linguistics, 2022, pp. 2655–2671.

[31] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," in *ICLR*. OpenReview.net, 2023.

[32] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," in *ICLR*. OpenReview.net, 2023.

[33] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang, and A. W. Senior, "Latent predictor networks for code generation," in *ACL (1)*. The Association for Computer Linguistics, 2016.

[34] R. Jia and P. Liang, "Data recombination for neural semantic parsing," in *ACL (1)*. The Association for Computer Linguistics, 2016.

[35] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *NeurIPS*, 2019, pp. 6559–6569.

[36] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *ACL (1)*. Association for Computational Linguistics, 2017, pp. 1139–1149.

[37] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *ACL (1)*. Association for Computational Linguistics, 2017, pp. 440–450.

[38] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," in *AAAI*. AAAI Press, 2019, pp. 7055–7062.

[39] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *AAAI*. AAAI Press, 2020, pp. 8984–8991.

[40] P. Yin and G. Neubig, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *EMNLP (Demonstration)*. Association for Computational Linguistics, 2018, pp. 7–12.

[41] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*. ACM, 2014, pp. 419–428.

[42] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *SIGSOFT FSE*. ACM, 2016, pp. 631–642.

[43] ——, "Deepam: Migrate apis with multi-modal sequence to sequence learning," in *IJCAI*. ijcai.org, 2017, pp. 3675–3681.

[44] R. Mukherjee, Y. Wen, D. Chaudhari, T. W. Reps, S. Chaudhuri, and C. M. Jermaine, "Neural program generation modulo static analysis," in *NeurIPS*, 2021, pp. 18 984–18 996.

[45] Y. Wang, W. Wang, S. R. Joty, and teven C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *EMNLP (1)*, 2021, pp. 8696–8708.

[46] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *ACL (1)*. Association for Computational Linguistics, 2022, pp. 7212–7225.

[47] T. Zhang, T. Yu, T. B. Hashimoto, M. Lewis, W. Yih, D. Fried, and S. I. Wang, "Coder reviewer reranking for code generation," *CoRR*, vol. abs/2211.16490, 2022.

[48] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," *CoRR*, vol. abs/2303.05510, 2023.

[49] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *CoRR*, vol. abs/2304.05128, 2023.

[50] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 195:1–195:35, 2023.

[51] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, O. Bousquet, Q. Le, and E. H. Chi, "Least-to-most prompting enables complex reasoning in large language models," *CoRR*, vol. abs/2205.10625, 2022.

[52] A. Lewkowycz, A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, Y. Wu, B. Neyshabur, G. Gur-Ari, and V. Misra, "Solving quantitative reasoning problems with language models," in *NeurIPS*, 2022.

[53] Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C. Szegedy, "Autoformalization with large language models," in *NeurIPS*, 2022.

[54] V. Sanh, A. Webson, C. Raffel, S. H. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, A. Raja, M. Dey, M. S. Bari, C. Xu, U. Thakker, S. S. Sharma, E. Szczechla, T. Kim, G. Chhablani, N. V. Nayak, D. Datta, J. Chang, M. T. Jiang, H. Wang, M. Manica, S. Shen, Z. X. Yong, H. Pandey, R. Bawden, T. Wang, T. Neeraj, J. Rozen, A. Sharma, A. Santilli, T. Févry, J. A. Fries, R. Teehan, T. L. Scao, S. Biderman, L. Gao, T. Wolf, and A. M. Rush, "Multitask prompted training enables zero-shot task generalization," in *ICLR*. OpenReview.net, 2022.

[55] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig, "Language models of code are few-shot commonsense learners," in *EMNLP*. Association for Computational Linguistics, 2022, pp. 1384–1403.

[56] L. A. Macaulay, *Requirements engineering*. Springer Science & Business Media, 2012.

[57] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with APPS," in *NeurIPS Datasets and Benchmarks*, 2021.

[58] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020.

APPENDIX

A. Details of Metrics

Pass@k. We use the unbiased version [8] 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 [57] 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 [58] 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-currence (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. Self-planning Prompt for HumanEval Series Benchmarks

The instruction *"Let's think step by step"* acts as a start-of-plan flag, and replacing it with $< plan >$ does not change the performance (Pass@1 also 60.3 on HumanEval).We keep it for semantic coherence.

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 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
Let's think step by step.
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)

Let’s think step by step.

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

Let’s think step by step.

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

Let’s think step by step.

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
Let’s think step by step.
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]
Let’s think step by step.
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
Let’s think step by step.
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.
”’

C. Self-planning Prompt for MBPP Series 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.
Let’s think step by step.
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.
Let's think step by step.
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.
Let's think step by step.
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.
Let's think step by step.
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.
Let's think step by step.
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.
Let's think step by step.
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.
Let's think step by step.
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.
Let's think step by step.
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.

D. Instances of Baseline Prompt

Instance of Chain-of-Thought Prompting

```
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'
    Let's think step by step.
    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'
    Let's think step by step.
    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 = ''
    for char in s:
        index = alphabet.find(char) + bias
        if index > 25:
            index = index - 26
        result += alphabet[index]
    return result
```

E. Human Evaluation Questionnaire

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

Human Evaluation

Please score the following codes in terms of correctness, readability, and robustness according to the given requirements.

Requirement:

Given an array of integers nums, find the minimum sum of any non-empty sub-array of nums.
Example:
minSubArraySum([2, 3, 4, 1, 2, 4]) == 1
minSubArraySum([-1, -2, -3]) == -6

Code 1:

```
def minSubArraySum(nums):
    max_sum = 0
    s = 0
    for num in nums:
        s += -num
        if (s < 0):
            s = 0
        max_sum = max(s, max_sum)
    if max_sum == 0:
        max_sum = max(-i for i in nums)
    min_sum = -max_sum
    return min_sum
```

Code 2:

```
def minSubArraySum(nums):
    min_sum = float('inf')
    curr_sum = 0
    for num in nums:
        curr_sum += num
        min_sum = min(min_sum, curr_sum)
        curr_sum = max(curr_sum, 0)
    return min_sum
```

Code 3:

```
def minSubArraySum(nums):
    '''
    1. Create a function to calculate the sum of a sub-array.
    2. Loop the input list, calculate the sum of each sub-array.
    3. Return the minimum sum.
    '''
    def subArraySum(nums):
        sum = 0
        for i in nums:
            sum += i
        return sum
    min_sum = subArraySum(nums)
    for i in range(len(nums)):
        for j in range(i+1, len(nums)+1):
            sum = subArraySum(nums[i:j])
            if sum < min_sum:
                min_sum = sum
    return min_sum
```

Code 4:

```
def minSubArraySum(nums):
    '''
    1. Initialize a variable "min_sum" with the first element of the input list.
    2. Initialize a variable "cur_sum" with the first element of the input list.
    3. Iterate over the input list.
    4. Add the current element to the current sum.
    5. If the current sum is less than the minimum sum, update the minimum sum.
    6. If the current sum is greater than 0, set the current sum to 0.
    7. Return the minimum sum.
    '''
    min_sum = nums[0]
    cur_sum = nums[0]
    for i in range(1, len(nums)):
        cur_sum += nums[i]
        if cur_sum  0:
            cur_sum = 0
    return min_sum
```

Fig. 6: Questionnaire Part 1

Code 5:

```
def minSubArraySum(nums):
    '''
    1. Create a variable to store the maximum sum.
    2. Loop the input list, add the negative number to the sum.
    3. If the sum is less than 0, set the sum to 0.
    4. Update the maximum sum.
    5. If the maximum sum is 0, find the maximum negative number.
    6. Return the minimum sum.
    '''
    if not nums:
        return 0
    min_sum = nums[0]
    for i in range(1, len(nums)):
        min_sum = min(min_sum, nums[i] + min_sum)
    return min_sum
```

01 Please score the 5 codes in terms of **Correctness**.

	0 (Worst)	1	2	3	4 (Best)
Code 1	○	○	○	○	○
Code 2	○	○	○	○	○
Code 3	○	○	○	○	○
Code 4	○	○	○	○	○
Code 5	○	○	○	○	○

02 Please score the 5 codes in terms of **Readability**.

	0 (Worst)	1	2	3	4 (Best)
Code 1	○	○	○	○	○
Code 2	○	○	○	○	○
Code 3	○	○	○	○	○
Code 4	○	○	○	○	○
Code 5	○	○	○	○	○

03 Please score the 5 codes in terms of **Robustness**.

	0 (Worst)	1	2	3	4 (Best)
Code 1	○	○	○	○	○
Code 2	○	○	○	○	○
Code 3	○	○	○	○	○
Code 4	○	○	○	○	○
Code 5	○	○	○	○	○

Submit

Fig. 7: Questionnaire Part 2