CoCoST: Automatic Complex Code Generation with Online Searching and Correctness Testing

Anonymous ACL submission

Abstract

Large Language Models have revolutionized code generation ability by converting natural language descriptions into executable code. However, generating complex code within real-world scenarios remains challenging due to intricate structures, subtle bugs, understanding of advanced data types, and lack of supplementary contents. To address these challenges, we introduce the CoCoST framework, which enhances complex code generation by online searching for more information with planned queries and correctness testing for code refinement. Moreover, CoCoST serializes the complex inputs and outputs to improve comprehension and generates test cases to ensure the adaptability for real-world applications. CoCoST is validated through rigorous experiments on the DS-1000 and ClassEval datasets. Experimental results show that CoCoST substantially improves the quality of complex code generation, highlighting its potential to enhance the practicality of LLMs in generating complex code.

1 Introduction

Automatic code generation from natural language descriptions is becoming more realistic, as large language models (LLMs) show their potential to generate accurate code (Li et al., 2023; Luo et al., 2023; Rozière et al., 2024). Various methods have been proposed to improve the quality of LLM code generation, such as retrieving offline documents (Zhou et al., 2023; Jiang et al., 2023) and debugging generated code (Zhang et al., 2023; Chen et al., 2023). However, complex code generation is a more difficult task, which involves intricate problem description, sophisticated code logic, and advanced data types (Lai et al., 2022; Du et al., 2023; He et al., 2023). The existing methods struggle to address the arising challenges:

Challenge 1: Offline documents cannot meet the diverse demands of code generation. In real-world scenarios, these demands often exceed the capabilities of limited offline documents. For example, problem descriptions may involve functions that are not covered by pre-collected documents. Additionally, complex code generation for diverse needs often entails highly complex logic and a series of transformation functions like the programming problem in Figure 1, where simple API examples in documents fail to provide adequate guidance.

Challenge 2: In real-world situations, there is often a shortage of test cases (*e.g.*, test cases in Figure 1) for automatic code generation. Most existing work depends heavily on pre-existing test cases in datasets (Zhang et al., 2023; Jiang et al., 2023) , which are difficult to acquire directly in practical scenarios.

Challenge 3: Hidden bugs in complex code require meticulous identification and refinement. Current techniques frequently enhance code by analyzing execution errors (Zhang et al., 2023; Jiang et al., 2023). But in the case of complex code, the executable code sometimes contains hidden bugs like the highlighted part of the initial code in Figure 1.

To address these challenges, we introduce a new code generation framework named **CoCoST**[1] (Automatic **Co**mplex **Co**de Generation with Online **S**earching and Correctness **T**esting) that improves the generation and refinement of complex code by LLMs through the planned online searching and automatic correctness testing steps. The intuition of CoCoST is straightforward: During the coding process, most human developers are not bothered by the above challenges, as illustrated in Figure 1. Developers can easily overcome these obstacles by searching online through engines (*e.g.*, Google and Bing) for solutions, experiences, and guidelines. In addition, they can create test cases and execute code to ensure the correctness of the code logic.

To address *Challenges 1*, CoCoST proposes

[1]The code will be publicly available upon acceptance.

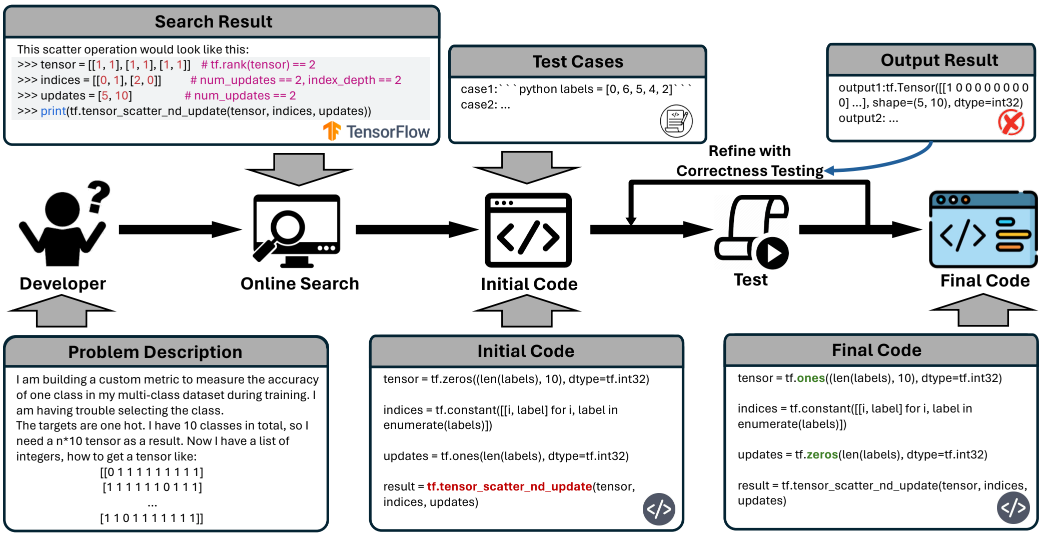

Figure 1: An Example of the Human Developer Code-writing Process Imitated by the CoCoST. After the problem is received, an online search is performed to simulate search results and create an initial version of code. Test cases are then generated, and the code is executed to produce output results. The code is refined based on the correctness of these results.

an **online search** methodology. This process involves querying web search engines and then extracting pertinent information to construct prompts for LLMs. The approach presents several benefits: (1) Retrieving information from the up-to-date blogs or Q&A platforms, such as StackOverflow, facilitates the emulation of commonly used code patterns, thereby reducing the complexity of generated code. (2) Online search extends beyond the scope of static offline documentation, covering a wider array of problems without being confined to a predetermined set. Meanwhile, it reduces the effort developers need to expend in assembling documentation, thereby increasing the framework's level of automation. However, using problem descriptions as search queries can be challenging due to the complexity of the problems, which often encompass multiple issues. Therefore, we propose an online search with **generation query through planning**.

To address the *Challenge 2*, we introduce **generation of test cases** during refinement. Several studies (Chen et al., 2022; Shinn et al., 2023) have attempted to generate tests. However, these methods often fall short when applied to the generation of complex code due to its intricate logic and outputs, which complicate the direct production of accurate tests (both the inputs and expected outputs for the solution code). CoCoST utilizes LLMs to automatically generate test cases (the inputs for the code). This strategy cleverly focuses on generating test cases without attempting to produce complete tests. It significantly simplifies the process of test case generation and facilitates its precise creation for complex code.

To address the *Challenge 3*, this work prioritizes **correctness testing** in refinement. During the refinement process, it is more critical to verify that the executed code produces the correct results rather than just checking for the existence of the errors. CoCoST incorporate both the execution output results and the errors into within the refinement prompts for LLMs to enhance the correctness. Moreover, during refinement, sophisticated data types and structures (within complex code itself, its inputs, and its execution results) are challenging for LLMs to understand, *e.g.*, large Pandas DataFrames, Matplotlib charts. Thus, CoCoST propose **serialization of input and output** to convert them into understandable sequences before being processed by LLMs. Particularly those are excessively long or non-textual modalities.

We evaluate the effectiveness of CoCoST on two complex code generation datasets (DS-1000 and ClassEval). By comparing with the existing state-of-the-art (SOTA) baseline, we achieve a 7.8% improvement on DS-1000 and average 9.47% on ClassEval.Moreover, we analyze and discovery that CoCoST requires models to have different capabilities such as planning, which vary according to the complexity of the problem.

In summary, our main contributions are:

- We propose the novel CoCoST framework to generate complex code, and can be automatic in real-world scenarios.

- We apply online search in code generation for the first time to our knowledge, in order to generate the complex code.
- We prioritize correctness testing in refinement with test case generation and serialization of input and output, in order to refine hidden bugs in complex code.
- We conduct experiments on DS-1000 and ClassEval datasets anddemonstrate the effectiveness and universality of CoCoST.

2 Related Work

Code generation datasets. The realm of automated code generation has been propelled by benchmark datasets such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021), which assess the proficiency of language models in generating executable code from descriptions. These datasets encompass a variety of programming problems, yet recent studies have sought to escalate the complexity of code generation tasks. Works like DS-1000 (Lai et al., 2022), ClassEval (Du et al., 2023) and Text2Analysis (He et al., 2023) have introduced datasets targeting specialized domains, including data science, object-oriented class generation, and data analysis. These endeavors reflect an emerging trend towards enhancing models' abilities to produce sophisticated and domain-specific code structures. In this paper, we select datasets with complex code generation to evaluate CoCoST.

Retrieval-augmented code generation. With the emergence of Large Language Models (LLMs), a variety of retrieval-augmented techniques have been developed to compensate for issues such as the inherent knowledge limitations. DocPrompt(Zhou et al., 2023) and SELVEVOLVE(Jiang et al., 2023) leverage document libraries or models as knowledge bases to improve code generation. However, their reliance on fixed document libraries limits the scope of information they can provide and confines the generated code to the context of these libraries. Furthermore, the prerequisite of preestablished document libraries prevents these approaches from being fully autonomous in real-world frameworks. Solutions such as WebGPT (Nakano et al., 2022), LaMDA (Thoppilan et al., 2022), and FreshLLMs (Vu et al., 2023) enhance the performance of natural language tasks by using online search or open web knowledge. However, because complex code generation often involves multiple steps and complexities, these methods struggle with direct application to complex code generation.

Code refinement. Refine iteratively enhances generated code for greater precision. Self-Debug(Chen et al., 2023), SELFEVOLVE(Jiang et al., 2023), and Self-Edit(Zhang et al., 2023) improve code generation by refining code through the resolution of errors identified during execution. These methods effectively address errors, while when it comes to complex code generation, subtle bugs also play a significant role in the overall error landscape. Moreover, relying on pre-existing tests from datasets in refinement limits their autonomy in real-world applications, where such tests may not be readily available. CodeT (Chen et al., 2022), Reflexion (Shinn et al., 2023), and CODECHAIN (Le et al., 2023) seek to strengthen code generation by creating tests. But the tests they generate include not only the inputs for the solution code but also the expected outputs. This poses a substantial challenge for complex code generation, where the logic can be intricate and certain problems may not lend to straightforward ground truth generation.

3 Methodology

The code generation task involves predicting a solution code W given a problem description D. When given an input i, the execution of code W produces an output result o and a potential error e, where both o and e can be empty $\varnothing$. The generated codes are evaluated against a set of test cases and ground truth $\{(t_j, g_j)\}_{j=1}^{J}$. The correctness of the code W is determined by verifying $o_j = g_j \wedge e_j = \varnothing$ when all $i_j = t_j, j \in \{1, \ldots, J\}$.

In this work, we adopt a two-step approach for code generation, mirroring the way humans write code. The first step is retrieval, where relevant information is obtained through online search and utilized by LLMs to generate initial code. The second step is refinement, where the initial code is refined based on the execution results, leading to the generation of the final version of the code.

3.1 Retrieval

The difficulty in achieving effective online retrieval lies in formulating optimal search queries. On the one hand, for complex code generation, the prob-

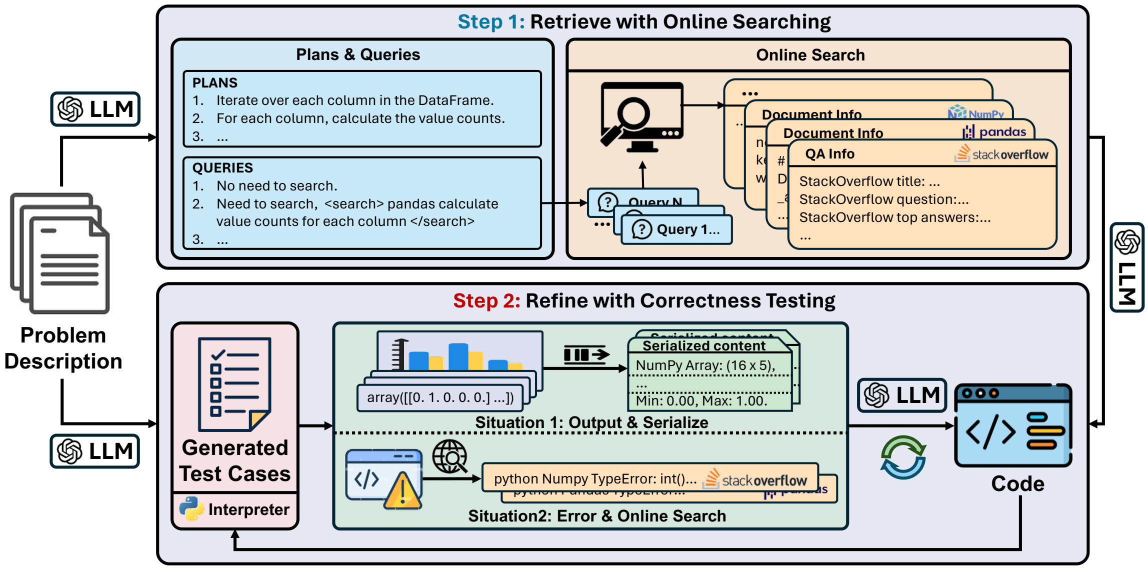

Figure 2: The Pipline of CoCoST. **Step 1:** LLM is employed to strategize the Problem and formulate queries based on the outlined steps. These queries enable the retrieval of diverse information from the internet. A high-quality initial code can be obtained through effective planning and leveraging internet information. **Step 2:** LLM generates test cases for testing the initial code. The test results serve as crucial inputs for the subsequent cycle of code refinement. Through iterative refinement processes, the quality of the initial code can be significantly improved.

lems are intricate and may involve multiple challenges. Directly searching for solutions to such problems is inaccurate and difficult. On the other hand, it is challenging that match queries directly through methods for offline documents like similarity calculations, due to the nature of online libraries. So we propose generating queries through planning to solve the challenge.

The retrieval process is divided into three steps: 1. Search queries $Q = \{q_1, \ldots, q_N\}$ are generated through planning. 2. Conducting online searches using these queries to obtain relevant background information $INFO = \{info_1, \ldots, info_M\}$. 3. The initial code W_0 is generated by the LLMs θ with the information obtained $INFO$:

$$\widehat{W_0} \sim p_\theta(.|D, INFO) \tag{1}$$

3.1.1 Generation Query through Planning

In order to generate more targeted queries, we initiate the process by using LLMs to do planning regarding the given problem. The planning phase involves outlining the natural language steps $P = \{plan_1, \ldots, plan_N\}$ required to address the problem. Later, the assessment involves utilizing LLMs to determine whether each planning step requires an online search. Subsequently, the planning steps identified as necessitating online search are translated into queries $Q = \{q_1, \ldots, q_N\}$ for use in the subsequent search process.

$$\widehat{P}, \widehat{Q} \sim p_\theta(.|D) \tag{2}$$

3.1.2 Online Search

For the above generated queries, we conduct an online search. In this study, we use the online search API[2] for the search process as Equation (3). CoCoST can also be applied to private or domain-specific knowledge repositories as long as they are accessible via query, with details in §A.

$$\{url_1, \ldots, url_{N_u}\} = search(q_j), j \in \{1, \ldots, N_q\} \tag{3}$$

where, N_q is the number of queries for the problem, N_u is the number of urls for one query. In this study, we use $N_q = 1$, $N_u = 1$.

Through the analysis of the website distribution Table 4, we observed that more than 90% of the URLs are concentrated on a total of 8 websites. Specific extraction rules are established for prominent websites such as StackOverflow to extract key information, facilitating a more comprehensive understanding of the website's content by subsequent models. Generic extraction rules are employed for extracting key information from other websites.

$$info_{j,k} = extract(url_k), k \in \{1, \ldots, N_u\}$$

The information $INFO$ is composed of details from each query q_j, each url url_k, with each piece of information $info_{j,k}$ extracted.

[2] https://github.com/Nv7-GitHub/googlesearch

3.2 Refinement

Existing work (Chen et al., 2023; Jiang et al., 2023) typically emphasizes the correctness of errors identified during the refinement process. However, we observe that refining code that produces error-free outputs is equally crucial during the refinement process. Therefore, we introduce correctness testing in §3.2.1. Additionally, we propose methods for the generation of test cases and serialization of inputs and outputs during the refinement process.

3.2.1 Correctness Testing

Correctness testing refers to the refinement of generated code based on correctness, determined by analyzing errors and output results obtained during code execution. In the context of complex code generation, the intricate logic of the code makes it challenging for the LLMs to consider every detail during code generation, and precisely ascertain the results obtained at each step of the execution process. Consequently, some code may be executed without errors, producing output results that do not align with what is expected. Incorporating both the error and the output result into the refinement process allows the model to take advantage of self-correction mechanisms.

$$\begin{cases} e_{j,k}, o_{j,k} &= execute(W_j, i_k), j \in \{1, \ldots, N_f\} \\ INFO_{e_{j,k}} &= \{e_{j,k}, extract(search(e_{j,k}))\} \\ \widehat{W}_{j+1} &\sim p_\theta(.|D, W_j, \{S_i, S_{o_j}, INFO_{e_j}\}_k), \\ & \qquad k \in \{1, \ldots, N_i\} \end{cases}$$

where, N_f is the total number of refinement steps, N_i is the number of inputs. i_K is the k-th input for the problem from Equation (4), S_i and S_{o_j} is the serialization of input and output from Equation (5).

3.2.2 Generation of Test Cases

Test cases are crucial, as they serve as indispensable inputs for the code execution in refinement. While, existing works in refining code predominantly rely on pre-existing test cases in datasets (Zhang et al., 2023; Jiang et al., 2023), which are challenging to obtain directly in real-world scenarios. Moreover, some existing work (Chen et al., 2023) even uses the ground truth output of the test case to refine the code, which is even more challenging to obtain for complex code problems in real-world scenarios. Because their problems involve various logical operations, deriving answers directly without code-based computations is demanding.

CoCoST introduces generation of test cases with LLMs to adapt to real-world scenarios.

$$\begin{cases} \widehat{I} &\sim p_\theta(.|D) \\ I &= \{i_1, \ldots, i_{N_i}\} \end{cases} \tag{4}$$

3.2.3 Serialization of Input and Output

Serialization of input and output makes them more intuitive and understandable for the model. For complex code, some inputs and outputs are intricate, such as Pandas DataFrames, PyTorch tensors, and Matplotlib PNG images. Understanding such inputs and outputs poses challenges for LLMs due to large matrices, image modalities, and so on.

In this study, we serialize common data structures in Python as follows:

1. For NumPy arrays, Pandas DataFrames, PyTorch tensors, and TensorFlow tensors, the serialization includes data truncated string, data type, data shape, and statistical information.
2. For image structures (such as PNG images generated by the Matplotlib library), we serialize them into SVG (Scalable Vector Graphics) format for LLMs to comprehend.

$$S_n = serialize(n), n \in \{i_k, o_{j,k}\} \tag{5}$$

4 Experiment

4.1 Experiment Setup

4.1.1 Datasets

We conduct experiences on two complex code generation datasets:

DS-1000 (Lai et al., 2022): DS-1000 is a code generation benchmark with a thousand data science questions spanning seven Python libraries. The complexity of this dataset is manifested in two aspects. First, complexity arises from intricate logical reasoning required during code generation due to the complex nature of the problems. For example, on the DS-1000 dataset, the average length of problem descriptions is 140 words, whereas other commonly used code generation datasets such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) have lengths of 23 and 15.7 words, respectively. Secondly, the input-output involves various complex data structures related to data science, making the code logic intricate during transformations of the data. Further details of DS-1000 implementation are shown in §B.1.

ClassEval (Du et al., 2023): ClassEval is the first class-level Python code generation benchmark designed to evaluate code generation models' performance on a diverse set of object-oriented programming tasks. The dataset comprises a curated

Table 1: Main Results and Ablation Study for DS-1000. The base model for CoCoST is GPT-4. All metrics are represented as percentages. For each metric, the **bold** number indicates the highest performance.

Method	Perturbation				Total/Avg.
	Origin	Surface	Semantic	Diff-Rewrite	
Codex	44.93	37.94	34.35	16.94	39.20
DocPrompting	53.95	50.00	38.39	21.05	43.30
Self-Debugging	63.38	59.21	45.65	28.40	53.00
SELFEVOLVE	66.23	67.11	48.70	33.95	57.10
Reflexion	58.99	73.03	52.17	48.77	57.90
CoCoST	**71.71**	74.34	**66.96**	**53.09**	**68.00**
w/o refinement of output	68.42	69.74	62.61	48.77	64.10
w/o refinement of error	68.20	73.03	62.61	49.38	64.60
w/o serialization	70.18	**75.00**	65.22	51.23	66.70
w/o generation of test case	66.23	71.05	59.57	45.68	62.10
w/o online retrieval	68.64	70.39	60.00	51.23	64.10
w/o all (GPT-4 only)	64.47	69.74	56.96	43.83	60.20

collection of 100 tasks. These tasks cover a wide range of concepts, including inheritance, polymorphism, encapsulation, etc. Each coding task is in the format of the class skeleton, outlining the target method description inside the class. The complexity of this dataset resides in its abstraction and hierarchical class structure. Tested models must generate large-scale code units and establish connections between each target method within the entire class, rather than focusing solely on individual functions.

The dataset provides two prompt designs for LLMs with or without IF ability. In our experiments, we employ the class skeleton as the prompt for GPT-based models, a system prompt along with task instructions for the WizardCoder.

4.1.2 Evaluation

We employ the same evaluation methodology as the original datasets for both DS-1000 and ClassEval.

DS-1000. We follow the original dataset using Pass@1 accuracy. This evaluation is conducted across total and perturbations: Origin, Surface, Semantic, and Diff-Rewrite.

ClassEval. We follow the original dataset using Pass@K metric. We calculate both class-leval and method-level Pass@K with $K = 1, 3, 5$.

4.1.3 Base LLMs

This work primarily utilizes the GPT (OpenAI, 2023) series as the LLM base model to validate the effectiveness of the framework. GPT-4 is utilized in *gpt-4-32k-0613* version, while GPT-3.5 is utilized in the *gpt-35-turbo-16k-0613* version. To further investigate the performance of CoCoST on both open-source and specialized code generation models, we have also employed WizardCoder (Luo et al., 2023) as a base model with *WizardCoder-Python-13B-V1.0* version.

4.1.4 Baselines

For the DS-1000, we selected four LLM-based frameworks as baselines: DocPrompt (Zhou et al., 2023), Self-Debugging (Chen et al., 2023), SELFEVOLVE (Jiang et al., 2023) and Reflexion (Shinn et al., 2023). DocPrompting enhances the LLM by employing a fine-tuned retriever to fetch problem-relevant documentation from offline document pools. Self-Debugging depends on a Python interpreter to instruct language models in revising Python code containing errors. SELFEVOLVE employs LLMs as both sources of knowledge and self-reflective programmers. Reflexion utilizes reflective feedback with generated tests and episodic memory to process task feedback. Details are shown in §B.3.

For the ClassEval, we select five LLM-based code generation models and frameworks as baselines: Instruct-CodeGen[3], SantaCoder (Allal et al., 2023), Instruct-StarCoder[4], WizardCoder (Luo et al., 2023) and Reflexion (Shinn et al., 2023).

4.2 Main Results

Regarding the DS-1000 dataset, the main results are shown in Table 1. CoCoST surpasses the current SOTA framework, SELFEVOLVE, by 10.9%, establishing itself as the new SOTA. Especially under the Diff-Rewrite perturbation setting, CoCoST

[3]https://huggingface.co/sahil2801/instruct-codegen-16B
[4]https://huggingface.co/GeorgiaTechResearchInstitute/starcoder-gpteacher-code-instruct

Table 2: Main Results and Ablation Study for ClassEval. All metric numbers are represented as percentages. For each metric, the **bold** number indicates the highest performance.

Method	Class-level			Method-level		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Instruct-StarCoder	10.2	12.7	14.0	23.1	26.5	27.7
SantaCoder	8.6	9.9	10.0	27.7	33.0	34.9
Instruct-CodGen	8.2	12.3	13.0	24.9	34.3	37.1
WizardCoder	12.2	20.0	23.0	35.2	47.1	51.1
Reflexion	24.1	30.7	35.2	43.4	51.6	61.8
CoCoST	**46.3**	**49.5**	**52.8**	**67.9**	**72.5**	**77.6**
w/o refinement of output	43.5	46.8	51.4	66.4	69.0	73.4
w/o refinement of error	46.2	**49.5**	51.7	**67.9**	**72.5**	77.2
w/o generation of test case	42.7	47.9	50.6	65.9	70.8	72.4
w/o online retrieval	37.2	42.5	44.9	60.4	65.7	69.8
w/o all (GPT-4 only)	36.2	39.3	43.5	58.6	64.9	67.3

exceeds SELFEVOLVE by 19.95%, which demonstrates the effectiveness of CoCoST in generating complex code. CoCoST employs online search and correctness testing to allow the model to imitate existing code patterns, thereby reducing the difficulty of generating new code, and refining the details to further enhance the correctness of the code.

For the ClassEval dataset, the results are shown in Table 2. Our experiments demonstrate that CoCoST has an overall higher performance on both class-level and method-level Pass@K evaluation. Specifically, CoCoST outperforms the Reflexion (best baseline model) significantly by an average of 19.5% and 20.4% on the Class and Method level.

4.3 Ablation Study

In this work, to validate the effectiveness of CoCoST, we conduct different ablation studies, with results presented in Table 1 and 2. Details on the ablation study are shown in §B.4.

CoCoST significantly enhances the base model's ability to generate complex code. Compared to the base model, CoCoST has shown improvements of 7.8% on the DS-1000 dataset and average 9.47% on ClassEval, demonstrating the effectiveness of the CoCoST.

Online search, generation of test cases, and serialization each contribute to the model's performance improvements. Compared to CoCoST, after performing ablation studies, these features showed a decrease in performance of 3.9%, 5.9%, and 1.3% respectively on the DS-1000 dataset. The online search improves the model by providing common code patterns, which reduces the difficulty of the model in generating initial code. Serialization, by converting inputs and outputs into a sequential format, allows the model to more intuitively observe inputs and outputs that are too lengthy or are in non-textual modalities, thereby strengthening its ability to solve complex code problems.

Online search outperforms offline retrieval in effectiveness and has a wider range of applicability. As shown in Table 1, using only online retrieval (the row w/o generation of test case) outperforms DocPrompting, which is an offline retrieval approach. Moreover, in real-world scenarios, as opposed to specific datasets, the types of problems encountered are more diverse. The scalability of online retrieval enables them to effectively address a wide range of problems. However, offline retrieval systems struggle to encompass all relevant information comprehensively.

During the refinement process, correctness testing is crucial, meaning that both the output result and error are equally important. After separately conducting ablation studies on the output result and error, CoCoST shows a decrease of 3.9% and 3.4% respectively on the DS-1000 dataset, and average 2.7% and 0.3% on the ClassEval dataset. This indicates that the output result contributes more to the refinement process than the error. However, in previous works, the output result is often overlooked, which should not be the case, especially in the generation of complex code. The evidence from the ablation study emphasizes the necessity of paying attention to the output results during the refinement phase to ensure the generation of high-quality, complex code.

4.4 Analysis of Different Base Models Performance

Table 3 shows the performance results of CoCoST on the DS-1000 dataset with different base models. We can see that GPT-4 has been comprehensively

Table 3: Different Base Models Results for DS-1000 and ClassEval. All metric numbers are represented as percentages. For each metric in each section, the **bold** number indicates the highest performance.

Method	DS-1000					ClassEval	
	Origin	Surface	Semantic	Diff-Rewrite	Total/Avg.	Class-level	Method-level
GPT-4	64.47	69.74	56.96	43.83	60.20	43.5	67.3
+ retrieve	66.23	71.05	59.57	45.68	62.10	50.6	72.4
+ refine	68.64	70.39	60.00	51.23	64.10	44.9	69.8
CoCoST	**71.71**	**74.34**	**66.96**	**53.09**	**68.00**	**52.8**	**77.6**
GPT-3.5	**57.02**	43.42	40.00	32.72	47.10	35.4	59.4
+ retrieve	47.15	25.00	36.96	25.31	37.90	41.9	61.7
+ refine	55.70	**50.66**	**44.35**	**35.80**	**49.10**	42.8	62.3
CoCoST	-	-	-	-	-	**45.8**	**64.7**
WizardCoder	**41.01**	**21.71**	**31.74**	**16.05**	**31.90**	**23.0**	**51.1**
+ retrieve	15.79	9.21	12.17	9.88	13.00	18.2	41.8
+ refine	39.69	21.71	30.00	15.43	30.80	22.3	50.7

improved with CoCoST, but the performance on GPT-3.5 and WizardCoder is mixed. This indicates that CoCoST requires the model to have the following capabilities to enhance its performance:

For code generation planning ability, the higher the complexity of the code that needs to be generated, the higher the demand for planning ability. Planning capability is key to online retrieval; only correct planning can generate appropriate queries to retrieve useful information. After incorporating online retrieval, GPT-3.5 has an increase of 4.75% on ClassEval, yet it decreased by 9.2% on DS-1000 as shown in Table 3. The challenge of ClassEval lies in how to generate the entire class and the interrelated functions, but the complexity of individual function codes is not as high as that of DS-1000. Therefore, the planning ability of GPT-3.5 can handle ClassEval, but it is inferior on DS-1000.

Code generation necessitates models to have in-context learning abilities. The generated code should be built on all the above provided contents, and the understanding of the preceding input prompt is of great importance in the refinement stage. In Table 3, it is observed that WizardCoder has a noticeable drop of 18.9% and 1.1% on the DS-1000 dataset when utilizing online retrieval and refinement respectively. And the overall performance of WizardCoder is comparatively interior to GPT models. This could be due to WizardCoder's limited in-context learning ability, especially with the complex and lengthy prompts, hindering accurate context comprehension and code modification.

4.5 Case Study and Error Analysis

For the case study on online retrieval, refer to Figure 4. It can be observed that by imitating the usage of functions found through online search, the model is better equipped to prepare the required parameters for the functions and to generate corresponding code. This significantly reduces the difficulty of generating complex code.

For the case study on correctness testing, refer to Figure 3. It is evident that, although the initially generated code did not show obvious errors, the output of the code did not align with the expected results. The model refines the code based on the output, thus improving hidden errors and generating the correct code.

Our framework consists of multiple components cascaded together, which results in certain intermediate steps that cannot be explicitly validated for effectiveness, as well as the potential generation of cascading errors. For the former, a discussion is provided in §C.2, while for the latter, an error analysis is conducted in §C.3.

5 Conclusion

In conclusion, CoCoST introduces a novel framework for generating complex code in real-world scenarios by emulating human coding processes like online searching and test case creation. It effectively overcomes challenges in code structure and logic, subtle bug detection, and handling of complex data. The framework's innovative use of online search, planning for query generation, correctness testing, and input-output serialization significantly improves code accuracy and model understanding. Tested on various datasets, CoCoST outperforms existing methods, demonstrating its efficacy in real-world code generation tasks.

Limitations

The main limitation of our research is that it has underlying issues of exceeding the allowed times of accesses due to multiple calls to the Google Search API. Similarly, we also have made multiple API calls to test and enhance the performance of the GPT models.

Ethics Policy

This research does not pose any ethical concerns. The datasets and other associated resources utilized in this study are publicly available and widely used in various other existing work.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh 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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *Preprint*, arXiv:2304.05128.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Xinyi He, Mengyu Zhou, Xinrun Xu, Xiaojun Ma, Rui Ding, Lun Du, Yan Gao, Ran Jia, Xu Chen, Shi Han, Zejian Yuan, and Dongmei Zhang. 2023. Text2analysis: A benchmark of table question answering with advanced data analysis and unclear queries. *Preprint*, arXiv:2312.13671.

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. *NeurIPS*.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Self-evolve: A code evolution framework via large language models. *Preprint*, arXiv:2306.02907.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *Preprint*, arXiv:2310.08992.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2022. Webgpt: Browser-assisted question-answering with human feedback. *Preprint*, arXiv:2112.09332.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam M. Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, Yaguang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Yanqi Zhou, Chung-Ching Chang, I. A. Krivokon, Willard James Rusch, Marc Pickett, Kathleen S. Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Hartz Søraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Díaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, V. O. Kuzmina, Joseph Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Rogers Croak, Ed Huai hsin Chi, and Quoc Le. 2022. Lamda: Language models for dialog applications. *ArXiv*, abs/2201.08239.

Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, and Thang Luong. 2023. Freshllms: Refreshing large language models with search engine augmentation. *Preprint*, arXiv:2310.03214.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations (ICLR)*, Kigali, Rwanda.

A Online Searching Detail

Website Base Station Distributions Table During the DS-1000 Online Retrieval Process:

Table 4: Website Base Station Distributions Table During the DS-1000 Online Retrieval Process.

Website	Proportion
https://stackoverflow.com	57.92%
https://numpy.org	8.59%
https://pandas.pydata.org	5.70%
https://www.geeksforgeeks.org	5.07%
https://docs.scipy.org	4.76%
https://matplotlib.org	3.20%
https://www.tensorflow.org	3.04%
http://scikit-learn.org	2.42%
The Others	9.29%

Moreover, CoCoST can be applied to specialized, proprietary, or domain-specific knowledge repositories as long as they are accessible via query. Moreover, implementing queries for private datasets is easily achievable and a growing trend in data management. Major companies such as Google and Microsoft already offer products designed to search private data; for example, Google Workspace's Cloud Search provides powerful capabilities for enterprises to search their private data. In this paper, to validate the effectiveness of our framework, we conducted tests on public online searches. Moving forward, the framework can be applied to an even broader range of knowledge repositories.

B Experiment

B.1 Datasets Detail

Further details of DS-1000 implementation are as follows:

• The dataset provides both Insertion and Completion style prompts, where the data is the same, differing only in prompt format, thus yielding similar results. In this paper, experiments are conducted with the Completion style prompt.

• We implement a filtering approach to prevent data leakage and model replication of existing solutions from Stack Overflow. The DS-1000 dataset originates from Stack Overflow, and concurrently, over 50% of the websites we encountered during our online searches are from Stack Overflow. Thus, to prevent data leakage, when conducting online searching, we filter out all Stack Overflow problems belonging to the source of the DS-1000 dataset by using the Stack Overflow question_id.

B.2 Base Models

The parameter details for each model in the experiment are as follows:

• GPT-4: model: *gpt-4-32k-0613*, temperature: 0, top_p: 0.95, max_tokens: 1024.

• GPT-3.5: model: *gpt-35-turbo-16k-0613*, temperature: 0, top_p: 0.95, max_tokens: 1024.

• WizardCoder: *WizardCoder-Python-13B-V1.0*, temperature: 0, top_p: 0.95, max_tokens: 1024.

B.3 Baselines Details

• DocPrompt (Zhou et al., 2023): DocPrompting enhances the LLM by employing a fine-tuned retriever to fetch problem-relevant documentation from offline document pools. The model then conditions on these documents, along with the problem description, to generate code.

• Self-Debugging (Chen et al., 2023): This approach depends on a SQL application or Python interpreter to instruct language models in revising SQL commands or Python code containing errors. For the sake of a fair comparison, we utilize its "simple" variant.

• SELFEVOLVE (Jiang et al., 2023): Employs LLMs as both sources of knowledge and self-reflective programmers. During the self-reflective process, it refines the code by addressing bugs.

• Reflexion (Shinn et al., 2023): Reflexion utilize reflective feedback with generated tests and episodic memory to process task feedback. For the sake of a fair comparison, we utilize GPT-4 as base model and set trail number = 1.

It is worth noting that the test cases involving the refinement process in the baselines mentioned above all use the test cases from the dataset designated for testing. However, within the context of the real-world scenario of CoCoST, test cases from the dataset should not be used within the framework. Without these test cases, they are entirely incapable of functioning.

B.4 Ablation Study Details

• Without refinement of output: During the refinement process, the output result is not refined; that is, refinement is conducted solely based on the error.

• Without refinement of error: During the refinement process, the error is not refined; that is, refinement is conducted solely based on the output result.

• Without serialization: During the refinement process, the input and output are not serialized; instead, their printout results are directly used as input.

• Without generation of test cases: Test cases are not generated. Since refinement cannot be performed without test cases, only online retrieval is conducted.

• Without online retrieval: Online retrieval is not performed, and the process is limited to refinement with correctness testing.

C Experimantal Results

C.1 DS-1000 Results

The main results for different packages in DS-1000 are shown in Table 5.

The results indicate that CoCoST shows a more pronounced effect on libraries whose inputs and outputs are more complex or more challenging for LLMs to intuitively understand, such as Matplotlib, TensorFlow, and PyTorch. On Sklearn, CoCoST experiences a slight decline due to its test cases containing complex objects, which present a significant challenge in generating test cases. Consequently, CoCoST's performance on Sklearn is not as strong as with the other libraries.

C.2 Analysis of Pipeline

Regarding the generation of test cases, to prove that the generated test cases are comparable to the ground truth test cases, we substitute the generated test cases in CoCoST with ground truth test cases and conduct experiment on the DS-1000 dataset using GPT-4. The results showed that the performance was 68.70%, only marginally higher by 0.7% compared to the use of generated test cases (68.00%). This proves that the effects of both are comparable and that the generated test cases rarely lead to errors.

Regarding generating plans, we attempt to bypass the planning step and directly perform online retrieval. On the DS-1000 dataset using GPT-4, this approach result in a performance of 55.70%, which is 6.4% lower than using planning for online retrieval (62.10%), and even 4.5% lower than only using GPT-4 to generate code (60.20%). This confirms that the generated plans are significantly effective.

C.3 Error Analysis

For cascading errors, some errors generated by retrieval processes can be corrected through refinement, while others may persist, necessitating future improvements. Take DS-1000 as an example: Compared to the baseline (GPT-4), we observed that among the instances that turned erroneous after retrieval, 39.6% were corrected, while 60.4% remained incorrect. For the former cases, it shows that CoCoST is able to fix some bad cases in the refinement stage even though the retrieval contents have some errors. For the later cases, we do observe some cases that are worthy of further research as potential directions for future work. First, the search content could be more detailed. E.g., Some basic steps that LLMs consider unnecessary to search for are not generating queries, but is exactly where the bug in the code. Second, the search query could be more targeted. The descriptions of some queries are not specific enough in terms of some complicated problems. Therefore, more sub-queries are needed to help model receive clearer instructions.

C.4 Case Study

Figure 4 demonstrates an example of online retrieval process in DS-1000 dataset. Given a problem description, the tested model firstly attempts to use the *tensor_scatter_nd_update* function from TensorFlow to achieve the solution, but encounters difficulty in selecting the appropriate class indices. After unsuccessful attempts, the model turns to an online search through a self-generated query and integrates the research results as part of the next prompt. Through the newfound knowledge online, the model re-implements a solution that efficiently generates the desired tensor representing the accuracy of each class. Overall, the process highlights how online retrieval can provide useful insights and potential solutions to complex challenges for LLMs' code generation.

Table 5: Table of Main Results for different packages in DS-1000. All metric numbers are represented as percentages. The **bold** number indicates the highest performance.

Method	Pandas	Numpy	Matplotlib	Tensorflow	Scipy	Sklearn	Pytorch	Total/Avg.
CoCoST	**59.45**	**75.91**	**75.48**	**71.11**	**61.32**	63.48	**77.94**	**68.00**
+ retrieve	51.89	70.91	68.39	66.67	52.83	**70.43**	60.29	62.10
+ refine	55.67	72.73	74.19	64.44	54.72	60.00	70.59	64.10
GPT-4 only	52.23	70.45	67.74	55.56	50.00	64.35	55.88	60.20

Problem Description

While nan == nan is always False, in many cases people want to treat them as equal, and this is enshrined in pandas.DataFrame.equals:
NaNs in the same location are considered equal.
......
Imagine I have a DataFrame which may contain some Nan:

```
   c0   c1   c2   c3   c4   c5   c6   c7   c8   c9
0  NaN  6.0  14.0  NaN  5.0  NaN  2.0  12.0  3.0  7.0
1  NaN  6.0  5.0  17.0  NaN  NaN  13.0  NaN  NaN  NaN
......
```

I just want to know which columns in row 0 and row 8 are different, please present them as pairs in a list. ...

Code before Correctness Testing Refinement

```
row0 = df.iloc[0]
row8 = df.iloc[8]
diff_cols = row0 != row8
result = list(zip(row0[diff_cols], row8[diff_cols]))
```

Output Result

[(nan, 18.0), (nan, nan), (nan, nan), (nan, 18.0), (17.0, 16.0), (16.0, nan), (nan, nan), (nan, nan), (nan, nan), (0.0, nan)]

Code after Correctness Testing Refinement

```
row0 = df.iloc[0]
row8 = df.iloc[8]
diff_cols = ~(row0.isnull() & row8.isnull()) & (row0 != row8)
result = list(zip(row0[diff_cols], row8[diff_cols]))
```

Output Result

[(nan, 18.0), (nan, 18.0), (17.0, 16.0), (16.0, nan), (0.0, nan)]

Figure 3: Case Study for Correctness Testing.

Problem Description

I am building a custom metric to measure the accuracy of one class in my multi-class dataset during training. I am having trouble selecting the class.
The targets are one hot. I have 10 classes in total, so I need a n*10 tensor as a result. Now I have a list of integers, how to get a tensor like:

```
[[0 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 0 1 1 1]
 ...
 [1 1 0 1 1 1 1 1 1 1]]
```

Code without Online Search

```
labels_tensor = tf.constant(labels)
ones_tensor = tf.ones_like(labels_tensor)
zeros_tensor = tf.zeros_like(labels_tensor)
result = tf.tensor_scatter_nd_update(ones_tensor, tf.reshape(labels_tensor, (-1, 1)), zeros_tensor)
```

Search Query

tensorflow tensor_scatter_nd_update usage

Search Results

This scatter operation would look like this:

```
>>> tensor = [[1, 1], [1, 1], [1, 1]]    # tf.rank(tensor) == 2
>>> indices = [[0, 1], [2, 0]]        # num_updates == 2, index_depth == 2
>>> updates = [5, 10]             # num_updates == 2
>>> print(tf.tensor_scatter_nd_update(tensor, indices, updates))
```

Code with Online Search

```
tensor = tf.ones((len(labels), 10), dtype=tf.int32)
indices = tf.constant([[i, label] for i, label in enumerate(labels)])
updates = tf.zeros(len(labels), dtype=tf.int32)
result = tf.tensor_scatter_nd_update(tensor, indices, updates)
```

Figure 4: Case Study for Online Retrieval.

D Prompts of CoCoST

Plan and Queries Generation Prompt
[System] Help me with the following problem, You need to write python code to solve the following problem. Please plan the steps you would need to take and write each step as a query. I can help you to search for relevant information online, if the query needs to be searchable, mark <search>. I can help you with google search through which you can search for real time information, python library document, error reporting information etc. Please return the queries that need to be searched in google. + First, [PLAN] plan the steps you would need to take and write each step as a query. Then, [SEARCH] list the query from [PLAN] that need to search. + You only need to plan that can complete the code snippet. You do not need to plan the codes before BEGIN SOLUTION block. + You can search for real-time information, python library documents, error messages, common usage, and other information. + Don't return duplicate query with similar semantics, return different queries. + Don't tag to search simple query that can be solved by yourself, return the most critical queries. [Example] ... [User] <problem description>
Predict
[PLAN] 1. ... ... [SEARCH] 1. No need to search. / <search> ... </search> ...

Figure 5: Plan and Queries Generation Prompt on DS-1000.

Plan and Queries Generation Prompt
[System] Help me with the following problem, You need to write python code to solve the following problem. Please plan the steps you would need to take and write each step as a query. I can help you to search for relevant information online, if the query needs to be searchable, mark <search>. I can help you with google search through which you can search for real time information, python library document, error reporting information etc. Please return the queries that need to be searched in google. + First, [PLAN] plan the steps you would need to take and write each step as a query. Then, [SEARCH] list the query from [PLAN] that need to search. + You only need to plan that can complete the code snippet. You do not need to plan the codes before BEGIN SOLUTION block. + You can search for real-time information, python library documents, error messages, common usage, and other information. + Don't return duplicate query with similar semantics, return different queries. + Don't tag to search simple query that can be solved by yourself, return the most critical queries. For each problem given, there will be a class with several functions inside you need to write subsequenct code. Please follow the rules below when you [PLAN] and [SEARCH]: + Do not PLAN and SEARCH the function with name: __init__(self), this function has been initialized for you as the setting of the class. + For each function in the class you need to implement, only SEARCH the query that you are unsure of the implementation. + For each function in the class you need to implement, you must limit the search up to 3 queries. [Example] ... [User] <problem description>
Predict
[PLAN] 1. Function: ... 1.1 ... [SEARCH] 1. Function: ... 1.1 No need to search. / <search> ... </search> ...

Figure 6: Plan and Queries Generation Prompt on ClassEval.

Online Retrieval Code Generation Prompt

[System]
You need to help me write code based on the PROBLEM as follows. Previously had a round of conversation about this problem, you made a PLAN of it and came up with a QUERY that needs to be searched. I've searched for the background information you might need. You can selectively refer to it when writing your code.

There are some rules that you must follow for writing the codes:
+ You only need to output codes that can complete the code snippet. You do not need to output the codes before the [insert] block.
+ Return the codes directly, if you want to add some explanation, please add them to the comments.
+ The execution result of the code must meet the requirements, including result formatting, etc. If the result is a table, it is also necessary to note that the header must be the same as the requirements, and the format of the table values must meet the requirements.
+ Background knowledge is for reference only and not all of the information you need to use in your code., please focus on code completion.
[Example]
...
[User]
<problem description>

Here's the plan you made earlier and the query to search for:
<plan and queries>

I've searched for the background information you might need. You can selectively refer to it when writing your code, noting that not all of the information you need to use in your code. The following information is the markdown text of the main information on the corresponding website.
<retrieve information>

Again, the PROBLEM is as follows:
<problem description>
Please generate codes in [insert] block following the format rules, and should !!!not!!! generate the code before the [insert] block.

Predict

```python
...
```

Figure 7: Online Retrieval Code Generation Prompt on DS-1000 and ClassEval.

Generation of Test Case Prompt
[System] I will give you a description of a PROBLEM which needs to be solved by generating code. I need test case (input for code) for testing if the generated code is correct. Generate up to 3 test cases for me. There are some rules that you need to follow: + If there is not input for the code, you should not generate test case and should not return any ```python. + If the input is fixed or it's not appropriate to have several different inputs, you can just generate one test case. + If the input has more than one variable, then the test case needs to contain all the variables. + Please keep the variable names the same as in the question. + If the input variable is a example for a function, you should retain variable names without "example". + You should return all variables or functions before "BEGIN SOLUTION", and make sure the variables or functions can directly be executed. E.g., you should not return the definition of load_data() function without using it, you should not load csv from local file, etc. [Example] ... [User] <problem description>
Predict
Test case1: ```python ... ``` Test case2: ...

Figure 8: Generation of Test Case Prompt on DS-1000.

Generation of Test Case Prompt

[System]
You are a Python Expert. Provided below is a problem of Python class skeleton with several functions inside empty. You will help me to generate test cases for the several empty functions in the class.
For each function you need to generate test cases, it will give you a instruction as the function comments. The instruction contains inforamtion:
1. The short problem description of the function
2. The input parameters' name, type, and its description of the function in order staring with ':param'
3. The return type of the function starting with ':return'
4. The example of the function usage starting with '>>>'
5. The result for the example of the function usage shown at the last line of the instruction.

Your response must follow the following rules:
+ Please keep the variable names the same as in the question.
+ For each function you need to write test cases, your response code MUST follow the format of: ```python \n <code> \n ```
+ You MUST generate test cases for any of the functions taking place in the given class except the constructor function "__init__" in the class.
+ You MUST generate three test cases for each function with the instruction comment, and MUST follow the format below, '##' is the separator of each test case:

```python
# <function_name>
##
# Test Case 1
<Test Case 1 code>
##
# Test Case 2
<Test Case 2 code>
##
# Test Case 3
<Test Case 3 code>
```

+ For each test case code above, first follow excatly the format of the example of the function usage in the instruction comment starting with '>>>', then assign the variable 'result' to the output of your tested function following the format: result = <code of the tested function result>.
[Example]
...
[User]
<problem description>

Predict

```
# <function_name>
   ##
   # Test Case 1
...
```

Figure 9: Generation of Test Case Prompt on ClassEval.

Refinement with Correctness Testing Code Generation Prompt

[System]
Help me rewrite the code. I will provide the PROBLEM description, the code for this PROBLEM, and the execution result of this code. Help me rewrite it into the correct code to solve this PROBLEM.

There are some rules that you must follow for rewriting the code:
+ Is the code execution result the right answer to the PROBLEM? If not, please rewrite the code, if yes, please do not return any code.
+ If you need to rewrite the code, you need to follow these format rules:
   + You need to first explain why the original code is incorrect in the comment.
   + You should directly answer the code in [insert] block, and should not generate the code before the [insert] block.
   + You should answer only one code snippet, not more than one.
   + You should directly answer the correct code, and don't offer the other possibilities.
   + You should output the code as the same format as the examples.
+ If you do not need to rewrite the code, return the original code in [insert] block.
[Example]
...
[User]
<problem description>

Here is a code snippet that may contain errors in solving the above PROBLEM:
<initial code>

Above is the code that GPT4 generated for me, here are the inputs as well as the execution results. You need to determine if the code is correct and suggest changes if it is not.
<serialized output or error>

I've searched for the background information you might need. You can selectively refer to it when writing your code, noting that not all of the information you need to use in your code. The following information is the markdown text of the main information on the corresponding website.
<retrieve information>

Again, the PROBLEM is as follows:
<problem description>
Please generate codes in [insert] block following the format rules, and should !!!not!!! generate the code before the [insert] block.

Predict

```python
...
```

Figure 10: Refinement with Correctness Testing Code Generation Prompt on DS-1000 and ClassEval.