Two Steps to Precision: Enhancing Reliable API Invocation in Code Generation

Anonymous ACL submission

Abstract

Automatic code generation is crucial in modern software development, yet large language models struggle with real-world challenges like code versioning and multi-API invocation. Existing approaches, including direct generation and retrieval-augmented methods, often fail to ensure precise API usage. This paper introduces a simple yet effective two-step framework: rough code generation or retrieval followed by fine code editing. Experiments on VersiCode and BigCodeBench show significant performance gains in version-specific code completion and function-level programming. These results demonstrate the framework's practicality in enhancing LLM-based code generation systems.

1 Introduction and Related Work

As software development becomes increasingly complex, automatic code generation has emerged as a critical research area in software engineering.In recent years, LLMs have shown impressive performance in benchmarks like HumanEval(Chen et al., 2021), generating code from natural language instructions. However, real-world tasks, involving code versioning and multi-API invocation, pose greater challenges. Direct LLM-based generation often fails to meet production-level requirements, revealing limitations in traditional approaches.

Existing research falls into two categories: LLMs relying solely on internal knowledge and those augmented with retrieval. Early efforts, such as Codex (Chen et al., 2021) and GPT-3 (Brown et al., 2020), focused on direct code generation, while later models like RECODE (Anugrah Hayati et al., 2018) and REDCODER (Lewis et al., 2020) leveraged retrieval-based techniques. However, both approaches also face challenges in ensuring accurate API calls for specific library versions(Wu et al., 2024) and multi-API invocation(Zhuo et al., 2024). Beyond these approaches, structured reasoning has shown potential: Chain of Thought prompting (Wei et al., 2022) improves performance by breaking tasks into steps, while ReAct (Yao et al., 2023) refines outputs through API or database queries. This raises a fundamental question: *Can increasing inference steps improve API invocation accuracy in code generation? What concrete benefits could this provide, such as mitigating specific failure cases, and what inherent limitations might still require alternative strategies?*

In this paper, we propose a simple yet effective two-step reasoning framework to explore the above questions, which adapts based on the availability of external knowledge. When external knowledge is not available, the framework first generates a rough code structure, which is then refined in the second step to meet specific coding standards. When external knowledge is available, the process begins with retrieving a rough code template from an external knowledge base, followed by knowledge-enhanced fine-grained code generation. Despite its simplicity, our results demonstrate that this framework significantly improves API usage and overall code quality. We summarize our main contributions:

- We investigate the two-step reasoning framework for API generation using only internal model knowledge, providing insights into LLMs' reasoning behavior per inference.
- We extend this framework to retrieval-augmented generation, integrating rough code retrieval to enhance API accuracy when external knowledge is available.
- Experiments on VersiCode and BigCodeBench demonstrate improved API invocation precision and version control, offering guidance on optimizing reasoning steps and encouraging further exploration in multi-agent AI for software engineering.

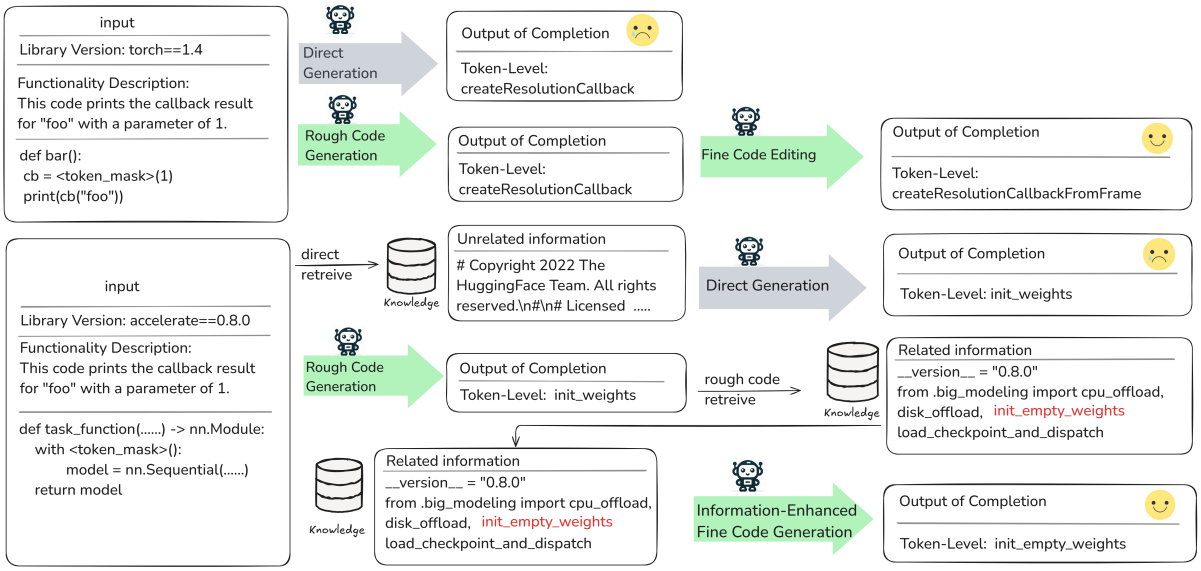

Figure 1: The figure illustrates the two-step framework. The upper section, **Code Generation without External Information**, involves two steps: (1) Rough Code Generation (C_{rough}), and (2) Fine Code Editing (C_{fine}). The lower section, **Code Generation with External Information**, introduces **Rough Code Retrieval**, fetching relevant knowledge fragments (V) via similarity search, followed by **Information-Enhanced Fine Code Generation**, which refines the code with retrieved knowledge for improved quality and context-awareness.

2 Two Steps to Precision

In this section, we elaborate on how we decompose the one-step reasoning framework used for direct code generation into a two-step reasoning framework. This approach applies to two scenarios: (1) the model generates code using only its memory and (2) it incorporates external information. Figure 1 illustrates the detailed flow in both cases.

The model is formulated as an end-to-end function $f : X \mapsto Y$, where the input space X consists of three primary components: *Missing code fragments* C_m, representing incomplete code snippets with placeholders for API calls that require completion; *Library requirements* R_l, specifying the necessary software libraries from which appropriate APIs should be selected to fulfill the task; and *Version constraints* V_c, ensuring compatibility with specific API versions.

2.1 Code Generation without External Knowledge

We decompose the process into two steps: rough code generation and fine code editing.

Rough Code Generation. The first step generates an initial code draft with API calls. Given the missing code fragments C_m and library requirements R_l, the model produces a rough code snippet $C_r = f(C_m, R_l)$. The rough code C_r produced in this step meets structural requirements but may contain API errors, version mismatches, or other inconsistencies.

Fine Code Editing. The second step refines the rough code to ensure correct API usage and version compliance. Given the rough code C_r, library requirements R_l, and version constraints V_c, the model generates the final output $C_f = f(C_r, R_l, V_c)$. The fine-tuned code C_f incorporates the necessary modifications, improving API correctness and compatibility with deployment standards.

2.2 Code Generation with External Knowledge

In retrieval-augmented generation (RAG), natural language queries often misalign with source code, leading to retrieved information that negatively impacts performance. We try to solve this problem through two-step reasoning. The process is also divided into two distinct steps: rough code retrieval and knowledge-enhanced fine code generation.

Rough Code Retrieval. To address alignment issues, we retrieve code snippets from an external knowledge base using a code-based query instead of relying on natural language. The process starts by generating rough code C_{r}, which is used as a query to retrieve relevant knowledge fragments via similarity search (Lewis et al., 2020). The retrieval

Model	token_level				line_level				block_level			
	PASS@1		CDC@1		PASS@1		CDC@1		PASS@1		CDC@1	
	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step
DeepSeek-7B	23.24	27.59	28.57	33.33	21.57	23.39	21.29	25.77	0	0.14	0	0.14
CodeGemma-7B	25.07	31.09	32.21	39.50	12.75	16.95	15.13	22.27	0	0	0.14	0.56
Deepseek-coder	54.62	55.46	71.43	72.27	24.37	42.02	21.85	37.82	2.52	15.97	4.20	23.53
llama-3-70b	46.08	47.49	54.06	56.72	30.95	31.65	28.43	29.41	14.15	14.29	22.55	22.69
Qwen2-72B	49.15	52.24	60.78	62.60	32.77	40.33	31.09	34.31	16.8	18.20	28.29	29.41
GPT-4o	60.36	62.04	75.21	77.17	28.29	42.85	25.91	42.30	16.66	24.23	22.83	29.55

Table 1: The Pass@1 and CDC@1 results of different LLMs without knowledge base support on VersiCode.

Model	token_level				line_level				block_level			
	PASS@1		CDC@1		PASS@1		CDC@1		PASS@1		CDC@1	
	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step
DeepSeek-7B	6.02	15.27	6.02	16.80	5.74	7.14	6.02	9.10	0	0.14	0	0
CodeGemma-7B	3.22	10.22	3.22	13.03	3.08	5.32	2.80	6.44	0	0	0	0
Deepseek-coder	39.50	40.34	48.74	48.74	23.53	24.37	18.49	21.85	1.68	1.68	1.68	1.68
llama-3-70b	14.84	22.68	14.01	22.55	14.99	17.65	13.87	15.69	4.20	5.46	7.70	8.60
Qwen2-72B	23.94	24.65	23.80	28.01	30.67	32.07	28.29	32.91	7.56	11.34	17.51	18.35
GPT-4o	43.13	44.95	47.47	52.38	37.11	37.81	37.25	38.51	9.66	9.94	9.66	12.46

Table 2: The Pass@1 and CDC@1 results of different LLMs with knowledge support on VersiCode.

Model	PASS@1	
	one-step	two-step
DeepSeek-7B	37.50	39.33
Llama-3-70b	45.71	51.65
Qwen-72B	52.64	53.96
DeepSeek-coder	58.40	59.40
GPT-4o	68.30	69.93

Table 3: The Pass@1 results of different LLMs on BigCodeBench.

process can be expressed as:

$$V = \arg\max_{v_i} \text{Sim}(C_{\text{r}}, k_i), \tag{1}$$

where k_i represents the code fragments in the knowledge base, and V denotes the most relevant retrieved fragment based on the similarity measure.

Knowledge-Enhanced Fine Code Generation. The second step refines the rough code by integrating external knowledge. Given the rough code C_r, library requirements R_l, version constraints V_c, and the retrieved knowledge fragment V, the model generates a fine-tuned code snippet, i.e., $C_f = f(C_r, R_l, V_c, V)$. The fine-tuned code C_f leverages external information to improve API correctness and ensure better compatibility with production-level requirements.

In both scenarios, the two-step framework generates rough code and refines it to meet more granular criteria. In the first scenario, the model relies solely on its internal memory. In the second, external knowledge is retrieved and incorporated to improve the handling of complex dependencies. See Appendix A for details on the prompts we use.

3 Experiment

We perform extensive experiments to answer three key research questions: ***RQ1***: What makes the two-step framework effective for LLM-based code generation without external knowledge, and how does it compare to single-step approaches? ***RQ2***: How does the two-step framework address misalignment in RAG, and when does it fall short? ***RQ3***: How well does two-step inference improve function-level code generation with multiple API calls, considering efficiency, correctness, and robustness?

3.1 Experimental Setup

Evaluation Datasets: We conduct experiments on two different types of code generation tasks, namely VersiCode (Wu et al., 2024) and BigCodeBench (Zhuo et al., 2024).

Knowledge Base: We build a structured knowledge base for retrieval-augmented code generation. Using automated scripts, we crawl Python libraries from PyPI and GitHub, extract source code across versions, and store metadata including library name, version, file paths, and code in JSON format for efficient retrieval. Each entry contains multiple API definitions, function signatures, and usage patterns specific to each library version.

Evaluation Metrics: We use *Pass@k(Chen et al., 2021)* and *Critical Diff Check (CDC@k)(Wu et al., 2024)* as evaluation metrics over task granularities.

Baseline: We compare two-step reasoning with one-step generation in scenarios without external

knowledge. For retrieval-based generation, we evaluate rough code retrieval against traditional natural language-based retrieval.

See Appendix C for detailed experimental setup.

3.2 Results and Analysis

API invocation and versioning are critical in software development, yet LLMs struggle with versioning and concurrent API calls. Our experiments (Table 1) show that even GPT-4, the best performer, scored only 60.36 Pass@1 on token-level tasks, dropping by 32.07 and 43.7 points at line and block levels, highlighting these challenges.

To Answer RQ1: The two-step reasoning framework significantly improves LLM performance on VersiCode. As shown in Table 1, CodeGemma-7B achieved a 6.02 increase in Pass@1 and 7.29 in CDC@1, with other models improving by 0.84 → 3.45 (Pass@1) and 0.84 → 4.76 (CDC@1) at the token level. For more complex tasks, the benefits were even greater. DeepSeek-Coder saw a 17.65 increase in Pass@1 and 15.97 in CDC@1 at the line level, with other models improving by 0.7 → 15.6 (Pass@1) and 0.98 → 16.39 (CDC@1). At the block level, DeepSeek-Coder and GPT-4 showed the largest gains (13.45 and 7.57 in Pass@1, 19.33 and 6.72 in CDC@1), while smaller models had minimal impact due to lower baselines. *The framework enhances accuracy at all levels, with stronger models benefiting more in high-level tasks.*

To Answer RQ2: Table 2 shows that natural language-based retrieval significantly hinders performance, introducing irrelevant context. For example, GPT-4o's Pass@1 drops to 47.47 with retrieval, compared to 75.21 for direct generation. The two-step framework effectively mitigates this misalignment, improving model performance across all levels. DeepSeek-7B gains +9.25 Pass@1 and +10.78 CDC@1 at the token level, while Qwen sees a +1.4 Pass@1 and +4.62 CDC@1 boost at the line level. Block-level gains are smaller, reflecting the increased difficulty of refining complete code. *By aligning retrieved information with task requirements, two-step reasoning enhances retrieval-augmented generation, ensuring more accurate and contextually relevant API usage.*

To Answer RQ3: We evaluate the two-step reasoning framework for function-level code generation on BigCodeBench. Table 3 shows that two-step reasoning consistently improves function-level code generation across all models. Correctness is significantly enhanced, with models like GPT-4o (68.30 → 69.93) and Llama-3-70B (45.71 → 51.65) benefiting the most. The improvements are robust across varying API complexities, ensuring better API sequencing and parameter usage. Larger and better-performing models gain more from two-step inference, indicating a trade-off between efficiency and effectiveness. *While additional refinement increases computational cost, it ensures more reliable and adaptable API usage, making it a practical approach for complex function generation.*

3.3 Discussion

While the two-step framework significantly improves API invocation accuracy, it has limitations. First, errors in the initial rough code (e.g., structural mismatches) may propagate to refinement, particularly in block-level tasks where gains were smaller (Table 1, line/block-level CDC@1). Second, performance depends on well-structured API knowledge—unstructured retrieval introduces noise, as seen in RAG misalignment (Table 2: GPT-4 Pass@1 dropped to 47.47 vs. 75.21 in direct generation). *Integrating version-specific knowledge graphs (modeling API evolution) or pretraining on API documentation may mitigate the limitations.*

The framework's simplicity positions it as a meta-structure for multi-agent systems. For example, specialized agents could handle retrieval, generation, and refinement, enabling scalable handling of complex dependencies (e.g., cross-library integration). This aligns with findings in BigCodeBench (Table 3: two-step improved Pass@1 by up to 5.94), suggesting adaptability for collaborative, domain-specific workflows. *Future work could explore hierarchical agent architectures for real-time API evolution and error recovery.*

4 Conclusion

We introduced a two-step reasoning framework that enhances API invocation accuracy and version control by separating rough code generation (or retrieval) from fine code editing. Experiments on VersiCode and BigCodeBench confirm its effectiveness, particularly benefiting larger models. While the framework improves correctness, limitations remain—structural errors can propagate, and unstructured retrieval may introduce noise. Future work should integrate version-aware knowledge graphs and explore hierarchical multi-agent architectures for dynamic retrieval, generation, and refinement.

5 Limitation

While our two-step reasoning framework improves API invocation accuracy, it has several limitations.

First, the framework is constrained by data limitations, as its evaluation relies on benchmarks like VersiCode and BigCodeBench, which primarily assess block-level code generation. However, real-world software development involves project-level tasks with cross-file dependencies and long-range API interactions, which these benchmarks do not fully capture. To better assess industrial applicability, future work should explore more challenging datasets featuring larger-scale, multi-file, and cross-library programming tasks.

Second, while the two-step reasoning approach improves API accuracy, its gains remain moderate. It mitigates some errors but does not fully resolve issues in API selection, version mismatches, or complex multi-API interactions. Further improvements require integrating richer external knowledge, such as API knowledge graphs, dynamic code retrieval, execution-based verification, and multimodal data, to enhance code refinement and correctness validation.

References

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. *CoRR*, abs/2306.10763.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 925–930. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

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

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, et al. 2024. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409.

Dejian Yang Zhenda Xie Kai Dong Wentao Zhang Guanting Chen Xiao Bi Y. Wu Y.K. Li Fuli Luo Yingfei Xiong Wenfeng Liang Daya Guo, Qihao Zhu. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, arXiv:2401.14196.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, and others. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. 2024. Versicode: Towards version-controllable code generation. *CoRR*, abs/2406.07411.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, , et al. 2024. Qwen2 technical report. *CoRR*, abs/2407.10671.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *CoRR*, abs/2406.15877.

A Prompt

We introduce prompt template for VersiCode and Bigcodebench which show in figure2,3,4,5,6 respectively.

B Similarity index experimental results

We used three other metrics to measure the similarity between the generated code and the groundtruth, respectively *ISM@K, PM@K, EM@K*.

Identifier Sequence Match (ISM@k) and *Prefix Match (PM@k)(Agrawal et al., 2023)* (for low-level generation): These two metrics measure how well the generated sequence matches the ground truth. In block-level generation, we calculate the average performance of each row, and each instance generates $n = 6$ independent samples. *Exact Match (EM@k)*: Regular expression matching is used to determine whether the generated code uses the specified API. The calculation method of EM@k is the same as Pass@k ($n = 6$, $k = 1$).

The detailed results can be seen in Table 4,5

C Setups

Evaluation Metrics: VersiCode evaluates large language models on version-specific code completion (VSCC), focusing on the dynamic nature of library updates and the ability to handle changing API requirements, and BigCodeBench evaluates LLMs on complex programming tasks requiring multiple function invocations, including various domains like data analysis and web development, with multiple test cases for each task.

Evaluation Metrics:*Pass@k(Chen et al., 2021)*: Evaluates model performance by generating $n \geq k$ correct samples ($n = 6$, $k = 1$) through executable testing. *Critical Diff Check (CDC@k)(Wu et al., 2024)*: Focuses on the difference between generated code and the reference answer, unlike traditional code similarity.

Baseline: We compare two-step reasoning with one-step generation in scenarios without external knowledge. For retrieval-based generation, we evaluate rough code retrieval against traditional natural language-based retrieval. Experiments are conducted on DeepSeek-Coder (Daya Guo, 2024), CodeGemma (CodeGemma Team et al., 2024), Llama-3 (Dubey et al., 2024), Qwen (Yang et al., 2024), and GPT-4o to assess performance across diverse model architectures.

You are a Python programming expert. Your task is to analyze a code snippet and infer the content masked by <mask>. Here are your instructions:
1. You will receive:
  - A Python library name and its version, which is relevant to the content masked by <mask>
  - A code snippet with one or more <mask> markers
2. Each <mask> in the snippet represents the same masked content.
3. Based on the provided library and its version, infer the specific token that <mask> is hiding.
4. Provide your response as follows:
  - Give only one answer, regardless of how many <mask> appear
  - Include only the inferred content
  - Wrap your answer with ```python and```to denote it as a code block
  - Omit any explanations or extra information
The Python library with its version and the code snippet are provided below:
Library and Version:
{task_description}
Code Snippet:
{masked_code}
Your response:

Figure 2: Rough Code Generation Prompt for Versicode

You are a Python programming expert. Your task is to analyze a code snippet and infer the content masked by <token_mask>. Here are your instructions:
1. You will receive:
  - A Python library name and its version, which is relevant to the content masked by <token_mask>
  - A code snippet with one or more <token_mask> markers
2. Each <token_mask> in the snippet represents the same masked content.
3. Based on the provided library and its version, infer the specific token that <token_mask> is hiding.
4. Provide your response as follows:
  - Give only one answer, regardless of how many <token_mask> appear
  - Include only the inferred content
  - Wrap your answer with ```python and ``` to denote it as a code block
  - Omit any explanations or extra information
The Python library with its version and the code snippet are provided below:
Library and Version:
{dependency_version}
Code Snippet:
{masked_code}
In the first stage of reasoning, we have generated some possible answers as follows:
{first_step_answer}
Please substitute the answer generated in the first step into the code snippet and determine whether it is correct. If it is wrong, please regenerate it. If it is correct, please keep the output unchanged.
Your response:

Figure 3: Fine Code Editing Prompt for Versicode

You are a professional Python engineer. Your task is to write Python code that implements a specific function based on the provided library and version. Here are your instructions:
1. You will receive:
  - The name and version of the library relevant to the code
  - A code snippet with a <type_mask> where you need to infer the missing code
2. Based on the library information, write the Python code that fills the <type_mask> and implements the feature.
3. Provide your response as follows:
  - Return only the code that fills the <type_mask> and implements the function
  - Enclose your code with ```python and ``` to denote it as a Python code block
  - Omit any explanations or extra information
The library information and partially masked code snippet are provided below:
Library and Version:
{dependency_version}
Code Snippet:
{masked_code}
Here I found the most relevant source code of the same version dependency, I hope that will be helpful:
{source_code}
Your Response:

Figure 4: Information-Enhanced Fine Code Generation Prompt for Versicode

Model	token_level		line_level						block_level					
	EM@1		EM@1		ISM@1		PM@1		EM@1		ISM@1		PM@1	
	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step
DeepSeek-7B	28.57	33.33	56.86	66.17	44.31	49.80	35.89	37.10	23.67	40.56	23.60	38.85	15.04	20.25
CodeGemma-7B	32.21	39.50	46.08	54.20	34.19	42.17	23.05	30.06	8.54	29.41	8.48	29.08	5.12	14.58
Deepseek-chat	71.43	72.27	77.31	78.15	39.49	57.98	45.35	59.04	41.18	66.39	13.44	63.02	20.18	48.81
llama-3-70b	54.06	56.72	65.97	66.39	50.47	50.90	46.29	46.67	61.06	61.34	61.01	61.34	36.10	36.35
Qwen2-72B	60.78	62.60	71.28	71.42	52.80	52.94	52.01	55.53	64.28	65.68	63.66	64.26	42.96	43.65
GPT-4o	75.21	77.17	77.17	78.85	41.87	58.68	46.58	58.72	67.92	68.20	55.88	66.80	48.90	50.59

Table 4: The EM@1, ISM@1 and PM@1 results of different LLMs without knowledge base support on VersiCode.

Model	token_level		line_level						block_level					
	EM@1		EM@1		ISM@1		PM@1		EM@1		ISM@1		PM@1	
	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step	one-step	two-step
DeepSeek-7B	6.02	16.80	15.55	26.61	11.06	16.30	8.45	12.85	6.16	6.16	6.16	6.16	2.73	2.73
CodeGemma-7B	3.22	13.03	9.66	19.47	7.35	11.70	4.68	7.74	7.98	17.93	7.93	17.80	4.55	9.70
Deepseek-chat	48.74	48.74	69.75	72.27	31.09	35.30	39.37	41.70	35.29	36.97	10.92	13.45	18.44	19.06
llama-3-70b	14.84	22.68	64.99	66.53	31.37	31.65	33.65	34.26	39.78	40.19	12.26	13.16	17.32	19.80
Qwen2-72B	23.80	28.01	72.12	75.07	51.94	54.03	49.12	52.87	54.06	55.32	42.50	47.55	27.42	30.97
GPT-4o	47.47	52.38	76.61	79.55	56.86	58.90	54.27	57.11	65.54	66.25	52.60	52.94	46.69	46.83

Table 5: The EM@1, ISM@1 and PM@1 results of different LLMs with knowledge base support on VersiCode.

You are a professional Python engineer. Your task is to write Python code that implements a specific function based on the provided task description and requirements. Here are your instructions:

1. You will receive:
   - A detailed description of the function to be implemented
   - A list of required libraries or packages to be used in the implementation
   - Information about the expected input arguments and return type
   - An example output to validate the implementation
2. Based on the provided information:
   - Write the Python code that satisfies the task description and requirements.
   - Ensure that the code includes proper imports for the specified libraries.
   - Adhere to the specified input and output formats.
3. Provide your response as follows:
   - Return the complete Python function implementation.
   - Enclose your code with ```python and ``` to denote it as a Python code block.
   - Omit any explanations or extra information.

The task description, requirements, and examples are provided below:

Task Description:
{task_description}
Requirements:
{requirements}
Example:
{example}
Your response:

Figure 5: Rough Code Generation Prompt for Bigcodebench

You are a professional Python engineer. Your task is to write Python code that implements a specific function based on the provided description and library requirements. Here are your instructions:
1. You will receive:
  - A detailed description of the task to be implemented
  - Some code references which may contain errors
  - The required libraries or packages for the task
2. Based on the task description and the provided information:
  - Ensure the correctness of the generated Python code.
  - Pay special attention to the following:
    * Whether the required libraries or packages are imported correctly.
    * Whether the return value type matches the task requirements.
    * Whether the libraries or packages are used correctly in the implementation.
3. Provide your response as follows:
  - Return the correct Python implementation for the described task.
  - Enclose your code with ```python and ``` to denote it as a Python code block.
  - Omit any explanations or extra information.
The task description, code references, and library requirements are provided below:
Task Description:
{task_description}
Code References:
{code_references}
Library Requirements:
{library_requirements}
Your response:

Figure 6: Fine Code Editing Prompt for Bigcodebench