MetaCoder: Generating Code from Multiple Perspectives

Xin Chen*
Zhijie Jiang*
chenxin19@nudt.edu.cn
jiangzhijie@nudt.edu.cn
National University of Defense Technology
Changsha, China

Shanshan Li†
Yong Guo†
shanshanli@nudt.edu.cn
yguo@nudt.edu.cn
National University of Defense Technology
Changsha, China

Zhouyang Jia
National University of Defense Technology
Changsha, China
jiazhouyang@nudt.edu.cn

Si Zheng
National University of Defense Technology
Changsha, China
zhengsi@qiyuanlab

Yuanliang Zhang
National University of Defense Technology
Changsha, China
zhangyuanliang13@nudt.edu.cn

Abstract

Large Language Models (LLMs) have already demonstrated excellent performance in code generation tasks. However, their proficiency varies considerably among different programming languages, performing well in languages like Python, but struggling with languages such as C++ and Java. This discrepancy limits their utility in scenarios requiring multi-language support. Existing methods aimed at enhancing the code generation capabilities of LLMs typically emphasize general performance improvements while overlooking discrepancies between languages, resulting in suboptimal outcomes for less proficient languages.

To address this challenge, we propose MetaCoder. Given a task description, MetaCoder first generates code in high-proficiency language, and then summarizes the code. Finally, MetaCoder generates target code using the task description, generated code, and summary. Additionally, MetaCoder detects and corrects syntax errors in the target code. We evaluate MetaCoder on HumanEval-x, and compared with Zero-Shot, the Pass@1 in generating C++ and Java code has improved by up to 13.09% and 16.98%, respectively.

CCS Concepts

• **Software and its engineering → Software notations and tools**.

Keywords

Code Generation, Large Language Model, Multi-language

*Co-first author
†Corresponding author

Internetware '25, Trondheim, Norway

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Xin Chen, Zhijie Jiang, Shanshan Li, Yong Guo, Zhouyang Jia, Si Zheng, and Yuanliang Zhang. 2025. MetaCoder: Generating Code from Multiple Perspectives. In *Proceedings of 16th Asia-Pacific Symposium on Internetware (Internetware '25).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Large Language Models (LLMs) have achieved excellent performance in code generation tasks and have been widely used in applications such as Copilot[3] and Cursor[4]. With continuous improvements in their capabilities, existing LLMs have achieved high accuracy in code generation. For example, the accuracy of DeepSeek Coder V2[45] and Llama 3.1[19] on the HumanEval [15] benchmark reached 89%. However, it is still a challenge to use LLMs to generate entirely correct code for complex requirements[17, 18].

Many methods have been proposed to improve LLM-based code generation. Some approaches[12, 25] fine-tune LLMs to generate correct code, such as OctoPack[31] and MultiPL-T[11]. However, these methods require a significant amount of resources. Other methods do not require fine-tuning the model, such as CoT[40] and SCoT[26], CoT prompts LLMs to generate a natural language reasoning process first, followed by the corresponding code. Some methods[18, 35] enables LLMs to adopt different roles, improving code generation by analyzing task requirements and checking code. To alleviate the Degeneration-of-Thought problem[27, 36] in code repair[17, 29, 33, 43], methods such as INTERVENOR[39] are proposed.

Recent studies[14, 19, 45] have found that the LLM-generated code exhibits varying accuracy across different programming languages. On the HumanEval and other benchmarks[9, 44], Llama 3.1 and DeepSeek Coder V2 show such differences. The accuracy of Llama 3.1 405B in generating Python code is 7% higher than in generating C++ code, and 8.6% higher than in generating Java code. Similarly, the accuracy of DeepSeek Coder V2 in generating Python code is 5.4% and 7.9% higher than that of generating C++ and Java, respectively. However, none of the existing methods can solve this problem at a low cost. By using the INTERVENOR method, the accuracy of LLMs generating Python is still 8.5% and 7.3% higher than

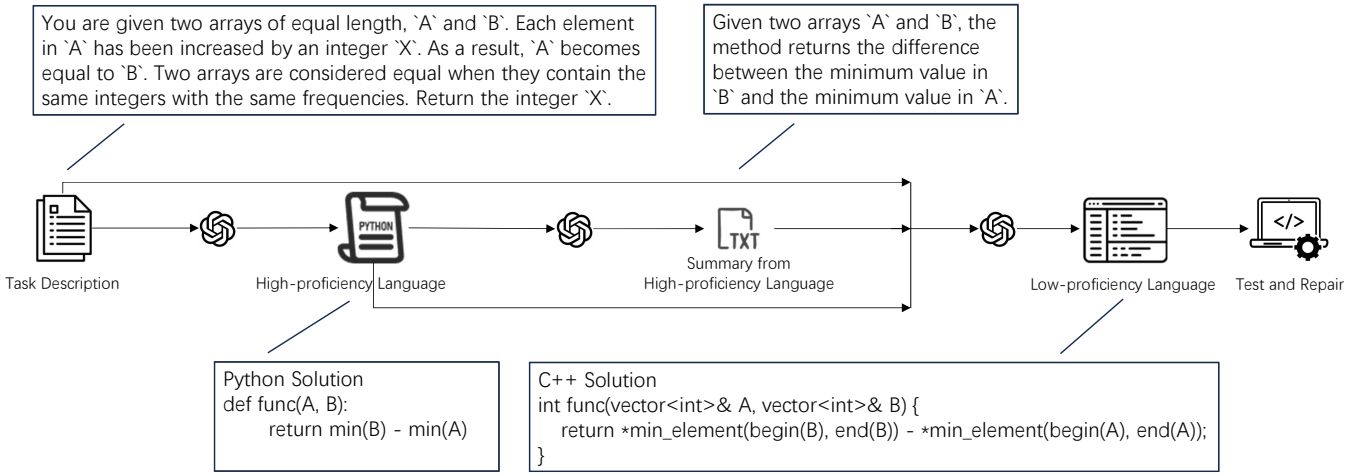

Figure 1: An example of using multiple perspectives to generate code. For this problem, the test cases ensure that for each input array A and B, there is an X satisfying condition. The Task Description and Summary from High-proficiency Language are two natural language descriptions that convey the same meaning but from different perspectives.

that of C++ and Java, respectively. In the following sections, we classify programming languages into high-proficiency languages and low-proficiency languages based on the accuracy of LLMs in generating code.

In this paper, we discuss how to enhance the performance of LLMs in generating code for low-proficiency languages. Since LLMs exhibit varying performance when generating code in different programming languages, why not take advantage of this difference? That is, we could use high-proficiency languages to improve the performance of low-proficiency languages. However, it may not be appropriate to use only the high-proficiency language as the prompt, since there is a syntactic gap between programming languages, which could lead to syntax errors in the generated code. Therefore, we propose using an intermediate language as an intermediary. We found that previous methods only used the natural language of functional description to generate code. In fact, there is another type of natural language description of the code, which is from the implementation perspective. We can use LLMs to summarize the code to obtain the natural language description from this perspective. In this way, we obtain two different perspectives: the functional perspective and the implementation perspective.

As shown in Figure 1, the task description for code generation is referred to as Perspective 1, which is described from the functional perspective. High-proficiency language serves as the realization of the task, and the summary of the high-proficiency language, referred to as Perspective 2, describes the task from the implementation perspective, while avoiding the syntactic complexities of the high-proficiency language. We observe that Perspective 1 for this code generation task is quite complicated. However, once it is implemented in Python, the task's underlying idea becomes much clearer. As a result, Perspective 2 is more direct than Perspective 1. Therefore, we infer that before generating low-proficiency language code, we could first generate the high-proficiency language code and its summary. High-proficiency language and multi-perspective natural language descriptions can provide additional information and help unlock the potential of LLMs.

To this end, we propose a method named MetaCoder, which is the first approach to use multiple perspectives to enhance the generation of low-proficiency language code. To implement this process, after receiving the task description, we first have LLMs generate the corresponding high-proficiency language code. This can be achieved using various methods, such as CoT, as mentioned earlier. Next, we summarize the generated high-proficiency language code to obtain a natural language description from the implementation perspective, while reducing some syntactic details. Then, we have LLMs generate the low-proficiency language code based on Perspective 1 and Perspective 2, with the high-proficiency language code provided as a reference. Although many syntactic details have been reduced in Perspective 2, there may still be some syntax errors in the generated low-proficiency language code. Therefore, we perform a syntax check at the end. We instruct LLMs to generate the corresponding test code and ensure its correctness. After checking for syntax errors, we guide LLMs to modify the code according to the compiler's feedback until the code compiles successfully or reaches the maximum number of iterations. Through this approach, we explore the potential of LLMs in low-proficiency code generation and achieve improved performance.

To validate the effectiveness of MetaCoder, we conducted extensive experimental evaluations on the widely used HumanEval-x dataset[44], including six popular LLMs with different number of parameters, GPT-3.5[1], GPT-4o, Llama 3.1 70B Instruct[19], DeepSeek V2.5, Qwen2.5 7/14/32/72B Instruct[41] and DeepSeek R1 7/8/14/32B[22]. We measured the correctness of the generated programs using unit tests and reported Pass@1. Our experimental results demonstrate that MetaCoder significantly improves the performance of LLMs in generating low-proficiency language code. Specifically, on the HumanEval-x dataset, the Pass@1 of GPT-3.5 for generating C++ code increased from 66.22% to 74.89%, while the Pass@1 of GPT-4o rose from 84.63% to 87.2%. For Java code generation, the Pass@1 of GPT-3.5 improved from 64.27% to 74.76%. On DeepSeek R1 8B, the Pass@1 of C++ code generation increased

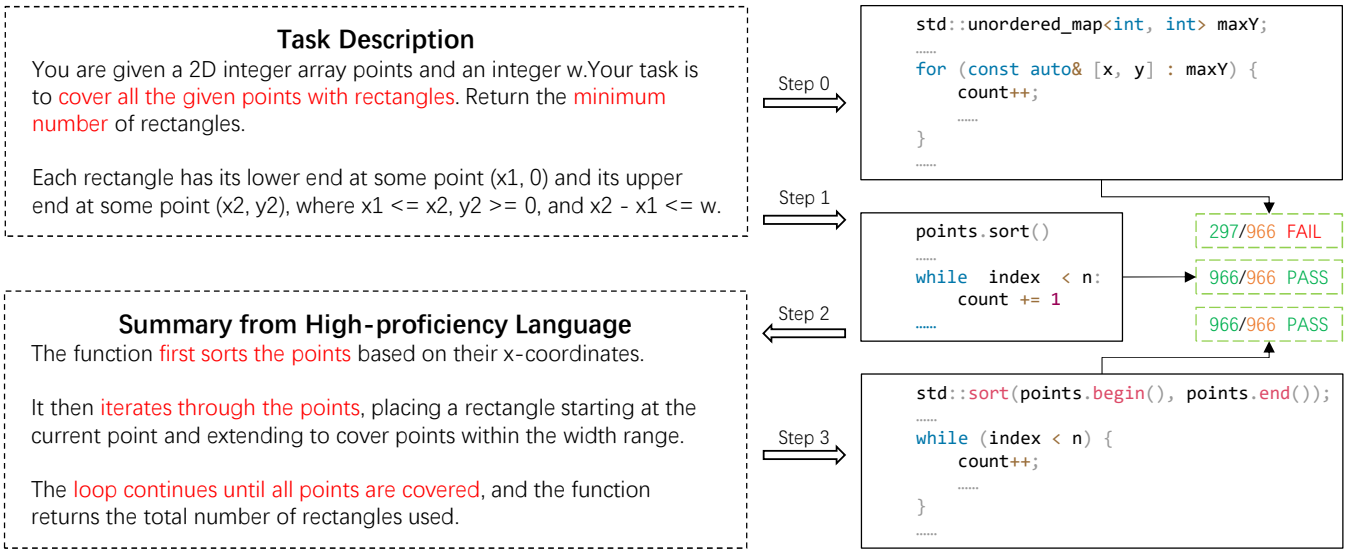

Figure 2: A motivating example. Directly generate C++ code failed (Step 0) , but succeeded when using the Task Description, Python solution, and Summary to generate C++ (Step 1 2 3) .

from 50.61% to 62.8%. These experimental results highlight the importance of multiple perspectives in facilitating code generation.

The contributions of our work can be summarized as follows:

- To the best of our knowledge, we are the first to propose a method that improves the performance of LLMs in generating low-proficiency language without requiring model fine-tuning, by leveraging the varying performance of LLMs across different programming languages.
- We propose MetaCoder, a method designed to improve the performance of LLMs in generating low-proficiency language. It utilizes multiple perspectives to mine the potential of LLMs in code generation.
- We conducted a comprehensive evaluation and demonstrated that MetaCoder is effective in improving the performance of LLMs in generating low-proficiency language.

The source code of MetaCoder and experiment data are publicly available at ***https://github.com/cx-hub/MetaCoder***.

2 MOTIVATION

2.1 A Motivating Example

In this section, we use a code generation example to illustrate the motivation of our method. This example is from LeetCode's 128th Biweekly Contest[6], problem number 3111. Through this problem, we demonstrate how to leverage task description, Python code, and the summary of Python code as multiple perspectives to generate correct C++ code.

The problem, shown in Figure 2, requires computing the minimum number of rectangles needed to cover all points given a set of coordinates and a threshold w. We asked ChatGPT[1] to provide solutions in both C++ and Python, with the results shown in Step 0 and Step 1. After testing, we found that the Python code from Step 1 passed all test cases, while the C++ code from Step 0 passed only 297 out of 966 test cases, with 669 failed. For example, when the input is "points = [[2, 1], [1, 0], [1, 4], [1, 8], [3, 5], [4, 6]]" and $w = 1$, the C++ code outputs 1, while the correct answer should be 2. Upon reviewing the C++ code, we discovered that ChatGPT failed to correctly interpret the problem statement, resulting in overly complex and incorrect logic. Specifically, although the coordinates include both x and y values, only x is relevant for determining the solution, and the y does not affect the final outcome. However, in Step 0, ChatGPT processed y, which led to a logical error in calculating the number of rectangles, resulting in an incorrect implementation.

Next, we asked ChatGPT to summarize the Python code generated in Step 1, and obtained the natural language description shown in the Figure 2, which we call Summary (Step 2). The initial problem description is referred to as Task Description (TD). We found that the semantics of Summary and TD are the same, as both aim to calculate the minimum number of rectangles. However, Summary and TD differ in their descriptions: TD is written from the functional perspective, explaining what tasks the code needs to complete, while Summary provides a description of the specific execution process of the code, such as how many steps the code includes and how each step is accomplished, while also reducing the syntax details of the Python code.

This led us to a key insight: if we add Summary to the process of generating C++ code, could we obtain correct C++ code? We then generated the C++ code shown in Step 3 of Figure 2. After testing, we found that the C++ code from Step 3 passed all test cases. The regenerated code was similar in structure and logic to the Python code from Step 1, and did not process y, unlike the C++ code from Step 0. We believe this improvement is due to the introduction of Summary, which helped ChatGPT avoid common pitfalls when generating C++ code.

2.2 Key Ideas

Inspired by the example above, we wonder whether we can improve the performance of LLMs in generating low-proficiency language

code in a similar way, by having high-proficiency languages guide the generation of low-proficiency languages. When generating code in a low-proficiency language, we first have LLMs generate the code in a high-proficiency language. Based on our previous findings, it is more likely to stimulate the potential of LLMs to generate correct code in high-proficiency languages. Afterward, we ask LLMs to summarize the generated high-proficiency language code to obtain a summary. This step primarily provides natural language descriptions from different perspectives, helping LLMs understand how to correctly complete the task from another viewpoint. Finally, we ask LLMs to generate the code in the low-proficiency language based on the task description, high-proficiency language code, and summary. By using the advantages of LLMs in high-proficiency languages, we can assist in generating low-proficiency language code and fully tap into the potential of LLMs. In a sense, this method is an alternative to CoT. Both our method and CoT enable LLMs to complete more complex tasks step by step, utilizing the inherent capabilities of LLMs. However, unlike CoT, our method generates high-proficiency language code, summary, and low-proficiency language code sequentially through interaction with LLMs, rather than in a single generation step.

However, we should acknowledge that this method is not infallible. It may not yield better results when LLMs are unable to solve the problem in the high-proficiency language. For example, if the generated code fails in the high-proficiency language, it is unlikely to succeed in the low-proficiency language.

3 Approach

In this section, we present the process of MetaCoder. MetaCoder utilizes multiple perspectives to generate low-proficiency language code. Additionally, our method incorporates syntax checking to effectively address syntax errors caused by long prompts. Our method consists of two main steps: 1.code generation and 2.test and repair. For a given programming task description and target programming language, MetaCoder first generates code in languages that LLMs are proficient in. Then MetaCoder summarizes the generated code, and finally generates code in the target language using the multiple perspectives. In the second step, MetaCoder generates function calls based on the task description and ensures their validity through continuous checks. MetaCoder then verifies whether the generated code from the first step can be compiled successfully. If it cannot, MetaCoder regenerates the code according to error messages until it compiles successfully or the number of iterations is exhausted. Below, we provide a detailed description of these two steps.

3.1 Code Generation

In this step, we use high-proficiency languages to guide the generation of low-proficiency languages, as shown in Figure 1. After obtaining the task description, MetaCoder first lets LLMs use the high-proficiency language to complete the programming task, and we refer to this generated code as Code_A. Several methods can be used to generate Code_A, including Zero-Shot, CoT, and others, which also makes our method extensible to some extent. After obtaining Code_A, MetaCoder then obtains natural language descriptions different from the Task Description. Here, we choose to have LLMs summarize Code_A directly. Existing LLMs have a strong ability to understand and analyze code, which allows us to leverage this ability to obtain a summary of Code_A.

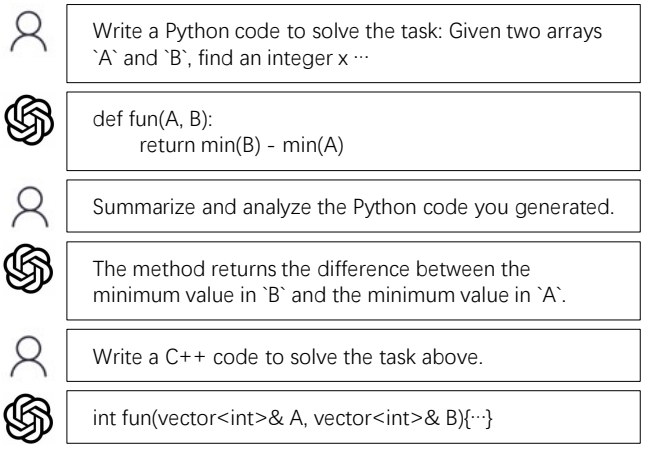

Figure 3: An example of code generation. The process of generating low-proficiency code in the form of a dialogue.

To ensure the summary is both comprehensive and concise, we ask LLMs to summarize Code_A without excessive detail, as overly fine-grained summary could hinder the generation of the target code. Therefore, we add a requirement to the prompt, such as "comprehensive and concise" when generating the summary. Compared with the task description, the summary obtained in this step provides a different angle of description and reduces many of the syntax details in Code_A.

Finally, we use the task description, Code_A, and summary as inputs to generate the code in the low-proficiency language, which we refer to as Code_B. To organize these information, we structure them in the form of a dialogue and add some necessary details. Since most LLMs have been fine-tuned to optimize this input-output format, we believe this construction method is more reasonable, even though it increases the calculation cost. Next, we present the entire process of the first step using an example.

Figure 3 illustrates the first step of MetaCoder, as discussed earlier. In this example, after obtaining the task description, Code_A, and summary, we organize them into a dialogue and add some user inputs to make the input received by LLMs more reasonable.

3.2 Test and Repair

Due to the potential for errors in code generated by LLMs [33], especially in longer contexts, we aim to correct syntax errors to further improve the code generation performance of MetaCoder . We first tested potential syntax errors using Phi3 14B [7] and Gemma2 27B [38] on MBCPP [9] and MBJP [9], and found the following common syntax errors: 1. Modification of function declarations, including function names and parameters. 2. Generating code in an incorrect programming language. 3. Uninitialized/undeclared variables. These syntax errors directly reduce the effectiveness of MetaCoder, so we aim to perform syntax check after obtaining the target code.

The second and third types of errors can be detected by directly compiling the generated code to check if errors exist, but the first type of error cannot be detected this way. Some methods[39] use

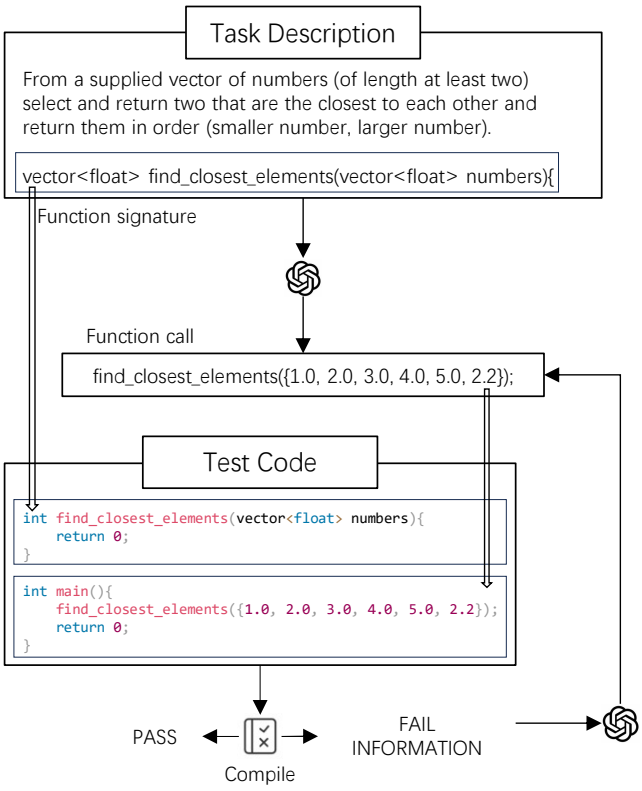

Figure 4: An example of function call generation. Given the task description, LLMs generate a valid function call.

test cases to detect errors and fix them, but not all datasets have example test cases, and using test cases may expose the test case information, leading to inflated accuracy. To address the first type of error, we plan to generate a function call that can detect this specific type of error.

3.2.1 Function Call Generation. As shown in Figure 4, to detect if the code generated from the Task Description contains the first type of error mentioned earlier, we simply need to construct a piece of code that calls the function as declared in the Task Description. Then, we concatenate this piece of code with the generated code to be checked, forming the Test Code, and compile it. We refer to this piece of code that successfully calls the function as a valid function call. Else, it is considered an invalid function call.

We provide the Task Description to LLMs and ask LLMs to return a function call. To ensure that the the return from LLMs is valid, we perform a check on the returned function call. We supplement the Task Description with simple additions, such as adding a return statement (e.g., return 0;), making the code compilable. The compilable code, along with the function call generated by LLMs, is then checked to ensure it compiles. If the code fails to compile, the error information is fed back to LLMs, prompting LLMs to regenerate a function call until a valid, compilable function call is produced.

We do not attempt to generate the output corresponding to this function call to verify the logical correctness of Code_B, as the reasoning process of LLMs may be flawed. For complex tasks, the output generated by LLMs may be incorrect, and using it to check the functional correctness of Code_B could lead to inaccurate results. While some existing studies suggest using techniques like voting to filter out incorrect reasoning, this is beyond the scope of our work. However, in practical scenarios, this step can be replaced with manually written inputs and outputs, making our method more extensible. Below is an example that illustrates this process.

Figure 4 shows an example of generating function call based on the Task Description and verifying its validity. The programming language here is C++, though a similar method applies to Java. The Task Description includes the problem description and a function signature. We provide this Task Description to LLMs, asking LLMs to generate one or more function calls. After receiving a function call, here is "find_closest_elements(1.0, 2.0, 3.0, 4.0, 5.0, 2.2); ", we construct a code snippet named Test Code that includes the function signature and the function call. We modify the function signature to complete the function, then add a main function to call the "find_closest_elements" function, using the generated function call. We check the syntax of the Test Code to determine whether the generated function call is valid. If no errors are found, the function call is considered valid; otherwise, the error information is sent back to LLMs, prompting it to regenerate the function call until a valid function call is produced. This process of generating valid function call from the Task Description is simple for LLMs, particularly because we've incorporated error correction, which ensures successful generation in all of our experiments.

3.2.2 Syntax Check and Repair. Finally, we use the valid function call generated in the previous step to check the syntax of Code_B, aiming to reduce syntax errors introduced by LLMs. Similar to the previous step, we concatenate Code_B with the valid function call and verify whether the combined code can be compiled. If the code compiles successfully, we take the resulting Code_B as the final output. If the code fails to compile, MetaCoder collects the error messages from the compiler.

MetaCoder then appends the error messages to the context of the conversation and prompts LLMs to modify Code_B to address the syntax errors. The modified code undergoes the same syntax check. This process continues until the code compiles successfully or the maximum number of iterations is reached. To prevent excessive computational overhead, we set a maximum loop limit, and stop the process if the number of repair times exceed this limit. We also check the error messages before adding them to the context. If any error message exceeds a certain length, we truncate it to avoid surpassing the context limit. We do not perform any additional processing on the error information, such as asking LLMs to create a modification plan or summarize the error information based on the errors, as we believe the error messages contain sufficient information. Current LLMs are capable of effectively interpreting and using this information to fix the errors.

4 EXPERIMENTAL DESIGN

To evaluate MetaCoder, we conducted a comprehensive experiment. This section outlines our experimental design.

4.1 Research Questions

We focus on the following research questions to assess the performance of MetaCoder:

Table 1: Selected LLMs. We choose several LLMs from four companies. These LLMs have different parameters.

Company	Model	Size	Release Date
OpenAI	GPT-3.5	-	November 2022
	GPT-4o	-	May 2024
Meta AI	Llama 3.1	70B	July 2024
Alibaba Cloud	Qwen 2.5	7B	September 2024
		14B	September 2024
		32B	September 2024
		72B	September 2024
DeepSeek	DeepSeek V2.5	236B	September 2024
	DeepSeek R1	7B	January 2025
		8B	January 2025
		14B	January 2025
		32B	January 2025

* We lack the parameter for GPT-3.5 and GPT-4o because OpenAI has not open-sourced them.

* Qwen 2.5 was released in May 2024, and was open source in September 2024.

RQ1: What is the effectiveness of our method? In this research question, we investigate whether MetaCoder can improve the code generation performance of LLMs for low-proficiency languages. We compare MetaCoder's performance against other methods by applying it to multiple LLMs, with Python guiding the generation of C++ and Java code.

RQ2: What is the effectiveness of each component of our method? This research question explores the importance of each component in MetaCoder. We examine the impact of removing specific components, such as Python solution generation, the summary of Python solution, and syntax check.

RQ3: What is the cost of our method? In this research question, we assess the cost introduced by MetaCoder. We calculate the computational resources required to generate code with MetaCoder and compare it to the cost of baseline methods.

RQ4: Detailed analysis. How much influence will the granularity of summary have on the results? What is the effectiveness of our method in the real world? Can our method be combined with other methods?

4.2 Selected LLMs

Numerous LLMs are available for code generation. However, MetaCoder requires LLMs to possess a high level of contextual understanding and conversational abilities. LLMs without instruction fine-tuning (e.g., InCoder[21] and CodeGen[32]) are not suitable for our approach. For our experiments, we selected several representative models from both proprietary and open-source categories, as Table 1 shows. These models include: GPT-3.5, GPT-4o, Llama 3.1 Instruct, Qwen 2.5 Instruct, DeepSeek V2.5, and DeepSeek R1. These models were chosen to evaluate the effectiveness of our approach across a range of different LLMs.

4.3 Benchmarks

Initially, we planned to conduct experiments using two public code generation benchmarks: HumanEval-x[44] and MBXP[9]. However, during the experimentation process, we found several issues with MBXP, including semantic ambiguity and errors in some test cases. Given our limited resources, it was not feasible to fully curate this dataset, which led to its abandonment. As a result, we decided to focus on the HumanEval-x benchmark.

HumanEval-x, proposed in 2023, is designed to evaluate multi-language code generation capabilities. This benchmark helps mitigate overfitting due to data leakage by including a diverse set of problems. HumanEval-x supports five programming languages: Python, C++, Java, JavaScript, and Go. Each language subset contains 164 problems, each with a task description, function signature, and test cases. The task descriptions are accompanied by 3 to 4 test cases. For our experiments, we used the C++ and Java subsets.

Although HumanEval-x also has some issues, such as errors in the example test cases of CPP/47, we were able to fix them easily. In our experiments, we used only one dataset, HumanEval-x, which may raise concerns about the generalizability of our results. While there are many code generation benchmarks available, most are limited to a single programming language. Currently, there is a lack of multilingual code generation datasets, so we chose the most widely used one from the available datasets to maximize the generalizability of our results.

4.4 Evaluation Metrics

In line with previous research[13, 15, 32, 44] in code generation, we use the pass@k metric to evaluate the performance of MetaCoder. Specifically, for each code generation task, we have LLMs generate k pieces of code. If any of the generated codes passes all test cases, the task is considered successfully completed. The pass@k score is then calculated as the percentage of tasks that were successfully completed out of all tasks. A higher pass@k value indicates better performance in generating correct code.

Previous work[9, 13, 15, 26] has demonstrated that the standard pass@k metric can exhibit high variance. To address this, an unbiased version of pass@k was proposed. Following this approach, we generate n codes (where $n > k$) for each task and count the number c of codes that pass the test. The unbiased pass@k is then computed using the following formula:

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

Previous code generation studies have employed text similarity-based metrics, such as BLEU[34]. However, these metrics were designed for evaluating natural language generation and are not well-suited for assessing the correctness of code. As such, we omit these metrics in our experiments.

For our experiments, we focus on pass@1, generating 5 samples for each task (i.e., $k = 1$ and $n = 5$).

4.5 Comparison Baselines

To evaluate the effectiveness of our method, we selected three widely-used prompting methods, one self-repair method, and one multi-agent method as baselines: Zero-Shot[15], Few-Shot[15], Zero-Shot CoT[23], INTERVENOR[39], and Self-Collaboration[18].

Table 2: The Pass@1 (%) of MetaCoder and baselines on HumanEval-X. The numbers in green denote MetaCoder's relative improvements compared to Zero-Shot.

	GPT 3.5		GPT 4o		DeepSeek V2.5		Llama3.1 70B		Qwen2.5 72B	
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
Zero-Shot	66.22	64.27	84.63	87.2	82.56	81.95	73.78	72.2	85.98	86.59
Few-Shot	60	67.68	80.98	85	85.37	83.54	78.05	78.65	86.58	85.37
CoT	64.51	66.34	84.39	85.73	**86.59**	**87.2**	76.22	**81.1**	85.98	86.58
INTERVENOR	66.95	71.95	84.02	88.41	83.41	80.73	71.95	74.27	88.41	85.98
Self-Collaboration	72.56	69.51	85.98	**90.85**	83.54	85.24	75.61	76.22	**90.24**	**89.02**
MetaCoder	**74.89**	**74.76**	**87.2**	85.49	**86.59**	86.34	**80**	78.65	86.58	87.8
Relative Improvement	13.09%↑	16.32%↑	3.04%↑	-1.96%↓	4.88%↑	5.36%↑	8.43%↑	8.93%↑	0.7%↑	1.4%↑

	Qwen2.5 32B		Qwen2.5 14B		Qwen2.5 7B		DeepSeek R1(C++)			
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**7B**	**8B**	**14B**	**32B**
Zero-Shot	78.9	74.02	72.2	77.44	68.29	69.39	47.93	50.61	54.88	53.05
MetaCoder	**84.76**	**86.59**	**73.54**	**79.63**	**71.95**	**71.59**	**63.41**	**62.8**	**80.49**	**81.1**
Relative Improvement	7.43%↑	16.98%↑	1.86%↑	2.83%↑	5.36%↑	3.17%↑	32.3%↑	24.09%↑	46.67%↑	52.87%↑

4.6 Implementation Details

Our experimental setup is as follows:

For LLMs, we set the temperature to 0.2 and the top-p to 0.95 across all experiments. For the R1 models, the maximum token limit was set to 32,768. For the non-R1 models, the maximum token limit was set to 1,024. Other parameters, such as frequency penalty, were left at their default values. For the R1 models, we used Ollama to load its q4_k_m quantized version. For the Llama 3.1 model, we used the complete unquantized version and loaded it in bfloat16 format. Other models use APIs to gain access.

For the prompts, we included format requirements for all code generation tasks, such as adding special symbols before and after the code, to ensure the complete code can be obtained later. we selected two examples, humanevalx/0 and LeetCode 3111, to serve as examples for Few-shot and MetaCoder. In other experiments, we did not add examples to the prompts.

5 RESULT

In this section, we report and analyze the experimental results for each research question.

5.1 Effectiveness of Our Approach

We conducted experiments using several LLMs on the HumanEval-x and compared the results with baseline methods. The Pass@1 are shown in Table 2. If we consider the table alone, MetaCoder appears to perform significantly well on DeepSeek R1. However, there is a misunderstanding here. Therefore, we will categorize the discussion based on whether the model is the R1 version.

5.1.1 Non-DeepSeek R1 series models. As evidenced by the results in Table 2, MetaCoder improves the performance of generating C++ and Java code across multiple LLMs, with significant improvement over existing methods. Compared with other baselines, MetaCoder generally outperforms existing methods, except in a few cases, such as with GPT-4o when generating Java code.

Notably, the largest improvements are observed in the GPT-3.5 and Llama 3.1 70B models. For C++ code generation, GPT-3.5 improved from 66.22% to 74.89%, a gain of 13.09%, while Llama 3.1 70B increased from 73.78% to 80%, with an improvement of 8.43%. For Java code generation, GPT-3.5 increased from 64.27% to 74.76%, an improvement of 16.32%, and Llama 3.1 70B improved from 72.2% to 78.65%, with an improvement of 8.93%. In comparison, these two models nearly achieved accuracies of 75% and 80% respectively, indicating that MetaCoder brings the performance of generating C++ and Java code closer to that of Python generation for these models (75% and 80.5%). The performance decline with the GPT-4o model, specifically when using MetaCoder to generate Java code, may be attributed to a pre-designed prompt that was not suitable for this model.

What surprised us is DeepSeek V2.5, which achieved the best results when using CoT rather than other methods. We conducted further analysis of this model's output and found that, compared to other models, it tends to generate longer natural language explanations when producing code. It is possible that CoT triggered its capabilities, enabling the model to analyze the task in more details and provide more effective outputs.

On smaller models, MetaCoder led to accuracy improvements, but the improvements are less pronounced. For the 7B and 14B models of Qwen 2.5, the performance enhancement is only about 3%. On the 32B model, there are significant improvements of 7.43% and 16.98%, respectively. We hypothesize that this discrepancy is due to the smaller models' difficulties in handling extensive context and their limited capacity due to fewer parameters.

5.1.2 DeepSeek R1 series models. For the distilled models in the DeepSeek R1 series, it is important to note that their effectiveness might be overstated. The best improvement observed was 52.87%, which may not accurately reflect their true capability. This discrepancy arises because, during Zero-Shot generation tasks, these distilled models often fail to generate C++ or Java code as required by the problem description. Instead, they frequently output Python or JavaScript code. This issue is particularly pronounced when generating Java code, which led us to exclude Java generation attempts for these models. This suggests that the results for this segment

Table 3: Component analysis. For the Python code and Python summary in Figure 3, as well as the Test and Repair in Figure 1, we removed one element at a time and recorded the Pass@1.

	GPT 3.5		GPT 4o		DeepSeek V2.5		Llama3.1 70B		Qwen2.5 72B	
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
Zero-Shot	66.22	64.27	84.63	**87.2**	82.56	81.95	73.78	72.2	85.98	86.59
- Python Solution	73.66	74.02	85.98	85.61	84.76	85.98	77.93	78.17	85	86.84
- Python Summary	71.34	73.78	86.71	84.51	85.98	84.39	76.22	77.44	83.41	84.27
- Test and Repair	72.56	71.95	86.59	85.12	85	85.37	76.71	76.45	86.21	86.34
MetaCoder	**74.89**	**74.76**	**87.2**	85.49	**86.59**	**86.34**	**80**	**78.65**	**86.58**	**87.8**

Table 4: Output tokens per task. The output tokens consumption of different models under different methods.

	GPT 3.5		GPT 4o		DeepSeek V2.5		Llama3.1 70B		Qwen2.5 72B	
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
Zero-Shot	90.8	177.26	274.74	572.6	673.27	737.4	465.9	542.93	582.32	614.86
Few-Shot	109.35	113.54	171.44	130.39	195.15	167.01	144.62	118.43	253.71	136.36
CoT	302.41	327.21	499.53	545.56	507.34	540.45	364.76	370.88	568.46	407.27
INTERVENOR	99.23	190.32	292.62	611.46	681.23	815.26	510.07	545.45	614.49	627.62
Self-Collaboration	730.81	750.21	1231.91	1833.1	1606.19	1556.74	919.16	908.2	1697.06	1608.95
MetaCoder	396.61	421.35	690.47	824.3	738.02	848.67	438.99	459.13	637.67	609.41

may be inflated—while these models may be capable of solving the given programming tasks, they do not always produce the correct programming language as output.

This issue is less common with the 70B distilled model and the 671B R1 model, both of which already achieve over 95% Pass@1. For these high-performing models, our approach provides limited room for further improvements.

To summarize, our method significantly improves the performance of generating C++ and Java code on LLMs, in some cases bringing their performance to a level similar to that of Python code generation. However, on smaller models, while there are improvements, the range of improvement is relatively small, which may be related to the model's ability to handle complex contexts.

5.2 Effectiveness of Components

In this section, we evaluate the effectiveness of each component of our design. First, we assess the influence of Python code on the final output when generating C++ code. Next, we investigate whether the Python Summary affects the final output when generating C++ code. Finally, we examine the impact of the syntax check on the final result. The results are shown in Table 3.

In our ablation study, we carefully analyze the contribution of each component to the overall performance of our proposed approach. Our analysis reveals that the Python summary plays a crucial role in enhancing the effectiveness of the generated code. This holds true across all tested models, indicating that summarization helps distill the core functionality, thereby guiding the generation process more accurately. However, the presence of the original Python code still contributes positively to the final output. The synergy between the summary and the Python code highlights the importance of both elements in our framework, demonstrating how they complement each other to achieve superior results.

Regarding the Test and Repair phase, our findings indicate that while this step is essential for ensuring code quality, its impact varies significantly across different models. For advanced models such as GPT-4o, DeepSeek V2.5, and Llama 3.1, the improvement in accuracy from incorporating syntax checks is relatively modest. For example, the accuracy of C++ code generated by GPT-4o only increased from 86.59% to 87.2%, while for Java code, it improved from 85.12% to 85.49%, which is hardly a significant improvement. This suggests that for more sophisticated models, which already possess a high level of syntax understanding, additional syntax check may offer less reward.

This comprehensive evaluation helps us understand which component of our approach contribute most significantly to its effectiveness and identifies direction that further optimization or research may be beneficial.

5.3 Cost Evaluation

In this section, we quantify the overhead associated with MetaCoder, focusing on one primary metric: the average output tokens consumption. We choose to account for output tokens consumption rather than input tokens consumption because the cost associated with output tokens is relatively higher. On many platforms, the cost ratio between output tokens and input tokens is typically 4:1. Therefore, it makes more sense to focus on the consumption of output tokens. The results are shown in Table 4.

The results show that our method requires more output tokens than Zero-Shot, as MetaCoder needs at least three LLM outputs to complete a code generation task. Despite this increase, it is important to note that MetaCoder consumes far fewer output tokens than Self-Collaboration, while achieving better performance.

We also observed some interesting trends: most models consume more output tokens in Zero-Shot than in Few-Shot, particularly DeepSeek V2.5. This is because, when using Zero-Shot, models

Table 5: The Pass@1 of MetaCoder at different summary granularities. Since we cannot directly control the granularity of the generated summary, we chose to include a word limit in the prompt to achieve a similar effect.

	GPT 3.5		GPT 4o		DeepSeek V2.5		Llama3.1 70B		Qwen2.5 72B	
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
MetaCoder	74.89	**74.76**	87.2	85.49	**86.59**	86.34	80	**78.65**	**86.58**	87.8
50 words	72.93	72.56	83.66	83.17	86.22	84.63	78.41	76.83	86.22	85.37
100 words	**75.24**	74.39	86.46	**85.86**	85.98	85.98	**80.13**	77.93	85.85	**88.54**
200 words	74.39	73.9	**87.32**	85.24	85.37	**86.59**	79.64	78.29	85.98	88.05

Table 6: The Pass@1 of MetaCoder in the real world. By introducing a filter in Figure 5, we can simulate real-world scenarios.

	GPT 3.5		GPT 4o		DeepSeek V2.5		Llama3.1 70B		Qwen2.5 72B	
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
MetaCoder	74.89	74.76	87.2	85.49	86.59	86.34	80	78.65	86.58	87.8
MetaCoder + Filter	81.59	81.22	90.73	91.22	90.85	90	86.58	86.58	91.46	92.07

	Qwen2.5 32B		Qwen2.5 14B		Qwen2.5 7B		DeepSeek R1(C++)			
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**7B**	**8B**	**14B**	**32B**
MetaCoder	84.76	86.59	73.54	79.63	71.95	71.59	63.41	62.8	80.49	81.1
MetaCoder + Filter	90.12	91.46	85.12	88.41	80.73	85.85	71.34	74.39	84.76	85.37

Table 7: The Pass@1 of MetaCoder combined with other methods. These methods are used for generating Python code.

	GPT 3.5		GPT 4o		DeepSeek V2.5		Llama3.1 70B		Qwen2.5 72B	
	C++	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**	**C++**	**Java**
MetaCoder	74.89	74.76	87.2	85.49	86.59	86.34	80	78.65	86.58	87.8
+ INTERVENOR	76.22	78.65	87.2	89.02	85.24	87.2	80.73	83.54	84.39	88.05
+ Self-Collaboration	78.05	80	86.59	89.02	86.59	88.41	81.1	83.54	85.24	90.24
+ Answer	82.93	91.46	92.07	94.51	91.23	93.29	89.02	93.29	90.85	93.29

typically generate not only code solutions but also explanations of the code and test cases, which are usually longer in Java, leading to higher costs. In contrast, other methods, such as Few-Shot, often provide examples, which helps reduce the overall token consumption. We also tested the output token consumption of the DeepSeek R1 series models. For the DeepSeek R1 model, output token consumption is particularly high, which exceeded 3000 tokens on average. DeepSeek R1 7B consumes the most tokens, reaching an astonishing 5265.53. This is due to the extensive "thinking" process before outputting results. The 7B model is more likely to reach the maximum output limit, as it often repeats outputs or overuses examples.

In conclusion, although our method requires higher token consumption than Zero-Shot, the trade-off is justified by its superior performance and more efficient resource utilization.

5.4 More Detailed Research

In this section, we study the impact of summary granularity on MetaCoder, the performance of MetaCoder in the real world, and the performance of combining MetaCoder with other methods.

5.4.1 Change the granularity of summary. In this section, we explore the influence of summary granularity on the results. Since we cannot directly control the granularity of the summary generated by LLMs, we instead control the length of the summary. The more words LLMs generates, the more detailed the summary becomes. We provide specific instructions in the prompt, specifying that the summary should be within 50 words, 100 words, or 200 words, as we found that the summaries generated by GPT-3.5 mostly fall within this range.

Our results, shown in Table 5, demonstrate that as the length of the summary increases, the accuracy of the code generated by each LLM generally improves. However, for specific tasks and models, selecting the appropriate level of granularity remains crucial and can be challenging. For example, when generating C++ code with GPT-3.5, a 100-word limit works better, whereas a 200-word limit is more effective when generating Java code with Llama 3.1.

5.4.2 Real world performance. In this section, we discuss the practicality of our method in real-world scenarios. After completing code on a competition website[5, 6] or during the production process, it is common to conduct tests to verify whether the code is correct. To simulate this process, we introduce test results. First, we have LLMs generate C++/Java code directly, then test it. If the test passes, the generated code is output. If it fails, we apply MetaCoder to this task. In this process, we do not expose any test cases. Essentially, we have added a filter, as shown in Figure 5. When LLMs generates incorrect code, we use MetaCoder to address the task.

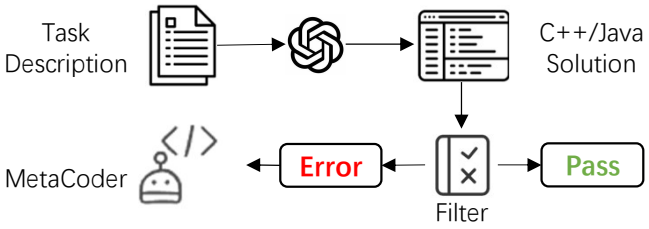

Figure 5: A filter before using MetaCoder. The filter first assess whether LLMs can directly generate the correct code. If they cannot, then use MetaCoder

Our results, shown in Table 6, demonstrate that after adding the filter, the accuracy improves significantly. This also highlights an important issue: during the implementation of our method, some tasks that LLMs can do right are finally wrong. This is a challenge worth considering and something we aim to address in future work.

This demonstrates that our method is highly practical and can enhance the diversity of LLM-generated code, incorporating more correct answers into the generation process. It truly unleashes the potential of LLMs for code generation.

5.4.3 Combine with other methods. In this section, we will discuss the effectiveness of combining MetaCoder with other methods. When generating Python code in the first step, we can use other techniques to enhance the accuracy of the Python code, which in turn improves the accuracy of the final generated code. To test the effectiveness of combining other methods, we use INTERVENOR and Self-Collaboration as the first steps in this process. Additionally, to evaluate the upper limit of our method, we include the Python answer as the first step.

The results, as shown in Table 7, indicate that our method has strong potential when combined with other methods. By directly using the answer as Python code, we found that the upper limit of MetaCoder is very high. For instance, GPT-4o achieved an accuracy of 94.51% when generating Java. MetaCoder also proved effective when combined with other methods, although the improvement was not as significant as when directly using the Python answer. This suggests that additional strategies may be needed to further enhance the effectiveness of MetaCoder, which will be a direction for our future work.

6 DISCUSSION

In this section, we will discuss the applicability of MetaCoder, its current limitations, and our future work.

First, MetaCoder has limited applicability for code generation at the code repository level. Due to the context length limitation of LLMs, generating code at this level often requires additional contextual information. While methods like Retrieval-Augmented Generation (RAG) can help handle information beyond the model's context window, MetaCoder is also constrained in such scenarios. However, if code generation at the repository level can be decomposed into several independent tasks, MetaCoder could still prove effective.

Second, the effectiveness of MetaCoder may be lower for smaller models. This is because the contextual understanding ability of smaller models is not as strong as that of larger models, making it difficult for them to process information comprehensively when dealing with long texts. In fact, this issue is common across many models. When the length of the prompt exceeds the model's context window, LLMs tend to selectively process certain pieces of information while ignoring others.

Finally, our future work will focus on improving MetaCoder's performance, such as determining the appropriate level of granularity for summary, preventing MetaCoder from making mistakes on tasks LLMs can directly handle, and finding more effective ways to combine MetaCoder with other methods.

7 RELATED WORK

In this section, we outline the most relevant directions of research related to our work.

Code generation is one of the most prominent topics in both software engineering and artificial intelligence research today. In recent years, LLMs have been extensively studied for code generation tasks. The development of using LLMs for coding tasks accelerated rapidly after CodeBERT[20] first linked code tasks with pre-trained models. This progress further gained momentum with the release of ChatGPT, which sparked widespread interest among researchers in applying LLMs to code generation tasks.

To date, numerous proprietary and open-source LLMs have been applied to code generation, including notable models such as GPT-4[8], Claude 3.5 Sonnet[2], and Llama 3.1. These LLMs exhibit strong natural language understanding and reasoning capabilities, often achieving outstanding results in code generation tasks. For example, on the HumanEval dataset, Claude 3.5 Sonnet achieves an accuracy of 92%, while Llama 3.1 and GPT-4 reach accuracy rates of 89% and 90.2%, respectively. However, it is still a challenge to use LLMs to generate entirely correct code for complex requirements. To enhance the performance of LLMs in code generation, several methods[10, 16, 24, 30, 42] have been proposed. These include prompting techniques, fine-tuning techniques, self-repair techniques, and multi-agent collaboration.

Prompting techniques[26, 40] propose different prompting techniques to help LLMs better analyze requirements and generate code. The SCoT prompting technique[26] instructs LLMs to generate a structured chain of thought (SCoT) using program structures such as sequences, branches, and loops. LLMs then generate the code based on the SCoT.

Fine-tuning techniques[25, 28, 37] fine-tune LLMs to make it more suitable for code generation tasks, such as UNICODER[37]. Found that CoT has different logical structure and expression form from code, Sun et al.[37] introduced universal code UniCode as an intermediate representation, collected data sets and fine-tuned LLMs, which improved the performance of LLMs in code generation tasks. Ma et al.[28] introduce the code data at the pre-training stage, instruction-tuning stage, and both of them respectively to explore the impact of code data.

Self-repair techniques[17, 29, 33, 43] use external tools to correct the code according to the feedback errors, making the code generation more accurate, such as INTERVENOR[39]. According to the feedback from external tools, Wang et al.[39] asked LLMs to generate suggestions for modifying the code, and modified the code according to the suggestions.

Multi-agent collaboration, such as Self-Collaboration[18], use multi-agent to verify the generated code agent and improve the quality of the generated code. Dong et al.[18] proposed a framework of self-collaboration, and built a team composed of analyst, coder and tester to solve the code generation task collaboratively.

While these methods have proven effective in improving the performance of LLMs in code generation, there remains a significant performance gap across different programming languages. This is because these approaches do not specifically address the languages in which LLMs are less proficient. Our method seeks to bridge this gap by using programming languages in which LLMs excel to guide the generation of those in which they are less skilled. In this process, we leverage natural language from multiple perspectives to reinforce the guiding effect and unlock the potential of LLMs in code generation.

8 CONCLUSION

In this paper, we propose a new code generation method—MetaCoder, which leverages the strengths of LLMs in high-level languages to enhance code generation in low-level languages. Additionally, MetaCoder integrates an iterative verification mechanism to detect and correct syntax errors in the generated code, further improving accuracy. Extensive experimental results demonstrate the effectiveness and versatility of MetaCoder. In summary, MetaCoder offers an effective approach to automatically generate code. This innovative method has the potential to significantly improve the quality of generated code, reduce human intervention, and accelerate the development of complex software systems.

Acknowledgments

The authors express thanks to the anonymous reviewers for their insightful comments. This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan Province (No.2023RC1001) and NSFC No.U2441238 and No.62202474.

References

[1] 2025. ChatGPT. https://chatgpt.com/ Accessed: 2025-2-26.
[2] 2025. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet Accessed: 2025-2-26.
[3] 2025. Copilot. https://copilot.microsoft.com/ Accessed: 2025-2-26.
[4] 2025. Cursor. https://www.cursor.com/ Accessed: 2025-2-26.
[5] 2025. HackerRank. https://www.hackerrank.com/ Accessed: 2025-2-26.
[6] 2025. LeetCode. https://leetcode.com/ Accessed: 2025-2-26.
[7] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219* (2024).
[8] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[9] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).
[10] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698.
[11] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 677–708.
[12] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).
[13] 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 arXiv:2207.10397* (2022).
[14] Jiawei Chen, Wentao Chen, Jing Su, Jingjing Xu, Hongyu Lin, Mengjie Ren, Yaojie Lu, Xianpei Han, and Le Sun. 2024. The Rise and Down of Babel Tower: Investigating the Evolution Process of Multilingual Code Large Language Model. *arXiv preprint arXiv:2412.07298* (2024).
[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[16] Simin Chen, Zexin Li, Wei Yang, and Cong Liu. 2024. DeciX: Explain Deep Learning Based Code Generation Applications. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2424–2446.
[17] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).
[18] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–38.
[19] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[21] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
[22] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[23] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.
[24] 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. *arXiv preprint arXiv:2310.08992* (2023).
[25] Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. 2024. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 182–203.
[26] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.
[27] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118* (2023).
[28] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298* (2023). https://doi.org/10.48550/arXiv.2309.16298
[29] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2023), 46534–46594.
[30] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996* (2023).
[31] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.
[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[33] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896* (2023).

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

[35] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* 6, 3 (2023).

[36] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[37] Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. UniCoder: Scaling Code Large Language Model via Universal Code. In *ACL (1)*.

[38] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118* (2024).

[39] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2023. Intervenor: Prompting the coding ability of large language models with the interactive chain of repair. *arXiv preprint arXiv:2311.09868* (2023).

[40] 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. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[41] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[42] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608.

[43] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087* (2023).

[44] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.

[45] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931* (2024).