香港中文大學計算機科學與工程學系
Department of Computer Science and Engineering
The Chinese University of Hong Kong

THE CHINESE UNIVERSITY OF HONG KONG

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LYU2301 Term-End Report : Large Language Models for Code Intelligence Tasks

Author:

Canran Liu (SID: 1155157250)

Xingyun Ma (SID: 1155157282)

Supervisor: Michael R. Lyu

Supervising TA: Wenxuan Wang

Acknowledgement

We would like to express the deepest gratitude to our supervisor, Professor Michael R. Lyu, and our advisor Mr. Wenxuan WANG, for their invaluable guidance and support throughout this Final Year Project.

Contents

1 Introduction

1.1 Background

Code generation tasks have garnered considerable attention from researchers and have evolved significantly since the introduction of Large Language Models (LLMs). There are many different perspectives in exploring LLM-based code generation. For instance, code readability, code accuracy, and code robustness. One of the most important metrics to measure is the efficiency of the code. Ensuring efficiency is a crucial aspect of programming, particularly when computational resources are limited or the program is utilized at a large scale[13].

The vast majority of existing research on LLM-based code generation focuses only on exploring the accuracy and the readability of the code. In these few studies that discuss the efficiency of generating code, Mandaan et al. and Chen et al. in their studies used fine-tuning CODEGEN and training seq2seq models respectively, and achieved impressive improvements in code efficiency [5, 13]. However, their approach requires a lot of computational resources and a massive code dataset to support model training. How to enable users who lack computing resources to realize the efficiency improvement of the code becomes a research topic worth pondering.

Inspired by many previous outstanding studies using LLM-based self-refinement and multi-agent collaboration frameworks to improve code accuracy [4, 8, 14, 15, 17], we would like to propose a novel LLM-based self-refinement and multi-agent collaboration

framework, since relatively small API charges were required to implement methods in their study. Our framework can achieve similar performance as previous studies, which can significantly improve code efficiency, under limited computational resources and training datasets.

Moreover, no existing benchmark can systematically measure the efficiency of code generation tasks. We would like to create a benchmark that makes it possible to measure the efficiency of different code generation methods under the same scale.

1.2 TimeEval

To achieve the aforementioned goals, We have introduced a benchmark named timeEval, rooted in two datasets: the APPS[7] and the CodeContests[12]. TimeEval is an abbreviation for time evaluation. The specific benchmark build process is described in detail in Section **??**.

The following components are included in the timeEval benchmark:

- **Problem set of size 111.** The problem set comprises 111 questions designed to assess the efficiency of generated code. These problems frequently admit multiple solutions, and opting for a different approach can lead to significant variations in execution time. These differences in execution time are usually caused by differences in the time complexity of the algorithms.
- **Canonical solution for each problem.** We provide an optimal solution for each

problem. So when exploring the efficiency of the generated code, the efficiency can be confirmed by comparing the execution time with that of the optimal solution. It's important to note that the term **optimal solution** in this context doesn't denote absolute optimality. Rather, a solution is deemed optimal if it attains the best possible time complexity and successfully passes all test cases. Consequently, we refer to it as the **canonical solution**.

- **Correct but slow solution for each problem.** We also provide a correct but less efficient solution for each problem, which is generated by the 'gpt-3.5-turbo-0613' model. And we ensure that this solution passes all the test cases, while it is less efficient compared to the canonical solution. We offer this solution firstly to demonstrate that all problems in our dataset have room for improvement in terms of code efficiency, and secondly to facilitate researchers in tasks such as code optimization or experiments like self-refinement that require original inputs.
- **Test cases for each question.** We prepared 30 test cases for each problem, which contained some very small-sized cases to check the correctness of the code, and some very large-sized cases to highlight the difference between the generated code, which has a large time complexity, and the canonical solution.
- **A framework for automated measurement of code efficiency.** We provide an automated code framework in benchmark to comprehensively measure the efficiency of the generated code.

1.3 Empirical Study of Code Efficiency

After establishing the timeEval benchmark, we first conducted an empirical study on the efficiency of generated code on our benchmark. In this empirical study, we focused on analyzing the performance in code efficiency of two frameworks: self-refinement and multi-agent collaboration. This successfully addresses the current research gap concerning the efficiency of generated code. The design and discussion of the experiments are detailed in Section 5.

1.4 New Framework for Code Efficiency

After conducting an empirical study, we identified several issues within the existing frameworks. To address these issues, we developed our own framework, which achieves a balance between improving code efficiency and accuracy. This tradeoff is crucial for enhancing the performance of code generation systems. The design and discussion of the experiments are detailed in Section 6.

1.5 Summary of Contributions

The following are some of the main contributions we have made during this semester's project.

- **Propose metrics.** Propose a set of metrics that comprehensively evaluate the execution efficiency of generated code.

- **Measure and process code_contests dataset.** We finished in-depth testing of the code_contests dataset. In total, we tested more than 13,000 problems with 520,000 corresponding solutions and 780,000 corresponding test cases. Proposed a reasonable process for processing the dataset, and filtered out the problems that were difficult to be solved efficiently by the gpt-3.5-turbo model through this process.
- **Improve timeEval benchmark.** We have successfully extended last semester's TimeEval benchmark from monolingual Python to encompass multilingual support for Python, C++, and Java. Additionally, we've developed a comprehensive automated code execution framework, marking it as the first multilingual code efficiency benchmark of its kind that we have known of.
- **Propose multiple frameworks.** We explore strategies for improving the efficiency of generated code from several perspectives. These include, but are not limited to: simple prompt engineering, chain-of-thought, In-context learning, self-refinement, and multiagent collaboration frameworks. We first migrated these frameworks to the code efficiency task and then improved them with encouraging results.

2 Related Works

2.1 Large Language Models for Code

With the continuous development of LLMs, their applications have expanded broadly across various domains. Among these, research in the area of LLMs for coding tasks has emerged as a popular topic, encompassing tasks such as code understanding, code completion, code generation, and code repair. Notably, the LLM CodeX, pioneered by Chen et al[2]., has achieved remarkable results in code-related tasks, leading to a surge of commercial products such as GitHub Copilot, as well as numerous coding models including StarCoder[10] and Code LLaMA[16][21]. Moreover, general large language models like ChatGPT, which are not solely focused on coding, also exhibit exceptional performance in coding tasks. Figure 1, taken from a survey on language models for code written by Zhang et al.[21], illustrates the current major types of language models for code and their typical representatives.

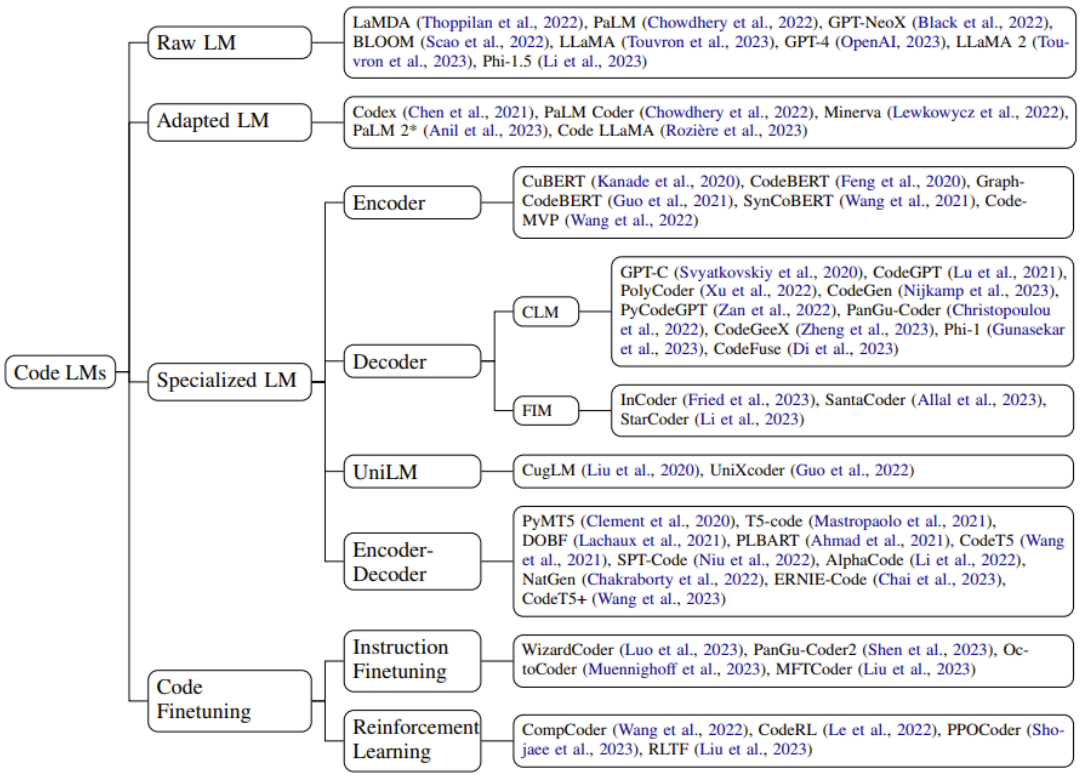

Figure 1: The overview of current language models for code[21]

2.2 Coding Benchmark for LLMs

Existing code benchmarks primarily focus on assessing the functional correctness of code. For instance, the most popular LLM code benchmark currently is the HumanEval dataset, proposed by Chen et al. in 2021[2]. It consists of 164 hand-crafted Python programming problems, primarily testing language understanding, reasoning, algorithms, and simple mathematical problems. Each problem is accompanied by several test cases and a canonical solution, allowing for the evaluation of the code's functional correctness. APPS is another Python coding dataset, comprising 10,000 coding problems, 131,836 test cases for checking solutions, and 232,444 human-written actual solutions, aimed at measuring

coding skills and problem-solving abilities[7]. There are also datasets for testing other languages, such as HumanEval-X[22], which includes Python, C++, Java, JavaScript, and Go, and the WikiSQL dataset for evaluating SQL[23], among others. However, we have observed that most current code datasets focus on the functional correctness of code generated by LLMs or LLMs' ability to understand text and code, lacking a dataset that can assess code efficiency. Therefore, we propose a new benchmark aimed at testing and evaluating the efficiency of code generated by LLMs.

2.3 Self-refinement

SELF-REFINE is a framework proposed by Madaan et al.[14], aiming to imitate human thinking to enable LLMs to improve their outputs through iterative feedback. The core concept of SELF-REFINE is to obtain an initial output generated by LLM and then make LLM provide feedback on its initial output; finally, the LLM refines its previous output based on its own feedback.

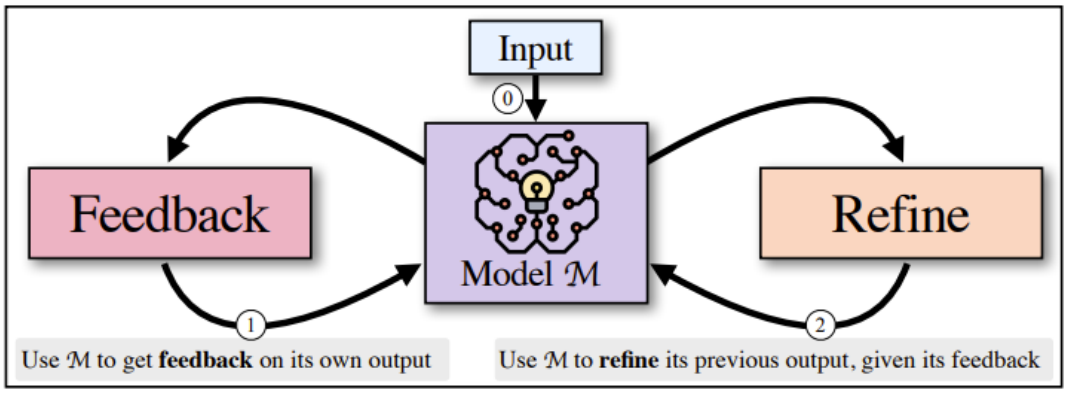

Figure 2: The process of SELF-REFINE[14]

The primary process of SELF-REFINE is depicted in Figure 2. It includes two iterative workflows: 'Feedback' and 'Refine'. Initially, the model generates an output based on the prompt, which is then fed back to the model, followed by obtaining feedback on this original output. This feedback is then provided to the model to refine the initial output. This process can be iterated multiple times to achieve optimal results, simulating the process of human thinking and correcting errors. The results indicated that in all seven different tasks, the outcomes generated by SELF-REFINE were an improvement over those produced directly by GPT-3.5 and GPT-4.

In the empirical study, we followed the experimental setup of the SELF-REFINE framework proposed by Madaan et al[14] to implement this framework on our dataset timeEval. This framework, which allows the model to think and adjust, inspired us and enabled us to propose our own framework.

2.4 LLM-based Multi-Intelligent Agents Collaboration

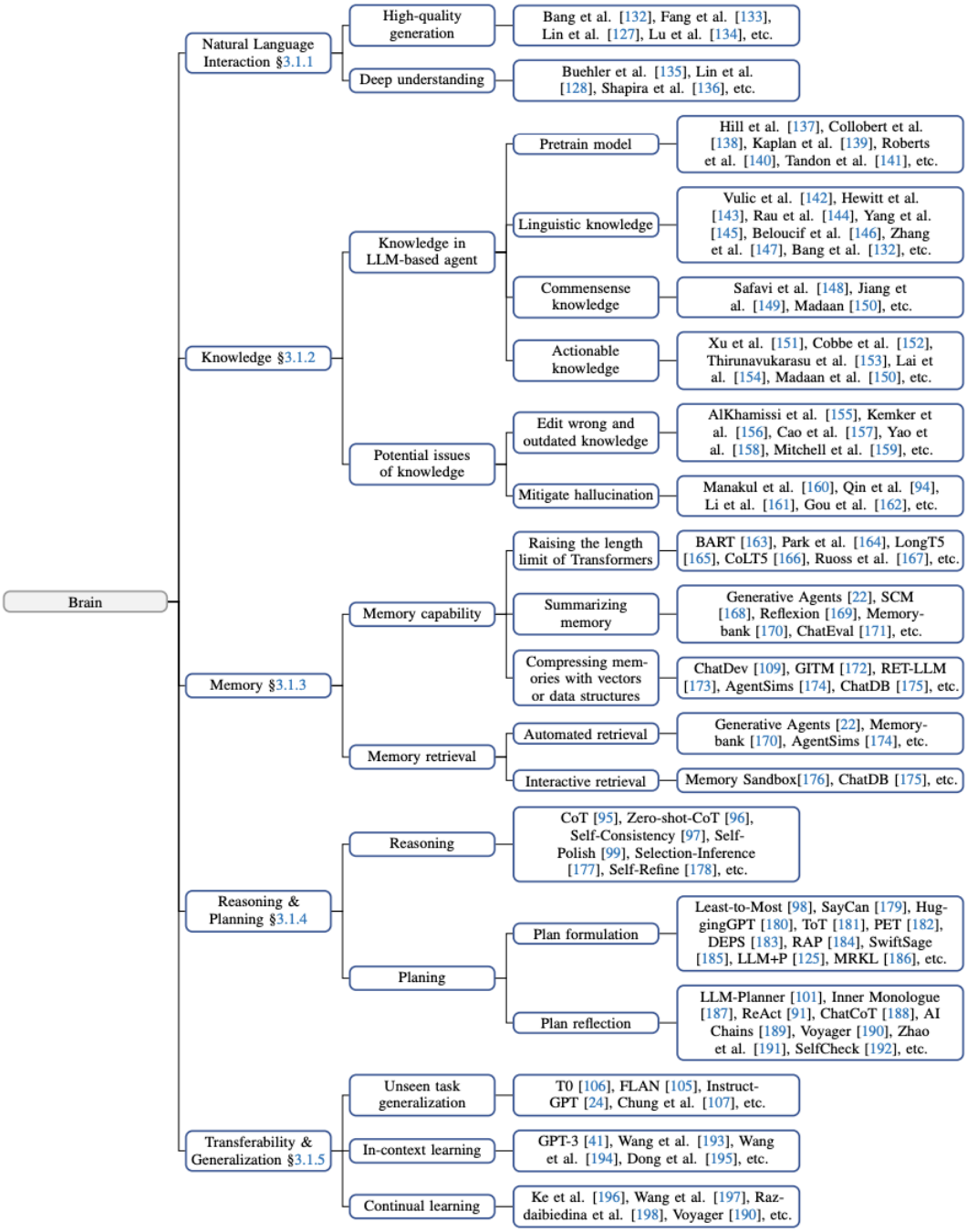

Figure 3: Related work on using LLMs as brains for agents[20].

LLMs possess characteristics of autonomy, reactivity, and pro-activeness, making them exceptionally well-suited to serve as the main part of the brains of AI agents [20]. As shown in Figure 3, there is a large amount of work that employs LLMs as a brain for intelligent agents. Among them, code generation task using LLM-based agents is a highly promising work. In certain studies, these forms of intelligence iteratively refine each other's actions over multiple conversational rounds. Notably, concerning the code generation task, previous research in references [3] and [8] addressed the critical coding aspect of software development by employing LLMs in the capacities of Manager, Tester, and Programmer. It's worth noting that the majority of existing research tends to involve the agent's role in software engineering development, with limited exploration of LLM-based agents aimed at enhancing the efficiency of specific code. There is still a gap in research on using multi-agent collaboration to improve code efficiency.

2.5 Prompt Engineering

The generation of outputs by large language models is fundamentally a process of predicting the next token, and this prediction is based on the prompt provided by the user. Therefore, the prompt is crucial to the model's output. Prompts can control the content generated by the model, and guide it to produce specific outcomes, and optimizing prompts can enhance the accuracy and efficiency of the model's outputs. Consequently, prompt engineering has emerged as a popular research trend. Here, we introduce several prompt methods that will be utilized in our subsequent experiments.

Zero-shot Prompt[1]

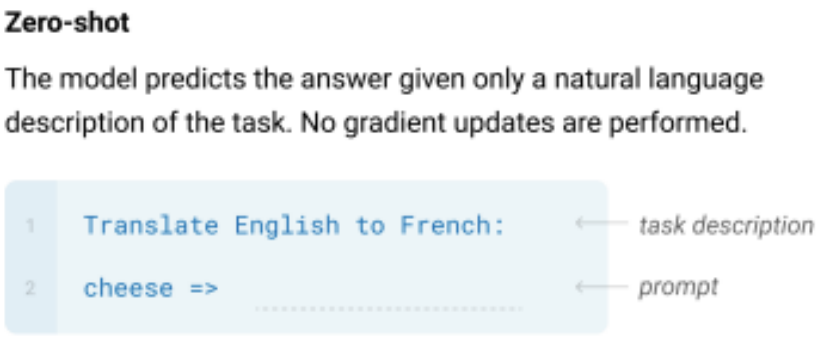

Figure 4: Example of zero-shot[1]

As illustrated in Figure 4, zero-shot learning involves providing the model directly with the task description without any examples. In this scenario, the model only relies on the task description to infer the answer. The advantage of zero-shot learning is its high flexibility. However, a drawback is that the model may not grasp the subtle nuances of some tasks, leading to answers that are either inaccurate or overly general.

One-shot Prompt/ Few-shot Prompt[1]

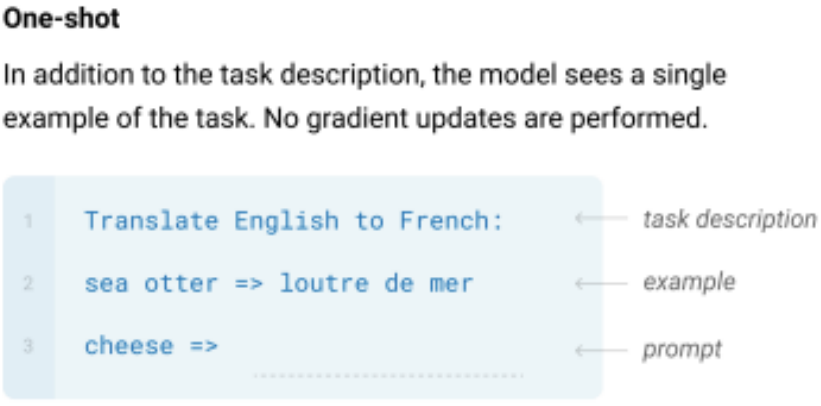

Figure 5: Example of one-shot[1]

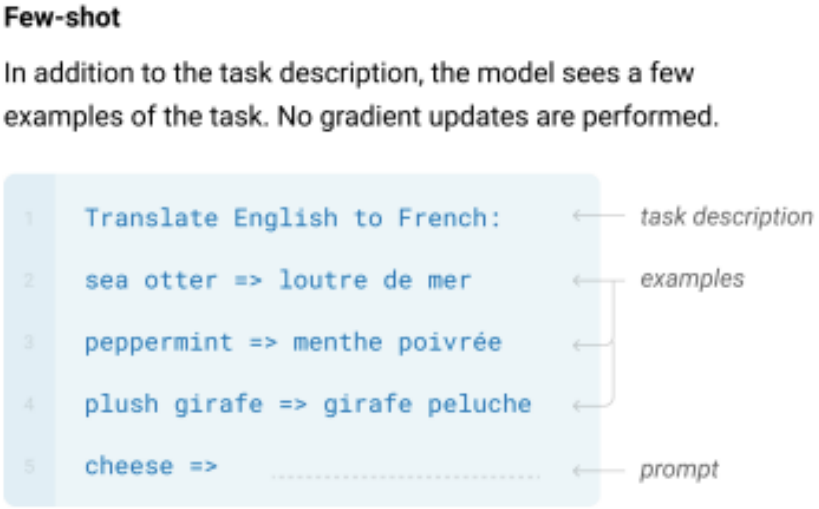

Figure 6: Example of few-shot[1]

One-shot learning involves providing the model with a single example along with a task description (as shown in Figure 5), while few-shot learning (as illustrated in Figure 6) entails presenting the model with multiple examples and a task description. These examples typically include both inputs and expected outputs, enabling the model to better understand the task requirements. Therefore, the outcomes of one-shot and few-shot learning are often superior to those of zero-shot learning. However, a downside is the potential to reach the limits of input and output length.

Chain of Thought (CoT)[19]

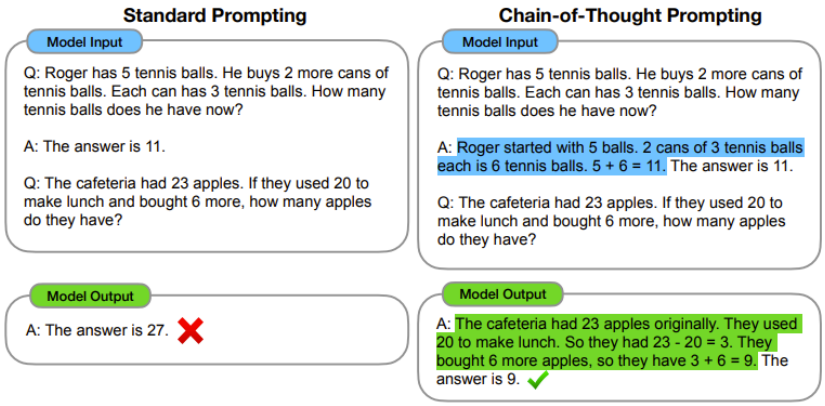

Figure 7: Example of CoT[19]

Chain of Thought (CoT) refers to the process of human thinking. Applying the CoT concept to language models can encourage the model to reason about questions, allowing it to decompose a complex problem into multiple intermediate steps to obtain more accurate answers. CoT is often used in conjunction with one-shot/few-shot learning, as its principal idea is to provide the model with examples that include explanatory reasoning processes. Consequently, the model tends to simulate this reasoning process in the examples. This type of reasoning often leads to more accurate results. Figure 7 illustrates an example of CoT, where the addition of a reasoning process in the example changes the model's answer from incorrect to correct.

2.6 Fine-Tuning

Fine-tuning is a common method for enhancing the performance of LLMs. Many studies have explored the effectiveness of fine-tuning LLMs. Particularly, in the paper 'Learning Performance-Improving Code Edits'[18], researchers utilized their own PIE dataset to

fine-tune pre-trained models, aiming to achieve better performance in code optimization. They addressed data imbalance issues during the fine-tuning process and implemented strategies such as introducing high-quality data subsets and performance tags. Their findings suggest that fine-tuning LLMs can effectively improve their performance in code optimization tasks. However, it's important to note that the fine-tuning process may face challenges, especially regarding its high costs, particularly with large-scale datasets or when numerous iterations are needed. Therefore, in practice, researchers need to balance the relationship between the performance benefits of fine-tuning and its associated costs to determine whether it is worthwhile.

One of the main reasons for our choice not to use fine-tuning is that other studies have already explored this method in terms of code optimization. Thus, we aim to explore different approaches to enhance the efficiency of LLMs for code. Additionally, the relatively high cost of fine-tuning, along with the high requirements for dataset quality and balance, presents certain challenges for us. Hence, we opted to use other different strategies to improve the efficiency of the code generated by LLMs.

2.7 Code Efficiency

Code efficiency is another critical aspect of code quality, in addition to correctness. We typically analyze it from two dimensions: time and space, represented by time complexity and space complexity, respectively. We employ a universal method, the Big O notation, to measure and describe complexity. The definition of Big O notation is as follows: A

function $g(n)$ is said to be $O(f(n))$ if there exist positive constants c and n_0 such that

$$0 \leq g(n) \leq c \cdot f(n) \quad \text{for all } n \geq n_0$$

In this case, we write $g(n) = O(f(n))$.

This allows us to ignore factors like hardware and analyze code efficiency more fundamentally and intuitively. There is an inherent trade-off between time complexity and space complexity. Generally, it is not possible to optimize both simultaneously. However, since space complexity is more challenging to assess and measure automatically, our research focuses on evaluating and improving the time complexity of code.

3 Dataset Processing

3.1 Select the Dataset

Last semester, we presented the Python code efficiency benchmark timeEval. This semester, we started to extend timeEval to a multilingual dataset. First, we have to pick a suitable dataset as a cornerstone. The data will be processed and filtered and then added to timeEval.

The code_contests dataset consists of 13328 samples in the training set, 117 samples in the validation set, and 165 samples in the test set. Additionally, the code_contests dataset covers multiple programming languages, including Python 2, Python 3, Java, and

C++[11]. In addition, the questions in code_contests are derived from programming competitions, which are generally answered by different algorithms, and the differences in these algorithms often bring about differences in time complexity and thus differences in code execution time.

To summarize, the codeContests dataset offers the advantages of large sample size, support for multiple programming languages, and challenging questions. Therefore, we have selected this dataset as the foundation for processing and filtering. The processing flow is as follows.

3.2 Processing Flow

First, we realized that we would face several problems if we randomly selected some questions directly from APPS as our dataset to evaluate the efficiency of the generated code.

- **Uneven number of test cases.** Certain problems within the dataset feature a limited number of test cases (less than 10), and these cases may involve only small-sized inputs. In such instances, the disparity in execution time between algorithms with varying efficiencies, such as different time complexities, is not notably significant. This can potentially impact the accuracy of the measurement results.
- **The quality of the problem varies.** Certain problems within the dataset are unsuitable for exploring code efficiency. Typically, these problems offer only one

fixed solution, leaving no room for potential efficiency enhancements.

- **The quality of the solutions in the dataset varies.** In the solutions that come with the dataset, certain solutions employ algorithms with high time complexity, making them unsuitable for evaluation as the ground truth for generated code.

Due to these considerations, in the initial phase of the process, For Python3, C++, and Java, we conducted tests on the first 20 self-contained solutions within the dataset, which comprises 13,328 problems. We excluded problems where the number of test cases was less than 10 or none of the self-contained solutions could pass all the test cases. In addition, we have also removed languages that only support Python2, as we consider Python2 to be a niche language nowadays. After completing this step, we can ensure that all remaining problems have a correct solution in at least one language and that there are enough test cases for each problem.

Following that, we measured the time and manually analyzed the solutions for each problem across all test cases to confirm optimal time complexity. After filtering out these solutions with optimal time complexity. We then select the solution with the shortest specific execution time from them. We ensured that the selected solution exhibited the least execution time among all the solutions tested for that specific problem. After this step, we are guaranteed to have at least one correct solution with optimal time complexity for the remaining problems, and this solution will be used as the canonical solution in our benchmark. This canonical solution plays a role similar to ground truth in our benchmark. We will execute the canonical solution and compare it with the

execution time of the generated code to discuss the efficiency of the generated code.

	A	B	C	D	E	F	G
1	problem	passed tests	wrong answers	time limit exceeded	gen_time	opt_time	opt_time/gen_time
3	80	3	0	7	105.06	0.13	0.001
7	323	2	0	8	120.06	0.14	0.001
9	396	4	0	6	108.6	0.14	0.001
12	457	3	0	7	105.11	0.14	0.001
16	854	2	0	8	120.07	0.13	0.001
18	1038	3	0	7	105.06	0.14	0.001
19	1088	2	1	7	105.07	0.15	0.001
20	1275	2	0	8	120.07	0.15	0.001
21	1326	3	0	7	105.07	0.14	0.001
22	1526	1	0	9	135.05	0.14	0.001
23	1718	1	0	9	135.06	0.15	0.001
24	4167	2	0	8	120.07	0.15	0.001
25	4237	3	0	7	105.08	0.15	0.001
27	267	5	0	5	75.13	0.14	0.002
28	309	6	0	4	60.14	0.14	0.002
29	486	5	0	5	75.09	0.14	0.002
30	739	4	0	6	90.08	0.14	0.002
31	773	3	1	6	90.07	0.16	0.002
32	806	5	0	5	75.08	0.13	0.002
33	866	5	0	5	77.93	0.15	0.002
34	1007	3	0	7	117.07	0.21	0.002
35	1124	3	0	7	105.07	0.23	0.002
36	1126	4	0	6	91.22	0.16	0.002
37	1175	5	0	5	75.09	0.14	0.002
38	1528	4	1	5	75.09	0.14	0.002
39	1810	6	0	4	60.33	0.15	0.002
40	1906	5	0	5	75.44	0.14	0.002
41	1925	6	0	4	60.1	0.14	0.002
42	175	7	0	3	45.15	0.13	0.003
43	246	7	0	3	45.11	0.13	0.003
44	1317	8	0	2	54.75	0.14	0.003
45	1428	4	0	6	90.09	0.31	0.003
46	1859	6	0	4	60.6	0.16	0.003
47	3798	5	0	5	75.12	0.23	0.003
48	3815	6	0	4	60.15	0.2	0.003
49	4154	6	0	4	69.66	0.2	0.003
50	4196	5	0	5	75.1	0.26	0.003
51	321	7	0	3	45.15	0.16	0.004
52	669	6	0	4	60.1	0.24	0.004
53	1087	3	0	7	105.08	0.43	0.004
54	81	8	0	2	30.38	0.15	0.005
55	88	8	0	2	30.15	0.14	0.005

Figure 8: Screenshot of some of the data in the flow of processing the APPS dataset.

Next, we employed the gpt-3.5-turbo model to generate code for the remaining problems. We assessed both the time and accuracy of these generated codes. In detail, we measured the execution results and execution times of the generated code for each test case in every problem. The execution results were categorized into three types: correct input/output correspondence (passed tests), incorrect input/output correspondence (wrong answers), and timeout (time limit exceeded). In this filtering round, we utilized the ratio of the execution time (opt_time) of canonical solutions to the execution time (gen_time) of the

generated solutions, ordering them in ascending order, as depicted in Figure 8. A smaller value of opt_time/gen_time indicates a larger gap between the execution time of the gpt-3.5-turbo generated code and the execution time of the optimal solution. We also make a preliminary inference based on this metric that the code generated by LLMs is less efficient on these problems. So we picked the problem in which opt_time/gen_time was in the interval $[0, 0.5]$.

Furthermore, we corrected erroneous considerations in how we handled the data last semester. We aimed to ensure that the chosen questions underwent modeling with gpt-3.5-Turbo to generate accurate yet inefficient code. In the prior semester, our selection process focused on questions with fewer than 20% wrong cases, which lacked rigor. According to the academic definition of code pass or fail, the presence of just one failed test case deems the code incorrect. Therefore, in the processing procedure, we eliminated all questions associated with incorrect answers. Moreover, we apply the same filtering criteria to our previous TimeEval data. Subsequently, the filtered data from the code_contests dataset and the old timeEval dataset are combined to create a new dataset.

In this new dataset, we are able to guarantee that all problems are gpt-3.5-turbo can directly generate correct but inefficient code, thus allowing us to more accurately measure the impact of the code generation framework on efficiency as well as correctness. The statistics of the dataset are shown in the table below:

Supported Language	Number of Problems
C++ only	52
Java only	18
Python only	32
Python and C++ only	1
Java and C++ only	7
C++, Java and Python	1
Intotal	111

Table 1: Supported Languages and Number of Problems

Note: Ours aims to provide a dataset that is challenging on code generation tasks. We expect that using the gpt3.5-turbo model directly on our dataset will generate slow but correct code. This allows for a more systematic measurement of methods to improve code efficiency. For these reasons, a situation may arise where the same problem occurs in the Python language that meets our filtering criteria, but not in the Java and C++ languages. We then use the problem only as data for testing code efficiency in the Python language.

3.3 Updated Dataset Structure

After the update, we have a total of 111 samples inside the dataset. The file structure of each of these samples is shown below.

```
├── question.txt
├── canonical_solution.cpp
├── canonical_solution.java
├── canonical_solution.py
├── input_output.json
└── metadata.json
```

canonical_solution.language denotes the ground truth solution for the corresponding programming language, as depicted in Figure 10. The content in the file **input_out.json**

are pairs of input and output test cases. **metadata. json** file has detailed data about the source of this question, and the types of programming languages supported. As shown in the example in Figure 9, the problem supports three languages C++, Java, and Python.

```
{
  "name": "p02470 Euler's Phi Function",
  "source": 6,
  "difficulty": 0,
  "cf_contest_id": 0,
  "cf_index": "",
  "cf_points": 0.0,
  "cf_rating": 0,
  "cf_tags": [
    ""
  ],
  "is_description_translated": false,
  "untranslated_description": "",
  "time_limit": {
    "seconds": 1,
    "nanos": 0
  },
  "memory_limit_bytes": 134217728,
  "language": [
    "C++",
    "Java",
    "Python"
  ],
  "dataset_source": "CodeContests_03751"
}
```


Figure 9: metadata.json in question 157 in the timeEval dataset

```
#include<iostream>
#include<map>
#define itr(it,a) for(auto it=(a).begin();it!=(a).end();++it)

std::map<int, int> prime_factor( int n )
{
        std::map<int, int> ret;
        for( int i = 2; i*i <= n; ++i ) while( n % i == 0 )
        {
                ++ret[i];
                n /= i;
        }
        if( n != 1 )
                ++ret[n];
        return ret;
}

int phi( int n )
{
        auto m = prime_factor(n);
        itr( it, m )
        {
                n /= it->first;
                n *= it->first-1;
        }
        return n;
}

int main()
{
        int n;
        std::cin >> n;
        std::cout << phi(n) << std::endl;
        return 0;
}
```

Figure 10: canonical_solution.cpp in question 157 in the timeEval dataset

4 Benchmark Creation

In addition to the dataset, we added a framework for measuring the efficiency of generated code to the timeEval benchmark.

4.1 Code Execution

Last semester, our dataset contained only Python, but now our dataset supports not only Python but also C++ and Java. Accordingly, we have added frameworks for executing C++ and Java programs to our benchmark.

As shown in figure 11. After running the timeEval benchmark's test framework, the test framework asks the user for the language to be tested. After the user enters the language to be tested, the code execution framework automatically detects the metadata.json file in each problem in the dataset. If a problem corresponds to the metadata.json file that exists in the language, it means that the problem supports that language. The generated solution and canonical solution of the problem will be executed.

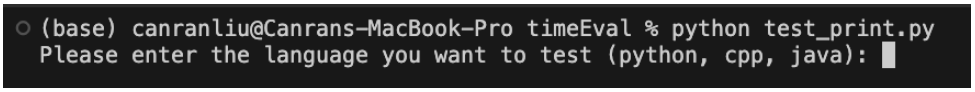

Figure 11: Test framework inquiry.

Python code execution Since Python is an interpreted language, executing Python does not require compilation. Python codes will be executed directly using *subprocess.check_output()* within the execution framework.

C++ code execution C++ code needs to be compiled for execution. We took g++ to

compile the C++ code. The compile command is executed by *subprocess.run()*. The relevant code snippet is shown in the figure 12. If compiled successfully, the *output_executable* file generated by the compilation will be executed using *subprocess.check_output()*. If the compilation is unsuccessful, the compilation error message will be displayed in the result.

It is worth noting that in order not to take up too much storage space. The output_executable file is deleted by the code execution framework after each execution. Moreover, the compilation time will not be counted as part of the program run time.

```
compile_command = ['g++', script_path, '-o', 'output_executable']
run_command = ['./output_executable']
compile_process = subprocess.run(compile_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
```

Figure 12: C++ code compilation

Java code execution Java code not only needs to compile but also to ensure that the name of the public class and the file name are the same. So the implementation of the framework to automatically execute the Java code process is more complex. We used the following strategy to execute the Java program:

First, as shown in Figure 13. We designed the *get_public_java_class()* function to detect the name of the public class in each .java file. Then copy the original .java program file to a temporary folder *tmp*. And change the name of the .java file to the public class name we got in the previous step. We then compile the Java file using a Java virtual machine and execute the compiled executable.

Similar to the execution of a C++ program, we delete the *tmp* folder after execution to

reduce the storage footprint. Also, when executing the framework execution to calculate the code runtime, the time spent on copying, compiling, etc. will not be counted.

```python
def get_public_java_class_name(java_source_file):
    with open(java_source_file, 'r') as file:
        content = file.read()

        # Modified regex to specifically match public class declarations
        match = re.search(r'\bpublic\s+class\s+(\w+)', content)

        if match:
            return match.group(1)
        else:
            return None
```

Figure 13: Get the public class name.

Our measure of the generated code's efficiency requires comparing the generated code's running time with the optimal solution's running time. To ensure that they run on the same hardware environment, e.g., CPU, we execute the canonical solution and the generated solution for each measurement and generate a report(shown in figure 14). If the result is correct, we label the case as *True*. If the result is incorrect, it is marked as *False*. In the case of a timeout, it is labeled as *Timeout*. Users employing our benchmark have the flexibility to adjust the value of the timeout parameter directly from the command line. Moreover, if the program has a compilation error, then all results will show a compilation error and the output will be empty.

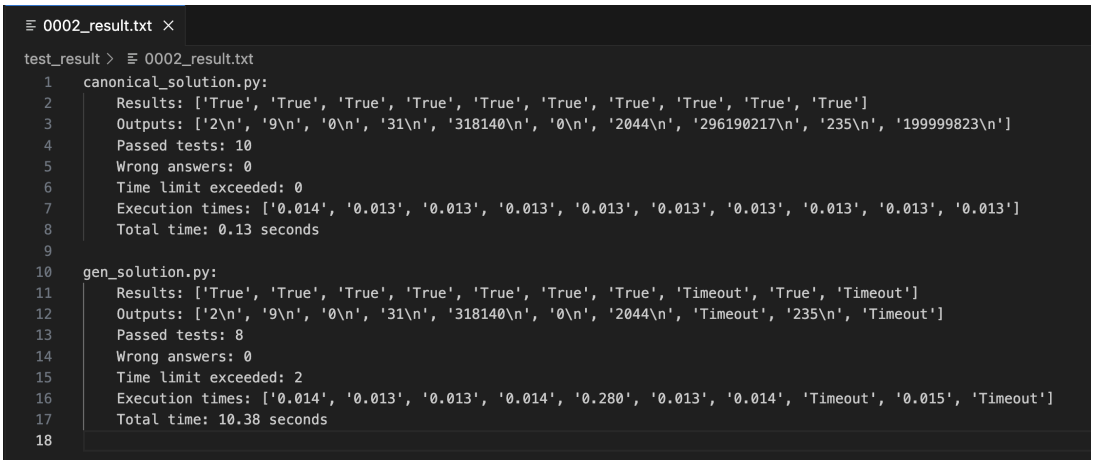

Figure 14: TimeEval performs the case-by-case evaluation of the generated code.

4.2 Metrics and Evaluation Framework

Unlike last semester's less rigorous metrics, we have updated our metrics for code efficiency this semester. Following the code execution in the previous step, we obtain a comprehensive record of the generated code's execution. Subsequently, our evaluation framework translates these statistics into concise quantitative metrics. In terms of metrics to measure the accuracy of the code, we have taken the metric *Pass@k*, which is commonly used in academia[2].

The five metrics we set up are as follows:

- **Total Time** TT: Total time(TT) measures the average time it takes for the generated code to finish executing all test cases. In our experiments, each time we encountered a timeout case, we added five seconds of penalty to the TT of that program. TT is defined as:

$$TT = \frac{1}{N} \sum t_{\text{gen}}$$

This metric allows the user to roughly determine the overall execution time of the code.

- **Efficiency Level** (%EL): In a problem, find the total execution time of the generated code on all passed test cases and find the total execution time of the optimal solution on the corresponding test cases. Efficiency Level (EL) is the ratio of the latter to the former. In our experiments, we used the average EL of all generated programs as a metric.

  Let's assume in the kth problem, there are n passed test cases, where each test case can be divided into two scenarios: the execution time of the generated code, denoted as G_i, and the execution time of the corresponding optimal solution, denoted as O_i. Then, two sets can be defined:

  $$G = \{G_1, G_2, ..., G_n\}$$

  $$O = \{O_1, O_2, ..., O_n\}$$

  Then, the Efficiency Level (EL) can be defined as the ratio of the total execution time of the optimal solution to the total execution time of the generated code:

  $$EL_k = \frac{\sum_{O_i \in O} O_i}{\sum_{G_i \in G} G_i}$$

  $$\%EL = \frac{1}{N} \sum_{K=1}^{N} EL_k * 100\%$$

  This metric allows us to determine the efficiency gap between the generated code

and the optimal solution. As this metric approaches 100, the efficiency of the generated code approaches that of the optimal code; conversely, as it approaches 0, the efficiency of the generated code diminishes relative to the optimal code.

- **Timeout Rate** %TR: Percentage of timeout test cases out of all test cases. We use this metric to see what percentage of test cases will make the generated code unable to complete execution within the time limit.

- **Pass@1**: The metric was proposed by Chen et al. [2] to measure the accuracy of the code and is defined as follows:

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

  Where c is the number of correct codes passed in all the test cases.

- **optimal solution ratio** %Opt: Proportion of programs where the execution time of the generated code is close enough to the execution time of the optimal solution in the test set (Canonical solution). That is, the code execution time that satisfies the equation,

$$\frac{t_{\text{gen}} - t_{\text{opt}}}{t_{\text{opt}}} < \theta$$

  where t_{gen} represents the execution time of generated code, t_{opt} represents the execution time of optimal code and θ represents the threshold. The execution time is defined as close enough when the LHS is less than the threshold. In our experiments, we set θ to 0.5.

Our benchmark will automatically measure the above metrics and provide feedback to the user. In addition, the user has the option of exporting the test results to Excel for more detailed analysis. (Shown in Figure 15) In the table, the user can observe some additional information, such as the problem index, the number of passed tests, the number of wrong answers, and the number of time limits exceeded.

	A	B	C	D	E	F	G	H	I	J
1	problem	passed tests	wrong answers	e limit exceed	opt_time	TT	EL	TR	Pass@1	Opt
2	98	30	0	0	1.13	11.89	0.095	0	1	0
3	99	30	0	0	0.85	1.28	0.667	0	1	0
4	111	0	30	0	1.3	9.19	0	0	0	0
5	116	1	29	0	0.87	1.61	0.508	0	0	0
6	118	30	0	0	1.4	1.38	1	0	1	1
7	124	6	18	6	1.01	36.41	0.552	0.2	0	0
8	133	30	0	0	0.83	1.26	0.662	0	1	0
9	135	0	30	0	0.82	1.27	0	0	0	0
10	136	30	0	0	0.89	1.27	0.703	0	1	1
11	140	30	0	0	0.82	1.33	0.615	0	1	0
12	145	30	0	0	0.84	1.3	0.651	0	1	0
13	157	30	0	0	0.83	1.92	0.434	0	1	0
14	172	30	0	0	1.12	11.85	0.094	0	1	0
15	184	0	30	0	0.83	1.27	0	0	0	0
16	185	0	30	0	0.87	1.3	0	0	0	0

Figure 15: An example of generated Excel form.

5 Empirical Study on Code Efficiency

5.1 Self-refine

5.1.1 Overview

To explore the practicality and effectiveness of self-refinement, we conducted an empirical investigation based on the self-refinement framework proposed in the paper “SELF-REFINE: Iterative Refinement with Self-Feedback”[14]. This framework is shown in the

figure 16 and mainly includes two steps: feedback and refinement. The main process is as follows: First, the model generates an initial output based on the given initial input; then, this output is fed back to the model to obtain feedback for improvement; next, the output is refined based on the feedback. This process is iterated continuously until it meets specific stopping criteria for the iteration.

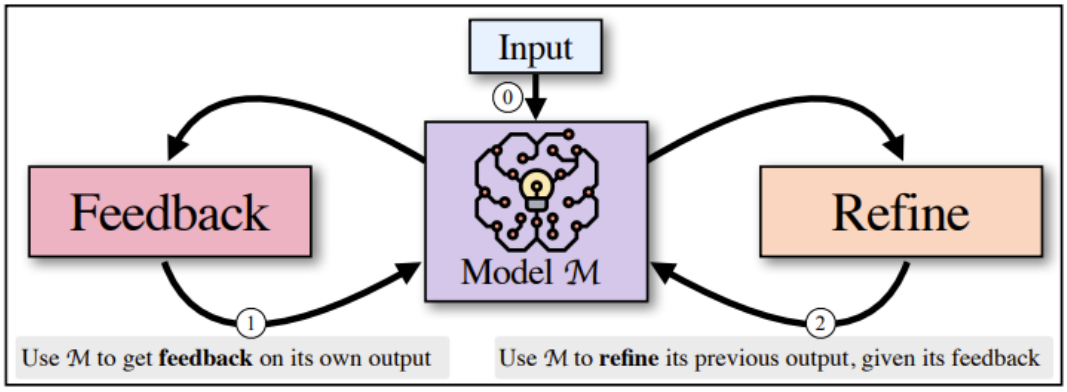

Figure 16: The process of SELF-REFINE[14]

5.1.2 Experimental Details

Framework and Process

While respecting the original experimental setup in the paper, we modified the framework according to the structure of timeEval to suit our dataset. The specific process is as follows:

- **Initialization Phase:** The model is first provided with a correct yet slow version of code, and it is tasked to directly generate an optimized version of this code.
- **Feedback Phase:** Then, the optimized version of code is given back to the model to obtain feedback.

- **Refine Phase:** After receiving the feedback, it is again passed back to the model for further improvements based on the feedback.

The feedback and refine steps are iteratively continued until the stopping conditions. There are two conditions for stopping the iteration: one is when the model's feedback indicates that the code is efficient enough, and the other is reaching the maximum number of iterations, which is set to four as per the experimental setup in the paper. For the first condition, our marker is to check whether the feedback included the words "is not slow."

Prompt

Madaan et al[14] also use the few-shot prompt approach to enhance the model's performance. Few-shot prompts were applied during the initialization and feedback phases. More specifically, the prompts used in the experiment are as follows:

- **Initialization Prompt:**

  # slower version:

```
a, b = input().split()
n = int(a + b)
flag = False
for i in range(n):
    if i ** 2 == n:
        flag = True
        break
print('Yes' if flag else 'No')
```

  # optimized version of the same code:

```
a, b = input().split()
```

```
n = int(a + b)

flag = False
for i in range(1000):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')
```

END

More examples...

 # slower version:

{*The correct but slow code provided by timeEval*}

 # optimized version of the same code:

- **Feedback Prompt:**

```
a, b = input().split()
n = int(a + b)

flag = False
for i in range(n):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')
```

 # Why is this code slow?

```
# This code is slow because it is using a brute force
   approach to find the square root of the input number.
   It is looping through every possible number starting
   from 0 until n. Note that the sqare root will be
   smaller than n, so at least half of the numbers it is
   looping through are unnecessary. At most, you need to
   loop through the numbers up to the square root of n.
```

END

More examples...

{*The correct but slow code provided by TimeEval*}

 # Why is this code slow?

- **Refine Prompt:**

  {*The correct but slow code provided by TimeEval*}

  # Why is this code slow?

  {*Feedback from the model*}

  How to improve this code? Please provide the improved version of the code.

5.1.3 Results Analysis

The experimental results (shown in Table 2, 3 & 4) showed that, compared to the baseline model, the code after self-refinement significantly improved in efficiency. Both the Efficiency Level and %opt, the metrics used to measure efficiency, showed significant improvement, indicating that the self-refinement framework is effective in enhancing

code efficiency. However, the drop in accuracy is also obvious. To address this, we conducted case studies to explore potential reasons for the decrease in accuracy.

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
Self-refinement	Python	7.0	41.9	2.5	61.8	11.8

Table 2: Experimental Results of Self-refinement in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
Self-refinement	C++	4.3	63.4	2.2	52.5	37.7

Table 3: Experimental Results of Self-refinement in C++

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0.0	100.0	3.8
Self-refinement	Java	7.5	30.7	2.6	46.1	7.3

Table 4: Experimental Results of Self-refinement in Java

Case Studies

A specific example is question 71 in the dataset. The question is: *"Once Max found an electronic calculator from his grandfather Dovlet's chest. He noticed that the numbers were written with seven-segment indicators. Max starts to type all the values from a to b. After typing each number Max resets the calculator. Find the total number of segments printed on the calculator. For example if $a = 1$ and $b = 3$ then at first the calculator will print 2 segments, then - 5 segments and at last it will print 5 segments. So the total number of printed segments is 12."*. And the slow but correct code provided by TimeEval is:

```
a, b = map(int, input().split())

segments = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]

total_segments = 0
for num in range(a, b+1):
    for digit in str(num):
        total_segments += segments[int(digit)]

print(total_segments)
```

After initialization, the model returned an optimized code:

```
a, b = map(int, input().split())

segments = [6, 2, 5, 5, 4, 5, 6, 3, 7, 6]

total_segments = sum(segments[(num // 10) % 10] +
   segments[num % 10] for num in range(a, b+1))

print(total_segments)
```

Analysis revealed that the original code converts each number to a string and then processed it character by character, which was especially time-consuming for larger numbers. The optimized code accesses each digitś units and tens directly through arithmetic operations, avoiding string operations. The optimized code uses the list segments to directly obtain and sum the corresponding display segment counts. Thus, the optimized version reduced internal loops and simplified calculations. However, the optimized code did not pass all the test cases because it assumes that all numbers are two digits, leading to incorrect results for numbers with more than two digits. For example, for a three-digit number like 123, the optimized code would only calculate the display segments for the tens and units of 12, ignoring the contribution of the highest digit 3, resulting in incorrect outcomes.

We manually analyzed 20 cases where errors occurred after self-refinement and found that in 12 cases, errors were present right from the initialization, 5 cases had errors after the first round of self-refinement, 1 case after the second round, and 2 cases after the fourth round. It is evident that most cases had errors from the initialization or after the first round, but since the feedback only concerned code efficiency, subsequent self-refinements did not correct the errors, leading to worse code. This discovery suggests that in the self-refinement framework, detection, and feedback regarding code correctness should be enhanced to ensure that code optimizations do not compromise accuracy.

5.2 Multi-Agent Collaboration

5.2.1 Overview

To explore the practicality and effectiveness of multi-agent collaboration, we conducted an empirical investigation based on the multi-agent collaboration framework proposed in the paper "Self-collaboration Code Generation via ChatGPT" [6], as shown in Figure 17 . We chose this framework because it clearly splits the roles into Analyst, Coder, and Tester. The division of responsibilities among these roles aligns with the conventional workflow in software engineering, and the structure of the framework is clear, which facilitated our further adjustments and optimizations.

5.2.2 Experimental Details

Experimental Framework and Process

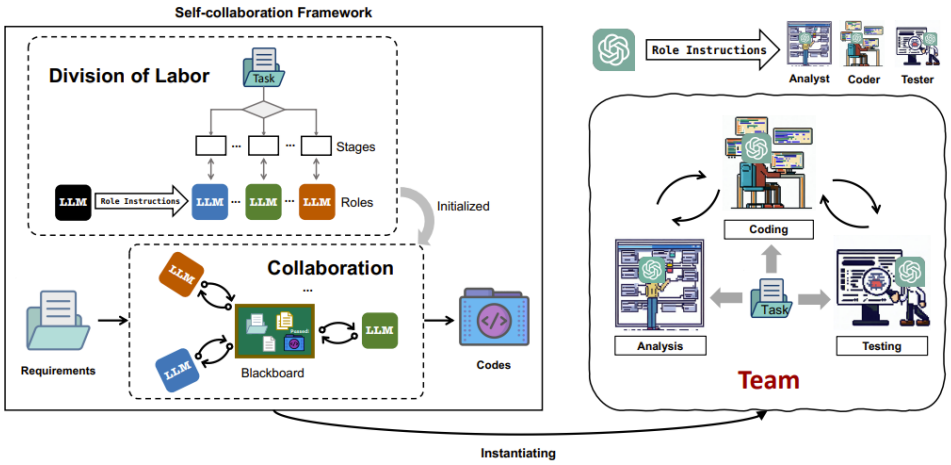

Figure 17: Multi-Agent Collaboration Framework

Based on the paper, the multi-agent collaboration framework we chose includes the following core steps:

- **Analysis Phase:** The task is first given to the Analyst, who then writes a high-level plan based on the task requirements.
- **Coding Phase:** Then, this plan is passed on to the Coder, who writes the corresponding code according to the plan.
- **Testing and Iteration Phase:** The completed code is handed over to the Tester for testing, and the Tester summarizes the test results into a report. If the code passes the test, the process ends, and the correct code is output. If the test fails, the test report is fed back to the Coder, who then tries to correct the code. This iterative process continues until reaching the maximum number of iterations mentioned in the paper, which is four.

Role Instructions and Prompts

Role Instructions = Team Description + User Requirment + Role Description	
Team Description	There is a development team that includes a requirements analyst, a developer, and a quality assurance tester. The team needs to develop programs that satisfy the requirements of the users. The different roles have different divisions of labor and need to cooperate with each others.
User Requirment	The requirement from users is '{Requirment}'. *For example: {Requirment} = Input to this function is a string containing multiple groups of nested parentheses. Your goal is to separate those group into separate strings and return the list of those. Separate groups are balanced (each open brace is properly closed) and not nested within each other Ignore any spaces in the input string*
Role Description	**Coder:** I want you to act as a developer on our development team. You will receive plans from a requirements analyst or test reports from a tester. Your job is split into two parts: 1. If you receive a plan from a requirements analyst, write code in Python that meets the requirements following the plan. Ensure that the code you write is efficient, readable, and follows best practices. 2. If you receive a test report from a tester, fix or improve the code based on the content of the report. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code. Remember, do not need to explain the code you wrote.

Figure 18: An example of role instruction for coder

To enhance the model's execution performance, the experiment in the paper employed the Role Instruction method, setting specific responsibilities for each agent through targeted prompt information. For example, figure 18 shows the prompt of Coder, which is composed of Team Description, User Requirement, and Role Description. In our experiment, we tried as much as possible to replicate the settings mentioned in the paper. But we made a practical change by splitting the Coder role into two: Coder and Repairer. The Coder is responsible for writing the original version of the code based on the Analyst's plan. The Repairer takes over to adjust the code after getting a report from the Tester. Below are the specific role prompts we used for the Analyst, Coder, Repairer, and Tester:

- **team_description:** "There is a development team that includes a requirements

analyst, a developer, and a quality assurance tester. The team needs to develop programs that satisfy the requirements of the users. The different roles have different divisions of labor and need to cooperate with each other."

- **user_requirement:** "The requirement from users is: {**problem**}."
- **Analyst:** **team_description** + "I want you to act as a requirements analyst on our development team. You will receive the requirements from users. Your job is: 1. Decompose the requirement into several easy-to-solve subproblems that can be more easily implemented by the developer. 2. Develop a high-level plan that outlines the major steps of the program. Remember, your plan should be high-level and focused on guiding the developer in writing code, rather than providing implementation details." + **user_requirement**.
- **Coder:** **team_description** + **user_requirement** + "I want you to act as a developer on our development team. You will receive plans from a requirements analyst. Your job is: to write code in language that meets the requirements following the plan. Ensure that the code you write is efficient, readable, and follows best practices. Remember, do not need to explain the code you wrote." + "The plans from a requirements analyst is: {**plan**}".
- **Repairer:** **team_description** + **user_requirement** + "I want you to act as a developer on our development team. You will receive test reports from a tester. Your job is: to fix or improve the code based on the content of the report. Ensure that any changes made to the code do not introduce new bugs or negatively impact

the performance of the code. Remember, do not need to explain the code you wrote." + "The code is script. The test reports from a tester is {**report**}".

- **Tester: team_description** + **user_requirement** + "I want you to act as a quality assurance tester on our development team. You will receive code from a developer. Your job is: 1. Test the functionality of the code to ensure it satisfies the requirements. 2. Write reports on any issues or bugs you encounter. 3. If the code or the revised code has passed your tests, write a conclusion 'Code Test Passed'. Remember, the report should be as concise as possible, without sacrificing clarity and completeness of information. Do not include any error handling or exception handling suggestions in your report." + "The code from a developer is: {**code**}".

5.2.3 Result Analysis

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
Multi-agent Collaboration	Python	24.9	53.0	17.0	20.1	5.9

Table 5: Experimental Results of Multi-Agent Collaboration in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
Multi-agent Collaboration	C++	11.9	39.8	6.0	55.7	16.4

Table 6: Experimental Results of Multi-Agent Collaboration in C++

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0.0	100.0	3.8
Multi-agent Collaboration	Java	14.8	40.5	6.2	57.7	3.8

Table 7: Experimental Results of Multi-Agent Collaboration in Java

This experimental result shows that, compared to the baseline result, the multi-agent collaboration framework performed significantly worse on our dataset. Although the Efficiency Level metric has slightly improved, the %Opt did not increase and even decreased in the Python language data. Moreover, pass@1 decreased significantly, particularly in the Python language data, where pass@1 was only 20.1%. This indicates that this multi-agent collaboration framework is not effective in solving coding problems on our dataset.

5.2.4 Multi-agent collaboration with new Tester for code efficiency

The original multi-agent collaboration framework completely follows the experimental setup in the paper[6]. However, the purpose of our project is to test and improve the efficiency of code generated by LLMs. Therefore, in this experiment, we modified the Tester to not only check the accuracy of the code but also to analyze its efficiency. The modified Tester is defined as follows:

Tester = team_description + user_requirement + "I want you to act as a quality assurance tester on our development team. You will receive code from a developer. Your job is: 1. Test the functionality of the code to ensure it satisfies the requirements. ***2. Test the efficiency of***

the code to ensure it has good time complexity. *3. Write reports on any issues or bugs you encounter. 4. If the code or the revised code has passed your tests, write a conclusion 'Code Test Passed'. Remember, the report should be as concise as possible, without sacrificing clarity and completeness of information. Do not include any error handling or exception handling suggestions in your report." + "The code from a developer is: {script}".*

Experimental Results

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
Original Multi-agent Collaboration	Python	24.9	53.0	17.0	20.1	5.9
Multi-agent collaboration with new Tester	Python	21.8	46.7	14.3	26.5	5.9

Table 8: Experimental Results of Multi-agent Collaboration with New Tester in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
Original Multi-agent Collaboration	C++	11.9	39.8	6.0	55.7	16.4
Multi-agent collaboration with new Tester	C++	8.3	39.0	4.1	50.8	18.0

Table 9: Experimental Results of Multi-agent collaboration with new Tester in C++

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0.0	100.0	3.8
Multi-agent Collaboration	Java	14.8	40.5	6.2	57.7	3.8
Multi-agent collaboration with new Tester	Java	16.0	39.0	6.7	57.7	3.8

Table 10: Experimental Results of Multi-agent collaboration with new Tester in Java

The test results are shown in the Tables 8, 9 & 10. For metrics measuring efficiency, only the %opt of C++ problems improved, but the 'Efficiency Level' all decreased. This indicates that the changes we made to the Tester do not effectively achieve the desired

effect of improving code efficiency. And the pass@1 of C++ decreased, while the pass@1 of Python increased. These results seem to be irregular, so next we analyze the reasons through a few case studies.

Case Studies

- **Case One: the 124th problem in the dataset**

  The **question** description for this problem is: *"One quite ordinary day Valera went to school (there's nowhere else he should go on a weekday). In a maths lesson, his favorite teacher Ms. Evans told students about divisors. Despite the fact that Valera loved math, he didn't find this particular topic interesting. Even more, it seemed so boring that he fell asleep in the middle of a lesson. And only a loud ringing of a school bell could interrupt his sweet dream. Of course, the valuable material and the teacher's explanations were lost. However, Valera will one way or another have to do the homework. As he does not know the new material absolutely, he cannot do the job himself. That's why he asked you to help. You're his best friend after all, you just cannot refuse to help. Valera's home task has only one problem, which, though formulated in a very simple way, does not have a trivial solution. Its statement looks as follows: if we consider all positive integers in the interval [a;b] then it is required to count the number of such numbers in this interval that their smallest divisor will be a certain integer k (you do not have to consider divisor equal to one). In other words, you should count the number of such numbers from the interval [a;b], that are not divisible by any number between 2 and k - 1 and yet are divisible by k."*

Then the **plan** given by Analyst is:

```
To decompose the requirement, we can break it down into
the following subproblems:
    1. Determine the range of numbers to consider, given
       the values of a and b.
    2. Iterate through each number in the range and check
        if it meets the conditions specified in the
       requirement.
    3. Count the number of numbers that meet the
       conditions.
    4. Print the count as the output.
Based on these subproblems, we can develop a high-level
plan for the program:
    1. Read the values of a, b, and k from the input.
    2. Initialize a count variable to 0.
    3. Iterate through each number in the range [a, b].
    4. For each number, check if it is divisible by any
       number between 2 and k-1, and not divisible by k.
    5. If the number meets the conditions, increment the
       count variable.
    6. Print the value of the count variable as the
       output.
```

This plan can correctly solve the problem, but it directly traverses the interval [a, b] and performs a division operation on each number to check if it meets the conditions. This direct iteration may cause significant performance issues when the distance between 'a' and 'b' is very large. Therefore, the test results of the code written by the Coder according to this plan show that 13 out of 30 test cases resulted in "timeout". And the Tester just outputs "Code Test Passed.", indicating that the tester cannot effectively judge the efficiency of the code.

- **Case Two: the 127th problem in the dataset**

  After analysis, the Analyst's high-level plan to solve this problem is also correct but inefficient. However, the code provided by the Coder has an error: "undefined

variable". And the Tester also simply outputs "Code Test Passed.", which shows that the Tester also cannot effectively detect errors in the code.

We manually analyzed a total of 20 cases and summarized four situations: Correct but low-efficient plan, timeout code and useless tester; Correct but low-efficient plan, wrong code, and useless tester; Wrong plan, wrong code and useless tester; Others. The specific results are shown in Table 11.

Type	Number
Correct but low-efficient plan, timeout code, and useless tester	6
Correct but low-efficient plan, wrong code, and useless tester	11
Wrong plan, wrong code, and useless tester	1
Others	2

Table 11: Analysis result of 20 cases

From the results, we believe that there are mainly two issues:

- The plans given by the Analyst are generally correct but often inefficient;
- The Tester is not able to effectively detect obvious errors in the code, nor can it effectively judge the efficiency of the code.

6 Methodology

6.1 Generative Executor Module

As mentioned in the previous sections, the accuracy of code testing is very low if it is done by LLM alone. Neither the self-testing in the self-refine framework nor the Tester

agent in the multi-agent collaboration framework does a satisfactory job of detecting problems in the generated code and providing feedback. So we tried to introduce an external executor to assist LLM in code testing. To involve an external executor, we must ensure that the test cases differ from those in the dataset to avoid potential test set leakage issues. Therefore, we aim to leverage the LLM's generation capabilities to automatically produce additional test cases.

Thus, we proposed the LLM-based Generative Executor Module to assist in code testing. The workflow of the Generative Executor Module is shown in the figure 19.

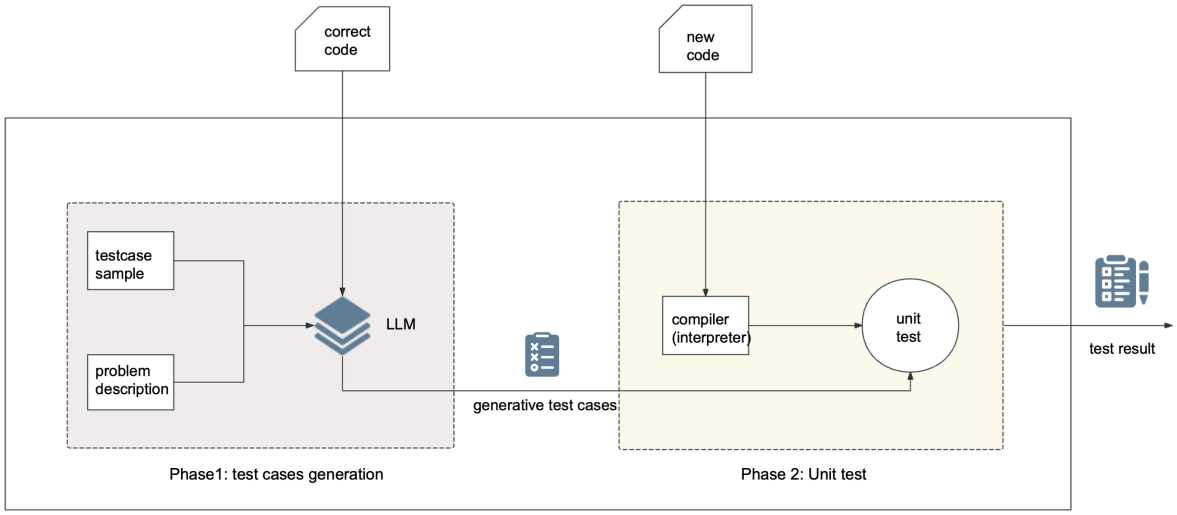

Figure 19: Workflow of generative executor module.

The module is divided into two parts: test case generation and unit test.

In the test case generation phase, we provide the problem description and a sample test case input to the LLM. The LLM then generates additional test case inputs for the problem. After obtaining these test case inputs, we execute the slow but accurate code

that needs optimization, utilizing the inputs generated in the previous step. This process yields the corresponding test case outputs. At the end of this phase, the module produces several input-output pairs for subsequent testing. Figure 20 shows the terminal output of the test cases generation phase in the module. The details of the prompt that triggered the LLM to generate test case inputs are depicted in Figure 21.

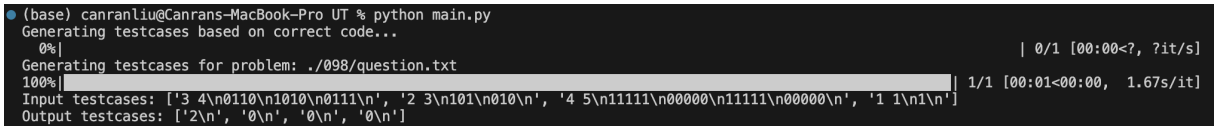

Figure 20: Terminal output after test cases generation phase.

```
def get_messages(prompt, example_input):
    messages = []
    system_prompt = f"I will give you a programming problem and
        a corresponding test input. The problem is: {prompt}.
       And the sample input case is: {example_input}. Please
       mimic the format of the test input to generate 5 test
       inputs and put them in a list. The list should only
       contain the test inputs and should not contain any other
        information. For example, if the sample input is '1 2 3
        4\\n', the list should be [\"1 2 3 4\\n\", \"4 3 2 1\\n
       \", \"1 1 1 1\\n\", \"0 0 0 0\\n\", \"1 0 1 0\\n\"]"
    messages.append(
        {"role": "user", "content": system_prompt}
    )
    return messages
```

Figure 21: Prompt of test inputs generation.

In the unit test phase, the module executes the new code intended for testing, verifies its correctness using the test cases generated in phase 1, and then presents the test results. Specifically, the module generates a file named *feedback.txt*. The details of the

feedback.txt file are as follows:

- **Test Result:** The first line of the file has only two possibilities: *Pass* or *Fail* If the first line is *Pass*, it indicates that the newly generated code was compiled successfully and passed all the generated test cases. Otherwise, if the first line is *Fail*, the subsequent content of the file elaborates on the reason for the failure.
- **Cause of Error:** If a syntax error or compilation error is detected, *An error occurred in the program:* will be displayed on the second line of the file. Subsequently, the output from the Python interpreter, C++, or Java compiler will be presented. As illustrated in Figure 22, an example of Java program error feedback is provided.

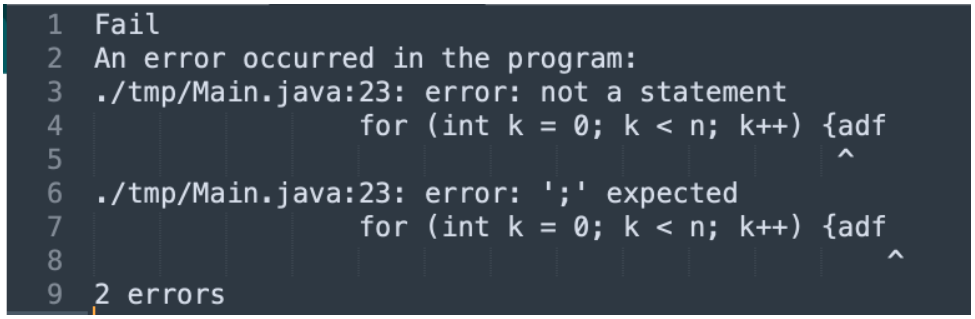

Figure 22: Error message output by the generative executor.

- **Failed Test Cases:** If the program compiles and runs without syntax errors but encounters logic errors leading to incorrect test cases, the second line of the file will state: "The new code failed following test cases:" followed by the specific errors. Each test case with errors will be presented in the following format: *When the input is...*, *The expected output is...*, and *The output is...*. Figure 23 illustrates the above error scenario.

```
Fail
The new code failed following testcases:
When the input is 3 4
0110
1010
0111
The expected output is 2
The output of the new code is -1

When the input is 2 3
101
010
The expected output is 0
The output of the new code is -2

When the input is 4 5
11111
00000
11111
00000
The expected output is 0
The output of the new code is -10
```

Figure 23: Error message output by the generative executor.

6.2 Self-Refine-Executor Framework

6.2.1 Motivation

In the previous empirical study, we conducted experiments with the self-refine framework, which did not perform well on our dataset. Through case studies, we identified that the main issue was the focus on code efficiency during feedback, which often led to errors in the refined code. Unfortunately, the feedback did not correct these errors. Therefore, we aimed to improve this aspect and consequently developed our framework.

6.2.2 Framework Description

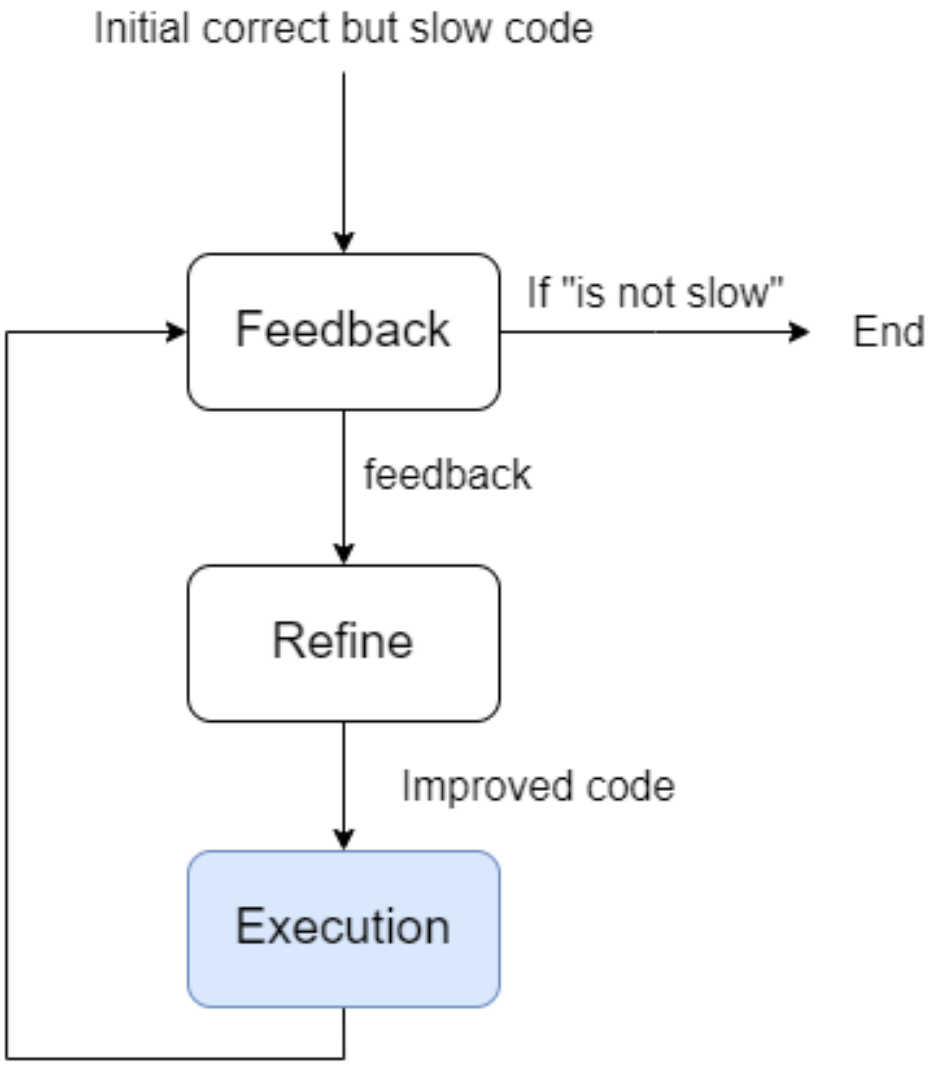

Figure 24: Workflow of Self-Refine-Executor Framework

A major improvement in our framework is the introduction of the generative executor module discussed earlier. This means that we use this external code execution module to obtain feedback on the correctness of the code. After each round of feedback and refinement, the code is tested, and only the code that passes the test is retained for the

next round of self-refinement. In essence, we always retain only the correct code. The specific process is as follows:

- **Initialization Phase:**

  Provide the model with a correct but inefficient version of code and ask for an optimized version.

- **Execution Phase:**

  Submit the code for testing by the execution module. If the test result is “pass,” the code is retained. If it fails, the code is discarded, and the previous correct code is used for the next feedback and refinement.

- **Feedback Phase:**

  The model provides feedback on the efficiency of the code. If the model determines the code is efficient enough, it outputs “is not slow,” signaling the end of the iteration.

- **Refine Phase:**

  The model refines or improves the code based on the feedback, and this code is sent for testing by the execution module.

The Execution, Feedback, and Refinement phases continuously iterate until the model believes no further optimization is needed or the maximum number of iterations is reached, which is set to four.

By incorporating an external module to check the correctness of code, we are able

to ensure that the code retained is always correct. The worst-case scenario would be retaining the original version we provided to the model, which was correct but inefficient.

6.3 Multi-Agent-Executor Framework

6.3.1 Motivation

In the previous research, we experimented with multi-agent collaboration, but the outcomes were not as expected. Through 20 case studies, we identified two main issues: the Analyst's plans were inefficient, and the Tester relying solely on the model itself without external feedback to review code often failed to detect errors in the code. Based on this, we proposed improvements in these two areas, which led to the development of our own framework.

6.3.2 Framework Description

Our Multi-Agent Executor Framework still consists of three agents: the Analyst, the Coder, and the Tester. And each agent is still assigned by specific role instructions. But unlike the original framework, here we added a new component called "Execution", realized by the "Generative Executor module" introduced before. This means we introduced an external code executor to obtain correct feedback on code correctness. Additionally, we have reduced the role of the Analyst, making it only be called upon when the initial code generated by the Coder is wrong. Therefore, the Analyst only serves a supportive function.

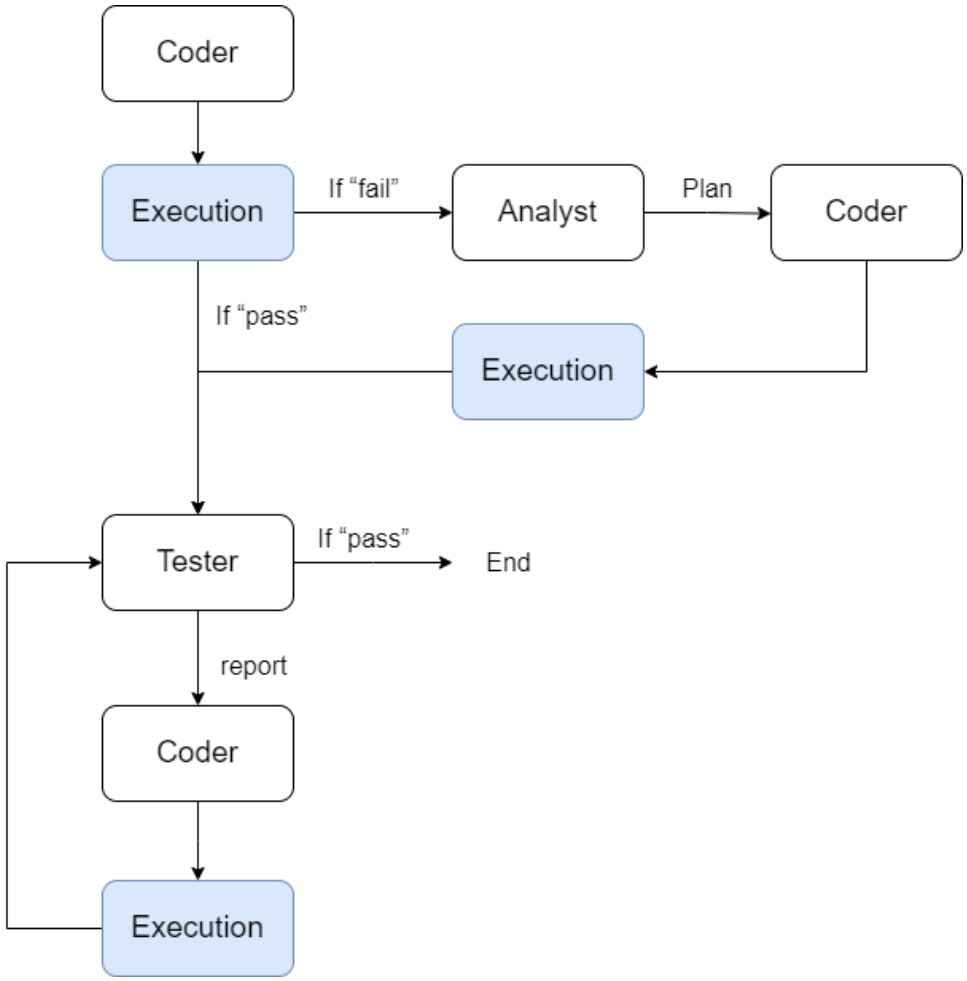

Figure 25: Workflow of Multi-Agent-Executor Framework

More specifically, the workflow of the Multi-Agent Executor Framework is as follows:

- **Initialization Phase:** The task is firstly given to the Coder, who will write code according to the requirements of users. The code will then be passed to the Executing Phase directly. If the execution result of this initial code is “Pass”, it then goes to the Testing Phase. If the code fails, the Analyst would be called to give a high-level plan for this task.
- **Coding Phase:** This plan is passed back to the Coder, and then the Coder will

write the code according to the plan.

- **Executing Phase:** The completed code will be executed through the external "Generative Executor module". The module returns a result, indicating "Pass" if the code passes all test cases, or "Fail" along with the test cases that failed and any error information (if available).
- **Testing Phase:** The execution result is given to the Tester. If the result is "Pass", the Tester analyzes whether there is room to improve the efficiency of the code; if the result is "Fail", the Tester drafts a report based on the error information. If the code is correct and the Tester believes it is efficient enough, the Tester will make a "Code Test Pass" conclusion, marking the end of the iteration.
- **Repairing Phase:** If the test is not passed, the test report is sent back to the Coder, who revises the code according to the report.

The Executing Phase, Testing Phase and Repairing Phase will continuously iterate until receiving the end signal "Code Test Pass" from the Tester or reaching the maximum number of iterations, which is set to 4 for this experiment.

7 Experiments

7.1 Setup

Metrics. We adhered to the evaluation strategy outlined in the timeEval benchmark. The assessment involves five metrics: **Pass rate**, **Fail rate**, **Timeout rate**, **Percent Optimized**, and **Speedup**, utilized to evaluate the efficiency of the generated code. For a detailed explanation of these metrics, please refer to Section 4.

Models. Our primary experimental foundation is the gpt-3.5-turbo model. Additionally, we conducted a comparative experiment in Section 7.6 using the gpt-4 model. In future work, we aim to explore other types of LLMs to provide a more detailed evaluation of the efficiency of code generated by different LLMs.

Platform. The timeEval benchmark executes both the canonical solution and the generated code during each test, comparing their running times. This approach enables tests to be conducted on different configurations or platforms while maintaining consistent results. Our experimental platform involves several different personal computers.

Interpreter and compiler version. We used Python 3.11.4 as the interpreter version, g++ 8.1.0 as the C++ language compiler, and java 18.0.2.1 as the Java language version in our experiments.

7.2 Baseline Experiment

The prompt for the baseline experiment is shown in Figure 26. We enforce a constraint on LLMs to refrain from generating text descriptions for two primary reasons. Firstly, we aim to eliminate the influence of chain-of-thought (CoT) processes on experimental outcomes, enabling us to isolate and analyze the impact of CoT on code efficiency in subsequent experiments. Secondly, in the majority of code-generation scenarios, it is unnecessary for LLMs to generate textual descriptions before code-generation

```
def get_messages(prompt, language):
    messages = []
    system_prompt = "Please generate " + language + "code that
       can be run directly to solve the following programming
       problem. Do not add any text description!"
    messages.append(
        {"role": "system", "content": system_prompt}
    )
    messages.append(
        {"role": "user", "content": prompt}
    )

    return messages
```

Figure 26: The baseline prompt.

7.3 Self-Refine-Executor

7.3.1 Experimental Details

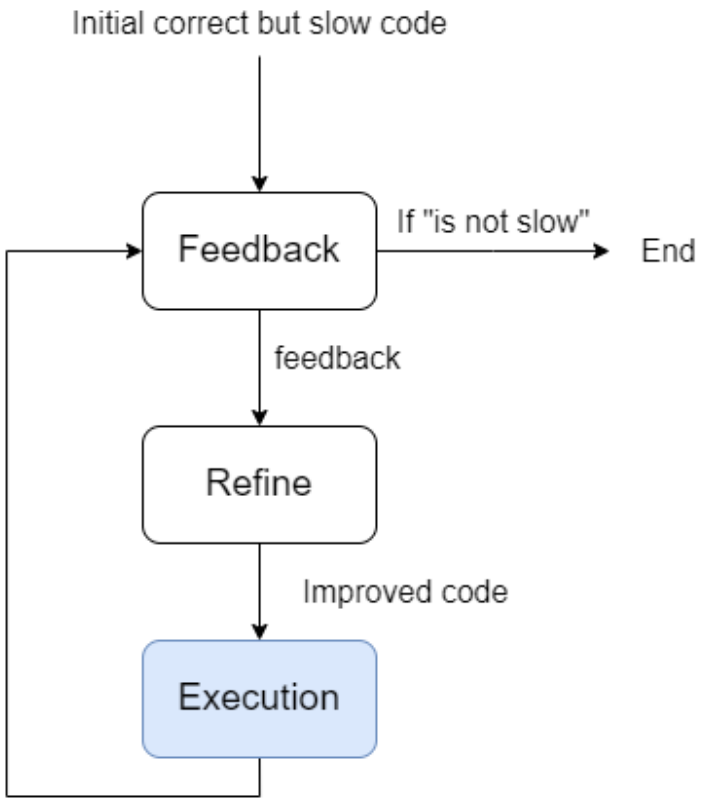

Figure 27: Workflow of Self-Refine-Executor Framework

We conducted the experiment of this Self-Refine-Executor framework following the process mentioned in the Methodology 6. The specific process is shown in Figure 27. For the prompt, we continue with the design used in the self-refine experiment, utilizing a few-shot prompt to enhance efficiency. The specific prompt is as follows:

- **Initialization Prompt:**

  # slower version:

```
a, b = input().split()
n = int(a + b)
flag = False
for i in range(n):
    if i ** 2 == n:
        flag = True
        break
print('Yes' if flag else 'No')
```

 # optimized version of the same code:

```
a, b = input().split()
n = int(a + b)

flag = False
for i in range(1000):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')
```

END

More examples...

 # slower version:

{*The correct but slow code provided by timeEval*}

 # optimized version of the same code:

- **Feedback Prompt:**

```
a, b = input().split()
n = int(a + b)
```

```
flag = False
for i in range(n):
    if i ** 2 == n:
        flag = True
        break

print('Yes' if flag else 'No')
```

 # Why is this code slow?

```
# This code is slow because it is using a brute force
   approach to find the square root of the input number.
   It is looping through every possible number starting
   from 0 until n. Note that the square root will be
   smaller than n, so at least half of the numbers it is
   looping through are unnecessary. At most, you need to
   loop through the numbers up to the square root of n.
```

END

More examples...

{*The correct but slow code provided by TimeEval*}

 # Why is this code slow?

- **Refine Prompt:**

  {*The correct but slow code provided by TimeEval*}

  # Why is this code slow?

  {*Feedback from the model*}

  How to improve this code? Please provide the improved version of the code.

7.3.2 Results Analysis

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
Self-refinement	Python	7.0	41.9	2.5	61.8	11.8
Self-Refine-Executor	Python	7.1	40.2	2.3	91.2	17.6

Table 12: Experimental Results of Self-Refine-Executor in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
Self-refinement	C++	4.3	63.4	2.2	52.5	37.7
Self-Refine-Executor	C++	3.4	53.8	0.7	90.1	29.5

Table 13: Experimental Results of Self-Refine-Executor in C++

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0.0	100.0	3.8
Self-refinement	Java	7.5	30.7	2.6	46.1	7.3
Self-Refine-Executor	Java	7.6	33.2	0.9	87.3	4.4

Table 14: Experimental Results of Self-Refine-Executor in Java

The experimental results are shown in the Table. From the data in the tables, compared to the baseline, our framework shows significant improvements in the Efficiency Level and %opt, indicating that the code generated through our framework generally has better time complexity than the baseline, while the decrease in pass@1 is minimal. Compared to the original self-refine framework, although the Efficiency Level and %opt are not as high as the original self-refine framework, the pass@1 of our framework remains at a

relatively high level. This indicates that our framework can ensure high accuracy while improving the efficiency of the code.

However, we also noticed that according to our hypothesis, the correctness of the code generated by this framework should be 100% because we always retain only the correct code. To explore why pass@1 is not 1, we conducted a case study. We identified two main reasons why the final output code did not pass all the test cases:

- After self-refine, the efficiency of the code actually decreased, but the executor-generated test cases were not large enough to detect timeout situations, so this inefficient version of the code passed the tests of the execution module and was retained.
- After self-refine, the optimized code had errors, but the executor-generated test cases were not comprehensive enough to detect these errors, so this erroneous version of the code passed the tests of the execution module and was retained.

7.4 Multi-Agent-Executor

7.4.1 Experimental Details

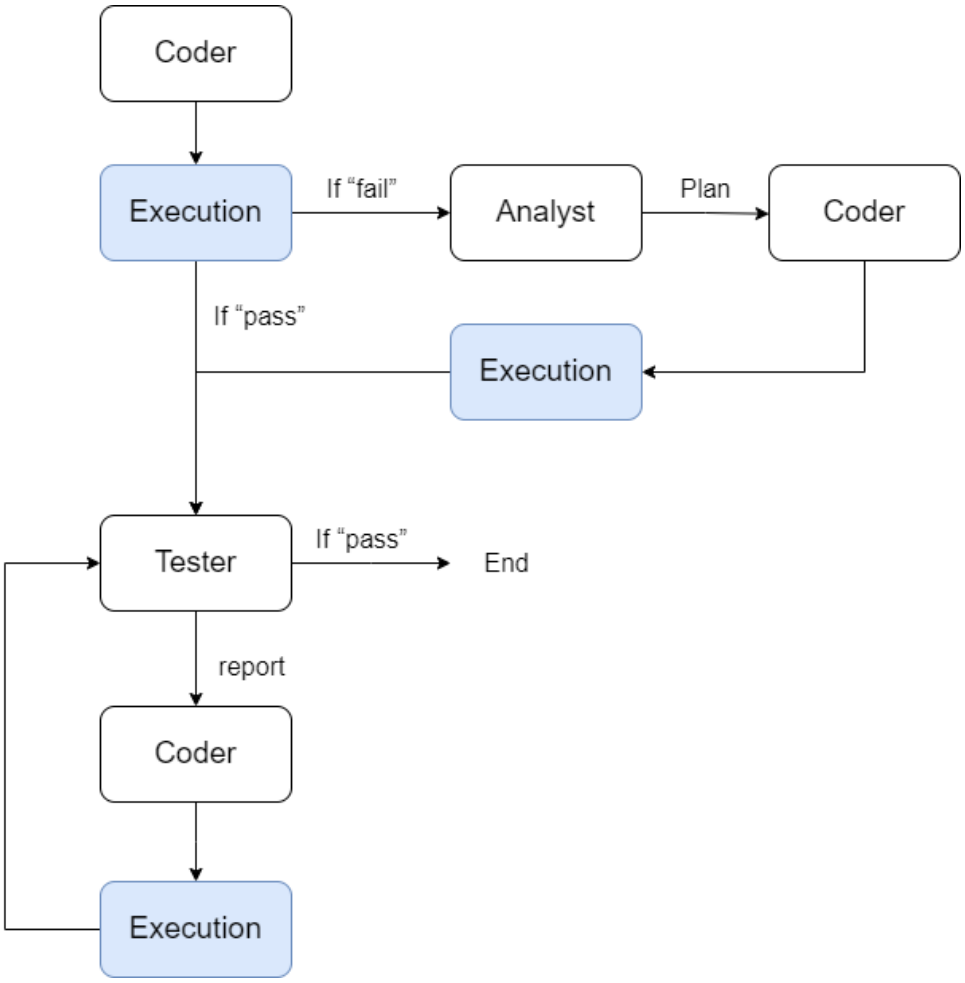

Figure 28: Workflow of Multi-Agent-Executor Framework

We conducted the experiment of this Multi-Agent-Executor framework following the process mentioned in the Methodology 6, and we have accordingly adjusted the prompt. Specifically, we divided the Coder into Coder1, Coder2, and Repairer. Coder1 generates code directly based on the task requirements without referring to the Analyst's plan,

while Coder2 generates code by referring to the Analyst's plan. The Repairer modifies or improves the previous code based on the Tester's report. We also emphasize to the Analyst that the plan provided should be efficient. The specific prompt for each agent is as follows.

- **team_description:** "There is a development team that includes a requirements analyst, a developer, and a quality assurance tester. The team needs to develop programs that satisfy the requirements of the users. The different roles have different divisions of labor and need to cooperate with each other."
- **user_requirement:** "The requirement from users is: {**problem**}."
- **Analyst:** **team_description** + "I want you to act as a requirements analyst on our development team. You will receive the requirements from users. Your job is: 1. Decompose the requirement into several easy-to-solve subproblems that can be more easily implemented by the developer. 2. Develop a high-level plan that outlines the major steps of the program. Remember, your plan should be high-level and focused on guiding the developer in writing code, rather than providing implementation details. **And your plan should have good time complexity.**" + **user_requirement**.
- **Coder1 (no plan needed):** **team_description** + **user_requirement** + "I want you to act as a developer on our development team. Your job is: to write code in language that meets the requirements of users. Ensure that the code you wrote is efficient, readable, and follows best practices. Remember, do not need to explain

the code you wrote.”.

- **Coder2 (plan needed):** **team_description** + **user_requirement** + “I want you to act as a developer on our development team. You will receive plans from a requirements analyst. Your job is: to write code in language that meets the requirements following the plan. Ensure that the code you write is efficient, readable, and follows best practices. The code you write also should always have the code for reading the inputs and printing the outputs of test cases from the tester later. Remember, do not need to explain the code you wrote.”.

- **Repairer:** **team_description** + **user_requirement** + “I want you to act as a developer on our development team. You will receive test reports from a tester. Your job is: to fix or improve the code based on the content of the report. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code. Remember, do not need to explain the code you wrote.” + “The code is {**code**}. The test reports from a tester is {**report**}”.

- **Tester:** **team_description** + **user_requirement** + “I want you to act as a quality assurance tester on our development team. You will receive code from a developer and **its execution result on the test cases**. Your job is: 1. If the execution result shows ‘pass’, analyze the efficiency of the code and see if it could have better time complexity. 2. If the execution result shows ‘fail’, find the reason for the incorrectness and give suggestions on how to fix the error. 3. Write reports of your findings and suggestions for the coder to repair the code. 4. Only if the execution

result of the provided code shows 'pass' and the provided code is efficient enough, write a conclusion 'Code Test Passed' at the end of the report, otherwise, write 'Code Test Failed'. Remember, the report should be as concise as possible, without sacrificing clarity and completeness of information. And remember your job is only to write the report for the Coder to repair the code, so please do not include any detailed code in the report." + "The code from a developer is: {**code**}. The execution result is:{**execution_result**}".

7.4.2 Results Analysis

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
Original Multi-agent Collaboration	Python	24.9	53.0	17.0	20.1	5.9
Multi-agent collaboration with new Tester	Python	21.8	46.7	14.3	26.5	5.9
Multi-Agent-Executor	Python	10.2	53.2	4.6	73.5	14.7

Table 15: Experimental Results of Multi-Agent-Executor with New Tester in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
Original Multi-agent Collaboration	C++	11.9	39.8	6.0	55.7	16.4
Multi-agent collaboration with new Tester	C++	8.3	39.0	4.1	50.8	18.0
Multi-Agent-Executor	C++	8.9	63.7	3.4	70.2	32.8

Table 16: Experimental Results of Multi-Agent-Executor with new Tester in C++

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0.0	100.0	3.8
Multi-agent Collaboration	Java	14.8	40.5	6.2	57.7	3.8
Multi-agent collaboration with new Tester	Java	16.0	39.0	6.7	57.7	3.8
Multi-Agent-Executor	Java	15.8	45.5	8.0	59.1	7.4

Table 17: Experimental Results of Multi-Agent-Executor with new Tester in Java

The results are shown in the Table 15, 16 & 17. Compared with the previous experiments of Original Multi-agent Collaboration and Multi-agent Collaboration with a new Tester, every metric in the result of our Multi-Agent-Executor framework has improved. The reduction in Total Time and Timeout Rate indicates shorter average execution times, while increases in Efficiency Level and %opt suggest that more problems are closer to the optimal solution, and on average each problem is closer to the optimal solution as well. Compared to the baseline, there is a notable improvement in Efficiency Level and %opt, while the decrease in pass@1 is not as much as in the previous two experiments. Overall, our Multi-Agent-Executor framework has found a balance between improving efficiency and accuracy, performing well in tasks that enhance code efficiency.

7.5 In-context Learning

7.5.1 Experimental Details

In this experiment, we take an in-context-learning approach to improve the efficiency of LLM-generated code. The experiment was specifically divided into the following steps:

Problem Collection In this step, selecting the appropriate case is crucial to steer LLM toward generating more efficient code. We've chosen four classic programming problem types from the LeetCode[9] website and selected a representative topic from each category. These four problem types include **Binary Search**, **Divide and Conquer**, **Dynamic Programming**, and **Sorting**. The details of the problem are represented in figure 29,30,31, and 32.

```
Given a string s, return the longest palindromic substring in s
   .

Example 1:

Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.

Example 2:

Input: s = "cbbd"
Output: "bb"

Constraints:

1 <= s.length <= 1000
s consist of only digits and English letters.
```

Figure 29: Dynamic programming question in in-context-learning experiment

```
Given two sorted arrays nums1 and nums2 of size m and n
   respectively, return the median of the two sorted arrays.

Example 1:

Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
Example 2:

Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2
    = 2.5.


Constraints:

nums1.length == m
nums2.length == n
0 <= m <= 1000
0 <= n <= 1000
1 <= m + n <= 2000
-106 <= nums1[i], nums2[i] <= 106
```

Figure 30: Binary search question in in-context-learning experiment

```
Given an integer array nums, find the
subarray
with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

1 <= nums.length <= 105
-104 <= nums[i] <= 104
```

Figure 31: Divide and conquer question in in-context-learning experiment

```
Given an integer array nums and an integer k, return the kth
   largest element in the array.

Note that it is the kth largest element in the sorted order,
   not the kth distinct element.

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2
Output: 5

Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4

Constraints:

1 <= k <= nums.length <= 105
-104 <= nums[i] <= 104
```

Figure 32: Sorting question in in-context-learning experiment

Sample Creation After successfully obtaining the four problems in the previous step, we find a positive and negative example pair for each problem. Positive example means efficient and code, negative example means inefficient but correct code. We designed two parallel experiments for in-context learning, one with only positive examples and one with both positive and negative examples. Where the time complexity of the positive example and negative example for each problem is shown in the table x. In addition, to eliminate the influence of the programming language of the sample itself on the results generated by LLM, we set up samples in different languages for different programming

languages. Samples in different languages are equivalent at the algorithmic level.

Sample Combination In addition to examining the impact of positive and negative examples in in-context learning techniques on the efficiency of generated code, we also explore the effects of combining different numbers of examples. Specifically, we experiment with combining examples in sets of 1, 2, and 4 for in-context learning, with the combinations being randomized.

Problem Type	Negative	Positive
Binary search	$O(m+n)$	$O(\log(m+n))$
Divide and conquer	$O(n^2)$	$O(n)$
Dynamic programming	$O(n^3)$	$O(n)$
Sorting	$O(n \log n)$	$O(n)$

Table 18: Time Complexity of Different Problem Types

7.5.2 Results Analysis

The experimental results of In-context learning are presented in Table 19, 20, and 21. In-context learning will be replaced with the abbreviation ICL below. From the experimental data, we observe that utilizing ICL (2 Positive and Negative Examples) in Python yields balanced efficiency and accuracy, resulting in improved performance. In the case of C++, employing ICL (2 Positive and Negative Examples) achieves balanced efficiency and accuracy with superior performance. For Java, employing ICL (4 Positive and Negative Examples) leads to balanced efficiency and accuracy, along with better performance. We posit that the optimal parameters for optimizing ICL experiments vary across programming languages, suggesting no significant correlation between them.

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
ICL(1 Positive Example)	Python	7.4	32.1	6.8	26.5	2.9
ICL(2 Positive Examples)	Python	3.1	36.4	2.0	50.0	8.8
ICL(4 Positive Examples)	Python	3.4	32.5	2.5	50.0	8.8
ICL(1 Positive and negative Examples)	Python	3.3	30.1	2.5	50.0	11.7
ICL(2 Positive and negative Examples)	Python	3.7	39.3	2.0	58.8	8.8
ICL(4 Positive and negative Examples)	Python	6.2	34.3	4.3	50.0	5.9

Table 19: Experimental Results of In-Context Learning in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
ICL(1 Positive Example)	C++	5.6	38.6	1.1	81.8	21.2
ICL(2 Positive Examples)	C++	5.2	57.1	1.3	90.2	42.6
ICL(4 Positive Examples)	C++	6.7	50.8	1.9	83.6	39.3
ICL(1 Positive and negative Examples)	C++	5.4	48.9	1.3	85.2	26.2
ICL(2 Positive and negative Examples)	C++	5.3	49.1	1.3	83.6	31.1
ICL(4 Positive and negative Examples)	C++	5.6	48.5	1.4	80.3	23.0

Table 20: Experimental Results of In-Context Learning in C++

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0	100.0	3.8
ICL(1 Positive Example)	Java	10.3	29.8	2.6	65.4	0
ICL(2 Positive Examples)	Java	8.8	28.4	1.5	73.0	0
ICL(4 Positive Examples)	Java	7.3	23.9	0.5	69.2	3.8
ICL(1 Positive and negative Examples)	Java	9.6	27.4	2.0	65.3	3.8
ICL(2 Positive and negative Examples)	Java	9.2	26.4	1.5	69.2	0
ICL(4 Positive and negative Examples)	Java	7.0	26.2	0	76.9	0

Table 21: Experimental Results of In-Context Learning in Java

7.6 Others

7.6.1 Simple Prompt Engineering

We explored the effect of different prompts on the efficiency of code generation by simply modifying the prompt. We try to remind LLM of the time complexity right in the prompt. The details are shown in Figure 33. The results of the experiment are presented in Table 22, 23, and 24. We can see from these data that a simple prompt change directly does increase the efficiency of the generated code. But the increase is relatively limited.

```
def get_messages(prompt, language):
    messages = []
    system_prompt = "Please generate " + language + "code that
       can be run directly to solve the following programming
       problem. Do not add any text description!" + "Please pay
        attention to the time complexity of your solution."
    messages.append(
        {"role": "system", "content": system_prompt}
    )
    messages.append(
        {"role": "user", "content": prompt}
    )

    return messages
```

Figure 33: Mind time complexity prompt.

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
Mind time complexity prompt	Python	7.49	42.2	5.2	67.7	11.7

Table 22: Experimental Results of Simple Prompt Engineering in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0	100.0	3.8
Mind time complexity prompt	Java	11.0	31.1	2.4	80.7	7.7

Table 23: Experimental Results of Simple Prompt Engineering in Java

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
Mind time complexity prompt	C++	4.8	50.1	0.6	80.3	37.7

Table 24: Experimental Results of Simple Prompt Engineering in C++

7.6.2 Chain-of-Thought(CoT)

We explored the impact of CoT on code generation. We formed the CoT by allowing LLMs to generate textual descriptions. The details are shown in Figure 34. The results of the experiment are presented in Table 25, 26, and 27. We can see from the results that CoT is useful for improving code efficiency. But it will reduce the accuracy of the code to some extent.

```
def get_messages(prompt, language):
    messages = []
    system_prompt = "Please generate " + language + "code to
       solve the following programming problem. Let's think it
       step by step."
    messages.append(
        {"role": "system", "content": system_prompt}
    )
    messages.append(
        {"role": "user", "content": prompt}
    )

    return messages
```

Figure 34: Mind time complexity prompt.

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Python	6.0	31.5	0.0	100.0	8.8
CoT	Python	14.6	40.5	11.3	50.0	11.7

Table 25: Experimental Results of CoT in Python

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	Java	12.6	24.0	0	100.0	3.8
CoT	Java	9.6	39.8	2.0	69.2	3.8

Table 26: Experimental Results of CoT in Java

Experiment	Language	Total Time(TT)	Efficiency Level (EL)	Timeout Rate (TR)	pass@1	%opt
Baseline	C++	3.5	40.4	0.0	100.0	16.4
CoT	C++	3.2	54.9	0.7	77.0	31.1

Table 27: Experimental Results of CoT in C++

8 Conclusion

The following are some of the main contributions we have made during this semester's project.

- **Propose metrics.** Propose a set of metrics that comprehensively evaluate the execution efficiency of generated code.
- **Measure and process code_contests dataset.** We finished in-depth testing of the code_contests dataset. In total, we tested more than 13,000 problems with 520,000 corresponding solutions and 780,000 corresponding test cases. Proposed a reasonable process for processing the dataset, and filtered out the problems that were difficult to be solved efficiently by the gpt-3.5-turbo model through this process.
- **Improve timeEval benchmark.** We have successfully extended last semester's TimeEval benchmark from monolingual Python to encompass multilingual support for Python, C++, and Java. Additionally, we've developed a comprehensive automated code execution framework, marking it as the first multilingual code efficiency benchmark of its kind that we have known of.
- **Propose multiple frameworks.** We explore strategies for improving the efficiency of generated code from several perspectives. These include, but are not limited to: simple prompt engineering, chain-of-thought, In-context learning, self-refinement, and multiagent collaboration frameworks. We first migrated these frameworks to

the code efficiency task and then improved them with encouraging results.

9 Division of Labor

In this project, our team made equal contributions throughout the entire year. Together, we strategized for the project, researched similar projects, wrote code, and collectively finished the project. In terms of dataset processing, we jointly surveyed existing datasets and discussed selecting the one that best suited our research purposes. Then, we worked together on data filtering and analysis, selecting 111 suitable problems for our dataset. Then for benchmark creation, We also collaborated on designing metrics for our benchmark and designing the framework for evaluating code efficiency. Details of the division of work are outlined in the table below:

Details of Work	**Responsible Member**
Research on various related works	Canran Liu & Xingyun Ma
Dataset Processing	Canran Liu & Xingyun Ma
Benchmark Construction	Canran Liu & Xingyun Ma
Empirical study of Self-refinement	Xingyun Ma
Empirical study of Multi-Agent Collaboration	Xingyun Ma
In-context Learning Experiment	Canran Liu
Generative Executor Module Implementation	Canran Liu
Self-Refine-Executor Framework and Experiment	Canran Liu & Xingyun Ma
Multi-Agent-Executor Framework and Experiment	Canran Liu & Xingyun Ma

9.1 Xingyun Ma

In this semester's final-year project, I took on several crucial tasks. Firstly, I was primarily responsible for conducting empirical study on our dataset, including implementing self-refinement and multi-agent collaboration. I strictly adhered to the experimental settings outlined in the referring papers, while also making necessary adjustments and modifications based on our dataset to ensure accurate and reference-worthy empirical research. Through in-depth analysis of experimental data and case studies, I identified issues existing in both frameworks and actively engaged in discussions with my teammate, advisor and supervisor, which were crucial for successfully proposing our framework. Ultimately, based on the outcomes of these discussions, we put forward our own framework, making a pivotal contribution to the project's success.

In addition to the individual responsibilities I undertook, I also collaborated with team members to ensure the project's consistency and refinement. Overall, through effective division of tasks and proactive discussions with team members, advisor and supervisor, I played a key role in this semester's graduation project.

9.2 Canran Liu

In this semester's final-year project, I embarked on a multi-faceted journey. Initially, I laid the groundwork by meticulously crafting a series of scripts engineered to meticulously measure the dataset's parameters and facilitate efficient data processing procedures. Following this preparatory phase, I delved into the development of the autorun framework

tailored specifically for the timeEval benchmark, streamlining its execution process for enhanced reliability and efficiency. Moreover, I undertook the intricate task of designing and coding the generative executor, a pivotal component essential to our project's success. This involved not only conceptualizing its architecture but also implementing robust coding solutions to ensure its seamless integration and functionality within the broader project framework. Additionally, my involvement extended beyond individual components to encompass broader project architecture, where I actively participated in the design and refinement of various frameworks. By collaborating with team members and contributing to the conceptualization of these frameworks, I played a key role in shaping the project's overall structure and ensuring its coherence and effectiveness.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. pages 16, 17

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. pages 10, 11, 32, 34

[3] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors, 2023. pages 15

[4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. pages 5

[5] Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in c/c++, 2023. pages 5

[6] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2023. pages 42, 47

[7] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021. pages 6, 12

[8] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, and Chenglin Wu. Metagpt: Meta programming for multi-agent collaborative framework, 2023. pages 5, 15

[9] ITCharge. Jump game ii. https://leetcode.cn/problems/jump-game-ii/solutions/1703944/by-itcharge-xn4b/. Accessed 28 November 2023. pages 73

[10] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023. pages 10

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

alphacode. *Science*, 378(6624):1092–1097, 2022. pages 21

[12] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. pages 6

[13] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2023. pages 5

[14] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. pages 5, 12, 13, 35, 36, 37

[15] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023. pages 5

[16] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xi-

aoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023. pages 10

[17] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. pages 5

[18] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2024. pages 18

[19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. pages 17, 18

[20] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey, 2023. pages 14, 15

[21] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. A survey on language models for code, 2023. pages 10, 11

[22] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023. pages 12

[23] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017. pages 12