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

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[11].

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, 11]. 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 muti-agent collaboration frameworks to improve code accuracy [4, 7, 12, 13, 16]. Relatively small API charges were required to implement methods in their study. Therefore, we

would like to propose a novel LLM-based self-refinement and muti-agent collaboration framework. The 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 the APPS dataset[6]. TimeEval is an abbreviation for time evaluation. The specific benchmark build process is described in detail in Section 3.

The following components are included in the timeEval benchmark:

- **Problem set of size 110.** The problem set comprises 110 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**.

- **Test cases for each question.** We prepared 10 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 building the timeEval benchmark. We did the first empirical study on the efficiency of generated code on our benchmark. In this empirical study, we comprehensively analyze the impact of diverse approaches on the efficiency of generated code. Our examination encompasses various perspectives, including prompt engineering, self-refinement, multi-agent collaboration, and a variety of language models. This successfully remedies the gap in current studies that under-explore the efficiency of generated code. The design and discussion of the experiment are described in detail in Section 4.

2 Related Work

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[15][19]. 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.[19], illustrates the current major types of language models for code and their typical representatives.

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

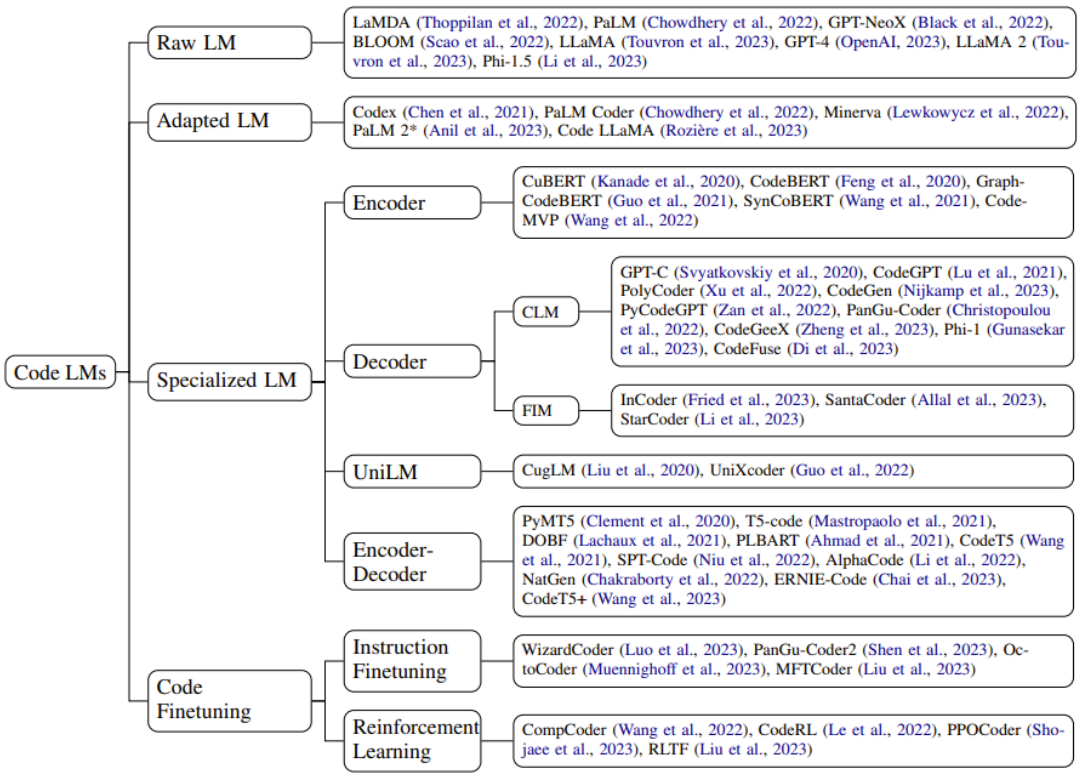

Figure 1: The overview of current language models for code[19]

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[6]. There are also datasets for testing other languages, such as HumanEval-X[20], which includes Python, C++, Java, JavaScript, and Go, and the WikiSQL dataset for evaluating SQL[21], 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.[12], 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.

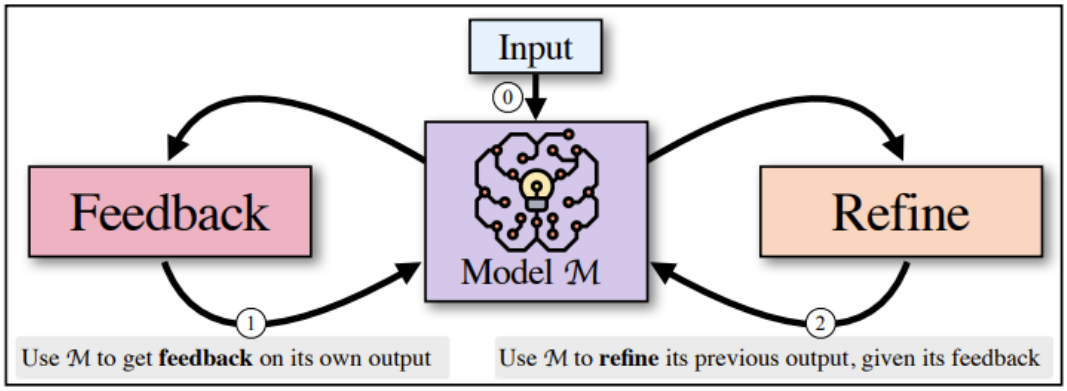

Figure 2: The process of SELF-REFINE[12]

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 our experiment, we also employed the concept of self-refinement. However, we did not strictly adhere to the SELF-REFINE framework as proposed by Madaan et al[12]. Instead, we adopted the idea of allowing the model to think and adjust, integrating it into our experiment according to our research objectives. Therefore, the self-refinement mentioned in the subsequent experiments refers not to this specific framework but to the method and concept itself.

2.4 LLM-based Multi-Intelligent Agents Collaboration

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 [18]. 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, with regard to the code generation task, previous research in references [3] and [7] 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

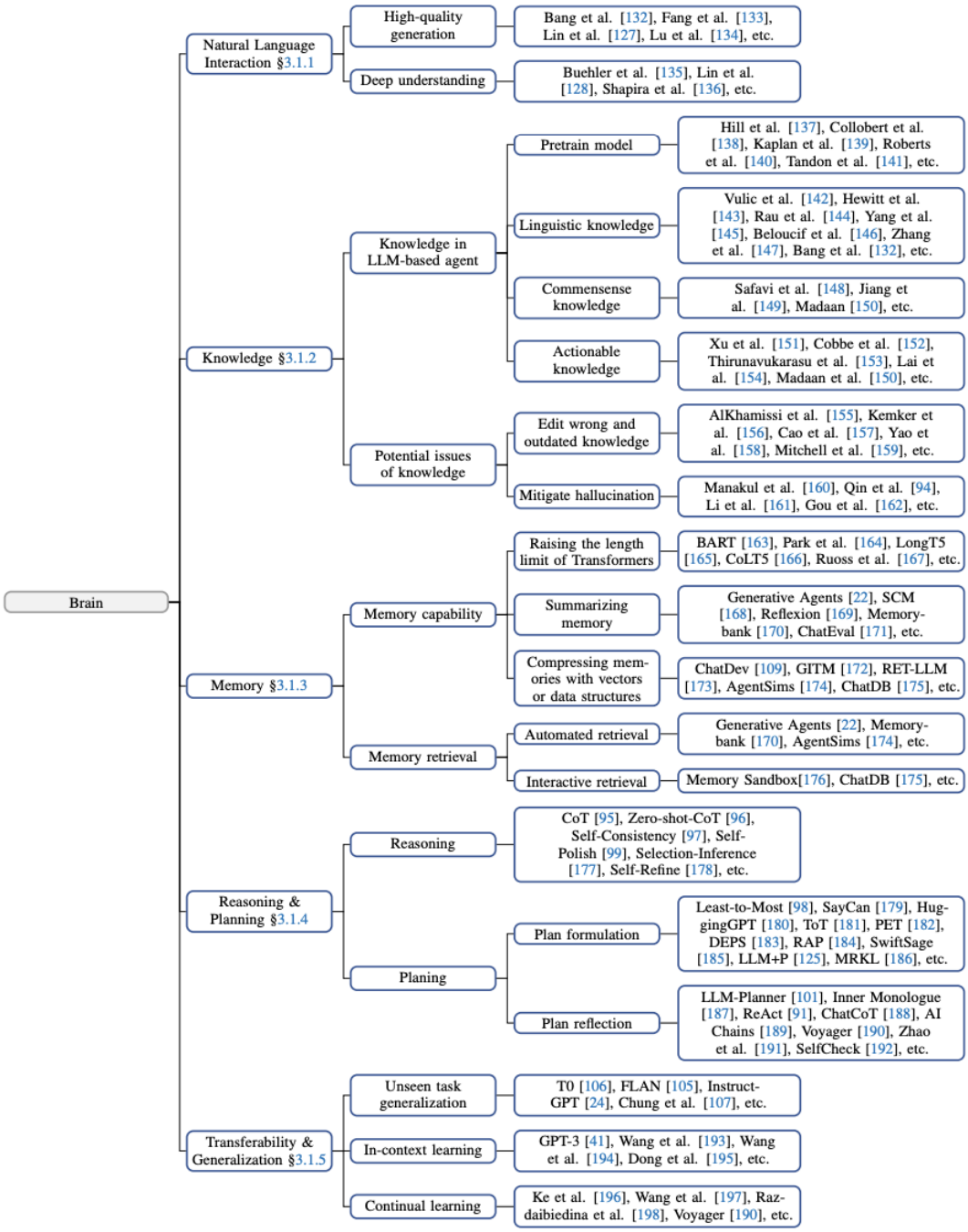

Figure 3: Related work on using LLMs as brains for agents[18].

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]

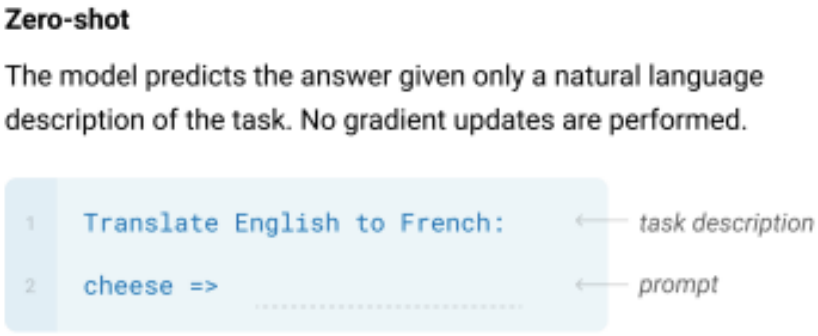

Figure 4: Example of zero-shot[1]

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]

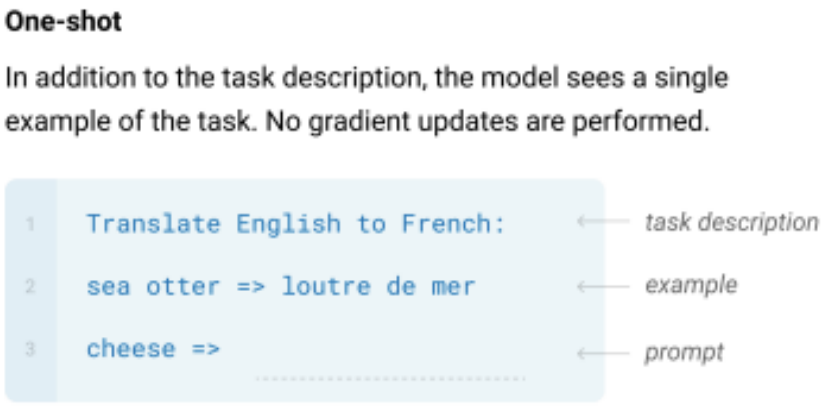

Figure 5: Example of one-shot[1]

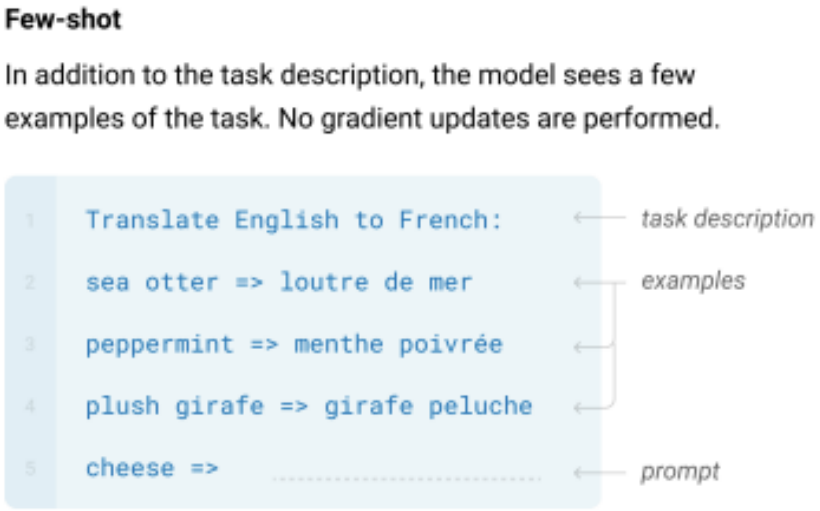

Figure 6: Example of few-shot[1]

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)[17]

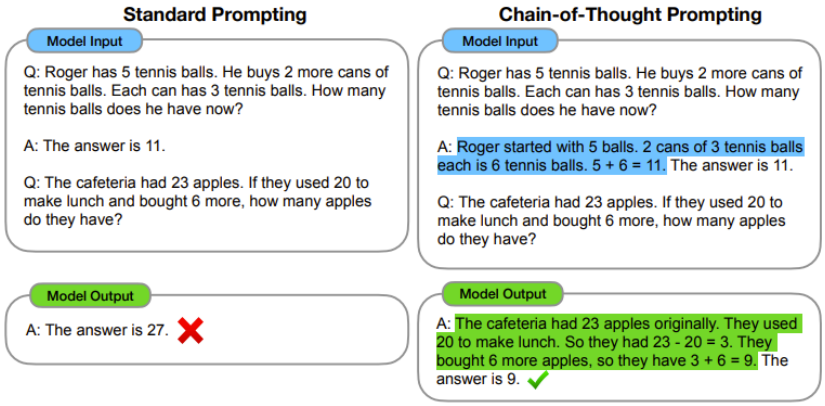

Figure 7: Example of CoT[17]

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 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 in a more fundamental and intuitive way. 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 & Benchmark Creation

3.1 Dataset Processing

APPS The Automated Programming Progress Standard, abbreviated APPS, consists of 10,000 coding problems in total, with 131,836 test cases for checking solutions and 232,444 ground-truth solutions written by humans. The data in the APPS are collected from different open-access coding websites such as Codeforces, Kattis, and more.[6]

The dataset part of our proposed timeEval benchmark is obtained from the APPS dataset after secondary processing. The specific processing flow is as follows.

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, we conducted tests on all self-contained solutions within the 5000 problems of the dataset. 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. After this step, we can ensure that all the remaining questions have at least one correct solution and that each question has enough test cases.

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. 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

	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.

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 $[0, 0.5]$. Additionally, we imposed a condition that the occurrence of wrong answers should not exceed 20% of all test cases for these selected problems. We introduced this constraint with the intention of incorporating problems where LLMs generate code correctly but inefficiently into our benchmark. This approach allows us to concentrate on evaluating the influence of different methods on the efficiency of the generated code.

In the final stage, we meticulously reviewed all the remaining questions from the preceding steps, making individual modifications to some of the test cases. Ultimately, we curated a set of 110 high-quality questions. As mentioned earlier, these questions were directly generated in the gpt-3.5-turbo model, featuring inherently inefficient code and typically compatible with multiple solutions with varying time complexities. Each problem was accompanied by 10 test cases of different sizes, spanning from very small to very large values, to underscore the efficiency differences among algorithms in the measurements.

The file structure for each problem is shown in Figure 9. The file structure corresponds to the description in Section 1.2. The specification of the problem is in the file *question.txt*. The canonical solution is in the *canonical_solution.py* file. The test cases corresponding to the problem are in *input_output.json* file. The specific provenance, difficulty, and optimal time complexity of the problem are in *metadata.json* file. A specific problem example from the dataset is shown below. Figure 10 illustrates the content in *question.txt*. Figure

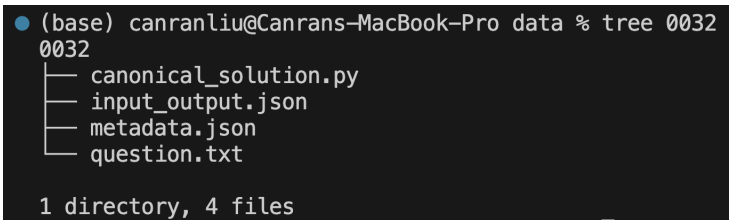

Figure 9: File structure for problem 0032 in the dataset.

13 illustrates the content in *canonical_solution.py*. Figure 11 illustrates the content in *input_output.json*. Figure 12 illustrates the content int *metadata.json*.

Figure 10: *question.txt* file for problem 0032.

Figure 11: *input_output.json* file for problem 0032.

Figure 12: *metadata.json* file for problem 0032.

data > 0032 > canonical_solution.py > ...

```python
import sys

#f = open('input', 'r')
f = sys.stdin

d = [[0] * 11 for _ in range(11)]

d[0][0] = 1
for i in range(10):
  for j in range(11):
    for k in range(10):
      if j+k <= 10:
        d[i+1][j+k] += d[i][j]

target = int(f.readline())
tt=target
target -= 1
val = 10
ans = ''
for i in range(10):
  ii=9-i
  for j in range(val+1):
    if j==10:
      continue
    jj=val-j
    if d[ii][jj] <= target:
      target -= d[ii][jj]
    else:
      val = jj
      ans += str(j)
      break
print(int(ans))
```

Figure 13: *canonical_solution.py* file for problem 0032.

3.2 Benchmark Creation

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

Execution code As depicted in Figure 14, for each problem measurement, we execute both *canonical_solution.py* and *gen_solution.py*, with the latter storing the generated code for evaluation. Each test case in the dataset, comprising input-output pairs, is utilized to test both codes, and we record the running time and correctness for each case. If

the result is correct, we label the case as *True*. If the result is incorrect, it is marked as *False*, and 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.

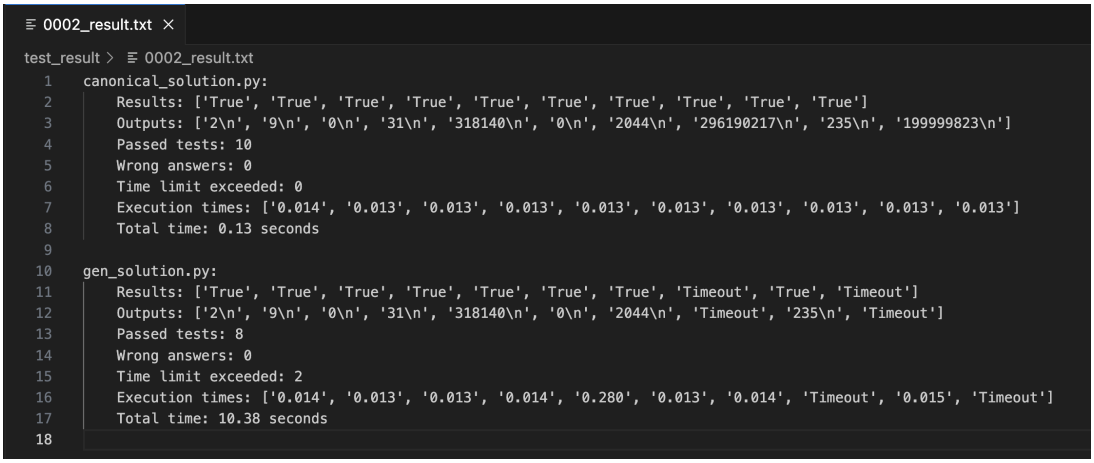

Figure 14: TimeEval performs the case-by-case evaluation of the generated code.

Metrics 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. Our metric framework is partially adapted from the work of Mandaan et al.[11], incorporating the following aspects:

- **Pass rate**: Percentage of test cases that passed the test out of all test cases.
- **Fail rate**: Percentage of test cases that failed the test out of all test cases.
- **Timeout rate**: Percentage of timeout test cases out of all test cases.

- **Percent Optimized** %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.

- **Speedup** %Sp: The ratio of the execution time of the optimal solution to the execution time of the generated program. This metric accurately describes how close in time the generated program is to the optimal solution.

$$SPEEDUP = \frac{t_{\text{opt}}}{t_{\text{gen}}}$$

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.

Time complexity analysis tool In addition to directly measuring the execution time of the generated code. We tried to create a tool that can estimate the time complexity interval of generated code. As shown in Figure 15, In the actual test we will get a dot plot with input size on the x-axis and run time on the y-axis. We will analyze it by combining the point plot and the time complexity function characteristics. For

example, the time complexity function $O(nlogn)$ can be expressed as $c * nlogn$ where c is a constant. The function has the property that the first-order derivative is greater than 0, the second-order derivative is greater than 0 and the third-order derivative is less than 0. We can combine this property with a dot plot as an additional measurement. In each calculation, we approximate the derivative of the time curve at $\frac{x_1+x_2}{2}$ by $\frac{y_1-y_2}{x_1-x_2}$.

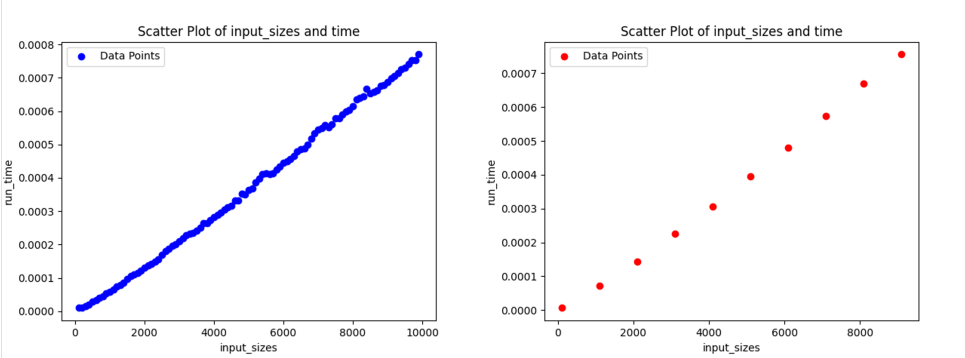

Figure 15: Left: Graph of a function of input size and program execution time with time complexity of $O(nlogn)$. Right: Simulate a benchmark test of the program shown in the left graph. This simulation is conducted by entering ten different test cases and recording the program execution time.

Nevertheless, the accuracy of this tool is currently not very high. Consequently, we opted not to utilize the estimation tool in this semester's experiment. Our plan moving forward involves continuous optimization efforts to enhance the accuracy of the tool, enabling it to provide more precise estimates of the time complexity of the code in future experiments.

4 Experiment

4.1 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 3.2.

Models. Our primary experimental foundation is the gpt-3.5-turbo model. Additionally, we conducted a comparative experiment in Section 4.3.4 using the gpt-4 model. In future work, we aim to explore a wider array 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.

4.2 Research Questions

RQ1: How do different prompts affect the efficiency of generated code? As discussed in the background section, the formulation of prompts critically influences the output of LLMs. The prompt can steer the LLM to produce specific, desired responses, effectively controlling the quality of LLMs output. Consequently, this RQ aims to investi-

gate the impact of various prompt types on the efficiency of code generated by LLMs. To this end, our experimental design will involve a comparative analysis of several classic prompt engineering techniques, including one-shot prompt, few-shot prompt, and the Chain of Thought approach. This comparison seeks to elucidate how different prompting strategies affect the performance and efficiency of LLMs in code generation tasks.

RQ2: Does self-refinement improve the efficiency of generated code? In prior studies, self-refinement has demonstrated its efficacy in enhancing accuracy in code generation tasks[4][12]. In this RQ, our objective is to investigate whether the application of self-refinement can also contribute to improved efficiency.

RQ3: How to enhance the refinement result when using the self-refinement technique. Several factors can influence outcomes when employing the self-refinement technique. In our exploration, we specifically investigated how varying the number of refinement rounds and altering the prompt can affect the results.

RQ4: Does the Multi-agent collaboration technique improve the efficiency of generated code? In this RQ, we aim to investigate whether the application of the multi-agent collaboration technique improves the efficiency of generated code and whether it ensures the accuracy of the code

RQ5: How different assignments of roles to agents and different collaborative structures will affect results. In this RQ we will explore the effect of different kinds

and different numbers of agents cooperating to generate code on the efficiency of generated code.

RQ6: The effect of other parameters or LLM types on the efficiency of generated code. In this RQ, we delved into the impact of temperature and LLM species on the results. Specifically, we explored the code generation results for temperatures ranging from 0.0 to 1.0 and evaluated the performance of two language models: gpt-3.5-turbo and gpt-4.

4.3 Experiments & Results Analysis

4.3.1 Prompt Engineering

Baseline Prompt:

In the initial stage of our experiment, it was necessary to design a baseline prompt, using the performance of the code generated by this prompt on our dataset as our baseline. Our objective was to ensure that this prompt was straightforward and concise, avoiding any excessive statements that could potentially influence the outcomes.

Consequently, the original prompt consisted of two parts. The first part involved directly inputting questions reading from our dataset into models. The second part was a specific condition: 'In the following question, you are required to generate only the code itself, producing directly executable Python code without any additional textual descriptions.' This instruction guaranteed that the answer obtained from models was Python scripts

that could be executed immediately, thereby facilitating our subsequent testing of the generated code. The exact format of this prompt is depicted in Figure 16. By utilizing the

```
def get_messages(problem):
    messages = []
    messages.append(
        {"role": "system", "content": "In the following question you only need to generate the code itself, "
         + "and directly generate a python code that can be run directly. Don't add any text descriptions!"}
    )
    messages.append(
        {"role": "user", "content": problem}
    )

    return messages
```

Figure 16: Format of baseline prompt

API, we were able to efficiently acquire a vast array of responses. The specific outcomes are presented in Table 1, serving as our baseline for comparison with the results obtained from other prompts. It is observable from the baseline that, while the code's accuracy was commendable at 0.67, both %OPT and %SP metrics were significantly low. This indicates that the efficiency of the code generated in the baseline was substantially inferior when compared to the optimal solution.

Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
67.0	1.6	31.4	1.8	11.9

Table 1: Result of baseline prompt

One-shot learning:

Our baseline prompt is zero-shot learning. Although LLMs demonstrate significant zero-sample learning capabilities, their performance remains subpar on complex tasks.

```
def get_messages(problem):
    messages = []
    messages.append(
        {"role": "system", "content":
         "Problem: You are given a 0-indexed array of integers nums of length n.
         + "Solution with poor time complexity:\ndef jump(self, nums: List[int])
         + "Solution with good time complexity:\ndef jump(self, nums: List[int])
    )
    messages.append(
        {"role": "user", "content": f"Problem: {problem}
         Please provide a solution with good time complexity.
         Please directly provide a python code that can be run directly.
         Don't add any text descriptions!"}
    )

    return messages
```

Figure 17: Format of one-shot prompt

The findings of Brown et al.(2020)[1] suggest that in-context learning can effectively enhance the capabilities of LLMs. In this experiment, considering the limitations of input text length, we employ one-shot learning. Specifically, we selected one problem from the LeetCode website and chose two solutions for this problem[9], one with poor time complexity and the other with good time complexity. We provided these to models in the format of 'Problem: problem content; Solution with poor time complexity: Python script with poor time complexity; Solution with good time complexity: Python script with good time complexity.' Then, as before, we input the problem and constraints to models, but this time we requested it to generate a solution with good time complexity. The specific format is shown in Figure 17.

Table 2 presents the results using one-shot learning. It is observed that after a single example prompt, the code generated by GPT-3.5 showed improved %Opt and %Sp

metrics on our dataset, indicating that the one-shot generated code indeed approximates the optimal solution more closely than the zero-shot. However, the improvement was marginal, and the accuracy decreased from 0.67 to 0.57, suggesting that this prompt led the model to focus more on the time complexity of the code, thereby overlooking its accuracy.

Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
56.6	23.4	20.0	22.7	37.2

Table 2: Result of one-shot prompt

Problem number	Prompt	Passed tests	Wrong answers	Time limit exceeded	Total time	Opt time	Opt
30	Zero-shot	10	0	0	1.75	0.57	0
30	One-shot	10	0	0	0.41	0.4	1

Table 3: Example: results of Problem 30

To more intuitively compare the outcomes of zero-shot learning and one-shot learning, let us consider one specific example, which is the 30th problem in our dataset. The results of two prompt on this problem are shown in Table 3 . The problem states: 'Consider the infinite sequence of integers: 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5.... The sequence is built in the following way: at first the number 1 is written out, then the numbers from 1 to 2, then the numbers from 1 to 3, then the numbers from 1 to 4 and so on. Note that the sequence contains numbers, not digits. For example, number 10 first appears in the

sequence in position 55 (the elements are numerated from one). Find the number on the n-th position of the sequence.' This problem requires the identification and analysis of a pattern in an infinite integer sequence to ascertain the value at a given position.

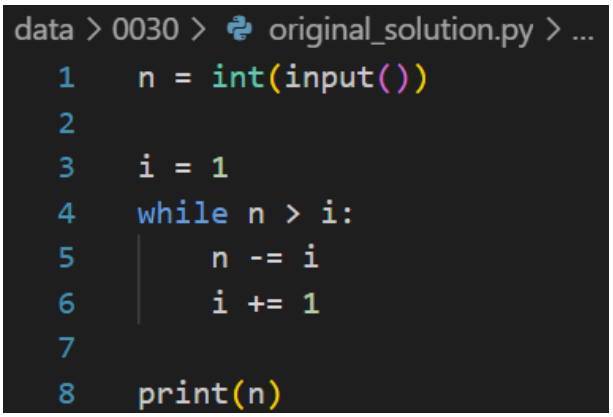

Figure 18: Zero-shot learning: the solution of Problem 30

The answer provided by zero-shot learning is illustrated in Figure 18. This solution simulates the sequence's construction process, gradually decreasing the value of n and correspondingly expanding the range of the sequence to locate the *n-th* position. It necessitates iterating until $i \geq n$, thus having a time complexity of $O(\sqrt{n})$.

In contrast, the one-shot learning solution, as shown in Figure 19, utilizes mathematical calculation to directly determine the number at the n-th position in the sequence without the need for iteration, resulting in a time complexity of O(1). Both solutions exhibit a 100% accuracy rate, but the latter's time complexity is significantly superior. This is evident from the %Opt and %Sp metrics, indicating that the one-shot learning solution approaches an optimal solution. Thus, this constitutes a successful example of one-shot learning.

data > 0030 > one-shot.py > ...

```python
def find_number(n):
    row = int((2 * n) ** 0.5)
    if row * (row + 1) // 2 < n:
        row += 1
    pos = n - row * (row - 1) // 2
    return pos

n = int(input())
print(find_number(n))
```

Figure 19: One-shot learning: the solution of Problem 30

4.3.2 Self-refinement

Prompt

a. Simple self-refinement:

In this experiment, our initial goal was to implement a basic form of self-refinement. Similar to the previously mentioned baseline prompt, at this initial stage, we aimed to keep the prompt as concise as possible to minimize unnecessary factors influencing the experiment. We input the problem and the corresponding solution from the baseline into the model, requesting improvements in time complexity. The specific format is illustrated in Figure 20.

Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
36.8	44.3	18.9	29.1	48.8

Table 4: Result of simple self-refinement

```
def get_messages(prompt, original_solution):
    messages = []
    messages.append(
        {"role": "system", "content": "In the following question you only need to generate the code itself, "
         + "and directly generate a python code that can be run directly. Don't add any text descriptions!"}
    )
    messages.append(
        {"role": "user", "content": f"There is a problem: {prompt}
         This is the code you generated previously for this problem: {original_solution}
         Please improve the time complexity."}
    )
    return messages
```

Figure 20: Prompt of the simple self-refinement

Table 4 presents the results of this experiment. We observed a noticeable improvement in both the %Opt and %Sp metrics after one round of self-refinement, indicating that for GPT-3.5, self-refinement indeed encourages the model to reassess and enhance the efficiency of its code in problems related to code efficiency. However, it is important to note a significant decrease in the accuracy of the code. We selected the second problem in our dataset as a case study to analyze this phenomenon in detail.

Problem	Experiment	Passed tests	Wrong answers	Time limit exceeded	Total time	Opt time	%Opt	%Sp
2	baseline	8	0	2	11.74	0.53	0	0.05
2	simple self-refinement	2	8	0	0.28	0.29	1	1.04

Table 5: An example of simple self-refinement

The first row of Table 5 shows the test results for the baseline solution. Out of 10 test cases, only two resulted in 'timeout', with the rest passing the test. We set our timeout threshold at 5 seconds, meaning that if a code runs for more than 5 seconds on a test case without producing a result, we stop testing that particular case, mark the result as 'timeout', and proceed to the next test case. When we increased the timeout limit to 60 seconds for retesting, it was found that all test cases were passed. This indicates

that the solution in the baseline is correct but performs poorly on test cases with larger input values due to its inferior time complexity. The second row of Table 5 shows the test results for the solution obtained after one round of simple self-refinement. We observed that this solution's test time was remarkably short, even surpassing the optimal solution, but it only passed two test cases. A detailed examination of its test results (Figure 21) revealed that out of 10 test cases, only two produced outputs, both being '0', with no results for the rest. The terminal displayed an error message: "ZeroDivisionError: integer division or modulo by zero". This suggests that the solution is fundamentally flawed, with the apparent speed improvement resulting from an error that terminated the run prematurely, not from an actual enhancement in time complexity. We do not consider such examples to be effective.

```
self_refine_solution.py:
    Results: ['False', 'False', 'True', 'False', 'False', 'True', 'False', 'False', 'False', 'False']
    Outputs: ['', '', '0\n', '', '', '0\n', '', '', '', '']
    Passed tests: 2
    Wrong answers: 8
    Time limit exceeded: 0
    Execution times: ['0.038', '0.027', '0.038', '0.048', '0.046', '0.040', '0.038', '0.043', '0.034', '0.037']
    Total time: 0.39 seconds
```

Figure 21: The detailed test result of simple self-refinement on problem 2

b. Self-refinement + test cases:

The results of the previous experiment indicate that simple self-refinement can, to some extent, enhance code efficiency but at the expense of sacrificing accuracy. Therefore, we sought to modify the prompt to enable the model to focus on both efficiency and accuracy simultaneously. Inspired by the work of Chen et al[4], we recognized that unit test is an effective approach of improving code accuracy in tasks of large language

models undergoing self-debugging. Unit test involves running the code through tests and returning the test results to the model for self-debugging, thereby enhancing accuracy.

```
Input: ['5 10\n', '2015 2015\n', '100 105\n', '1 100\n', '1 1\n', '1 2\n', '1 3\n']
Correct output: ["'2\\n'", "'1\\n'", "'0\\n'", "'16\\n'", "'0\\n'", "'1\\n'", "'1\\n'"]
Output of original solution: ["'2\\n'", "'1\\n'", "'0\\n'", "'16\\n'", "'0\\n'", "'1\\n'", "'1\\n'"]
```

Figure 22: The format of inputs and outputs

In our experiment, we write a Python script to obtain inputs for 10 test cases for each problem and their corresponding correct outputs, along with the outputs from the baseline solution for these test cases. The specific format is illustrated in Figure 22. Due to the limitations on the input length of the model, if the inputs or outputs are excessively lengthy, we will not input them into the model. We provided these test cases and their results to GPT-3.5, requesting it to improve both the efficiency and correctness of the code. The specific content of the prompt is shown in Figure 23.

```
def get_messages(prompt, original_solution, input_output):
    messages = []
    messages.append(
        {"role": "system", "content": "In the following question you only need to generate the code itself, "
         +"and directly generate a python code that can be run directly. Don't add any text descriptions!"}
    )
    messages.append(
        {"role": "user", "content": f"There is a problem: {prompt}
         This is the code you generated previously for this problem: {original_solution}
         These are inputs and output of test cases and outputs of the original code: {input_output}
         Please improve both accuracy and time complexity of the original solution according to these information"}
    )

    return messages
```

Figure 23: Prompt of the self-refinement + test cases

Table 6 presents the results of this experiment. The accuracy, compared to the baseline, still declined, but the degree of decline was not as pronounced as with simple self-

refinement. Meanwhile, both the %Opt and %Sp metrics showed improvements over the baseline, although not as significantly as with simple self-refinement. These results suggest that providing test cases to the model can indeed mitigate the decrease in accuracy caused by self-refinement to some extent, but it also tends to constrain the enhancement of code efficiency.

Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
58.3	18.1	23.6	24.5	33.5

Table 6: Result of self-refinement + test cases

c. Self-refinement + one-shot learning :

In section 4.3.1, we experimented with one-shot learning to enhance the efficiency of the code generated by GPT-3.5 and obtained positive results. Consequently, we considered applying one-shot learning in our self-refinement experiments, providing the model with an example for contextual learning to guide the direction of self-refinement.

```
def get_messages(prompt, original_solution):
    messages = []
    messages.append(
        {"role": "system", "content": "Problem: You are given a 0-indexed array of integers nums of length n
         + "Original solution:def jump(self, nums: List[int]) -> int:\n    size = len(nums)\n    dp = [float
         +  "Solution with improved time complexity:\ndef jump(self, nums: List[int]) -> int:\n    end, max_
    )
    messages.append(
        {"role": "user", "content": f"Problem: {prompt}
         Original solution: {original_solution}
         Please provide a solution with improved time complexity.
         Please directly provide a python code that can be run directly. Don't add any text descriptions!"}
    )
```

Figure 24: Prompt of the self-refinement + one-shot learning

In this experiment, we used the same example as in 4.3.1 but with slight modifications to the format. The specific format is shown in Figure 24, where 'original solution' refers to the code with inferior efficiency. Following the same format, we provided the problem and the baseline solution to the model, requesting it to emulate the example and generate a solution with improved time complexity.

Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
58.9	22.0	19.1	25.5	35.4

Table 7: Result of self-refinement + one-shot learning

The results, as shown in Table 7, were somewhat unexpected. This experiment did not show a significant improvement over simple self-refinement or one-shot learning alone. The accuracy was the highest among the three experiments, but both %Opt and %Sp metrics were at intermediate values. We speculate that such results might be attributed to that the example only provides the problem, original solution, and solution with improved time complexity, without an analytical process. The lack of guidance through an intermediate process might be the reason that its effectiveness was similar to simple self-refinement and one-shot learning.

c. Self-refinement + one-shot learning + Chain of Thought:

We hypothesized that the unremarkable results of the previous experiment were due to the lack of guidance in the intermediate analysis process. Naturally, we want to

incorporate an analysis process in the example, namely 'Chain of Thought' (CoT). CoT can simulate human thought processes by breaking down a multi-step reasoning problem into several intermediate steps. CoT is often used in conjunction with few-shot learning. By providing the model with examples and the thought process from the problem to the answer, the model's reasoning and problem-solving abilities can be effectively enhanced. In this experiment, we added the thought process from the 'original solution' to the 'solution with improved time complexity' to our previous example: "The time complexity of the original solution is $O(n^2)$ due to the nested loops that iterate through each position in the input list. To improve the time complexity, you can use a greedy algorithm because, for each position i, all the positions it can jump to can be considered as the next possible starting points. In order to minimize the number of jumps, you should aim to jump as far as possible in the next step. This means that at each step, you should choose a position within your jumping range that can take you the farthest in the next jump. This approach only requires traversing the loop once, so the time complexity is O(n)." The process can be summarized as the following questions:

- What is the time complexity of the original solution?
- Is there a better algorithm in terms of time complexity?
- What is the time complexity of this algorithm?
- How to implement this algorithm?

The specific prompt content is shown in Figure 25. Due to the need for an analytical process, we did not require the model to directly provide executable code. We saved the

```
def get_messages(prompt, original_solution):
    messages = []
    messages.append(
        {"role": "system", "content": "Problem: You are given a 0-indexed array of integers nums
         + "Original solution:def jump(self, nums: List[int]) -> int:\n    size = len(nums)\n
         + "How to improve the time complexity?\n"
         + "Answer:The time complexity of the original solution is O(n^2) due to the nested loops
         + "So the solution with improved time complexity is:\ndef jump(self, nums: List[int]) ->
    )
    messages.append(
        {"role": "user", "content": f"Problem: {prompt} \n
         Original solution: {original_solution} \n
         How to improve the time complexity?
         Answer:"}
    )
```

Figure 25: Prompt of the self-refinement + one-shot learning + CoT

responses obtained in a text file and then manually extracted the code implementation parts from each response into a Python file.

Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
35.8	55.4	8.8	60.0	84.8

Table 8: Result of self-refinement + one-shot learning + CoT

Table 8 presents the results of this experiment. We observed significant improvements in both %Opt and %Sp metrics compared to all previous experiments, indicating that guiding the model with CoT to perform reasoning analysis indeed effectively enhances the efficiency of the code. However, we also need to note a significant decrease in accuracy, the lowest of all experiments so far. This result might be due to that the model focus solely on the efficiency of the code during analysis, thereby overlooking its correctness.

Problem	Experiment	Passed tests	Wrong answers	Time limit exceeded	Total time	Opt time	%Opt	%Sp
13	baseline	7	0	3	15.49	0.53	0	0.03
13	self-refinement + one-shot + CoT	10	0	0	0.72	0.81	1	1.13

Table 9: An example of self-refinement + one-shot learning + CoT

We analyze two specific examples here. The first example is the 13th problem in our dataset. The specific results of this problem and its comparison with the baseline are presented in Table 9. The baseline test results included three 'timeouts', while the result of this experiment shows that the updated solution not only passed all test cases but also executed very quickly, surpassing even the optimal solution. Let's examine GPT-3.5's reasoning process for this problem: "The original solution checks whether the area of the remaining piece is prime by iterating from 2 to the square root of the area and checking for divisibility. This has a time complexity of $O(sqrt(area))$. To improve the time complexity, we can use a more efficient primality testing algorithm, such as the Miller-Rabin primality test. The Miller-Rabin test is a probabilistic algorithm that can determine whether a number is composite with a high probability. It has a time complexity of $O(k \cdot \log^3 n)$, where k is the number of iterations and n is the number being tested. Here is an updated solution that uses the Miller-Rabin primality test: Python script." This process follows the four analytical steps we outlined in the example and ultimately yields a correct and efficient solution. This is a successful example.

The second example is the 22nd problem in our dataset. The specific results of this problem and its comparison with the baseline are presented in Table 10. After the

Problem	Experiment	Passed tests	Wrong answers	Time limit exceeded	Total time	Opt time	%Opt	%Sp
22	baseline	5	0	5	25.3	0.51	0	2.0
22	self-refinement + one-shot + CoT	0	10	0	0.61	0.71	1	1.16

Table 10: Second example of self-refinement + one-shot learning + CoT

updates in this experiment, the solution did not pass any test cases. Let's look into GPT-3.5's reasoning process: "The original solution has a time complexity of $O(n*log(n))$ because it iterates through each number from 1 to n and calculates the product of its digits, which takes log(n) time. To improve the time complexity, we can use a dynamic programming approach. We can define a function dp(n) that returns the maximum product of digits among all integers from 1 to n. We can calculate dp(n) by considering the last digit of n and recursively calculating dp(n // 10) and dp(n - 1). Here is the updated solution with improved time complexity: Python script. This solution has a time complexity of $O(log(n))$ because it only needs to calculate dp(n) once for each digit in n." This process also follows the four steps outlined in the example and seems unproblematic at first glance. However, a closer examination of the updated code (Figure 26) reveals that while the main implementation of the code is correct, it lacks the crucial step to obtain the input value that is unrelated to algorithmic efficiency. This issue was not isolated to this problem alone. We believe that this phenomenon occurred because the model excessively focused on enhancing the time complexity and logical implementation of the algorithm, neglecting parts unrelated to the main body of the algorithm.

```python
def dp(n):
    if n < 10:
        return n
    last_digit = n % 10
    if last_digit == 9:
        return 9 * dp(n // 10)
    return max(last_digit * dp(n // 10), dp(n - 1))

print(dp(n))
```

Figure 26: The updated solution of problem 22

Round While using the self-refinement technique, we noticed that the number of rounds in the refinement may have an effect on the results. So we explored the results after going through different rounds of self-refinement. Table 11 shows the test results for refinement round = 0, 1, 2, 3. During round 1, the initial self-refinement process prioritizes the improvement of code efficiency at the cost of accuracy. This is evident in the increase in both Opt and Sp metrics, indicating that the code is running closer to the optimal solution. In the two subsequent rounds, the self-refinement technique shifts its focus, sacrificing code efficiency to enhance accuracy. The process stabilizes during these rounds, resulting in a relatively consistent performance. Based on the above analysis, we can infer that when the round of self-refinement is greater than or equal to two, we can achieve a balanced result in terms of accuracy and efficiency.

The study conducted by Huang et al.[8] raises questions about the effect of self-refinement. LLMs generate multiple samples during the self-refinement process, such as three outputs produced in two rounds of self-refinement: the initial output and the

Round	Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
0	68.5	1.6	29.8	0.0	8.3
1	61.3	17.0	21.6	20.0	32.5
2	67.8	5.2	27.0	4.5	12.9
3	66.6	6.7	26.6	3.6	12.6

Table 11: Impact of refinement rounds on generated code.

outputs generated in the subsequent rounds. Hence, there is a reasonable suspicion that the enhancement in results may be more attributed to the increase in the number of generated samples rather than the self-refinement technique itself. We therefore designed controlled experiments to dispel this suspicion.

We generate K samples at once for each problem in the process of generating code and select the code with the highest efficiency among these K samples. The results are shown in Table 12.

Best of K	Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
K = 1	68.5	1.6	29.8	0.0	8.3
K = 2	67.8	3.4	28.8	0.9	9.7
K = 3	67.7	3.3	29.0	1.8	10.0
K = 4	66.9	4.3	28.7	3.6	14.3

Table 12: Generate k samples for each problem and select the optimal sample.

Based on the above design, we can compare the round of self-refinement to the best of K where they generate an equal number of samples when round $= K - 1$. As depicted in Figure 27, even after controlling for an equal number of generated samples, self-refinement continues to play a significant role, particularly in round $= 1$ and round $= 2$. The efficiency of the generated code surpasses that of the best of K by a considerable margin.

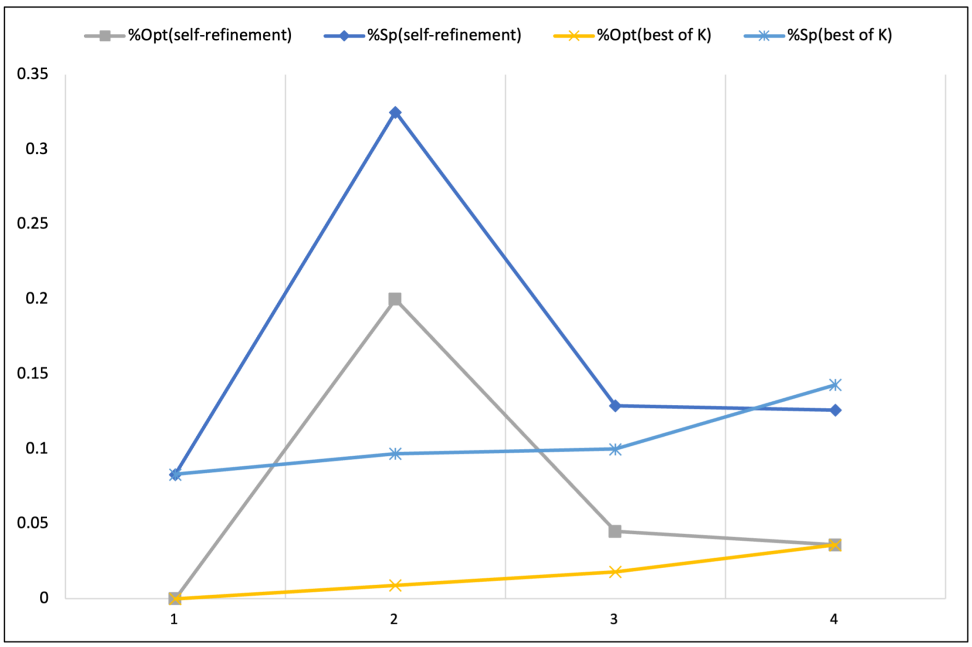

Figure 27: Comparison of self-refinement and best of K.

4.3.3 Multi-agent Collaboration

Planner+Coder In this experiment, we try to use two agents cooperating for code generation. The collaboration pattern of the two agents is illustrated in Figure 28.

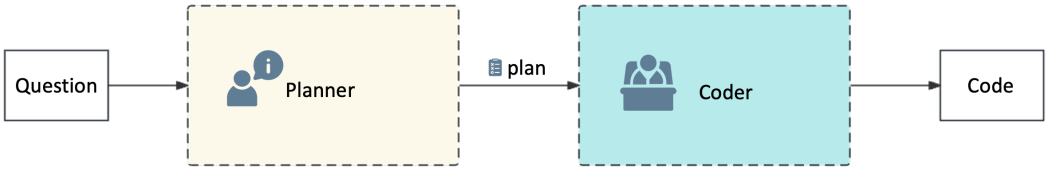

Figure 28: Planner + Coder framework

In each code generation process, the Planner first analyzes the problem, and after analyzing the problem, the Planner generates a step-by-step schedule. The Coder executes and generates code according to this plan. Figure 29 illustrates a successful plan. During this planning process, the Planner not only clearly developed a detailed plan for the subsequent code generation task. During the planning process, Planner successfully realized the existence of more efficient algorithms, thus reducing the algorithm for solving the problem from time complexity $O(n^3)$ to $O(n^2)$. Such an improvement is huge in terms of code efficiency.

```
To solve this problem, we can use a brute force approach to check all possible
combinations of side lengths a, b, and c. However, this approach would have a time
complexity of O(n^3), which would be too slow for large values of n.

Instead, we can optimize the solution by using a more efficient approach. We can
iterate over the values of a and b, and for each combination, calculate the value of
c using the Pythagorean equation. If the calculated value of c is less than or equal
to n, we can increment the count of valid triangles.

Here is a plan to solve the problem with low time complexity:

1. Read the input value of n.
2. Initialize a variable count to 0 to keep track of the number of valid triangles.
3. Iterate over the values of a from 1 to n.
   - For each value of a, iterate over the values of b from a to n.
     - For each combination of a and b, calculate the value of c using the
     Pythagorean equation: c = sqrt(a^2 + b^2).
     - If the calculated value of c is less than or equal to n, increment the count
     of valid triangles.
4. Print the value of count as the output.

By following this plan, we can solve the problem with a time complexity of O(n^2),
which is much more efficient than the brute force approach.
```

Figure 29: A step-by-step plan generated by the Planner.

The experiment results presented in Table 13 reveal that while the inclusion of Planner can enhance the efficiency of code generation to some extent, it significantly diminishes the accuracy of the generated code. This outcome is attributed to the fact that the Coder strictly adheres to the Planner's plan at each step without an error correction mechanism. Consequently, if there is a flaw in Planner's plan, there is no mechanism to revise the plan, leading to potentially irreversible issues in subsequent code generation.

To prevent such problems, we will introduce a new agent Tester to correct errors in the code generation process.

Agents	Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
Coder	68.5	1.6	29.8	0.0	8.3
Planner + Coder	37.1	44.5	18.2	29.0	46.1

Table 13: The experimental results employing Planner + Coder.

Planner+Coder+Tester In this experiment, in addition to the Planner and Coder agents from the previous experiment, we also introduced a new agent Tester. The collaboration pattern of the two agents is illustrated in Figure 30. In this scenario, Tester acts as a mediator between the Planner and the Coder. If the Tester identifies an issue with a plan generated by the Planner, the issue is communicated back to the Planner for plan correction. Once it is confirmed that the plan is error-free, it is forwarded to Coder. The coder then generates the code based on the plan and sends the code back to Tester. If the generated code does not align with the plan, the Tester reports the bugs back to

the Coder for necessary adjustments. Finally, when the Coder receives the pass message from the Tester, it outputs the generated code.

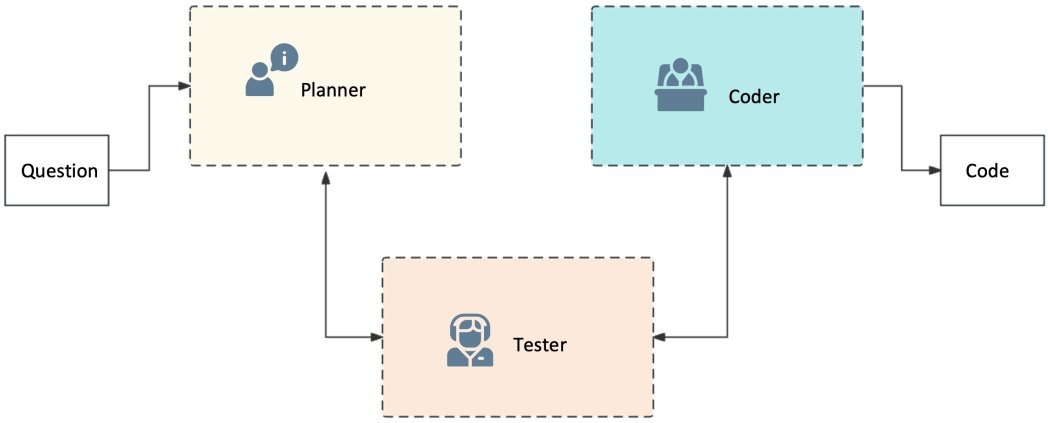

Figure 30: Planner + Coder + Tester framework

The detailed results of this experiment are outlined in Table 14. It was observed that the utilization of Tester significantly enhances the accuracy of the generated code while maintaining code efficiency, as compared to the scenario of solely employing Planner + Coder.

Agents	Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
Coder	68.5	1.6	29.8	0.0	8.3
Planner + Coder	37.1	44.5	18.2	29.0	46.1
Planner + Coder + Tester	55.6	26.6	17.7	29.1	46.9

Table 14: The experimental results employed Planner + Coder + Tester.

4.3.4 Others

Temperature Temperature is a parameter provided by OpenAI for user adjustment. The choice of sampling temperature ranges from 0 to 2. Higher values like 0.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic[14].

In the following experiments, we explored the effect of the parameter temperature on the efficiency of the generated code. As shown in Table 15, we tested the generation of temperature parameters for the gpt-3.5-turbo model in the range $[0, 1]$. This range also covers three different temperatures defined by OpenAI as low, medium, and high.

Temperature	Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
0.0	68.5	1.6	29.8	0.0	8.3
0.1	67.6	5.3	27.1	5.4	16.3
0.2	65.3	10.2	24.5	10.0	20.3
0.3	65.5	9.4	25.2	9.1	18.0
0.4	60.3	13.3	26.5	11.8	18.4
0.5	59.5	14.5	26.0	10.9	22.3
0.6	58.7	15.9	25.4	13.6	20.5
0.7	54.5	20.8	24.6	20.9	28.9
0.8	55.7	18.4	25.9	17.3	26.5
0.9	53.1	25.3	21.6	18.2	31.8
1.0	46.9	31.4	21.7	23.6	41.8

Table 15: Accuracy and efficiency of code generation under different temperatures.

The results indicate that as the temperature increases, the outcomes tend to become more creative and random. This change has resulted in a decrease in code accuracy and an increase in code efficiency. The change in code efficiency over time is well shown in the line graph shown in Figure 31.

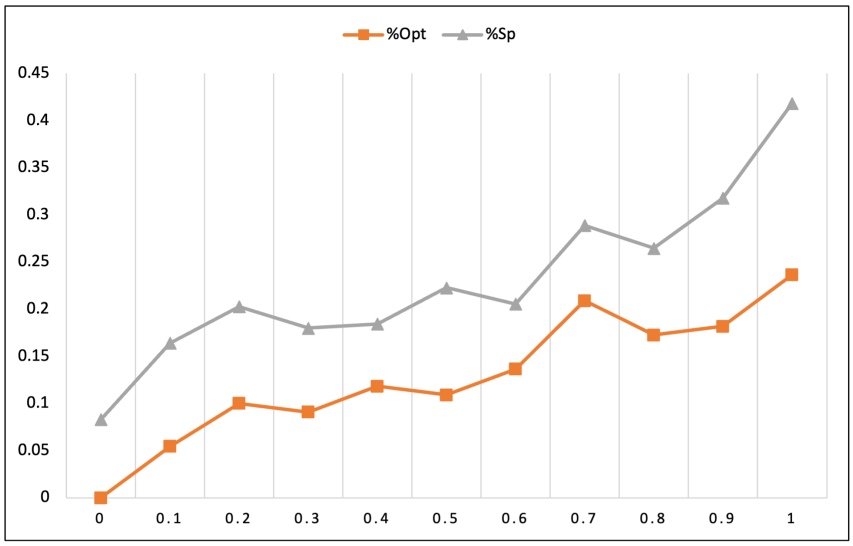

Figure 31: Variation of code efficiency with temperature.

Comparison of different LLMs In the final experiment, we compare the difference in efficiency between two LLMs, gpt-3.5-turbo and gpt-4, in terms of code generation. By observing the results we found that gpt-4 tends to sacrifice accuracy to generate code with less time complexity. The experimental results are shown in Table 16, where the efficiency of the code generated by gpt-4 is much higher than that of the code generated by gpt-3.5-turbo. However, the accuracy of the code generated by gpt-4 is slightly lower than the accuracy of the code generated by gpt-3.5-turbo.

Model	Pass Rate	Wrong Rate	Timeout Rate	%Opt	%Sp
gpt-3.5-turbo	68.5	1.6	29.8	0.0	8.3
gpt-4	58.9	34.8	6.2	57.2	73.3

Table 16: Accuracy and efficiency of generated code for different LLMs

5 Conclusion

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

- **Proposed timeEval benchmark.**We introduce the timeEval benchmark, currently the only known benchmark designed for evaluating the efficiency of generated Python code. We posit that this contribution addresses a gap in the existing dataset, providing a unified metric to measure code efficiency. We plan to publish the benchmark on GitHub after the project is finished.
- **Explored methods to improve the efficiency of the generated code.** We explore the role of many different approaches in improving the efficiency of generated code. These include but are not limited to, methods of prompt engineering, self-refinement, and multi-agent collaboration. In addition, we also explore the impact of, e.g., temperature and LLM type on the efficiency of generated code. We preliminarily demonstrate the feasibility of using the above approach to improve

the efficiency of generated code.

- **Proposed several frameworks to improve code efficiency.** Throughout the experimental process, we introduced several frameworks grounded in prompt engineering, self-refinement, and multi-agent collaboration. Some of these frameworks have demonstrated notable success in enhancing the efficiency of code generation.

6 Future Work

- **Continue to measure the different models as well as the framework on our benchmark.** Numerous LLMs, such as Codex and Code Llama, have not yet been assessed for their efficiency in code generation. Additionally, we plan to evaluate the performance of some state-of-the-art frameworks, including Language Agent Tree Search and Reflexion, not only in terms of accuracy but also in generating code efficiency.
- **Trying to create a more efficient framework.** We hold the belief that there exist more advanced code generation frameworks capable of optimizing efficiency to a greater extent. Our commitment is to continue exploring various approaches with the aim of developing improved frameworks in the future.
- **Begin an exploration of the space complexity of the generated code.** When exploring code efficiency, it is important to explore space complexity from a spatial perspective in addition to its time complexity. Time complexity is well worth

exploring as an important indicator of code efficiency. Realizing the same functionality with less memory is a very challenging task. In some cases, there is a trade-off between time complexity and space complexity. How to balance the time complexity and space complexity is also one of the focuses of our future research.

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 11, 12, 28

[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 6

[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 9

[4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. pages 3, 25, 33

[5] Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in c/c++, 2023. pages 3

[6] 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 4, 7, 15

[7] 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 3, 9

[8] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2023. pages 41

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

[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 6

[11] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2023. pages 3, 21

[12] 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 3, 8, 9, 25

[13] 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 3

[14] OpenAI. Api reference - openai api. pages 47

[15] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing 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 6

[16] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. pages 3

[17] 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 13

[18] 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 9, 10

[19] 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 6, 7

[20] 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 7

[21] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017. pages 7