2023 30th Asia-Pacific Software Engineering Conference (APSEC) | 979-8-3503-4417-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/APSEC60848.2023.00025

Refactoring Programs Using Large Language Models with Few-Shot Examples

Atsushi Shirafuji[†], Yusuke Oda[‡], Jun Suzuki[‡], Makoto Morishita[§], Yutaka Watanobe[†]
[†]*University of Aizu, Japan*
[‡]*Tohoku University, Japan*
[§]*NTT Communication Science Laboratories, Japan*
Email: {m5261161, yutaka}@u-aizu.ac.jp, {yusuke.oda.c1, jun.suzuki}@tohoku.ac.jp, makoto.morishita@ntt.com

***Abstract*—A less complex and more straightforward program is a crucial factor that enhances its maintainability and makes writing secure and bug-free programs easier. However, due to its heavy workload and the risks of breaking the working programs, programmers are reluctant to do code refactoring, and thus, it also causes the loss of potential learning experiences.**

To mitigate this, we demonstrate the application of using a large language model (LLM), GPT-3.5, to suggest less complex versions of the user-written Python program, aiming to encourage users to learn how to write better programs. We propose a method to leverage the prompting with few-shot examples of the LLM by selecting the best-suited code refactoring examples for each target programming problem based on the prior evaluation of prompting with the one-shot example.

The quantitative evaluation shows that 95.68% of programs can be refactored by generating 10 candidates each, resulting in a 17.35% reduction in the average cyclomatic complexity and a 25.84% decrease in the average number of lines after filtering only generated programs that are semantically correct. Furthermore, the qualitative evaluation shows outstanding capability in code formatting, while unnecessary behaviors such as deleting or translating comments are also observed.

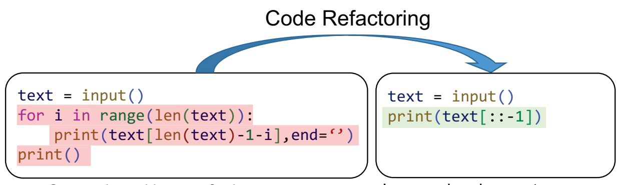
***Index Terms*—code refactoring, large language models, few-shot prompting, software complexity, programming education**

I. Introduction

Programmers and learners often write unreadable, redundant, or complicated programs because they lack the knowledge to write less complex programs or are in a hurry to meet minimum requirements by sacrificing the complexity. Writing readable and maintainable programs from the beginning is difficult for not only novices but also experts. They often iteratively make minor modifications to the previous version of the program to improve the code readability and maintainability, called *code refactoring* (Figure 1).

Complicated and unreadable programs have risks of making them harder to maintain (i.e., difficult to add, modify, or remove functions) and to hand over to a successor because the program can be understood by the author only, as well as the risks of making bugs and the difficulty of finding and fixing the potential bugs.

Programmers sometimes have no choice but to write such complicated and unreadable programs because they lack the knowledge to write the program in a less complex and more straightforward way than the one that comes to their mind, or they intentionally leave them complicated and unreadable because of the high cost of modifying them (e.g., lack of time for delivery) although they recognize that they should deal with them. According to one of the definitions of code refactoring by Fowler et al. [1], "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure," it should keep the external behavior of the previous version of the program. In fact, code refactoring also has a risk of breaking the already working program, which is another reason programmers are reluctant to do code refactoring.


Code Refactoring
Correct but with room for improvement. (Before)
Improved and correct program. (After)

```
text = input()
for i in range(len(text)):
    print(text[len(text)-1-i],end='')
print()
```

```
text = input()
print(text[::-1])
```

Fig. 1: Example of code refactoring to improve a correct program with room for improvement.

In recent years, large language models (LLMs) trained on texts in programming languages as well as natural languages [2]–[7] have shown the potential to support programming in both programming education [8]–[11] and software development [12]–[14]. From a prompt engineering perspective, White et al. [15] proposed several prompt patterns for code refactoring to support software engineering activities. As the most related work, Madaan et al. [16] used two popular LLMs, Codex and CodeGen, to improve the program's time efficiency. In another work, Madaan et al. [17] used Codex to improve the readability of variable names and comments. However, to the best of our knowledge, no prior work has demonstrated code refactoring to reduce the software complexity using LLMs.

To address the issue rising in both the fields of software development and programming education, we propose the use of an existing LLM, GPT-3.5, to suggest a complexity-improved version of a user-written complex Python program to motivate them to learn how to write better programs, as

This work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Number JP23H03508.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

well as supporting programming instructors by reducing their workloads to think and answer the questions for each learner. To leverage the prompting with few-shot examples[1] of the LLM, we propose a method to select the best-suited code refactoring examples used for few-shot prompts in each target programming problem based on the prior evaluation results of one-shot prompting for each code refactoring example. Since the LLM has the potential to break the input program, the generated programs are validated, and only functionally correct programs are suggested to a user.

In the experiments, we collect functionally correct Python programs to be refactored from a set of 44 introductory programming problems provided on Aizu Online Judge (AOJ) [18] and randomly select 20 unique programs in each programming problem. We generate 10 code refactoring candidates for each of the correct programs and verify if the generated program is syntactically and semantically correct. We demonstrate the applicability of the LLM to generate less complex programs aligned to the user-written program, evaluated both quantitatively and qualitatively.

The contributions of this work are as follows:

- We demonstrate that the LLM can generate correct and less complex programs for a majority of the input programs.
- We propose leveraging few-shot prompting by selecting refactoring examples to help guide the LLM to better align with user-written programs.
- Our quantitative and qualitative evaluations exhibit the LLM's performance in code refactoring and its potential to assist programming.
- We discuss the limitations of the current approach, providing insights for future research in leveraging LLMs for code refactoring.

II. Related Work

Code refactoring is a well-known practice in software development that aims to improve the internal structure and readability of a program without changing its external behavior [1]. Most code refactoring tasks are performed manually by developers based on experience and best practices [19]. However, there have been several research efforts to assist or automate code refactoring using various techniques.

One important target of code refactoring is code clone or code duplication, which refers to reusing several portions of code across different parts of the codebase instead of defining modular functions. Code clones increase complexity, reduce maintainability, and violate the *Don't Repeat Yourself* (DRY) principle. Refactoring the duplicated or near-duplicated code fragments makes the program more concise and easier to maintain [20], [21], and code clone detection techniques [22] can also be applied.

Several different approaches have been proposed to assist code refactoring. WitchDoctor [23] is a tool that can detect refactoring behaviors of programmers and automatically complete the refactoring while they are being performed. Blue-Pencil [24] empowers the refactoring feature of Visual Studio IntelliCode[2] to suggest repetitive edits automatically. Search-based techniques, which find code fragments that can improve program readability or complexity, have also been applied in refactoring suggestions [25]–[27].

In addition to automated code refactoring approaches mentioned above, refactoring is also used in a more narrow context within integrated development environments (IDEs). In IDEs, refactoring often refers to the process of automatically renaming identifiers, such as classes, functions, and variables, throughout the codebase. Many IDEs, such as Visual Studio[3], Eclipse[4], and IntelliJ IDEA[5], provide built-in functionality to facilitate this renaming process, ensuring consistent updates of identifiers throughout the codebase.

It is worth noting that code refactoring can be considered a code-to-code generation task in the natural language processing field since a potentially complex program written in a programming language is converted into a less complex program written in the same language. While code refactoring aims to improve already correct programs, code repair focuses on converting incorrect programs into correct ones [28]–[30]. LLMs, such as Codex [2], have shown the capability of repairing programs [11], [31]–[33]. Code editing [34]–[36] is a more generalized task that involves learning the code editing behavior. Pre-trained models can be fine-tuned on downstream tasks such as code repair [35], [36].

In recent work, White et al. [15] proposed a catalog of prompt patterns for software engineering activities using ChatGPT as a representative LLM. In particular, they provided six prompt patterns for code refactoring tasks, such as making a given program follow certain coding principles. As the most related work, Madaan et al. [16] demonstrated the capability of LLMs (i.e., Codex [2] and CodeGen [7]) to improve the time efficiency of programs by proposing a large-scale dataset consisting of (slower, faster) pairs of programs written by the same user. Similarly, Madaan et al. [17] used Codex to improve the readability of variable names and comments by leveraging the iterative refinement of the generated programs using the feedback generated by Codex, named `Self-Refine`. These works are closely related to ours because they focus on improving already correct programs in terms of performance or readability by keeping the functional correctness. In contrast, our work focuses on reducing program complexity.

III. Methodology

A. Overview

The proposed methodology is illustrated in Figure 2.

Firstly, we define code refactoring examples used in prompts. Secondly, we evaluate each example in each programming problem, and based on the performance of each

[1]In this paper, we denote *n-shot prompting* for prompting with n-shot examples.

[2]https://devblogs.microsoft.com/visualstudio/refactoring-made-easy-with-intellicode/.
[3]https://visualstudio.microsoft.com/.
[4]https://www.eclipse.org/.
[5]https://www.jetbrains.com/idea/.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

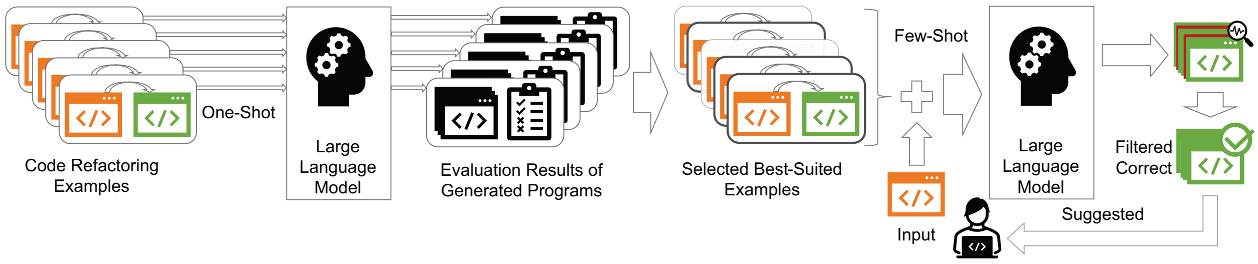

Code Refactoring Examples
One-Shot
Large Language Model
Evaluation Results of Generated Programs
Selected Best-Suited Examples
Few-Shot
Input
Large Language Model
Filtered Correct
Suggested

Fig. 2: Illustration of the proposed approach selecting the best-suited code refactoring examples for few-shot prompting for each programming problem based on the performance of one-shot prompting. Only the filtered programs are suggested to a user.

example, we select the best-suited examples for each programming problem. Thirdly, the user-written program is passed to the LLM with the selected few-shot examples, and the LLM generates several refactoring candidates. Finally, we validate the generated programs using an automatic judge system, and only functionally correct programs are suggested to the user.

B. Examples Preparation

Few-shot prompting [37] provides a few examples for LLMs to demonstrate the expected inputs and outputs of conversations, whereas zero-shot prompting provides no examples.

Figure 3 illustrates the prompting in this work. The system instruction and the user's program are always passed to the LLM. No examples are passed in the zero-shot, one in the one-shot, and three in the few-shot (i.e., 3-shot). The program on the LLM (left) side is not actually generated by the LLM but prepared by us manually. It imitates that it is generated to demonstrate the generation of the LLM.

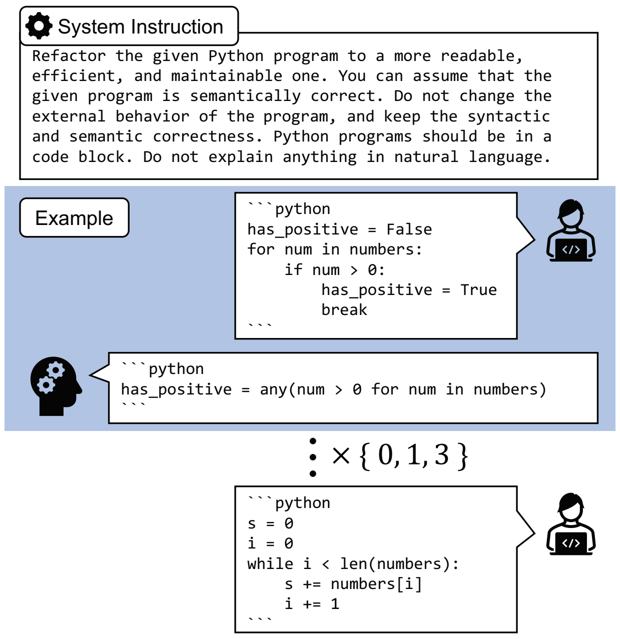
We manually define the following 10 code refactoring examples, aiming to reduce the software complexity by utilizing the defined functions and statements in Python, as well as some techniques that make the program more readable but do not directly reduce the software complexity.

- Use a formatted string.
- Use a built-in (i.e., radians) function.
- Use a logical operator instead of a nested if.
- Use a for-loop instead of a while-loop.
- Use list comprehension instead of a for-loop.
- Use the map function instead of list comprehension.
- Use a throwaway variable.
- Use the enumerate function instead of the range function.
- Use the zip function instead of the range function.
- Use a ternary operator instead of an if-branch.

C. Examples Evaluation

In this phase, each code refactoring example is used for one-shot prompting and evaluated by a validator.

Let $\mathbf{E} = \{(\text{original}, \text{refactored})\}$ be a set of code refactoring examples, where $(\text{original}, \text{refactored}) \in \mathbf{E}$ is a pair consisting of a correct program with room for improvement and its refactored version. Hereafter, let us denote $e = (\text{original}, \text{refactored})$ for short, and thus, $e \in \mathbf{E}$. Note



```
Refactor the given Python program to a more readable,
efficient, and maintainable one. You can assume that the
given program is semantically correct. Do not change the
external behavior of the program, and keep the syntactic
and semantic correctness. Python programs should be in a
code block. Do not explain anything in natural language.
```

~~~
```python
has_positive = False
for num in numbers:
    if num > 0:
        has_positive = True
        break
```
~~~

~~~
```python
has_positive = any(num > 0 for num in numbers)
```
~~~

~~~
```python
s = 0
i = 0
while i < len(numbers):
    s += numbers[i]
    i += 1
```
~~~

Fig. 3: Illustration of prompting consisting of (1) a system instruction, (2) zero/one/few-shot examples, and (3) the user's input program. The conversation in blue is the code refactoring example.

that the number of code refactoring examples in $\mathbf{E}$ is a hyper-parameter. We use $|\mathbf{E}| = 10$ in our experiments, as defined in Section III-B.

Let $\mathbf{P}$ be a set of programming problems. Then, each programming problem $P \in \mathbf{P}$ has a set of original correct programs written by users $\mathbf{X}_P = \{x\}$. Moreover, $\mathbf{Y}_{ex} = \{y\}$ is a set of code refactoring candidates generated by a pre-trained LLM using a code refactoring example $e \in \mathbf{E}$ and an input program $x \in \mathbf{X}_P$. In our experiments, we use $|\mathbf{X}_P| = 20$ programs for $|\mathbf{P}| = 44$ programming problems and generate $|\mathbf{Y}_{ex}| = 10$ code refactoring candidates.

A validator $\mathcal{V}_P$ to validate the correctness of a generated program y for the programming problem P can be denoted as

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

Formula 1.

$$\mathcal{V}_P(y) = \begin{cases} 1 & \text{if } y \text{ solves } P, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

In practice, the validator actually executes the given program using test cases and returns the correctness in binary, as described in Section IV-D.

Then, for each code refactoring example $e \in \mathbf{E}$ and each programming problem $P \in \mathbf{P}$, the score S_{eP} can be denoted as Formula 2. This score indicates how suitable the code refactoring example e is for the programming problem P.

$$S_{eP} = \sum_{x \in \mathbf{X}_P} \sum_{y \in \mathbf{Y}_{ex}} \mathcal{V}_P(y) \quad (2)$$

Finally, for each code refactoring example $e \in \mathbf{E}$, the cumulative score S_e can be denoted as Formula 3. This score reflects how suitable the code refactoring example e is overall.

$$S_e = \sum_{P \in \mathbf{P}} S_{eP} \quad (3)$$

D. Examples Selection

Let $\text{argtopk}(\mathbf{x}, k)$ a function to return the arguments at which the function returns the top-k values, the best-performed k code refactoring examples $\mathbf{E}_P$ for each programming problem $P \in \mathbf{P}$ is calculated by Formula 4.

$$\mathbf{E}_P = \text{argtopk}_e(1000 S_{eP} + S_e, k) \quad (4)$$

After selecting the best-performed code refactoring examples for few-shot prompting, we construct the few-shot prompts for the LLM. As a few-shot prompting, we adopt 3-shot prompting, which provides $k = 3$ sets of best-performed code refactoring examples.

IV. Experimental Setup

We demonstrate the applicability of an LLM in code refactoring to reduce software complexity. In this section, we describe the original dataset consisting of target programs to be refactored, the model from the representatives of off-the-shelf LLMs used to generate the refactored programs, and the objective metrics to evaluate the effectiveness.

A. Dataset

We construct a dataset consisting of the original programs to be refactored by an LLM.

1) Data Collection: To construct a dataset, we collect a variety of correct programs written in Python3 from AOJ, an online judge system where users submit programs to solve given programming problems. AOJ provides approximately 3,000 programming problems and stores 8,000,000 programs submitted by 100,000 users, including wrong or incomplete programs [38]. The source code submitted to AOJ is available for research or educational purposes and can be downloaded from the official source archive[6] or via public datasets such as CodeNet [39] or CodeContests [6].

[6]http://developers.u-aizu.ac.jp/index.

We limit the target problems from a popular course problem named *Introduction to Programming I* (ITP1)[7], which has 44 introductory programming problems, ranging from basic operations of input/output to class definitions.

Each submission of a program on AOJ has a verdict resulting from executing hidden test cases by the judge system, such as *Accepted*, *Wrong Answer*, or *Runtime Error*. We only use the submissions judged as *Accepted* because we only focus on the code refactoring task to improve correct but complex programs.

By the above collecting conditions, we collect 296,885 correct Python programs from 44 programming problems on AOJ.

TABLE I: Statistics of collected programs in each of the data preprocessing phases. LOC indicates the average number of lines, Chars and Tokens indicate the average number of characters and tokens, respectively, and CC indicates the average cyclomatic complexity.

	#Programs	LOC	Chars	Tokens	CC
Collected	296,885	14.59	332.56	108.79	5.85
Unique	161,670	15.24	346.19	111.88	6.01
Filtered	158,081	14.60	329.40	107.26	5.79
Selected	880	14.82	331.35	106.15	5.65

2) Preprocessing: After collecting the correct programs from AOJ, we preprocess the data to select partial programs for the experiments in the following phases. The statistics of the data in each phase are summarized in Table I. Also, the metrics used in the table are explained in Section IV-C.

a) Duplicate Deletion: Firstly, we remove duplicate programs in each problem. The duplicate detection is based on the characters of the raw source code, neither based on tokens nor trees. Therefore, the programs are considered different if only one space or empty line is different. After removing the duplicated programs, the number of unique programs is 161,670.

b) Outlier Deletion: Secondly, we remove outliers in each problem. The outlier criterion is $2\sigma + \mu$, where σ is the standard deviation, and μ is the mean of the number of tokens in programs in each problem. After removing outlier programs, the number of filtered programs is 158,081.

c) Randomly Selection: Finally, we randomly select 20 programs for each problem, and the final dataset contains 880 programs in total (20 programs $\times$ 44 problems $=$ 880 programs).

B. Model

To demonstrate the few-shot prompting using the pre-trained LLM, we use the representative LLM from the off-the-shelf models. We use the `gpt-3.5-turbo` engine served from the OpenAI API[8], often referred to as ChatGPT or GPT-3.5 models. More precisely, in this work,

[7]https://onlinejudge.u-aizu.ac.jp/courses/lesson/2/ITP1/all.

[8]https://platform.openai.com/docs/models/gpt-3-5.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

we use gpt-3.5-turbo-0301, a snapshot version from March 1st, 2023, for reproducibility. gpt-3.5-turbo is an InstructGPT [40]-based model, trained to follow user instructions and provide detailed responses using a technique called *reinforcement learning from human feedback* to align with the user's instructions. InstructGPT is also based on Codex [2], which is trained on massive source code. Therefore, gpt-3.5-turbo has a high capability in understanding and generating programming languages as well as natural languages.

We set the *temperature*, which determines the creativity of generated texts, to 0.2 to be slightly creative for all generations in this work. We set the *max_tokens*, which limits the maximum number of tokens to generate, to 1,024. Although gpt-3.5-turbo supports up to 4,097 tokens, it counts both the prompt and generation tokens. Given that the average number of tokens in the input programs is 106.15, as shown in Table I, the limit of 1,024 tokens is sufficient for generating refactored programs. We use the following system instruction to ask the LLM to do code refactoring: *"Refactor the given Python program to a more readable, efficient, and maintainable one. You can assume that the given program is semantically correct. Do not change the external behavior of the program, and keep the syntactic and semantic correctness. Python programs should be in a code block. Do not explain anything in natural language."* The system instruction is used in all generations, including zero-shot prompting. In addition, as instructed in the system instruction, each Python program communicated with the LLM is represented as a code block, enclosed with three backticks.

C. Metrics

Since our purpose is to suggest less complex programs as well as functionally correct programs for the user, we evaluate the generated programs from the perspective of correctness and complexity. We also report other software metrics for analysis, such as Levenshtein distance (or edit distance), lines of code (LOC), the number of characters, the number of tokens, and the character-based similarity.

a) Compilability: is the syntactic correctness of the generated program, whether the program passes the compilation, denoted by Compilability $= P/(P + F)$, where P is the number of programs that passed the compilation, and F is the number of programs that failed in the compilation. Since all the collected programs for code refactoring initially solved the problem, the compilability of the original programs is 100%.

b) Pass@k: is used to evaluate the semantic (functional) correctness of the generated programs, proposed by Chen et al. [2]. Although the metric is designed to evaluate *if the problem is solved* by generating k samples, we adopt this metric to evaluate *if the program is refactored* by generating k samples. Semantic correctness is validated by a virtual judge system, where hidden test cases are given to the program and considered correct if the program passes all the hidden test cases. Pass@k is denoted as Formula 5, where $n \geq k$ is the number of samples and $c \leq n$ is the number of correct samples.

$$\text{pass@}k := \underset{\text{Problems}}{\mathbb{E}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (5)$$

We generate $n = 10$ samples for each program and report pass@k at $k = 1, 10$. In this work, pass@1 indicates the ratio of correct programs out of all generated programs. Also, assuming to suggest the programs only validated as correct, pass@10 indicates the ratio of programs that at least one refactored program is suggested within the 10 generation tries.

c) McCabe's Cyclomatic Complexity (CC): is a software metric to measure the program's complexity [41]. CC increases with the number of if branches, for loops, and their nesting. A smaller CC is preferable, and CC < 10 is categorized as a *little risk* in software development. We employ the radon[9] library to calculate the CC.

d) Chars: is a length based on the number of characters in the program. This metric is highly influenced by the longer tokens, such as string literals, comments written in natural language, and white spaces, as well as the variable or function names. However, this metric can reflect the raw length of the program compared to the other metrics.

e) Tokens: is a length based on the number of tokens in the program. This metric can particularly reflect the number of elements (tokens) directly influencing the program's behavior, not affected by superficial texts that are already measured by the chars. We utilize the official Python tokenizer[10] to tokenize the programs. In tokenization, we exclude comments and special tokens, such as newline, indent, and dedent, to better reflect the length of the semantic elements of the program.

f) Lines of Code (LOC): is a length based on the number of lines in the program, including empty and comment lines. Another metric named Source Lines of Code (SLOC) excludes empty and comment lines in counting. Although either metric can be used in the experiments, we use LOC since it can reflect the number of insertions and deletions of empty and comment lines.

g) Distance: is a Levenshtein distance [42] calculated between the original and generated programs based on characters, also known as edit distance. Each inserted, deleted, or substituted character is counted as one distance, whereas there is another method to count substitution as two distances (i.e., an insertion and a deletion).

h) Similarity: is a syntactic similarity between the original and generated programs based on characters using the Levenshtein distance. In this work, the similarity of two programs, a, b, of length $|a|, |b|$, is defined as Formula 6, where Distance(a, b) is the Levenshtein distance between a and b.

$$\text{Similarity}(a, b) = 1 - \frac{\text{Distance}(a, b)}{\max(|a|, |b|)} \quad (6)$$

[9]https://github.com/rubik/radon.
[10]https://docs.python.org/library/tokenize.html.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

D. Execution Environment

To validate the semantic correctness of generated programs, we execute the generated program using test cases for each problem. The program is judged correct if it passes all test cases and is incorrect otherwise, similar to unit testing in software development. The test cases for each problem are available through AOJ API[11]. However, several prior works mentioned that LLMs trained on public source code have a risk of generating malicious or vulnerable programs, which may harm the host computer [2], [43]. Therefore, we prepare a virtual judge system on our isolated sandbox environment to not be affected by the malicious generated programs.

V. Results

A. Quantitative

TABLE II: Pass@k, compilability, and CC of generated programs, along with the input programs. *One-shot* indicates the best one-shot using the *list comprehension* example that performed the best among one-shot prompting on both pass@1 and pass@10. *Few-shot* indicates our proposed approach using 3-shot prompting. CC indicates cyclomatic complexity. The best score in each metric is in bold.

	Pass@1	Pass@10	Compilability	CC
Input	—	—	100.00%	5.65
Zero-shot	87.44%	93.30%	**99.99%**	4.71
One-shot	89.77%	94.77%	99.95%	4.78
Few-shot	**91.11%**	**95.68%**	99.93%	**4.67**

1) Main Results: Table II shows the main evaluation results of our proposed methodology. Both pass@1 and pass@10 scores are improved by giving more examples in prompts, and the proposed 3-shot prompting resulted in the best performance. In addition, the CC is improved (reduced) by 17.35% compared to the input programs. Although the compilability slightly decreases, we prioritize the higher scores of pass@k because the semantically wrong programs cannot be suggested to the user even if they are compilable.

2) Detailed Scores for Each Prompting: Figure 4 and Figure 5 show the detailed scores for each prompting, representing the difference from the zero-shot prompting. While the 3-shot prompting yielded the best performance on pass@10, several one-shot prompting also performed better than the zero-shot prompting. This variation of performance among the one-shot prompting highlights the need for selecting the best-suited examples for each problem.

3) Conflict Between Pass@10 and Cyclomatic Complexity: When we refer to the relationship between pass@10 and CC, we observe that these two metrics conflict. For Figure 4, the best (highest) one-shot prompting uses the *list comprehension* example, and the worst (lowest) prompting uses the *zip function* example. In contrast, as shown in Figure 5, the *zip function* example that performed the worst in pass@10 is the best (lowest) in CC. Similarly, the *list comprehension* example that performed the best in pass@10 is the worst (largest) in CC. This conflict of metrics is verified in Figure 6. We identify a strong positive correlation between the pass@10 and CC, as the correlation coefficient is 0.7916. It indicates the difficulty of reducing the CC while keeping the pass@10 simultaneously. However, to raise the availability of suggesting at least one candidate of code refactoring for each program, we prioritize raising the pass@10 in this work, and reducing the CC is our secondary importance.

4) Decrease in Cyclomatic Complexity: We test the difference in CC between the input and generated programs using the Wilcoxon signed-rank test, where the significance level is set to 0.05. We find a statistically significant difference, as the P-value is less than 0.001. As shown in Figure 7, the CC in the generated programs generally decreases.

[11]http://developers.u-aizu.ac.jp/index.

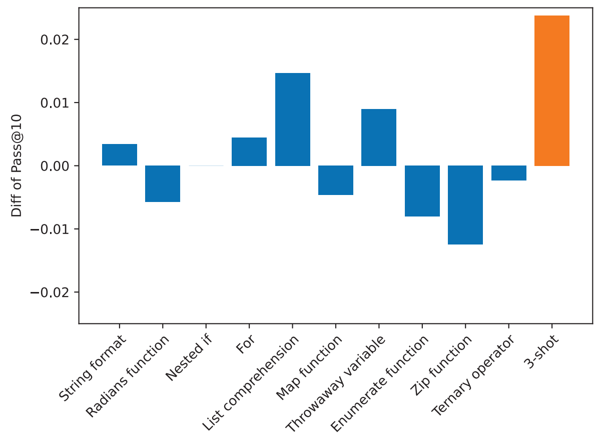


Fig. 4: The difference of pass@10 from zero-shot prompting for each prompting. Higher is better.

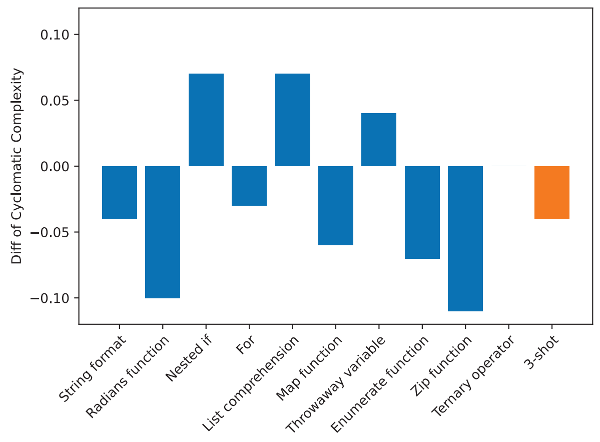


Fig. 5: The difference of CC from zero-shot prompting for each prompting. Lower is better.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

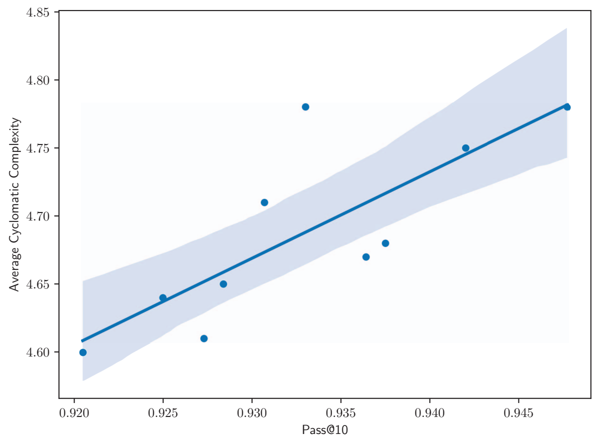


Fig. 6: A strong positive correlation (coefficient is 0.7916) between pass@10 and CC, indicating the difficulty of reducing the CC while keeping the pass@10 simultaneously.

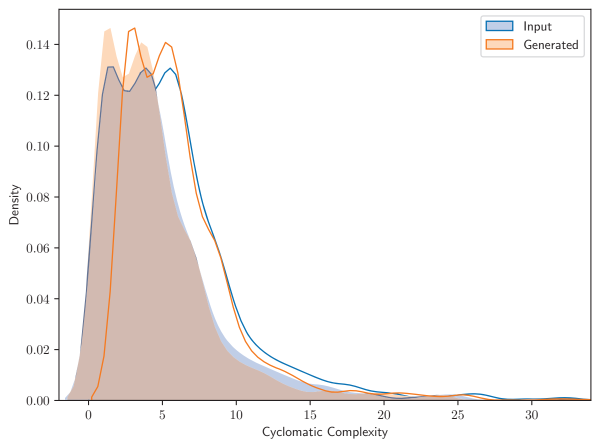


Fig. 7: Distributions on the CC for input and generated programs.

5) *Correlations in Levenshtein Distance:* Figure 8 shows a positive correlation between the CC of the input programs and the Levenshtein distance, as the correlation coefficient is 0.4886. It indicates that the LLM makes more extensive changes to the program with larger CC trying to make it less complex, whereas the LLM makes only small changes if the given program is already simple. Similarly, Figure 9 shows a strong positive correlation between the number of characters of the input programs and the Levenshtein distance, as the correlation coefficient is 0.7442. It indicates that the LLM makes longer changes to the longer programs.

6) *Decrease in LOC, Chars, and Tokens:* As shown in Table III, all of the metrics of LOC, Char-based length, and Token-based length are reduced in the generated programs compared with the input programs. Although the metrics do not directly indicate the readability and maintainability of the programs (i.e., longer programs may have descriptive variable names and comments), it shows that the generated programs are more concise while keeping the semantic correctness and reducing the complexity.

7) *Levenshtein Distance and Similarity:* In addition, as shown in Table IV, we identify that the generated programs contain at least one or more edits, as the minimum Levenshtein distance is 1 and the maximum similarity is 99.46%. This further indicates that there is no *cheating case* where the correct program is just returned without any edits to pass the semantic validation. However, on the other hand, this suggests

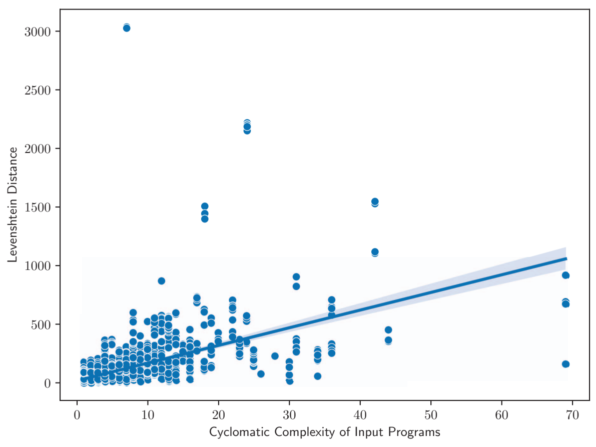


Fig. 8: A positive correlation (coefficient is 0.4886) between the CC of input programs and the Levenshtein distance, indicating that the LLM made larger modifications to the programs with higher complexity.

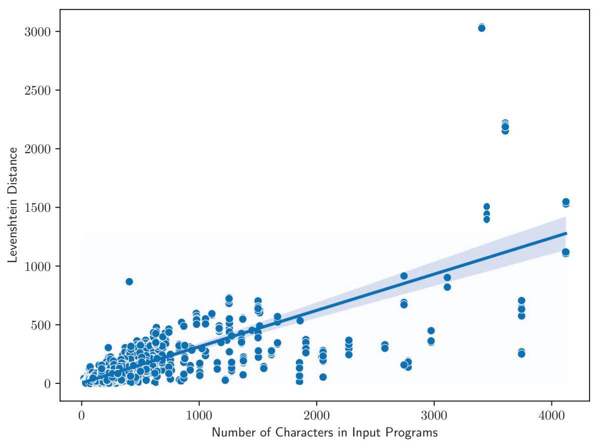


Fig. 9: A strong positive correlation (coefficient is 0.7442) between the number of characters in input programs and the Levenshtein distance, indicating that the LLM made larger modifications to longer programs.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Average values of LOC, Chars, and Tokens of the programs. The values in parentheses indicate the standard deviation.

	LOC	Chars	Tokens
Input	14.82 (± 17.29)	331.35 (± 462.24)	106.15 (± 116.70)
Generated	10.99 (± 10.76)	263.17 (± 333.94)	85.18 (± 95.68)

TABLE IV: Character-based Levenshtein distance and similarity, compared with the input programs.

	Mean (Std)	Min ∼ Max
Distance	95.59 (± 162.89)	1 ∼ 3035
Similarity	67.54% (± 16.95%)	10.76% ∼ 99.46%

that modifications can be forcibly made, even if the input program is already readable.

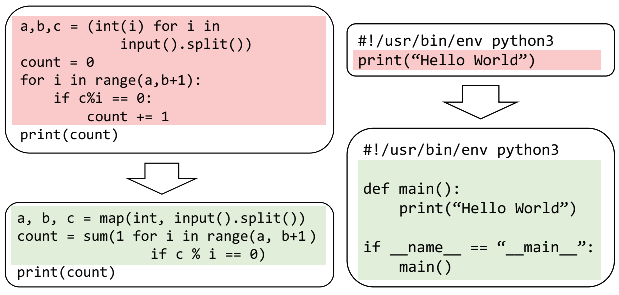
B. Qualitative

1) Main Results: For a qualitative evaluation, we manually inspect the generated programs randomly. Overall, the suggested programs contain helpful modifications to make the input programs less complex, as well as the improvement of readability and maintainability. For the readability improvement, we observe the cases of variable renaming for more descriptive variables and code formatting of adding or removing white spaces and white lines appropriately.

Figure 10 shows the representatives of improved and worsened refactorings generated by the LLM. Figure 10a is the improved example, changing from for and if statements to an effective generator. Figure 10b is the worsened example, defining an excessive function that worsens the complexity.

2) Code Formatting: To support the observation of the capability in code formatting, we perform additional analysis to calculate the character-based Levenshtein distance between the programs and their formatted programs. We employ the `yapf`[12] library to format the Python programs. While the input programs have a 21.96 (± 50.83) distance from the formatted programs on average, the generated programs have only a 7.95 (± 32.78) distance. The significantly smaller distance in the generated programs indicates that the generated programs are well-formatted and less required to be formatted.

[12]https://github.com/google/yapf.



```
a,b,c = (int(i) for i in
          input().split())
count = 0
for i in range(a,b+1):
    if c%i == 0:
        count += 1
print(count)
```

```
a, b, c = map(int, input().split())
count = sum(1 for i in range(a, b+1)
            if c % i == 0)
print(count)
```

(a) Improved example.

```
#!/usr/bin/env python3
print(“Hello World”)
```

```
#!/usr/bin/env python3

def main():
    print(“Hello World”)

if __name__ == “__main__”:
    main()
```

(b) Worsened example.

Fig. 10: Examples of refactoring by the LLM.

TABLE V: The average number of comments in the input and generated programs. Incl. and excl. indicate whether to include inline comments. The comment ratio is calculated by dividing the number of comment lines excluding inline comments by the number of lines of code including white lines.

	Comments (incl.)	Comments (excl.)	Comment Ratio
Input	0.60 (± 2.59)	0.53 (± 2.44)	2.39% (± 7.92%)
Generated	0.16 (± 0.75)	0.14 (± 0.70)	0.89% (± 4.42%)

3) Comment Deletion: Although we observe many cases improving the code readability, we also observe that the LLM deletes many comments that can help understand the program. We count the number of comments using the standard `tokenize`[13] library to support this observation. As shown in Table V, we verify that the average number of comment lines decreased from 0.60 to 0.16, which reduced the comment ratio from 2.39% to 0.89%.

TABLE VI: The proportion of natural languages used in code.

	Input		Generated	
1	English	95.11%	English	97.16%
2	Japanese	3.98%	Japanese	1.58%
3	Unknown	0.91%	Unknown	1.26%

4) Comment Translation: Similar to deleting comments, we also observe a few cases of non-English comments (e.g., Japanese) being translated into English without a demand. To support this observation, we count the languages used in the code by asking GPT-3.5 to determine the natural language. We use the engine `gpt-3.5-turbo-0301` with the parameters of $temperature = 0$ to be deterministic and $max\_tokens = 10$ to be answered concisely. We use the system instruction of *"What natural language is used in this code? Select from [English, Japanese, Korean, Chinese, Other, Unknown, None]."* to be easier to aggregate. We compare the changes in the language proportion since it is difficult to quantitatively detect what language is translated into what language for each comment, as the whole program is refactored by the LLM. The decrease in the proportion of Japanese and the increase in English in Table VI suggests that the Japanese comments are translated into English without demand. Note that the reason only English or Japanese comments are used is that the AOJ is a Japanese web service that supports only English and Japanese.

However, the decrease in the number of comments (i.e., natural language explanations) and the preferred use of English might be affected by the system instruction we used. Firstly, the instruction *"Do not explain anything in natural language."*

[13]https://docs.python.org/3/library/tokenize.html

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

might suppress the natural language explanations in the comments, although we intended to suppress explanations outside the code block (program). Secondly, the LLM might decide that the comments should be rewritten in English so that the users using English can understand them since the instruction is written in English.

VI. Limitations

As described in Section V-A, the LLM makes at least some modifications to the given program. It can worsen an already straightforward, readable, or concise program that is not required to be refactored. To avoid this, applying some techniques to make modifications only if the program requires refactoring can be beneficial. For instance, detecting whether the program should be refactored before the refactoring request, or providing some instructions to control the LLM to make modifications only when there is a particular need for refactoring, can be considered. Furthermore, since code refactoring is usually performed iteratively, suggesting the fully refactored program with extensive modifications at once can be overwhelming for users. Iterative suggestions containing a small fraction of the modifications would be more educational.

In this work, we employ GPT-3.5 as the representative LLM from the publicly available LLMs to demonstrate the effectiveness of our proposed approach. Many LLMs capable of solving programming problems have been proposed recently [44]–[48], including those that may perform even better than GPT-3.5. However, our results, which show the refactoring ability of an LLM that can be further improved by selecting better few-shot examples, can be a foundation for further advancements in applying LLMs for code refactoring tasks.

VII. Conclusion

This paper presents a methodology for using a large language model (LLM), GPT-3.5, to suggest less complex versions of user-written Python programs. The proposed approach leverages few-shot prompting with carefully selected examples to encourage users to learn how to write better programs.

The quantitative evaluation demonstrates that the LLM can generate correct and less complex programs for a majority (95.68%) of the input programs by generating 10 candidates for each. The average cyclomatic complexity is reduced by 17.35%, and the average number of lines of code is decreased by 25.84%, indicating the effectiveness of the LLM in code refactoring. Furthermore, we observe several associations in metrics, suggesting the capabilities and limitations of the LLM.

The qualitative evaluation shows that the LLM can improve code readability through variable renaming and formatting. However, we also observe that the LLM made unnecessary modifications to already straightforward and readable programs, as well as deletion and translation of comments.

Overall, this work demonstrates the potential of LLMs in supporting code refactoring and improving program complexity. Future research includes exploring using different LLMs and further improvements on making code refactoring examples to enhance the pass rates and effectiveness of the suggested refactorings.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series, Addison-Wesley Professional, 1999.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, *et al.*, “Evaluating large language models trained on code,” *arXiv preprint*, 2021.

[3] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, *et al.*, “PaLM: Scaling language modeling with pathways,” *arXiv preprint*, 2022.

[4] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, *et al.*, “PanGu-Coder: Program synthesis with function-level language modeling,” *arXiv preprint*, 2022.

[5] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, *et al.*, “InCoder: A generative model for code infilling and synthesis,” in *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

[6] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, *et al.*, “Competition-level code generation with AlphaCode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[7] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, *et al.*, “CodeGen: An open large language model for code with multi-turn program synthesis,” in *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

[8] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The robots are coming: Exploring the implications of openai codex on introductory programming,” in *Proceedings of the 24th Australasian Computing Education Conference (ACE)*, p. 10–19, 2022.

[9] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, “Automatic generation of programming exercises and code explanations using large language models,” in *Proceedings of the 2022 ACM Conference on International Computing Education Research (ICER)*, p. 27–43, 2022.

[10] M. Wu, N. Goodman, C. Piech, and C. Finn, “Prototransformer: A meta-learning approach to providing student feedback,” *arXiv preprint*, 2021.

[11] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, *et al.*, “Repairing bugs in python assignments using large language models,” *arXiv preprint*, 2022.

[12] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “IntelliCode Compose: Code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, p. 1433–1443, 2020.

[13] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models,” in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA)*, 2022.

[14] F. F. Xu, B. Vasilescu, and G. Neubig, “In-ide code generation from natural language: Promise and challenges,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, 2022.

[15] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, “Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design,” *arXiv preprint*, 2023.

[16] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, *et al.*, “Learning performance-improving code edits,” *arXiv preprint*, 2023.

[17] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, *et al.*, “Self-refine: Iterative refinement with self-feedback,” *arXiv preprint*, 2023.

[18] Y. Watanobe, “Aizu Online Judge,” 2004.

[19] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[20] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, “Assessing the refactorability of software clones,” *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[21] N. Tsantalis, D. Mazinanian, and S. Rostami, “Clone refactoring with lambda expressions,” in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 60–70, 2017.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.

[22] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[23] S. R. Foster, W. G. Griswold, and S. Lerner, “Witchdoctor: Ide support for real-time auto-completion of refactorings,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, p. 222–232, 2012.

[24] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, *et al.*, “On the fly synthesis of edit suggestions,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019.

[25] O. Seng, J. Stammel, and D. Burkhart, “Search-based determination of refactorings for improving the class structure of object-oriented systems,” in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, p. 1909–1916, 2006.

[26] M. Harman and L. Tratt, “Pareto optimal search based refactoring at the design level,” in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 1106–1113, 2007.

[27] F. Adler, G. Fraser, E. Gründinger, N. Körber, S. Labrenz, *et al.*, “Improving readability of scratch programs with search-based refactoring,” in *Proceedings of the 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 120–130, 2021.

[28] M. M. Rahman, Y. Watanobe, and K. Nakamura, “A bidirectional lstm language model for code evaluation and repair,” *Symmetry*, vol. 13, no. 2, 2021.

[29] T. Matsumoto, Y. Watanobe, and K. Nakamura, “A model with iterative trials for correcting logic errors in source code,” *Applied Sciences*, vol. 11, no. 11, 2021.

[30] A. Shirafuji, M. M. Rahman, M. F. I. Amin, and Y. Watanobe, “Program repair with minimal edits using codet5,” *arXiv preprint*, 2023.

[31] H. Joshi, J. Cambronero Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, “Repair is nearly generation: Multilingual program repair with llms,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, pp. 5131–5140, 2023.

[32] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, “Examining zero-shot vulnerability repair with large language models,” in *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2339–2356, 2023.

[33] J. A. Prenner, H. Babii, and R. Robbes, “Can openai’s codex fix bugs? an evaluation on quixbugs,” in *Proceedings of the 3rd International Workshop on Automated Program Repair (APR)*, p. 69–75, 2022.

[34] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, “CODIT: Code editing with tree-based neural models,” *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[35] J. Li, G. Li, Z. Li, Z. Jin, X. Hu, *et al.*, “CodeEditor: Learning to edit source code with pre-trained models,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, 2023.

[36] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, “CoditT5: Pretraining for source code and natural language editing,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, *et al.*, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[38] Y. Watanobe, M. M. Rahman, T. Matsumoto, U. K. Rage, and P. Ravikumar, “Online judge system: Requirements, architecture, and experiences,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 4, pp. 1–30, 2022.

[39] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, *et al.*, “CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks,” in *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks (Round 2)*, 2021.

[40] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, *et al.*, “Training language models to follow instructions with human feedback,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[41] T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[42] V. I. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[43] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the keyboard? assessing the security of github copilot’s code contributions,” in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, pp. 980–994, 2022.

[44] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint*, 2023.

[45] OpenAI, “Gpt-4 technical report,” *arXiv preprint*, 2023.

[46] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, *et al.*, “Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x,” *arXiv preprint*, 2023.

[47] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, *et al.*, “Starcoder: may the source be with you!,” *arXiv preprint*, 2023.

[48] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, *et al.*, “Wizardcoder: Empowering code large language models with evol-instruct,” *arXiv preprint*, 2023.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:52:24 UTC from IEEE Xplore. Restrictions apply.