ThinkRepair: Self-Directed Automated Program Repair

Xin Yin
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
xyin@zju.edu.cn

Chao Ni*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
chaoni@zju.edu.cn

Shaohua Wang
Central University of Finance and Economics
Beijing, China
davidshwang@ieee.org

Zhenhao Li
Concordia University
Montreal, Canada
zhenhao.li@ieee.org

Limin Zeng
Zhejiang University
Hangzhou, China
limin.zeng@zju.edu.cn

Xiaohu Yang
Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

Abstract

Though many approaches have been proposed for Automated Program Repair (APR) and indeed achieved remarkable performance, they still have limitations in fixing bugs that require analyzing and reasoning about the logic of the buggy program. Recently, large language models (LLMs) instructed by prompt engineering have attracted much attention for their powerful ability to address many kinds of tasks including bug-fixing. However, the quality of the prompt will highly affect the ability of LLMs and manually constructing high-quality prompts is a costly endeavor.

To address this limitation, we propose a self-directed LLM-based automated program repair, ThinkRepair, with two main phases: collection phase and fixing phase. The former phase automatically collects various chains of thoughts that constitute pre-fixed knowledge by instructing LLMs with the Chain-of-Thought (CoT) prompt. The latter phase targets fixing a bug by first selecting examples for few-shot learning and second automatically interacting with LLMs, optionally appending with feedback of testing information.

Evaluations on two widely studied datasets (Defects4J and QuixBugs) by comparing ThinkRepair with 12 SOTA APRs indicate the priority of ThinkRepair in fixing bugs. Notably, ThinkRepair fixes 98 bugs and improves baselines by 27%∼344.4% on Defects4J V1.2. On Defects4J V2.0, ThinkRepair fixes 12∼65 more bugs than the SOTA APRs. Additionally, ThinkRepair also makes a considerable improvement on QuixBugs (31 for Java and 21 for Python at most).

CCS Concepts

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

Keywords

Automated Program Repair, Large Language Model, Prompt Engineering

ACM Reference Format:
Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. ThinkRepair: Self-Directed Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680359

1 Introduction

Automated Program Repair (APR) is a promising approach to automatically fix bugs in computer programs, which can significantly reduce debugging time and enhance software reliability. Traditional APR techniques can be classified into heuristic-based [25, 26, 59], constraint-based [10, 24, 38, 43], and template-based [15, 19, 35, 36, 42] approaches. Template-based APRs can fix a large number of bugs using predefined templates, but are limited to these patterns and lack generalizability to other types of bugs. To address this limitation, techniques based on Neural Machine Translation (NMT) have been extensively studied in recent years [11, 20, 21, 44, 64, 65, 67]. These approaches treat fixing bugs as an NMT problem, where the goal is to translate buggy code into correct code, rely heavily on bug-fixing datasets obtained from open-source repositories.

To overcome the limitations of NMT-based APR, researchers are exploring the use of pre-trained LLMs, which generate correct code directly based on context, mitigating the need for translation from buggy code by pre-training on large amounts of open-source code snippets. AlphaRepair [62] is the first tool for cloze-style APR and its performance indicates that LLM-based APR outperforms the widely studied NMT-based APR techniques. Following that, researchers [23, 50] adopt Codex to generate a repaired code function based on the buggy one. Recently, Xia et al. [61] conducted an extensive study of LLM-based APR using various LLMs [5, 8, 14, 57] by In-Context Learning, further demonstrated the superiority of LLM-based APR.

Instead of directly generating an answer, Chain-of-Thought (CoT) prompting [58] instructs LLMs to obtain an answer with a step-by-step process, which largely improves performance on reasoning. Despite the success of CoT, studies [34, 39] have shown that the strength of LLMs depending on the few-shot examples, and low-quality examples are unable to guide LLMs to engage more profound inferential reasoning.

*Chao Ni is the corresponding author.
He is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680359

In this paper, we aim to advance APR by introducing ThinkRepair, an approach with strong analyzing and reasoning capabilities for program repair tasks. **First, we propose an LLM-based framework for APR.** LLMs are trained in an unsupervised fashion using up to billions of text/code tokens. This large-scale unsupervised learning process allows LLMs to have strong reasoning thinking and can be applied for program repair without relying on historical bug fixes. Therefore, we propose a novel LLM-based approach ThinkRepair for APR since representative conversational LLM model provides advanced capabilities for several tasks, including natural language processing [47], code generation [27], and bug-fixing [18, 56]. **Second, we develop a self-directed framework to enhance the LLM's capabilities.** Fixing bugs requires logical thinking and a coherent series of intermediate steps and few-shot CoT enables LLMs to improve their analytical and reasoning capabilities through a step-by-step process instead of generating fixed code directly. However, previous studies [28, 33, 60] have shown that the quality of prompts will highly affect LLM's reasoning abilities across various tasks. Meanwhile, manually constructing high-quality prompts is a costly endeavor. Therefore, we propose ThinkRepair, which contains two phases: (1) the collection phase aims to construct chains of thoughts that constitute pre-fixed knowledge pool and (2) the fixing phase aims to fix a buggy function by selecting high-quality examples from pre-fixed knowledge pool for few-shot learning and interacting with LLM with optional feedback testing information.

We conduct experiments on two widely studied dataset (i.e., Defects4J [22] and QuixBugs [32]) by comparing ThinkRepair with 12 state-of-the-art APR approaches. The results indicate the priority of ThinkRepair over baselines. For example, on Defects4J V1.2, ThinkRepair totally fixes 98 bugs and improves baselines by 27%~344.4%. ThinkRepair also achieve the best performance on Defects4J V2.0 and fixes 12~65 more bugs than SOTAs. Our results also show that ThinkRepair has a complementary results to the SOTAs and exclusively fixes 32 bugs (out of 98) that the SOTAs can not fix. We also collect bugs from real-world projects to evaluate data leakage. ThinkRepair can fix 19 out of 44 bugs on RWB V1.0, and 10 out of 29 bugs on RWB V2.0.

In summary, the key contributions of this paper include:

A. Novel Self-directed LLM-based APR: ThinkRepair advances LLM-based APR for program bugs. We show that LLM-based APR can achieve comparable and complementary results as other APR directions.

B. Automatic Reasoning for APR: (1) Few-shot CoT that largely enhances analyzing and reasoning capabilities to understand the semantics of the buggy function; (2) The framework with automated chains of thoughts collection, few-shot selection and interaction feedback to promote reasoning for APR.

C. Extensive Evaluation: (1) We evaluate ThinkRepair against current state-of-the-art NMT-based and LLM-based tools on the widely studied Defects4J [22] and QuixBugs [32] datasets; (2) We also conduct a further study about the data leakage in ThinkRepair by collecting new datasets from real-world projects.

2 Motivation

2.1 A Motivation Example

Fig. 1 shows the bug (Closure-56) and its fix of a Java project named Closure. The function's purpose is to extract a specified line of text from the given text content. It takes a *"lineNumber"* as input and returns the content of the corresponding line. The function first retrieves the text content and searches line by line from the beginning until it finds the specified line number or reaches the end of the file. If it successfully finds the specified line, it returns the content of that line. However, there is a logical error inside the function (i.e., Line 6). When it fails to retrieve the text content or the specified line number is invalid, this buggy function incorrectly returns *"null"*. To fix this bug, a developer modified the return statement inside the *"if (js.indexOf('\n', pos) == -1)"* block. The updated code snippet resolves the bug by checking if the variable *"pos"* has exceeded the length of the *"js"* string. If it does, it returns *"null"*, indicating that the requested line number is out of bounds. Otherwise, it returns the *"substring"* from *"pos"* to the end of the *"js"* string, effectively returning the content of the requested line.

To fix the logic error in Fig. 1, one needs to understand the semantics of the function. For example, the purpose of *"getLine"* is to extract a specified line of text from the given text content, with two additional handling cases when it cannot find the next line break. Fixing this bug requires to add additional code, which is more challenging than modifying or deleting code to fix a bug, as it demands a greater understanding ability (e.g., considering new code boundaries or new code functionality).

```
01 public String getLine(int lineNumber) {
02   ......
03   lastOffset = pos;
04   lastLine = lineNumber;
05   if (js.indexOf('\n', pos) == -1) {
06 -   return null;
07 +   if (pos >= js.length()) {
08 +     return null;
09 +   } else {
10 +     return js.substring(pos, js.length());
11 +   }
12   } else {
13     return js.substring(pos, js.indexOf('\n', pos));
14   }
15 }
```

Figure 1: Closure-56: a code logic error in Closure project

Observation 1. Fixing the above bug requires powerful code understanding and reasoning about the code logic for the given buggy function. Over the years, several NMT-based APRs [20, 21, 44, 64, 65] have been proposed, and show strong bug fixing capabilities through training on large amounts of labeled data. However, none of them has possessed powerful analytical reasoning capabilities to auto-fix the above bug, such as KNOD [20] and SelfAPR [64]. If there are no similar repair patterns in their training data, it becomes difficult to correctly fix the issue, as none of them can understand and reason to add new logic into the code for fixing.

Observation 2. Powerful models need to be empowered and guided with prior fix knowledge. Unlike current NMT-based APRs using limited number of bug fixes as training data, LLM is directly pre-trained using millions of code snippets from open-source projects, allowing it to provide a variety of edit types to fix different bugs. LLM has shown dominantly superior reasoning capabilities over any other existing AI models in natural language and code understanding [4]. We observed that LLM can correctly understand the code in Fig. 1, but cannot auto-fix it due to the missing reasoning required by fixing. The performance of LLM is influenced by prompts, and low-quality prompts are unable to guide LLM to engage in more profound inferential reasoning to fix such bugs.

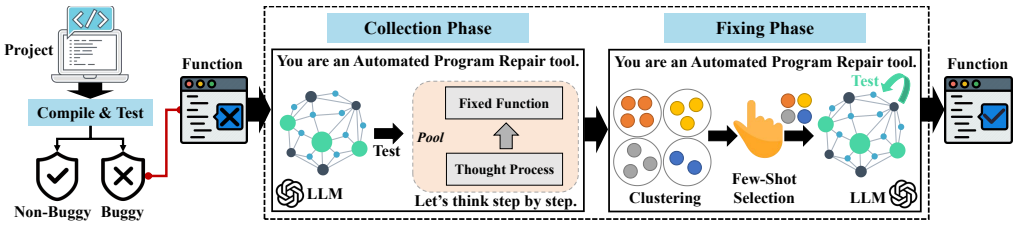

Figure 2: Overview of ThinkRepair

2.2 Key Ideas

Based on the above observations, we propose an LLM-based APR framework using chain-of-thought reasoning combined with few-shot learning to enhance the analysis and reasoning capabilities for understanding the semantics of functions.

(1) LLM-based APR. Unlike the specifically designed NMT-based APR models, LLMs are unsupervised trained using up to billions of text/code tokens. This large-scale unsupervised learning process allows LLMs to have strong reasoning capabilities and be applied for program repair without relying on training with a large amount of historical bug fixes. Therefore, we propose a novel self-directed LLM-based APR, namely ThinkRepair, since representative conversational LLM model provides advanced capabilities for several tasks, including bug-fixing [18, 56, 63].

(2) Automated few-shot CoT for APR. Automated program repair is not a trivial task since it requires logical thinking to understand the semantics of the buggy function and a coherent series of intermediate steps to fix the bugs. Even with powerful reasoning capabilities, LLMs still require some guidance in orchestrating the steps for fixing. Instead of directly generating fixed code, chain-of-thought inside LLMs can help analyze and reason the code logic. Meanwhile, few-shot examples can help LLMs to better under the faced task, but the quality of examples will highly affect the capabilities. Thus, we design an automated approach that extracts the chains of thoughts from the LLMs, selects effective examples for few-shot learning, and composes a prompt with CoTs for fixing.

3 Our Approach: ThinkRepair

ThinkRepair has two main phases (illustrated in Fig. 2 and Algorithm 1): ❶ **collection phase** and ❷ **fixing phase**. The former phase is to collect chains of thoughts that constitute pre-fixed knowledge, and the latter phase fixes a bug with CoT-based prompting and few-shot learning. In this paper, we adopt ChatGPT [47], CodeLlama [53], DeepSeek-Coder [3], and StarCoder [27] as the backend LLMs. ThinkRepair is flexible to include other state-of-the-art LLMs as the backend model. The details of ThinkRepair are presented in the following subsections.

3.1 Collection Phase

This phase aims to collect a variety of chains of thoughts that constitute the knowledge pool. To achieve this, we need to address three tasks: **(1) Prompt Preparation**, **(2) Chains of Thoughts Collection**, and **(3) Function Verification**.

3.1.1 Task 1: Prompt Preparation. The prompt used in ThinkRepair involves four important components as illustrated in Fig. 3:

- **Role Designation** (marked as ①). ThinkRepair starts a role for LLM with an instruction like "`You are an Automated Program Repair tool`".

Algorithm 1: Collection Phase and Fixing Phase

Collection Phase (Section 3.1)
Input: Buggy functions F;
for f *in* F **do**
  1: Combine the buggy function f into the prompt;
  2: Generate CoT and fixed function by prompting the LLM;
  3: Test the fixed function, retaining the valid function (with their buggy function and CoT) into K;
Output: Knowledge Pool K;

Fixing Phase (Section 3.2)
Input: Knowledge Pool K, Buggy function f, interaction=1;
1: Select few-shot examples E from knowledge pool K;
2: Combine the example E and buggy function f into the prompt;
3: Generate CoT and fixed function by prompting the LLM;
4: **while** *Fixed Function is invalid && interaction++ < 5* **do**
  Add test failure information to prompt and regenerate;
Output: Fixed function f';

- **Task Description** (marked as ②). LLM is provided with the description constructed as "`// Provide a fix for the buggy function`". Since we illustrate an example in Java, we use the Java comment format of "`//`" as a prefix.
- **Buggy Function** (marked as ③). ThinkRepair provides the buggy function to LLM in our single-function fixing scenario. We also prefix the buggy function with "`// Buggy Function`" to directly indicate LLM about the context of the function.
- **Chain-of-Thought Indicator** (marked as ④). LLM is instructed to think step-by-step when fixing a bug. In this paper, we follow the best practice in previous work [58] and adopt the same prompt named "`Let's think step by step`".

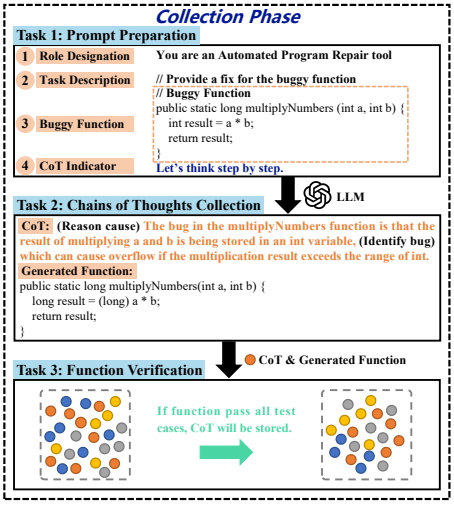

Figure 3: The Process of the Collection Phase

3.1.2 Task 2: Chains of Thoughts Collection. Given a corpus of buggy functions, ThinkRepair uses the decorated prompt to collect chains of thoughts on fixing the buggy functions. The output of Task 2 is a collection of samples, where a sample includes a buggy function, its fixed version, and the chain of thought. As an example in Fig. 3, the *multiplyNumbers* function initially fails to consider the possibility of an overflow bug when performing multiplication operations. LLM reasons that the result of multiplying *"integer a"* and *"integer b"* inside *"multiplyNumbers"* function can cause a bug. The result is originally planned to be stored in another integer variable, which may cause an overflow bug if the multiplication result exceeds the range of an integer. Thus, it suggests converting the type of multiplication results (i.e., *"result"*) into a *"long"* type and consequently resolving the overflow problem.

3.1.3 Task 3: Function Verification. To get effective samples, it is imperative to filter out low-quality thought processes. ThinkRepair runs a test suite (originally supported by the studied dataset, cf. Section 4.1) to test the fixed functions extracted from LLM's output in Task 2, retaining only the fixed functions (with their buggy functions and CoTs) that successfully pass the entire test suite. Meanwhile, LLM may not always correctly fix one buggy function at the first attempt. Thus, we execute the process at most 25 attempts for each bug (refer to Section 4.3 for more details).

3.2 Fixing Phase

In this phase, ThinkRepair first selects diverse and effective samples from the knowledge pool in the collection phase. ThinkRepair automatically utilizes selected examples (i.e., buggy function as well as its corresponding fixed version appended with reasoning process) and the targeted buggy function (i.e., function to be fixed) to compose a prompt to interact with LLM. Finally, ThinkRepair obtains the output from LLM including both the chains of thoughts and the candidate fixed function to the buggy one. Notice that each candidate function generated by LLM will be passed through a function verification step and the feedback from the verification step will be appended to LLM for further refinement. Overall, the fixing phase has three tasks: **(1) Few-Shot Selection**, **(2) Automatic Fixing**, and **(3) Interaction Verification**.

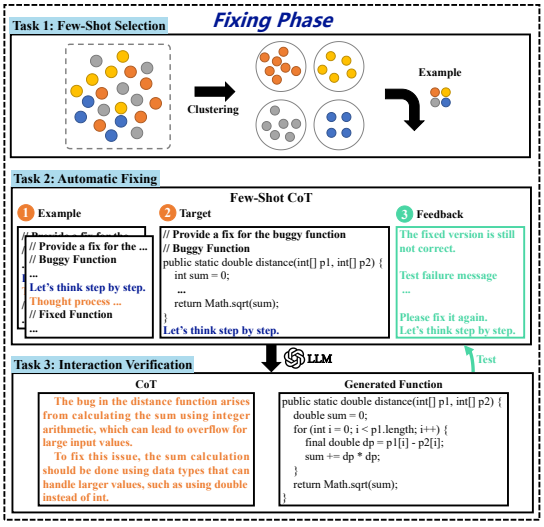

Figure 4: The Process of the Fixing Phase

3.2.1 Task 1: Few-Shot Selection. LLM needs a high-quality prompt to instruct itself to finish the downstream tasks, which is also the focus of prior works [12, 60, 63]. Similarly, we aim to reduce labor involvement in instructing LLM by guiding it to learn from the solved problems. To achieve this, we need to identify the most beneficial examples from the knowledge pool. Moreover, previous work [66] concludes that a few diverse examples may help LLM achieve a better generalization ability. Therefore, ThinkRepair clusters these examples inside the knowledge pool on the basis of their semantic similarity to pick out distinct ones [28, 66]. Furthermore, considering the limitations of LLMs' conversation windows, all examples in the knowledge pool are clustered into two clusters, with one sample selected from each cluster (i.e., two shots used in this paper). We employ two advanced embedding strategies (i.e., Semantic-based Selection and Contrastive-based Selection) to select semantically similar examples. For comparison, we also use IR-based Selection and Randomly Selection, and the details are elaborated as follows.

- **Semantic-based Selection** adopts a pre-trained model (i.e., UniXcoder, which effectively comprehends code semantic information [17, 45]) to embed all the buggy functions and then uses the K-means algorithm [41] for clustering. During the fixing phase, the most semantic similar examples are picked out from each cluster based on cosine similarity.
- **Contrastive-based Selection** utilizes the contrastive learning framework R-Drop [46] to further fine-tune UniXcoder for better semantic embedding. We input one function twice to get the embeddings E_1 and E_2. The training objective is the distance between E_1 and E_2 should be as small as possible. For clustering, it remains the same operation as Semantic-based Selection.
- **IR-based Selection** builds indexes for code in the knowledge pool and then retrieves similar examples by BM25 score [52].
- **Randomly Selection** randomly selects a few examples from the knowledge pool built in the collection phase.

3.2.2 Task 2: Automatic Fixing. ThinkRepair utilizes the selected examples and the target buggy function (i.e., to be fixed) to construct a prompt (i.e., ① + ② marked in Fig. 4). Then, ThinkRepair uses this prompt to interact with LLM and help it to infer the bug-fixing solution. Following that, we can obtain the model outputs, which usually contain the process of LLM's thought and the candidate fixed function, which will be verified in the next step.

3.2.3 Task 3: Interaction Verification. ThinkRepair complies and runs test suite to verify all candidate fixed functions generated by LLM. In case a candidate function fails to pass all test cases, ThinkRepair first collects the failing test information, which can aid LLM in understanding the failure causes and provide guidance to generate the correct fix. In particular, the test failure messages can be divided into four categories: (1) *Compile Fail*, (2) *Time Out*, (3) *Syntax Error*, and (4) *Failing test*: *TestClass::TestFunction*. Then, ThinkRepair reconstructs the prompt and appends the failing information (i.e., ③ marked in Fig. 4) to the back of the original prompt (i.e., ① + ② marked in Fig. 4). Here, the failing information is added into a template: *"The fixed version is still not correct. {test failure message}. Please fix it again. Let's think step by step."* Then, ThinkRepair interacts with LLM using the new prompt (i.e., ①+②+③) to generate a new fixed solution. LLM can avoid generating similar

mistakes and also learn from previous interactions based on new prompt. This iterative process continues until a fixed function is obtained (i.e., successfully passing the entire test suite) or the maximum number of interactions exceeds (i.e., five times for a better balance between effectiveness and cost, refer to Section 5.3).

4 Experimental Design

We present the experimental design, including studied datasets, baselines, evaluation metrics, and experiment settings.

4.1 Datasets

For the evaluation, we follow previous studies [61–63] to adopt the two APR benchmarks (**Defects4J dataset** [22] and **QuixBugs dataset** [32]) spanning across two popular programming languages (i.e., Java and Python). Similar to prior APR studies [61–63], we separate Defects4J into Defects4J 1.2 and Defects4J 2.0. Defects4J 1.2 consists of 391 bugs in 6 different Java projects, while Defects4J 2.0 consists of 438 new bugs across 9 additional projects. We also focus on scenarios where the fix is solely located in a single function since it is the focal point of most recent APR work [21, 40, 61, 62, 65, 67]. We filter out the datasets to contain single function bugs. The statistic of each evaluation dataset is presented in Table 1.

Table 1: Statistics of studied dataset

Dataset	# Total Bugs	# SF Bugs	# SH Bugs	# SL Bugs
Defects4J 1.2	391	255	154	80
Defects4J 2.0	438	228	159	78
QuixBugs-Java	40	40	37	37
QuixBugs-Python	40	40	40	40
# Sum	**909**	**563**	**390**	**235**

*Defects4J 1.2 and Defects4J 2.0 are two completely independent versions, with no overlapping bugs between them.

4.2 Baselines and Evaluation Metrics

Studied Baselines. We compare ThinkRepair with the twelve state-of-the-art APR approaches in Table 2, including eight NMT-based and four LLM-based SOTAs in APR. AlphaRepair is an LLM-based repair tool that employs the pre-trained CodeBERT model [13] with cloze-style APR, which means that it does not require fine-tuning on bug-fixing data. Codex and GPT-NeoX directly applied LLMs for APR without fine-tuning [61]. ChatRepair is similar to our work, which repairs a bug with a conversational ChatGPT. The NMT-based APR models require fine-tuning for better performance, while LLM-based APR approaches do not require fine-tuning.

Evaluation Metrics. Following previous work [20, 21, 29–31, 40, 44, 61, 62, 64, 65], we adopt two widely used metrics for evaluating approaches: *(1) number of correct patches* and *(2) number of plausible patches*. A plausible patch is a patch that can pass all test cases but is not semantically equivalent to the actual fix. Following the previous work [61, 62], we also manually check and identify the plausible patches that are semantically equivalent to the actual fixes.

4.3 Implementation

We developed the generation pipeline in Python, utilizing PyTorch [49] implementations of CodeLlama 13B, DeepSeek-Coder 7B, and StarCoder 16B. We use the Hugging Face [1] to load the model weights and generate outputs. For ChatGPT, we utilize OpenAI's API access [7], complying with the recommended best practices [55] for each prompt. We utilize the gpt-3.5-turbo model from the ChatGPT family, which is the version used uniformly for our experiments. A sampling temperature of 1 is utilized to obtain a diverse set of potential patches [16]. The maximum number of repair attempts is set to 25 (i.e., 25 independent sessions) for ThinkRepair as the default for auto-fixing the single function bugs. Since LLM has a maximum input limit, we employ two few-shot examples and set the maximum interaction number to 5 (i.e., up to 5 interactions in a session). The evaluation is conducted on a 16-core workstation equipped with an Intel(R) Xeon(R) Gold 6226R CPU @ 2.90Ghz, 192GB RAM and NVIDIA RTX 3090 GPU, running Ubuntu 20.04.1 LTS. Following previous APR works [62, 67], we use a default end-to-end timeout of 5 hours to fix one bug. In practice, the total time required is on average lower than 20 minutes since we sample small-scale patches (i.e., 25 attempts × 5 interactions) for each bug.

Table 2: The studied baselines

Approach	Category	Fine-Tuning	Time/# Patch	Venue
KNOD [20]	NMT-based	✓	5 Hour	ICSE 2023
TENURE [44]	NMT-based	✓	5 Hour	ICSE 2023
RewardRepair [65]	NMT-based	✓	200	ICSE 2022
SelfAPR [64]	NMT-based	✓	no setting reported	ASE 2022
CURE [21]	NMT-based	✓	10,000	ICSE 2021
DeepDebug [11]	NMT-based	✓	100	arXiv 2021
CoCoNuT [40]	NMT-based	✓	10,000	ISSTA 2020
DLFix [29]	NMT-based	✓	5 Hour	ICSE 2020
ChatRepair [63]	LLM-based	✗	≤ 200	arXiv 2023
Codex [61]	LLM-based	✗	200	ICSE 2023
GPT-NeoX [61]	LLM-based	✗	200	ICSE 2023
AlphaRepair [62]	LLM-based	✗	5,000	FSE 2022
Ours: ThinkRepair	LLM-based	✗	≤ 125	This work

"**Time/# Patch**": indicates the longest time set for fixing a bug or the maximum number of candidate patches that a model was set to loop through via running test cases before a correct patch is obtained.

5 Experimental Results

To investigate the effectiveness of ThinkRepair on bug fixing, our experiments focus on the following three research questions:

- **RQ-1 Comparable Study on LLM-based APRs.** *How does the performance of ThinkRepair compare with the LLM-based APRs?*
- **RQ-2 Comparable Study on NMT-based APRs.** *How does the performance of ThinkRepair compare with the state-of-the-art NMT-based APRs?*
- **RQ-3 Sensitivity Analysis.** *How do different configurations affect the overall performance of ThinkRepair?*

5.1 RQ-1: Compare with LLM-based APRs

RQ1-Analysis Procedure. We evaluate ThinkRepair against four LLM-based APRs: ChatRepair [63], Codex [61], GPT-NeoX [61], and AlphaRepair [62]. We adopt ChatGPT [47], CodeLlama [53], DeepSeek-Coder [3] (we refer to as DeepSeek), and StarCoder [27] as the backend LLMs for ThinkRepair. In addition, we build baseline approaches, BaseChatGPT, BaseCodeLlama, BaseDeepSeek, and BaseStarCoder, which utilize the basic LLMs to perform a repair (i.e., directly using existing bug-fixing data as examples) without any Chains-of-Thoughts reasoning process and feedback information.

Data Splitting. Our ThinkRepair has two phases: the *Knowledge Collection* and *Fixing*. We use the buggy functions in Defects4J V2.0 (i.e., 228) for collection and the ones in Defects4J V1.2 (i.e.,

Table 3: RQ1: ThinkRepair vs. Basic LLMs for different projects on Defects4J V1.2

Projects	ThinkRepair with perfect fault info				Basic LLM with perfect fault info			
	ChatGPT	CodeLlama	DeepSeek	StarCoder	BaseChatGPT	BaseCodeLlama	BaseDeepSeek	BaseStarCoder
Chart	11	10	10	8	5	6	6	5
Closure	31	20	14	20	13	11	9	14
Lang	19	13	11	12	11	10	7	7
Math	27	21	20	16	16	12	13	9
Mockito	6	4	4	2	6	4	3	2
Time	4	2	4	1	1	1	1	0
# Sum	98	70	63	59	52	44	39	37

255) for the fixing phase. Defects4J V1.2 and Defects4J V2.0 are two completely independent versions, with no overlapping bugs between them. In addition, we also conduct an experiment to use functions in Defects4J V1.2 for collection and the ones in Defects4J V2.0 for the fixing phase. Since QuixBugs dataset has the limited number of functions (i.e., 40 for both Java and Python), for comprehensively evaluating the performance differences, we adhere to the best-practice guide [55], manually designed 2 examples for fixing bugs without the collection phase.

Fault Information. Following previous work [21, 40, 61, 62, 65], we focus on two settings: (1) No fault localization is performed, the perfect fault information (i.e., including statement-level fault information) is provided; (2) The fault information at method-level is provided but no statement-level fault information.

Experiment Settings. Since all of the studied baselines were already evaluated on Defects4J, following the convention in APR work [61, 62], we directly compare the auto-fix results obtained in previous studies [61–63] under the same settings.

As for the few-shot selection strategy during the fixing phase, we adopt the *Contrastive-based Selection* strategy since it has the overall best performance. Meanwhile, we set the number of interaction feedback as five since it achieves a better balance between effectiveness and cost (cf. Section 5.3).

In ThinkRepair, we study 3 different repair scenarios used in previous works [61, 63]: *single function*, *single hunk*, and *single line*. Note that *single hunk* fix is a subset of *single function* fix and *single line* fix is a subset of *single hunk* fix. In QuixBugs-Java, *single hunk* is equal to *single line* (i.e., all *single hunk* bugs require fixing just one line of code), while in QuixBugs-Python, all fixes are *single line*.

The experimental settings for ThinkRepair: only *single function* scenario is considered, the results for *single hunk* and *single line* come from *single function* results. Whereas, the experimental settings for ChatRepair, Codex, and GPT-NeoX are that *single function*, *single hunk*, and *single line* scenarios are experimented separately, and the results of three experiments are independent to each other. These models conducted experiments on the single function bugs in the same setting as ours, we therefore compared with them on single function setting only.

RQ1-Results. ThinkRepair vs. Basic LLMs. Table 3 illustrates the number of bugs successfully repaired by ThinkRepair and Basic LLMs for different projects in the scenario of *single function* bug-fixing. We observe that with perfect fault information, ThinkRepair demonstrates superior performance across all projects compared to Basic LLMs. In particular, the performance of ThinkRepair is significantly better than Basic LLM with improvements ranging from 59.1%~88.5%. This not only demonstrates the superiority of our ThinkRepair approach, but also its universal applicability, as it is not tailored for any specific LLM, but is suitable for various LLMs.

Table 4: RQ1: ThinkRepair vs. LLM-based APRs (SF: Single Function, SH: Single Hunk, SL: Single Line)

Models	Defects4J						QuixBugs		
	V1.2			V2.0			Java		Python
	SF	SH	SL	SF	SH	SL	SF	SH	SF
Method-level fault info.									
Codex	63	-	-	-	-	-	32	-	37
GPT-NeoX	18	-	-	-	-	-	8	-	19
BaseChatGPT	36	28	20	39	29	23	38	36	35
ThinkRepair*	63	46	33	62	44	25	38	36	38
ThinkRepair	80	64	44	90	69	41	39	36	40
Perfect fault info.									
AlphaRepair	67	57	48	-	-	35	28	27	27
ChatRepair	76	-	-	-	-	-	39	-	40
BaseChatGPT	52	44	31	46	35	25	38	36	37
ThinkRepair*	70	55	37	72	53	36	38	36	38
ThinkRepair	98	78	52	107	81	47	39	36	40

"-": indicates no results reported in the original work, or cannot be directly compared since different experimental settings.

This underscores its LLM-agnostic design paradigm. To facilitate comparison, we employ the top two performing models, ChatGPT and CodeLlama, in the following sections of our results, denoting them as ThinkRepair and ThinkRepair*, respectively.

ThinkRepair vs. ChatGPT-based APRs. With the perfect fault information provided, ThinkRepair can auto-fix 98 bugs and 22 more bugs (28.9% improvements) than ChatRepair for single function bugs in Defects4J V1.2. For the QuixBugs dataset, ThinkRepair and ChatRepair can auto-fix the same number of bugs. Both models can auto-fix all bugs in QuixBugs-Python, but miss one bug in QuixBugs-Java.

ThinkRepair can outperform the BaseChatGPT in all settings. For example, with the method-level fault information, ThinkRepair can auto-fix 122.2%, 128.6%, and 120% more bugs than BaseChatGPT for single function, single hunk, and single line bugs in Defects4J V1.2, respectively. When provided with perfect fault information, ThinkRepair can auto-fix 88.5%, 77.3%, and 67.7% more bugs for single function, single hunk, and single line bugs in Defects4J V1.2. Similarly, on Defects4J V2.0, ThinkRepair achieves an overwhelming better performance than BaseChatGPT.

Table 5 illustrates the number of bugs successfully repaired by ThinkRepair and LLM-based APRs for different projects in the scenario of *single function* bug-fixing. ThinkRepair and ThinkRepair* represent for the results of our approach with ChatGPT and CodeLlama as the backend LLMs, respectively. We observe that with or without perfect fault information, ThinkRepair demonstrates superior performance across all projects compared to BaseChatGPT. For example, for the project Closure, ThinkRepair improves the BaseChatGPT by 280% and 138.5% with the method-level and perfect fault information, respectively.

ThinkRepair vs. other LLM-based APRs. For single function bugs in Defects4J V1.2, ThinkRepair can auto-fix 31 more bugs

Table 5: RQ1: ThinkRepair vs. LLM-based APRs for different projects on Defects4J V1.2

Projects	Method-level fault info					Perfect fault info				
	ThinkRepair	ThinkRepair*	BaseChatGPT	GPT-NeoX	Codex	ThinkRepair	ThinkRepair*	BaseChatGPT	ChatRepair	AlphaRepair
Chart	9	8	3	-	-	11	10	5	-	8
Closure	19	19	5	-	-	31	20	13	-	22
Lang	15	11	7	-	-	19	13	11	-	11
Math	27	20	16	-	-	27	21	16	-	19
Mockito	7	4	4	-	-	6	4	6	-	4
Time	3	1	1	-	-	4	2	1	-	3
# Sum	80	63	36	18	63	98	70	52	76	67

"-": indicates no results reported in the original work [61, 63].

(46.3%) than AlphaRepair with the perfect fault information provided, and auto-fix 17 (27%) and 62 (344.4%) more bugs than Codex and GPT-NeoX with only method-level fault information provided, respectively. As for QuixBugs, ThinkRepair obtains the best result with the perfect and method-level fault information. As shown in Table 3 and Table 4, BaseCodeLlama demonstrates relatively poor bug-fixing capabilities. However, under our unsupervised setting, CodeLlama (i.e., ThinkRepair*) can achieve performance on par with or surpass supervised approach (i.e., AlphaRepair).

By observing Table 5, we find that with perfect fault information, ThinkRepair outperforms AlphaRepair in all projects. In particular, the performance of ThinkRepair is significantly better than AlphaRepair in three projects (Closure, Lang, and Math), with improvements ranging from 8 to 9 bugs (40.9%~72.7%).

Fig. 5 illustrates the Venn diagram depicting the bugs fixed by ThinkRepair, BaseChatGPT, and AlphaRepair on Defects4J V1.2. It is noteworthy that ThinkRepair successfully fixes 31 unique bugs that BaseChatGPT and AlphaRepair are unable to resolve. Meanwhile, BaseChatGPT can exclusively correctly fix four bugs (i.e., Closure-61, Lang-53, Lang-57 and Mockito-12), which ThinkRepair fails to fix. The reason may be attributed to ThinkRepair's inclination to excessively modify bugs stemming from over-inferring.

A specific example is Mockito-12, where ThinkRepair incorrectly revises the non-buggy line *"if (generic != null && generic instanceof ParameterizedType)"* to *"if (generic instanceof ParameterizedType"* even though it has passed all test suites after adding the *"if (num.intValue() < 0)"* and *"else if (actual instanceof ParameterizedType)"* condition. As a consequence, this modification results in divergent semantics compared to the correct code repair.

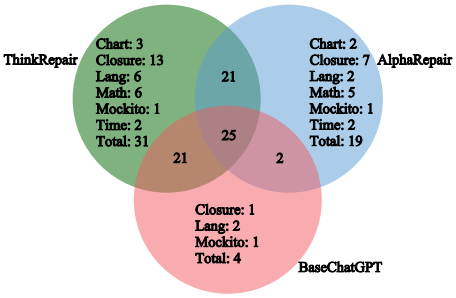

Figure 5: RQ1: Bug-fixing Venn diagram on Defects4J V1.2 of ThinkRepair, BaseChatGPT and AlphaRepair

To illustrate the effectiveness of ThinkRepair compared to BaseChatGPT, we present an instance of a bug (Mockito-29) in Defects4J V1.2 that is exclusively fixed by ThinkRepair, as depicted in Fig. 6. The bug causes an incorrect description generation when the variable *wanted* is null. Both BaseChatGPT and ThinkRepair offer distinct solutions to resolve this bug. In this case, BaseChatGPT's solution mistakenly alters the line *description.appendText("same(")* by substituting it with *description.appendText("is ")*, causing a deviation from the intended code functionality. Conversely, ThinkRepair accurately recognizes the importance of preserving the original line and concentrates on enhancing other segments of the code. It introduces additional conditional statements to handle scenarios where the *wanted* variable could be null, which guarantees proper behavior of the code when *wanted* is not null and finally repairs the bug with the conservation of intended functionality.

Test Name: NPEWithCertainMatchersTest ()
Cause: When wanted is null, it may cause a NullPointerException.
Thought Process: The bug in the describeTo function is that it is assuming that the wanted object has a to String() method, which may not always be the case. This can result in a NullPointerException. To fix this, we can check if wanted is null and handle it accordingly.

ThinkRepair

```
01 public void describeTo(Description description) {
02     description.appendText("same(");
03     appendQuoting(description);
04 + if (wanted != null) {
05         description.appendText(wanted.toString());
06 + } else {
07 +     description.appendText("null");
08 + }
09     appendQuoting(description);
10     description.appendText(")");
11 }
```

BaseChatGPT

```
01 public void describeTo(Description description) {
02 - description.appendText("same(");
03 + description.appendText("is ");
04     appendQuoting(description);
05     description.appendText(wanted.toString());
06 - appendQuoting(description);
07 - description.appendText(")");
08 }
```

Figure 6: RQ1: Unique bug fixed by ThinkRepair on Defects4J

Answer to RQ-1: *The Basic LLMs have limited ability to fix bugs and ThinkRepair can improve it with suitable adaptions. Overall, ThinkRepair performs better than LLM-based APRs which indicates the priority by properly knowledge collection, combining few-shot selection as well as interaction feedback.*

5.2 RQ-2: Compare with NMT-based APRs

RQ2-Analysis Procedure. We compare ThinkRepair with 8 NMT-based SOTA baselines: KNOD [20], TENURE [44], SelfAPR [64], RewardRepair [65], CURE [21], DeepDebug [11], CoCoNuT [40], and DLFix [29]. Benefiting from the powerful learning capability of deep neural networks, these NMT-based SOTAs have also been verified for their effectiveness on bug-fixing tasks.

Data Splitting, Fault Information and Experiment Settings. Like RQ-1, we use Defects4J and Quixbugs datasets and consider the same data splitting. Following previous works [11, 20, 21, 29, 40, 44, 64, 65], we directly adopt the results from the original papers when the perfect fault information is provided. This comparison setting is the preferred or the only one for comparing recent NMT-based APRs as it removes the influence of other factors, such as fault localization, thus showing the pure potential of different approaches [62]. We also focus on single function bug-fixing scenarios and use the same evaluation settings (e.g., *Contrastive-based Selection* for few-shot selection and five interaction feedbacks).

Table 6: RQ2: ThinkRepair vs. NMT-based APRs with the perfect fault information provided on Defects4J V1.2

Projects	ThinkRepair	ThinkRepair*	KNOD	TENURE	SelfAPR	RewardRepair	CURE	CoCoNuT	DLFix	DeepDebug
Chart	11	6	9	6	7	5	9	6	5	-
Closure	31	19	22	21	16	15	13	8	10	-
Lang	19	16	11	12	10	7	9	7	7	-
Math	27	22	18	16	18	18	16	12	12	-
Mockito	6	5	5	3	2	2	4	4	1	-
Time	4	2	2	3	3	1	1	1	2	-
# Sum	98	70	67	61	56	48	52	38	37	-

"-": indicates no results reported in the original work [11].

RQ2-Results. Table 6 shows that **ThinkRepair can outperform the studied 8 state-of-the-art NMT-based APR approaches on Defects4J V1.2**. With the perfect fault information, ThinkRepair achieves the best overall performance with a total of 98 bug fixes, representing a significant improvement over the NMT-based baselines, in the range of 46.3% (67 of KNOD)~164.9% (37 of DLFix). In addition, ThinkRepair performs the best in all six projects with improvements ranging from 1 to 9 bugs, which indicates the superiority of our proposed approach. On the open-source LLM (i.e., CodeLlama), ThinkRepair* outperforms supervised APRs. Specifically, ThinkRepair* fixes 3~33 more bugs than the SOTA APRs.

Furthermore, ThinkRepair generates far fewer patches ($\leq$ 125 in total per bug) than the NMT-based approaches that may generate up to 10,000 patches per bug [21, 40], which indicates ThinkRepair can also ensure a good efficiency.

We draw a Venn diagram to further illustrate the performance difference on bug-fixing. For a better presentation, we independently illustrate the Top-3 best baselines (i.e., KNOD, TENURE and SelfAPR) on the basis of the number of correctly fixed bugs and divide the rest methods into one group named "Other" for easy reference. As for the "Other" group, we union all distinct correctly fixed bugs by the rest methods for comparison. Fig. 7 shows the illustrated results and we can also obtain two observations: (1) Individual approaches have varying capabilities of fixing bugs and each of them can fix some specific bugs that other approaches cannot address. Therefore, to some extent, these methods have a complementary performance. (2) Overall, ThinkRepair has a more powerful ability than baselines since it can auto-fix the most number of unique bugs (i.e., 32) that other baselines can hardly fix.

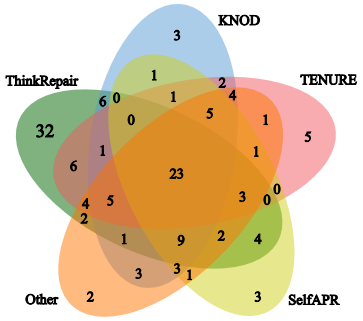

Figure 7: RQ2: Bug-fixing Venn diagram on Defects4J V1.2 of ThinkRepair and all studied NMT-based APRs

Case Study. To further understand why ThinkRepair has an outstanding performance in fixing unique bugs, we further analyze one example (i.e., Math-69 in Defects4J V1.2) as a case study in Fig. 8. The function calculates a matrix of p-values of a 2-sided, 2-sample t-test. The bug is caused by a precision error when the function call is extremely close to 1. This is a hard-to-fix bug since the change is quite subtle and it does not fit any of the common templates used in traditional APRs. To generate the correct patch, ThinkRepair needs to understand the goal of the function (i.e., p-value calculation) and correctly use statistical formulas. As shown in Fig. 8, ThinkRepair undergoes two attempts (i.e., marked as *"Round 1"* and *"Round 2"*), and the *"Thought Process"* reveals how ThinkRepair thinks to solve this bug. Specifically, the first attempt (i.e., Lines 13-14) modifies the buggy lines (i.e., Lines 11-12) to *"tDistribution.cumulativeProbability(-t) * 2"* while overlooking the necessity of using the absolute value of the *t-value*. In the second attempt (i.e., Lines 15-16), ThinkRepair comprehends the semantics gleaned from the test failure message and thereby successfully fixes the bug. This example further exemplifies the capability of ThinkRepair to leverage previously overlooked semantic information within failing tests to directly guide the repair process.

Test Name:
testPValueNearZero ()
Interaction Number: 2
Cause:
Precision error when the function call is extremely close to 1.

Thought Process:
1. The calculation of the p-value itself is incorrect, as it uses 1 - tDistribution.cumulativeProbability(t)` instead of `tDistribution.cumulativeProbability(-t) * 2`, which properly accounts for both tails of the t-distribution.
2. The p-value calculation should use the absolute value of the t-value to compute the cumulative probability of the t-distribution.

```
01 public RealMatrix getCorrelationPValues() throws MathException {
02     TDistribution tDistribution = new TDistributionImpl(nObs - 2);
03     int nVars = correlationMatrix.getColumnDimension();
04     double[][] out = new double[nVars][nVars];
05     for (int i = 0; i < nVars; i++) {
06         for (int j = 0; j < nVars; j++) {
07             if (i == j) {
08                 out[i][j] = 0d;
09             } else {
10                 double r = correlationMatrix.getEntry(i, j);
11 -               double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
12 -               out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(t));
13 +               double t = r * Math.sqrt((nObs - 2)/(1 - r * r));
14 +               out[i][j] = tDistribution.cumulativeProbability(-t) * 2;
15 +               double t = r * Math.sqrt((nObs - 2)/(1 - r * r));
16 +               out[i][j] = 2 * tDistribution.cumulativeProbability(-Math.abs(t));
17             }
18         }
19     }
20     return new BlockRealMatrix(out);
21 }
```

Buggy Line — Round 1 — Round 2

Figure 8: RQ2: Unique bug fixed in Defects4J V1.2

Table 7 shows that compared with the NMT-based APRs, ThinkRepair can achieve the best performance on Defects4J V2.0, QuixBugs-Java, and QuixBugs-Python, similar to the findings obtained Defects4J V1.2. ThinkRepair can auto-fix 107 bugs on Defects4J V2.0, with 60~65 (127.7%~154.8%) more bugs than the current state-of-the-art NMT-based APRs. Furthermore, ThinkRepair* fixes 72 bugs and improves baselines by 53.2%~71.4%. Additionally, ThinkRepair has improved NMT-based baselines by fixing 14~26 more bugs on QuixBugs-Java and 19~21 bugs on QuixBugs-Python.

Answer to RQ-2: *All APR approaches have complementary advantages in fixing different bugs. Overall, ThinkRepair outperforms NMT-based approaches in terms of the number of auto-fixed bugs.*

Table 7: RQ2: ThinkRepair vs. NMT-based APRs with perfect fault information provided on the other three datasets

Models	Defects4J	QuixBugs	
	V2.0 (228 bugs)	Java (40 bugs)	Python (40 bugs)
KNOD	47	25	-
TENURE	43	-	
SelfAPR	42	-	-
RewardRepair	44	20	-
CURE	-	21	-
DeepDebug	-	-	21
CoCoNuT	-	13	19
DLFix	-	-	-
ThinkRepair*	72	38	38
ThinkRepair	107	39	40

"-": indicates no results reported in the original work.

5.3 RQ-3: Configurations of ThinkRepair

RQ3-Analysis Procedure. ThinkRepair has two important components: ① **CoT few-shot learning (knowledge collection + few-shot selection)** prompts LLM to construct a chain of thought pool and selects diverse and effective examples for LLM to better understand the downstream task with few-shot learning, and ② **interaction feedback** provides feedback to LLM with the test failure information interactively during the process of function verification. Therefore, in this RQ, we aim to conduct a comprehensive experiment to evaluate the impact of different components on the ThinkRepair's performance. Furthermore, we study the impact of the number of interactions with LLM and the impact of the four few-shot selection strategies on the performance of ThinkRepair.

Data Splitting and Fault Information. In this RQ, we utilized ChatGPT as the backend LLM. Considering the cost of invoking the ChatGPT API multiple times for this ablation study, we concentrate our experiments on Defects4J V1.2, which has the most number of single function bugs (i.e., 255) among our studied datasets. Also, we only provide ThinkRepair with the method-level fault information but no statement-level fault information.

Experiment Settings. As for the interaction number, the maximum interaction number (cf. Section 3.2) dictates the amount of history/feedback within each individual repair query. Therefore, when interacting once, it is equivalent to directly using the initial prompt without any feedback for ThinkRepair. We treat the original ChatGPT as the basic model for comparison and investigate its initial performance in the zero-shot learning setting and few-shot learning setting. Zero-Shot means that we directly prompt ChatGPT to fix a bug without providing any other information. Few-Shot means that we provide ChatGPT with two examples of fixes but do not include CoT information (i.e., directly using existing bug-fixing data as examples). Moreover, *"knowledge collection"* is a preparation step for *"few-shot selection"* when fixing bugs. Therefore, the two parts are used together. We build a variant of ThinkRepair without interaction feedback as ThinkRepair-v1 and another variant without CoT few-shot learning as ThinkRepair-v2. All of our ablation experiments utilize the default settings outlined in Section 4.3.

RQ3-Results. Impacts of Components. According to the results in Table 8, we can observe that: (1) In few-shot, ChatGPT exhibits better bug-fixing performance compared to zero-shot, but the performance improvement is limited (i.e., 34→36). (2) Two components have their own advantages in a single-function bug-fixing scenario, achieving a varying performance and significantly improving the performance of ChatGPT. Both of them can contribute to the performance of ThinkRepair. (3) The combination between *"Knowledge Collection"* and *"Few-Shot Selection"* is effective for ThinkRepair, which improves ChatGPT a lot (i.e., 36→57) and emphasizes the importance of prompting ChatGPT by selecting high-quality of examples. (4) The *"Interaction Feedback"* seems to contribute the performance of ThinkRepair, which brings ChatGPT with a large improvement (i.e., 36→62). It indicates that the information obtained from test failures can help ChatGPT understand the reasons for failures and provide guidance for generating plausible fixes. (5) A combination of these two components can improve ChatGPT. ThinkRepair is capable of generating 44 more correct fixes than ChatGPT with a few-shot setting (i.e., 36→80).

Table 8: RQ3: The performance difference among different components

Models	CoT Few-Shot Learning	Interaction Feedback	# Correct Fixes
Zero-Shot	✗	✗	34
Few-Shot	✗	✗	36
ThinkRepair-v1	✓	✗	57
ThinkRepair-v2	✗	✓	62
ThinkRepair	✓	✓	80

Impacts of Interaction Number. According to the results in Fig. 9, we observe that: (1) Different interaction numbers have varying impacts on ThinkRepair's performance and the performance of ThinkRepair increases as the number of interactions increases. (2) Sampling directly from the ChatGPT (i.e., interact once) may not ensure a good performance. For example, when directly interacting with ChatGPT without information feedback, ThinkRepair achieves the lowest number of correct fixes (i.e., 57). (3) Feedback information promotes ChatGPT to reason, but more interaction times may not guarantee additional performance improvement. Notice that ThinkRepair has a big improvement when interacting twice with ChatGPT (i.e., 57→75). However, when continuously increasing the number of interactions, the rate of performance improvement decreases (i.e., 75→80) and meanwhile, the interaction cost with ChatGPT is increasing. Considering both the performance improvement and the communication cost caused by ChatGPT, we adopt five times of interactions as the default setting.

Table 9: RQ3: The performance difference among four different few-shot selection strategies

Datasets	CSelect	SSelect	ISelect	RSelect
Defects4J V1.2 (255 bugs)	**80**	73	70	68
Defects4J V2.0 (228 bugs)	**90**	84	78	77
QuixBugs-Java (40 bugs)	**39**	38	38	38
QuixBugs-Python (40 bugs)	**40**	**40**	39	39
# Sum	**249**	235	225	222

"CSelect": Contrastive-based Selection, "SSelect": Semantic-based Selection, "ISelect": IR-based Selection, "RSelect": Randomly Selection.

Impacts of Few-Shot Selection Strategy. According to the results in Table 9, we observe that: (1) Both *Contrastive-based Selection* and *Semantic-based Selection* help to generate more correct fixes than *IR-based Selection* and *Randomly Selection*, which indicates the importance of the semantic similarity among samples. Particularly, *Contrastive-based Selection* and *Semantic-based Selection* can help ThinkRepair fix 27 and 13 more bugs than does *Randomly Selection*

help, respectively. (2) Overall, *Contrastive-based Selection* performs the best and shows promising results in selecting high-quality examples, which may benefit from the well-trained semantic encoder (i.e., UniXcoder) fine-tuned with contrastive learning.

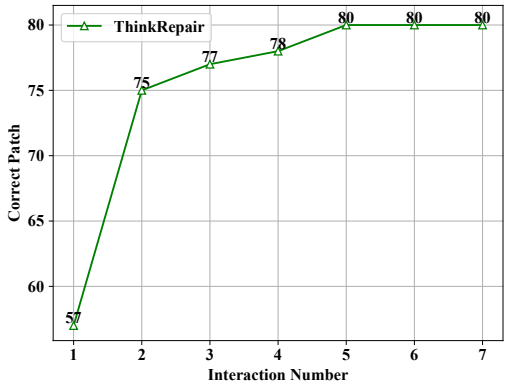

Figure 9: RQ3: The varying performance of ThinkRepair with different interaction number

We also discuss the cost of different selection strategies. It has some difference in time cost. On average, both *Semantic-based Selection* and *Contrastive-based Selection* strategies take about 0.01 seconds to embed a function, while *IR-based Selection* and *Randomly Selection* strategies take 0.9 milliseconds. However, considering that a repair process is mainly consumed in the output of the LLM (which usually takes several seconds), the time cost of selection is negligible. Therefore, to strike a balance between efficiency and effectiveness, we adopt *Contrastive-based Selection* as default.

Answer to RQ-3: *(1) The two components (i.e., CoT few-shot learning and interaction feedback) contribute substantially to ThinkRepair, and combining them achieves the best performance. (2) Increasing the interaction number (e.g., from 1 to 5) can significantly improve the performance of ThinkRepair, but more interactions may not gain larger performance improvement. (3) Semantically similarity-based example selection strategies can pick out high-quality examples for ThinkRepair than random selection and Contrastive-based Selection is the best choice.*

6 Discussion

This section discusses open questions regarding the data leakage and threats to the validity of ThinkRepair.

6.1 Evaluation of Data Leakage

6.1.1 Similarity between generated patches and the ground-truth. We follow the work [62] and initially calculate the number of correct patches, which lexically matches the ground-truth in Defects4J V1.2. We find that out of 98 correct patches, 24 of them lexically match the ground-truth (24.5%), while the other patches semantically match the ground-truth. We exemplify with Fig 8, the ground-truth fix is *"double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r))); out[i][j] = 2 * tDistribution.cumulativeProbability(-t);"*. Our ThinkRepair generates a fix that is different from the ground-truth but semantically equivalent, namely *"double t = r * Math.sqrt((nObs - 2)/(1 - r * r)); out[i][j] = 2 * tDistribution.cumulativeProbability(-Math.abs(t));"*.

6.1.2 Study on Real-World Projects. We follow Defects4J and collect bug-fixing commits from high-quality open-source projects included in Defects4J. The resulting dataset is referred to as RWB (Real-World Bugs). We evaluate data leakage on both ChatGPT and DeepSeek, denoting them as ThinkRepair and ThinkRepair*, respectively. Notice that the pre-training data for ChatGPT was collected before September 2021 [47], while the pre-training data for DeepSeek was collected from GitHub before February 2023 [3]. Consequently, we collected two datasets to assess data leakage. The first dataset (RWB V1.0) comprises bug-fixing commits after October 2021, while the second dataset (RWB V2.0) includes bug-fixing commits after March 2023, resulting 113 and 61 bugs, respectively.

Since ThinkRepair focuses on single-function bugs, we perform two steps on the original datasets to obtain valid functions:

Step-1: Each commit is considered a mini-version of a project. We use the commit IDs to request commit histories of the projects, and for each commit, we extract the code changes between before and after fixing a bug. Finally, we use the code change information to obtain the buggy and fixed version of a function.

Step-2: To clean and normalize the dataset, we keep only single-function bugs. In this step, we finally obtain the RWB V1.0 dataset, which comprises 44 single-function bugs (4 of Cli, 5 of Codec, 1 of Collections, 8 of Compress, 2 of Csv, 6 of Lang, and 18 of Jsoup), and the RWB V2.0 dataset, which comprises 29 single-function bugs (4 of Cli, 4 of Codec, 4 of Compress, 6 of Lang, and 11 of Jsoup).

Finally, we conducted a comparative analysis on ThinkRepair and AlphaRepair with perfect fault information, and AlphaRepair was identified as the baseline with the best performance (cf. Section 5.1). Table 10 presents the results of ThinkRepair on RWB. According to the results, we can conclude that ThinkRepair possesses the ability to fix bugs in real-world projects, not just limited to the bugs present in the Defects4J dataset and QuixBugs dataset. Furthermore, both ThinkRepair and ThinkRepair* outperform AlphaRepair, underscoring the practicality and viability of our approach.

In summary, we believe that data leakage will not significantly affect the performance of our ThinkRepair.

Table 10: ThinkRepair vs. AlphaRepair on RWB

Projects	RWB V1.0 (44 bugs)		RWB V2.0 (29 bugs)	
	ThinkRepair	AlphaRepair	ThinkRepair*	AlphaRepair
Cli	4	3	4	3
Codec	3	1	1	1
Collections	1	0	-	-
Compress	1	1	-	-
Csv	1	0	-	-
Lang	3	2	3	1
Jsoup	6	2	2	1
# Sum	19	9	10	6

6.2 Threats to Validity

Internal Validity. The first internal threat arises from the manual validation employed to determine the correctness of the plausible patches. To mitigate this concern, we conduct a thorough examination and comprehensive discussion of each patch, following prior work [21, 61, 62, 64, 65, 67]. The second one comes from potential data leakage since referenced developer patches may be part of the training data of LLM. As discussed in section 6.1.1, 24.5% of correct

patches aligned with the reference developer fix. Moreover, even after excluding all the correct patches (24) that aligned with the reference developer patch, ThinkRepair is still capable of generating the correct patch for 29 unique bugs, none of which could be fixed by any previous methods. Additionally, compared to BaseChatGPT, ThinkRepair achieves 46 more correct fixes. This demonstrates that the improved results achieved by ThinkRepair are not merely a result of memorizing the training data. We also collected bugs from real-world projects to evaluate data leakage. ThinkRepair can fix 19 out of 44 bugs on RWB V1.0, and 10 out of 29 bugs on RWB V2.0.

External Validity. The effectiveness observed in ThinkRepair's performance may not be applicable across different repair datasets. We conduct evaluations not only on the widely-used Defects4J dataset but also on QuixBugs dataset. This broader evaluation scope aims to showcase the generalizability of our approach.

7 Related Work

7.1 Large Language Model

Large Language Models (LLMs) [6] have been widely adopted since the advances in Natural Language Processing (NLP) which enable LLM to be well-trained with both billions of parameters and billions of training samples, and consequently, they bring a large performance improvement. LLMs can be easily used for a downstream task by being fine-tuned [51] or being prompted [37] since they are trained to be general and they can capture different knowledge from various domain data. Fine-tuning is used to update model parameters for a particular downstream task by iterating the model on a specific dataset. Meanwhile, prompting can also be directly used by providing natural language descriptions or a few examples of the downstream task. Compared to prompting, fine-tuning is expensive since it requires additional model training and has limited usage scenarios, especially in cases where sufficient training datasets are unavailable.

ChatGPT [47] is a successor of InstructGPT [48] and is fine-tuned with the Reinforcement Learning with Human Feedback (RLHF) approach [9, 48, 68]. RLHF first fine-tunes the model with the input (i.e., a small dataset of prompts) and the desired output (i.e., usually human-written). Following that, a reward model will be trained on a larger set of prompts by sampling a few outputs that are generated by the fine-tuned model and these outputs are reordered by humans. Finally, reinforcement learning [54] is adopted to calculate the reward of each output that is generated based on the reward model and eventually updates the LLM parameters accordingly. Benefiting from fine-tuning as well as human preference alignments, LLM has a better understanding of input prompts and instructions to perform better on various downstream tasks [4, 48].

7.2 Automated Program Repair

Automated Program Repair (APR) can assist developers in generating patches for specific bugs based on their potential fault locations. Traditional APR techniques can be classified into heuristic-based [25, 26, 59], constraint-based [10, 24, 38, 43] and template-based [15, 19, 35, 36, 42] approaches. Template-based APR tools have gained recognition as state-of-the-art due to their ability to fix a large number of bugs. These tools utilize human-defined or automatically-mined templates to identify potential buggy code patterns and apply corresponding fixes. However, template-based tools are limited to the patterns within their predefined set and lack the ability to generalize to other types of bugs or fixes. To address this limitation, researchers have proposed learning-based APR techniques by leveraging recent advancements in Deep Learning. Techniques based on Neural Machine Translation (NMT) have been extensively studied in recent years [11, 20, 21, 29–31, 40, 44, 64, 65, 67] and they treated APR as an NMT problem that is translating buggy code into correct code. However, these methods heavily rely on historical bug-fixing data and noise may impact their performance.

In order to overcome the limitations of NMT-based tools, researchers have explored the potential of utilizing LLMs directly to generate correct patches. By pre-training on large amounts of open-source code snippets, LLMs have the ability to generate correct code directly based on the surrounding context, eliminating the need for translation from the buggy code. AlphaRepair [62] is the first tool for cloze-style APR and its performance indicates that LLM-based APR outperforms the widely studied NMT-based APR techniques in real-world systems. Following that, researchers [23, 50] directly adopt Codex to generate a fixed function based on a buggy one. Recently, Xia et al. [61] conducted an extensive study of LLM-based APR techniques using various LLMs [5, 8, 14, 57] and their results further demonstrated the superiority of LLM-based APR. ChatRepair [63] simply uses the chat capability of ChatGPT and iteratively enters test information to obtain the final patch. However, the capabilities of LLMs are influenced by high-quality prompts. In this paper, we propose a self-directed framework, ThinkRepair. The objective of ThinkRepair is to enable the LLM to engage in self-reflection to construct a high-quality knowledge pool, and select few-shot examples from the knowledge pool to better guide LLM.

8 Conclusion

This paper proposes a novel approach ThinkRepair, which is a single function APR tool. ThinkRepair has two main phases: the collection phase and the fixing phase. The former phase adopts knowledge collection to generate a series of thought processes that provide high-quality examples for subsequent phases. The latter phase targets fixing a bug by first selecting examples for few-shot learning and second automatically interacting with LLM, optionally appending with feedback of testing information. Through this repair paradigm, ThinkRepair has strong analytical and reasoning abilities and is capable enough to repair complex bugs. Therefore, ThinkRepair achieves state-of-the-art performance on both Defects4J V1.2 and Defects4J V2.0, surpassing the baselines by 17∼62 and 12∼65 more bugs, respectively.

9 Data Availability

The replication of this paper is publicly available [2].

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant No.62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), and the State Street Zhejiang University Technology Center.

References

[1] 2024. Hugging Face. https://huggingface.co

[2] 2024. Replication. https://github.com/vinci-grape/ThinkRepair

[3] DeepSeek AI. 2023. DeepSeek Coder: Let the Code Write Itself. https://github.com/deepseek-ai/DeepSeek-Coder.

[4] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[5] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745* (2022).

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] chatgptendpoint. 2023. Introducing ChatGPT and Whisper APIs. https://openai.com/blog/introducing-chatgpt-and-whisper-apis.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

[10] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*. 30–39.

[11] Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2021. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. *arXiv preprint arXiv:2105.09352* (2021).

[12] Sidong Feng and Chunyang Chen. 2023. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. *arXiv preprint arXiv:2306.01987* (2023).

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[15] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[16] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. 2023. Chatgpt outperforms crowd-workers for text-annotation tasks. *arXiv preprint arXiv:2303.15056* (2023).

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[18] Md Asraful Haque and Shuai Li. 2023. The Potential Use of ChatGPT for Debugging and Bug Fixing. *EAI Endorsed Transactions on AI and Robotics* 2, 1 (2023), e4–e4.

[19] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 888–891.

[20] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. Knod: Domain knowledge distilled tree decoder for automated program repair. *arXiv preprint arXiv:2302.01857* (2023).

[21] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[22] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[23] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*.

[24] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[25] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[26] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[27] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[28] Xiaonan Li and Xipeng Qiu. 2023. MoT: Pre-thinking and Recalling Enable ChatGPT to Self-Improve with Memory-of-Thoughts. *arXiv preprint arXiv:2305.05181* (2023).

[29] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[30] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Improving automated program repair using two-layer tree-based neural networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 316–317.

[31] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*. 511–523.

[32] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[33] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. *arXiv preprint arXiv:2305.08360* (2023).

[34] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? *arXiv preprint arXiv:2101.06804* (2021).

[35] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[36] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[37] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[38] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[39] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786* (2021).

[40] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[41] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[42] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 441–444.

[43] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[44] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1456–1468.

[45] Chao Ni, Liyu Shen, Wei Wang, Xiang Chen, Xin Yin, and Lexiao Zhang. 2023. FVA: Assessing Function-Level Vulnerability by Integrating Flow-Sensitive Structure and Code Statement Semantic. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 339–350.

[46] Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing Look-Alike Innocent and Vulnerable Code by Subtle Semantic Representation Learning and Explanation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1611–1622.

[47] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. (2022). https://openai.com/blog/chatgpt/.

[48] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022.

Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[50] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[52] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[53] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[54] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[55] Jessica Shieh. 2023. Best practices for prompt engineering with OpenAI API. *OpenAI, February https://help.openai. com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api* (2023).

[56] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[57] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[58] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[59] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*. 1–11.

[60] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[61] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[62] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[63] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[64] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[65] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*. 1506–1518.

[66] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).

[67] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

[68] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2019. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593* (2019).

Received 2024-04-12; accepted 2024-07-03