CREF: An LLM-Based Conversational Software Repair Framework for Programming Tutors

Boyang Yang*†
School of Information Science and Engineering, Yanshan University
China
yangboyang@jisuanke.com

Haoye Tian*
CIS, University of Melbourne
Australia
haoye.tian@unimelb.edu.au

Weiguo Pian
SnT, University of Luxembourg
Luxembourg
weiguo.pian@uni.lu

Haoran Yu
Jisuan Institute of Technology, Beijing JudaoYouda Network Tech. Co. Ltd.
China
yuhaoran@jisuanke.com

Haitao Wang
Jisuan Institute of Technology, Beijing JudaoYouda Network Tech. Co. Ltd.
China
wanghaitao@jisuanke.com

Jacques Klein
SnT, University of Luxembourg
Luxembourg
jacques.klein@uni.lu

Tegawendé F. Bissyandé
SnT, University of Luxembourg
Luxembourg
tegawende.bissyande@uni.lu

Shunfu Jin‡
School of Information Science and Engineering, Yanshan University
China
jsf@ysu.edu.cn

Abstract

With the proven effectiveness of **L**arge **L**anguage **M**odels (LLMs) in code-related tasks, researchers have explored their potential for program repair. However, existing repair benchmarks might have influenced LLM training data, potentially causing data leakage. To evaluate LLMs' realistic repair capabilities, (i) we introduce an extensive, non-crawled benchmark TutorCode, comprising 1,239 C++ defect codes and associated information such as tutor guidance, solution description, failing test cases, and the corrected code. Our work assesses LLM's repair performance on TutorCode, measuring repair correctness (TOP-5 and AVG-5) and patch precision (RPSR). (ii) We then provide a comprehensive investigation into which types of extra information can help LLMs improve their repair performance. Among these types, tutor guidance was the most effective information. To fully harness LLMs' conversational capabilities and the benefits of augmented information, (iii) we introduce a novel conversational semi-automatic repair framework Cref assisting human programming tutors. It demonstrates a remarkable AVG-5 improvement of 17.2%-24.6% compared to the baseline, achieving an impressive AVG-5 of 76.6% when utilizing GPT-4. These results highlight the potential for enhancing LLMs' repair capabilities through tutor interactions and historical conversations. The successful application of Cref in a real-world educational setting demonstrates its effectiveness in reducing tutors' workload and improving students' learning experience, showing promise for code review and other software engineering tasks.

CCS Concepts

• **Software and its engineering** → *Software defect analysis*; Software testing and debugging.

Keywords

Program Repair, Large Language Model, Open Source

ACM Reference Format:
Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F. Bissyandé, and Shunfu Jin. 2024. CREF: An LLM-Based Conversational Software Repair Framework for Programming Tutors. 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/10.1145/3650212.3680328

1 Introduction

In code-related scenarios, such as programming education, providing efficient and precise automated feedback, especially auto-generated program repairs, is essential for effectively guiding a large number of students and reducing the workload of human tutors [1, 28, 79]. Data from a company's online programming education platform show that 54.5% of students need debugging assistance while completing programming tasks. Each tutor spends an average of 26.7 minutes resolving a single issue, which leads to high labor costs for the company. This extensive resolution time also adversely affects students' learning experience. Program repair techniques, vital in both programming education and software

*Co-first authors who contributed equally to this work.
†Also affiliated with Jisuan Institute of Technology, Beijing JudaoYouda Network Tech. Co. Ltd.
‡Corresponding author.

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

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680328

development, significantly reduce the manual labor involved in debugging [16, 21, 31, 32, 61, 63]. Nowadays, machine learning-based approaches have gained prominence in the field of program repair research. These techniques predominantly rely on **N**eural **M**achine **T**ranslation (NMT) to correct code [25, 38, 43, 48, 50, 53, 76].

Code-targeted pre-trained LLMs [14, 36, 44], have demonstrated promising results in the field of learning-based automated program repair. These LLMs have been trained to repair codes through various paradigms, including zero-shot approaches, which utilize the original incorrect code either with [13, 30, 52] or without accompanying instructions [15]; and few-shot approaches that incorporate a small set of patch examples [49]. With the using of training datasets including large-scale code-related data crawled from the internet [9, 46, 64], there has been a significant enhancement in LLMs' program repair capabilities [36]. However, the inherent stochastic and opaque nature of LLMs makes the generated patches unreliable [5]. Yet, many of the current LLM-based repair techniques typically treat the repair task as an automated procedure, often overlooking the collaborative and interactive aspects inherent in programming [80]. To harness the conversational capabilities of LLMs and address the unpredictability of generated code, there is a growing need for engaging in interactive program repair [18, 73]. For instance, Gao *et al.* [17] introduced interactive program repair where developers are involved in reviewing and selecting automatically generated patches. Building on the success of recent LLM-as-a-service deployments, Sobania *et al.* [58] enhanced the repair performance of ChatGPT by incorporating human-authored hints. They have validated their approach on QuixBugs, based on a corpus of 40 bugs. From a different perspective, Xia *et al.* [73] proposed ChatRepair, where when an LLM-generated patch fails to pass a test case, a new prompt is constructed by combining the invalid patch with the failing test case information, towards generating the next prompt. This process is executed in at most three turns of dialogs, and has been validated on 337 bugs from QuixBugs and Defects4J. Considering the importance of utilizing ChatGPT judiciously and with expertise, Azaria *et al.* [5] proposed an LLM-based repair strategy designed for experts who are well-versed in the respective domains. Unfortunately, they did not perform any performance evaluation.

Overall, however, interactive program repair using LLMs still faces several limitations:

① Data Leakage Concerns: The effectiveness of LLMs in interactive program repair often relies on the scale of code-related data collected from the internet, which can be associated with **data leakage** issues. Indeed, the benchmarks used for evaluation may have been part of the training data for these LLMs [2, 37, 62, 71]. For example, Tian *et al.* [62] discovered that ChatGPT's correctness on *2022 LeetCode questions* was significantly lower than in previous years, where the questions were available online when ChatGPT data was being collected (i.e., before September 2021). This finding suggests that, to ensure more accurate assessment of LLMs, careful benchmark selection is essential to reduce data leakage risks.

② High Computational Overhead: Current interactive repair methods can be computationally intensive. For instance, Xia *et al.* [73]'s ChatRepair requires an average of 10 independent dialog sessions to repair an incorrect code [15]. This high computational cost can limit the practicality of these approaches.

③ Limited Evaluation on Large-Scale Datasets: To the best of our knowledge, existing interactive repair methods have not been evaluated on large-scale datasets. This includes recent research by Xia *et al.* [73], Azaria *et al.* [5], and Sobania *et al.* [58]. Assessing the performance of these state-of-the-art methods on larger datasets would provide a more comprehensive understanding of their capabilities and limitations.

④ Need for Augmented Information: Existing LLM-based interactive program repair methods primarily rely on problem descriptions and failing test cases to generate patches [72, 73]. However, this reliance might not give LLMs enough information to understand the programs and identify the necessary fixes. Prior research [5, 8, 70] indicated that incorporating human interaction into the repair process can significantly improve the quality and accuracy of generated patches. Furthermore, leveraging diverse augmented information, such as solution descriptions, can deepen LLMs' understanding of programs [8, 21, 42, 74]. Consequently, it is necessary to investigate approaches to integrating augmented information with human expertise, specifically **interaction with tutors or developers**, into the repair process, further enhancing LLMs' repair capabilities.

This paper. In this study, we address the challenge of data leakage and evaluate the practical benefits of different forms of augmented information within conversation-based program repair methods. To do this, we introduce an extensive benchmark dataset, which we refer to as "*uncrawled*" to emphasize that it has not been incorporated into the training data of any pre-trained LLM. This benchmark originates from a company specializing in training novice developers to become experienced professionals through data structure and algorithm courses.

Our dataset, TutorCode, comprises 1,239 incorrect C++ code samples contributed by 427 students, covering 35 distinct programming challenges distributed across 12 difficulty levels. It also includes tutor guidance provided by human tutors as well as corresponding corrected code. The 35 challenges were designed to cultivate various cognitive skills, including abstract reasoning, procedural thinking, and conceptual understanding. Sourced from the real-world experience of repair processes, we expect this benchmark to serve as a valuable and reliable asset[1] for evaluating a range of coding-related tasks, such as program synthesis, fault localization, and program repair. Using TutorCode, we explore the realistic repair capabilities of LLMs and investigate LLMs as conversational repair tools for tutors in programming education scenarios. First, we investigate the practical program repair capabilities of 12 prominent LLMs. Our analysis reveals that GPT-4 and GPT-3.5 consistently outperform other LLMs in program repair tasks. Second, we explore the influence of different types of augmented information in prompts on LLM-based program repair performance. Our findings demonstrate that providing LLMs with tutor guidance significantly enhances their performance, and this can be further improved by incorporating solution descriptions

[1]With the risk of it being leaked into training data of LLMs, TutorCode has been made available to open science through API: **http://tutorcode.org**.

and failing test cases. Finally, to leverage the conversational abilities of LLMs and capitalize on the aforementioned three types of augmented information, we introduce a novel semi-automatic LLM-based **C**onversational program **RE**pair **F**ramework (Cref) for tutors, prompting LLMs with tutor guidance, solution description, and failing test cases, interactively. Leveraging LLM conversational capabilities, Cref achieves an AVG-5 score of 76.6% when utilizing GPT-4, showcasing remarkable program repair capabilities with only 7.5% of input tokens (i.e., expense cost) per request compared to ChatRepair. In practical application within a company, Cref acts as a semi-automatic repair tool that aids tutors by utilizing LLMs to repair incorrect codes based on preliminary feedback, thus reducing debugging times by 71.2% and decreasing costs by 69.9%. Cref enhances the tutoring effectiveness and improves the learning experience by providing more timely and accurate debugging assistance.

Contributions. The main contributions of our work are as follows:

- We introduce a large-scale uncrawled benchmark TutorCode for realistic evaluation of LLM-based repair approaches. TutorCode includes 1,239 defective C++ programs, to which human tutor guidance, programming problem description, solution description, test cases, and ground truth corrected codes are attached.
- We assess the realistic repair capabilities of state of the art 8 open-source [14, 44, 55, 56, 65, 81] and 4 closed-source [4, 46, 47, 51] LLMs on the TutorCode.
- We evaluate the enhancements of different augmented information on the repair capabilities of LLMs, measuring repair correctness (AVG-5) and patch precision (RPSR).
- We introduce a conversational semi-automatic repair framework, termed Cref, to leverage the conversational capabilities of any LLMs and different augmented information for repair tasks. The effectiveness of Cref is validated using TutorCode. The artifact of this study is publicly available at **https://github.com/buaabarty/CREF**.
- We deploy Cref as an **assisting tool for programming tutors** in a company, achieving a 71.2% response time reduction and a 69.9% cost decrease for students' debugging requests.

The structure of this paper is organized as follows: Section 2 offers a review of background and related works. In Section 3, we design the methodology employed in this study. Section 4 provides a detailed account of the experimental findings, while Section 5 investigates the outlier results and introduces the real-world application. Potential threats to the validity are qualified in Section 6. Section 7 summarizes this paper's key points and findings.

2 Background & Related Work

2.1 Large Pre-Trained Language Model

Encoder-Decoder Models

Encoder-decoder architectures, such as BART [34] and T5 [54], employ an encoder to transform an input sequence into a continuous representation, capturing essential information. A decoder then generates the output sequence based on this representation. **CodeT5p** [67], an advancement of CodeT5 [68], is developed by Salesforce and is pre-trained on a wide array of tasks, incorporating both unimodal code data and bimodal code-text data.

Decoder-Only Models

Decoder-only architectures, such as GPT-3 [9] and CodeGen [44], utilize the transformer's decoder in an auto-regressive manner to generate tokens based on preceding tokens. Both Codex [11] and InstructGPT (**GPT-3.5**) [47] are derivatives of the GPT-3 model. Additionally, ChatGPT [45] served as a dialog-optimized adaptation of GPT, developed through Reinforcement Learning from Human Feedback (RLHF) and drawing from the frameworks of both Codex and InstructGPT. The successor to GPT-3.5, known as **GPT-4** [46], boasts enhanced performance and an extended context window. GPT-3.5 and GPT-4 are proficient in code generation and debugging, guided by natural language conversational contexts [62]. **Claude-instant** [4], based on Anthropic's Constitutional AI [7], is trained on an extensive corpus of text and code and thus capable of code-related tasks [19]. **Bard** [51], Google's experimental conversational AI, based on LaMDA [60], is pre-trained on a 1.56 TB dataset, comprising public dialog data and web documents, which includes 12.5% of code-related documents. **StarChat** [65] is a derivative of BigCode's StarCoder, pre-trained as a Transformer decoder-only model named StarCoderBase, and is subsequently fine-tuned for Python coding tasks. Salesforce's **CodeGen** [44] employs a multi-step paradigm for program synthesis that outperforms single-turn methods. **Incoder** [14]uses a causal-masked objective for training and specializes in code insertion and generation. **Replit-code** [55], a 2.7B Causal Language Model, focuses on code completion and employs Flash Attention and AliBi positional embeddings for efficient training and inference. **Vicuna** [81] is fine-tuned from Llama [64] utilizing shared conversational data. **CodeLlama** [56] represents a leading family of LLMs tailored for coding tasks. Given the numerous LLMs that can repair code, we list 12 prominent LLMs in Section 3.5 for subsequent experiments.

2.2 Interactive Program Repair

Given the LLM's inherent unpredictability in coding tasks, engaging human interactions in a dialogic collaboration becomes imperative for dependable results [5, 18]. Gao *et al.* [17] introduced an interactive repair methodology, enabling developers to review and select patches generated by automated techniques, translating the patch into interrogative frameworks, resulting in a more straightforward selection process for developers without delving into the semantics of the patch. Sobania *et al.* [58] explored the utility of dialog-based interactions in ChatGPT and demonstrated the efficacy of dialogic hints in enhancing the program repair capabilities. Xia *et al.* [73] proposed ChatRepair, a paradigm that prompts ChatGPT with generated incorrect patches and failing test cases, where each repair case consisted of approximately 10 distinct conversation sessions, with three turns of conversation, demonstrating the effectiveness of providing automated feedback in a conversational manner for program repair. In our work, we investigate the potential of the interactive capabilities of LLMs with extra information to enhance LLM-based semi-automated program repair.

2.3 Intelligent Tutoring System for Programming

Intelligent **T**utoring **S**ystem (ITS) for programming education shows promising results in helping novice students learn programming [1, 13, 22, 23, 78]. Clara [22] is an automated program repair tool for

introductory C/Python programming assignments, using the existing correct student solutions to repair the incorrect attempts through clustering. Refactoring [23] generates Python introductory program repairs in real-time, using re-factoring rules to generate a correct solution with the same control flow as the incorrect program. ITSP [77] generates partial repairs that serve as hints to guide students toward the reference solution. Verifix [1] is an automated program repair technique for introductory C programming assignments that generates repairs by aligning a student's incorrect program with a reference solution using control flow. MMAPR [78] was proposed to use Codex [11] with few-shot prompting to build an APR system for introductory Python programming assignments. Yasunaga *et al.* proposed DrReapir [75], a graph-based program repair technique that learns to repair bugs by leveraging compiler diagnostic feedback, employs a self-supervised learning paradigm that generates extra training data by corrupting unlabeled codes and obtains diagnostic feedback. Fan *et al.*'s study revealed that given bug location information provided by a statistical fault localization approach, Codex with edit mode is similar to or better than existing Java state-of-the-art repair tools in fixing incorrect codes [13].

Due to the unavailability of the closed-source Codex used by Fan *et al.*'s approach as well as MMAPR, and the lack of C++ language support in Clara, ITSP, Verifix, and Refactoring, this paper focuses on evaluating the repair capabilities of DrRepair.

3 Study Design

3.1 Research Questions

- **RQ-1**: ***How effective are state-of-the-art LLMs in repairing incorrect code?*** We evaluate the realistic performance of 12 prominent LLMs in repairing code based on TutorCode (an uncrawled dataset that enables fair and unbiased evaluation of LLMs for code repair). We assess both the correctness as well as the precision of the generated patches. A baseline for evaluation is established using input data consisting of incorrect code and associated programming problem description. Furthermore, this study delves into the influence of code length and difficulty of programming tasks on the repair capabilities of LLMs.
- **RQ-2**: ***Can augmented information assist in strengthening the repair capabilities of LLMs?*** This study employs three types of augmented information to enhance the repair capabilities of LLMs: *solution description* as general hints, *tutor guidance* as specific guidance, and *failing test cases* to provide automated testing feedback. The study provides a thorough analysis of the impact of various combinations of augmented information on the repair capabilities of LLMs.
- **RQ-3**: ***To what extent can conversation-based repair further exploit the repair capabilities of LLMs?*** Utilizing the conversational strengths of LLMs and incorporating human interactions, this study introduces Cref, a semi-automatic conversational repair framework. A large-scale automated experiment is conducted using the TutorCode to evaluate its effectiveness. Cref incorporates three types of augmented information to unlock the potential repair capabilities of LLMs through multiple rounds of dialogues. The study validates the efficacy of conversation-based repair methods by comparing the performance disparities when historical dialogue entries are included or excluded in each conversation turn.

3.2 Benchmark Selection Criteria

In this paper, C++ is selected as the repair benchmark's programming language. Given its widespread application in high performance and interfaces, including game development, large-scale data platforms, and operating systems [24, 59]. Furthermore, C++ is a multi-paradigms programming language, posing additional complexities for program repair tools [59].

3.3 Dataset

LLMs like ChatGPT often incorporate code from various sources, including GitHub, into their training datasets [9, 46, 56]. Most C++ repair benchmarks, such as IntroClass [20] and ITSP [77], are hosted on GitHub. Additionally, benchmarks like Bugs-C++ [3] and ManyBugs [20] compile various historically popular open-source projects from GitHub. Qingyuan *et al.* highlighted that frequently incorporating GitHub-hosted program repair benchmarks into conventional training datasets leads to data leakage and subsequent inaccuracies in performance evaluation [35]. This suggests a widespread potential for data leakage across existing C++ repair benchmarks. To mitigate the risk of data leakage during our experimental analysis, we introduce a large-scale uncrawled C++ repair benchmark, denoted as TutorCode. TutorCode originates from proprietary, internal data collated by a company, specifically from data structure and algorithm courses aimed at training novice developers to experts. TutorCode is publicly available through API to mitigate the risk of unauthorized crawling into the training corpus of LLMs. Under the usage license, API users are strictly limited to uploading TutorCode to the public network. TutorCode comprises 1,239 incorrect codes written by 427 students and covers 35 programming problems, adapted from real-world development context to single-task problems assessed using standardized input-output test cases. Each incorrect code is accompanied by problem description (Figure 1a), tutor guidance (Figure 1b), solution description (Figure 1c), and failing test cases (Figure 1d). The public benchmark TutorCode will be **continuously** updated with new problems and buggy codes in the future, and will serve as a fundamental resource for future code-related research.

Table 1: 12 Tiers of TutorCode.

Tiers Range	Summary	Key Objectives
T1, T2, T3	Introduction to C++ Grammar	Abstract Thinking, Program Thinking
T4, T5, T6	Introduction to Data Structures and Basic Algorithms	Object Thinking, Relationship Modeling
T7, T8, T9	Common Data Structures and Algorithms	Algorithm Modeling, Abstract Stateful Thinking
T10, T11, T12	Advanced Data Structures and Algorithms	Time-space Trade-off, Mathematical Deduction

All the programming problems in TutorCode are derived from software development scenarios. Figure 1a illustrates that each problem description includes a title, task objective, input/output formats, and constraints/limitations. The constraints describe the accepted data formats and ranges of input values, and the limitations impose bounds on time and memory usage. The problems cover various data structures and algorithms, thoroughly evaluating LLMs' capabilities. For each programming problem, TutorCode contains 5-10

```
# Artificial Intelligence

 In order to make users believe they are talking to an AI instead of a real person, it is not an easy task to take on this position. Mr. Garlic is one of the best in this role, and his performance is outstanding.
 The company's top management praised Mr. Garlic's abilities and organized a learning activity for him to share his work experience. However, when everyone came to Mr. Garlic's workstation, they found no one in the chair, and only saw a program running on the computer:
 Accept user input, then:
  1. Change all capital English letters in the original text to lowercase, except for `I`;
  2. Replace all standalone `can you` and `could you` in the original text with `I can` and `I could`, respectively;
  3. Replace all standalone `I` and `me` in the original text with `you`;
  4. Convert `?` to `!`;
 Finally, output the result as the reply to the user. Can you understand this program?
 ### Input Format
 The first line contains an integer n, indicating that there will be n operations. The next n lines contain a string representing user input (len ≤ 100, no more than 100 lines).
 ### Output Format
 For each user input, output a corresponding line of string, representing the AI's output content.
 ### Time Limit
 1000ms
 ### Memory Limit
 65536KB
```

(a) Programming problem description

```
Since line 30 has judged f[i+1], it should let i++ and skip f[i+1].
Also, note that there are multiple sets of data, and you need to output for each input. Remember to clear the f array.
```

(b) Tutor guidance

```
Use `getline()` to read each line of the string, and use the `isalnum` function to determine whether it is a number or a letter. If it is a number or a letter, a non–0 value is returned. If it is, check if it is an `I`, as we need to convert it to lowercase. If it is not a number or a character, insert a space (the reference code for handling spaces uses `stringstream`, but other methods can also be used). After processing the spaces, split the words in the sentence for judgment. Pay attention to the spaces before and after the words and the ultimate string operation. Pay attention to the details, and remember to output the original sentence first!
```

(c) Solution description

```
[INPUT]   5
          Can YoU aNswer me ?
          .....
[OUTPUT]  I can answer you !
          .....
```

(d) Failing test cases

Figure 1: Examples of programming problem description and three types of augmented information.

sets of paired input-output test cases, as detailed in Figure 1d. Since 2017, these test cases have been continually refined to ensure their quality. Tutor guidance is provided as targeted hints after a code review without revealing the corrected code, as shown in Figure 1b. Solution descriptions offer a high-level approach rather than specific code implementations, depicted in Figure 1c.

Table 2: Statictics of TutorCode.

Title	Category	Tier	Codes[1]	Lines[2]	Hunks[3]
Absolute Value Sorting	Branch	T1	85	29.5	2.9
Number of Days in a Month	Branch	T1	64	20.5	2.6
Binary Tree Sorting	Binary Tree	T10	36	45.6	4.9
Magical Key	Char	T2	81	23.8	3.0
Receipt	Loop	T2	15	32.6	2.0
Chasing the Enemy	Loop	T3	88	15.1	2.1
Compression Technology	Array	T3	22	33.3	2.9
Find the Longest Word	Loop	T3	67	24.9	3.7
Advanced Integer Sorting	Sort	T4	66	32.8	3.1
Gem Collector II	Sort	T4	26	46.3	4.0
Maximum Submatrix	Enumeration	T4	33	36.7	3.2
Artificial Intelligence	Ad Hoc	T5	63	41.5	4.4
Onion Girl's Flying Chess	Recursion	T5	20	29.4	4.2
Annoying Queue Jumping	Queue	T6	60	41.6	3.3
Complex Stacks	Stack	T6	22	74.6	5.9
Talent Show	Vector	T6	54	35.0	3.9
Four Squares Theorem	Enumeration	T7	30	25.5	3.5
High Precision Factorial	Big Integer	T7	11	46.8	4.4
p Nodes	Tree Structure	T7	48	40.3	4.4
Perfect Match	Prefix Sum	T7	15	31.5	3.3
Program Design T2	Binary Search	T7	8	31.9	3.4
Program Design T3	Big Integer	T7	23	27.0	2.8
2n Queens Problem	DFS	T8	37	52.3	4.9
Elevator	Maths	T8	26	31.3	5.1
Going Outdoors	DFS	T8	19	47.3	2.8
Information Parser	Maths	T8	34	65.4	6.1
Super Bookshelf 2	DFS	T8	14	27.9	2.7
Escape	DP	T9	24	74.2	5.8
I Want to Stay Healthy Today	DP	T9	33	38.4	4.0
Noble Shops	DP	T9	8	33.5	5.1
Furious Stones	DP	T10	30	30.0	3.4
Maze	BFS	T10	20	76.2	9.3
Foodie Mr. Garlic	DP	T11	23	47.6	5.6
Highways	Graph Theory	T12	13	104.8	13.3
Mr. Garlic's Treasure Hunt	Graph Theory	T12	21	145.6	10.3

[1] "Codes" refers to the number of incorrect codes.
[2] "Lines" refers to the average lines of incorrect codes.
[3] "Hunks" refers to the average changed hunks between incorrect codes and corrected codes.

Contrastingly, TutorCode is uniquely structured into 12 difficulty tiers, each labeled by software developer experts. These tiers are further subdivided into four stages. The objectives for each stage are outlined in Table 1 and correspond to critical cognitive stages in the professional development of novice developers. Each tier has at least two programming problems and a minimum of 34 incorrect codes. A hierarchical arrangement of 20 knowledge tags, organized by tier, is provided in Table 2. As one ascends the difficulty tier, there is a corresponding increase in algorithmic complexity and the intricacy of code implementation. On average, each programming problem in TutorCode includes 7.27 domain expert-designed test cases that have been consistently evaluated as sufficiently valid for assessing code correctness from 2017 to the present. To avoid flaky tests for the repaired codes, we offer a public testing API for each programming problem within TutorCode, facilitating the direct acquisition of results and eliminating environmental instabilities.

We have preprocessed incorrect codes by removing duplicates, disregarding whitespace differences, to ensure the uniqueness of bugs within TutorCode. Compared to widely-used Java repair benchmark Defects4J [26], TutorCode exhibits significant assessment diversity, not only in the availability of tutor guidance information but also in terms of bug complexity. Only 18.6% of bugs in TutorCode involve one modified hunk versus 64.2% in Defects4J. TutorCode spans a wider array of complexities, from one to more than six modified hunks. Additionally, in TutorCode, 47.8% of incorrect codes contain multiple functions, and 38.4% of incorrect codes exhibit defects across multiple functions, while all the buggy codes in Defects4J are single-function modifications. TutorCode provides a foundation for future investigations into the repair capabilities of tools across various complexities.

3.4 Evaluation Metrics

Given LLM's inherent ***randomness***, multiple requests with an identical prompt may yield different outputs. Multiple requests are made to LLMs using the same prompt to mitigate this randomness and facilitate robust analytical outcomes. Prior research has shown that typically, five instances are generated using LLMs for each incorrect code.

1. Repair Correctness. This study evaluates the correctness of LLM-based repairs through two key quantitative metrics: TOP-5

and AVG-5. **TOP-5** quantifies the likelihood that LLMs will produce at least one accurate repair out of five attempts. This measurement is designed to align with the highest number of code revisions most developers are willing to review [29]. Additionally, **AVG-5** is a more stable metric than the somewhat unpredictable **TOP-1** that LLMs produce; it reflects the mean number of accurate repairs across five trials.

2. Patch Precision. In **programming education scenarios**, minimizing code changes is the key to preserving intent for students, aiding their understanding of how to fix bugs [22, 57]. Existing program repair tools often generate larger patches that deviate from the original codes [66], while optimal patches should be minimalistic, addressing code defects without introducing unnecessary modifications [12]. Therefore, evaluating the size of patches generated by LLMs becomes vital in assessing the precision of LLM-based repair methods. Gulwani *et al.* [22] introduced the Relative Patch Size (RPS) as a metric to quantify patch size. RPS is formulated as follows:

$$RPS(AST_i, AST_r) = \frac{TED(AST_i, AST_r)}{Size(AST_i)}$$

Here, AST_i and AST_r denote the Abstract Syntax Tree (AST) of the incorrect code and LLM-generated repaired code, respectively, while $TED(AST_i, AST_r)$ is the Tree-Edit-Distance (TED) between them. $Size(AST_i)$ represents the AST node count of the incorrect code. According to the formula, RPS scores can exceed 1.0 and approaching infinity for null programs, and they are inherently affected by the distribution of successful repairs. This study introduces a new metric, the Relative Patch Size Ratio (**RPSR**), to address this limitation and utilize ground truth corrected codes. RPSR is defined as:

$$RPSR(AST_i, AST_c, AST_r) = \frac{RPS(AST_i, AST_r)}{RPS(AST_i, AST_c)} = \frac{TED(AST_i, AST_r)}{TED(AST_i, AST_c)}$$

Where AST_c means the abstract syntax tree of ground truth corrected code. For each LLM-fixed code, a lower RPSR means a more precise patch, while RPSR values lower than 1.0 indicate smaller patch sizes produced by LLMs compared to ground truth patches.

3.5 Models

This study evaluates twelve state-of-the-art LLMs, divided into eight open-source and four closed-source LLMs. The LLMs are selected based on the number of downloads from official repositories hosted by HuggingFace. The open-source LLMs are shown in Table 3, featuring parameter sizes ranging between 6b and 16b. Within this table, the column labeled *#Param* specifies the size of the model parameters, and the *Downloads* column provides the aggregate number of downloads for each LLM up to September 2023 across all sub-categories. The *HumanEval TOP-1* column presents the TOP-1 performance metric for each LLM, as measured by the HumanEval [11] benchmark.

Table 4 enumerates the closed-source LLMs, with the *Institution* column describing each model to its respective affiliated organization. The remaining columns in this table shared the same attributes as those outlined in Table 3.

Table 3: 8 selected open-source LLMs (Sept. 2023).

Model	Training Dataset	#Param	Downloads	HumanEval TOP-1
CodeLlama-instruct-13B	N.R.	13B	522.6k	42.7 [56]
Vicuna-13B	BigQuery	13B	440.2k	15.5 [39]
CodeGen-6B / 16B	BigQuery	6B / 16B	91.1k	27.7 / 32.2 [39]
StarChat-alpha	Stack Dedup v1.2	16B	82.6k	30.0 [6]
CodeT5p-16B	CodeSearchNet	16B	21.7k	30.9 [40]
Incoder-6B	N.R.	6.7B	12.4k	15.6 [39]
Replit-code-v1	Stack Dedup v1.2	2.7B	2.3k	21.9 [55]

Table 4: 4 selected closed-source LLMs.

Model	Institution	#Param	Usage Type	HumanEval TOP-1
GPT-4-0613-8k	OpenAI	1800B	API / Web	67.0 [46]
GPT-3.5-Turbo-0613-4k	OpenAI	175B	API / Web	48.1 [46]
Claude-instant-v1	Anthropic	52B	Web	47.6 [40]
Bard	Google	137B	Web	44.5 [40]

In the subsequent experiments, specific abbreviations are used for ease of reference: GPT-4 to GPT-4-0613-8k, GPT-3.5 to GPT-3.5-Turbo-0613-4k, Claude to Claude-instant-v1, StarChat to StarChat-alpha, CodeLlama to CodeLlama-instruct-13B. The sampling temperature for these selected LLMs is set to a consistent value of 1.0, which aligns with the default settings employed across most LLMs.

3.6 Prompts and Augmented Information

The performance of LLMs in repair tasks is significantly influenced by the prompt's design [10, 27, 62]. The optimal prompt format, depicted in Figure 2, mitigates the adverse impact of lengthy prompts. This format includes a problem description, an incorrect code encased in triple backticks, and a task prompt.

```
This is a programming problem description:
{{description}}
This is an incorrect code to the problem:
{{incorrect code}}
You are a software engineer. Can you repair the incorrect code?
```

Figure 2: The prompt format of the baseline.

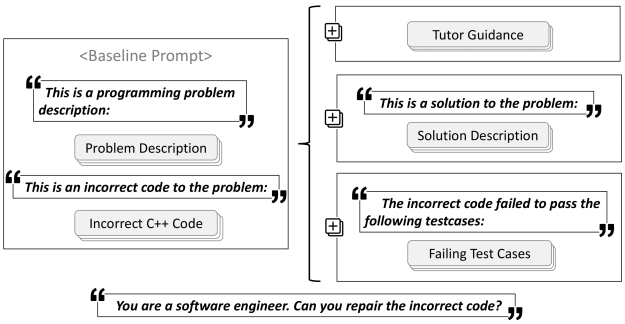


Figure 3: The baseline's prompt structure entails three augmented information types.

To assess the impact of augmented information on LLM's repair capabilities, comprising tutor guidance, solution description, and failing test cases, a uniform prompt structure was used in the following experiments to avoid sensitive prompts for LLMs. This framework, displayed in Figure 3, initiates with a problem description and incorrect code consistent with the baseline prompt. It then incorporates one or more types of augmented information. The solution description commences with the phrase *"This is a solution to the problem:"*, followed by the complete content of the solution

description. In the context of tutor guidance, content is explicitly provided. When presenting failing test cases, the framework employs the introductory phrase *"This incorrect code failed to pass the following test cases:"* succeeded by the corresponding failed input-output pairs. The concluding task description retains alignment with the baseline, thus facilitating a rigorous comparative analysis of the impact of augmented information on the repair performance of LLMs.

4 Experiment & Result

4.1 Realistic Repair Performance of LLMs

[Experiment Goal]: We aim to investigate the realistic repair performance of 12 state-of-the-art LLMs and an existing ITS technique, utilizing the TutorCode as our benchmark.

[Experiment Design]: This experiment offers LLMs with prompts in the baseline format, as described in Section 3.6. The repair capabilities of LLMs are assessed in terms of three metrics: TOP-5, AVG-5, and RPSR, as detailed in Section 3.4. The AVG-5 results of 5 best-performing LLMs across 12 tiers are calculated to compare the repair performance of LLMs across various tiers and analyze their trends. Additionally, we analyze the impact of code length on the repair capabilities of LLMs. We calculate the boxplot distribution of the length of incorrect codes for correct and incorrect predictions generated by LLMs.

Table 5: TOP-5, AVG-5, RPSR results of 12 selected LLMs on TutorCode.

Model	TOP-5	AVG-5	H-TOP-1*	RPSR
GPT-4	66.5%	52.0%	67.0%	3.748
GPT-3.5	56.8%	41.5%	48.1%	5.072
Claude	34.1%	20.3%	47.6%	4.083
Bard	27.9%	16.8%	44.5%	4.728
CodeLlama	15.9%	6.8%	42.7%	3.635
StarChat	11.7%	5.7%	30.0%	8.028
Vicuna-13B	1.4%	0.5%	15.5%	6.140
CodeGen-multi-16B	0.8%	0.3%	32.2%	1.745
CodeGen-multi-6B	0.0%	0.0%	27.7%	/
CodeT5p	0.3%	0.1%	30.9%	1.013
Incoder	0.2%	0.1%	15.6%	0.860
replit-code-v1	0.1%	0.1%	21.9%	0.959
DrRepair	0.2%	0.2%	/	1.731

* "H-TOP-1" represents the TOP-1 of LLMs on the HumanEval [11].

[Experiment Result]: Table 5 presents the repair performance of 12 LLMs, comprising 4 closed-source LLMs first, followed by 8 open-source LLMs. The table shows that the closed-source LLMs exhibit superior performance compared to the open-source LLMs across multiple evaluation metrics, including TOP-5, AVG-5, and RPSR. Regarding correctness metrics, TOP-5 and AVG-5, GPT-4 demonstrates the highest performance, achieving a TOP-5 of 66.5% and an AVG-5 of 52.0%, closely followed by GPT-3.5. The gap in performance between Claude/Bard and GPT-3.5/4 is notably significant within the TOP-5 and AVG-5 metrics. Among the open-source LLMs, CodeLlama and StarChat emerge as the best-performing for correctness, indicated by TOP-5 and AVG-5. In terms of AVG-5, CodeLlama achieves 6.8%, while StarChat achieves 5.7%. Except for CodeLlama and StarChat, Vicuna-13B exhibits the highest performance in terms of correctness, achieving merely a TOP of 1.9% and an AVG-5 of 0.6%. This result suggests that other open-source LLMs are ineffective for program repair tasks. Furthermore, among all the LLMs with AVG-5 scores exceeding 1%, CodeLlama stands out with the lowest RPSR, indicating that CodeLlama excels in generating precise code patches. DrRepair, having only successfully repaired two incorrect codes, showcases the limited repair capabilities of traditional ITS techniques on TutorCode compared to most selected LLMs.

The table also presents the TOP-1 results of LLMs on the HumanEval benchmark, denoted as H-TOP-1. Open-source LLMs, such as CodeLlama, CodeGen-multi-16B, StarChat, and CodeT5p, demonstrate H-TOP-1 results that are closely aligned with closed-source LLMs. Notably, the H-TOP-1 result of CodeLlama is nearly on par with that of GPT-3.5. However, when evaluated on TutorCode, the AVG-5 (i.e., the average TOP-1) results of open-source LLMs are considerably inferior to those of closed-source LLMs, which indicates considerable potential for improving open-source LLMs in practical repair scenarios.

Table 6: AVG-5 results of 5 best-performing LLMs across 12 tiers.

Tier	GPT-4	GPT-3.5	Claude	Bard	CodeLlama
T1	0.946	0.879	0.715	0.577	0.262
T2	0.885	0.615	0.198	0.158	0.083
T3	0.706	0.576	0.267	0.190	0.051
T4	0.664	0.576	0.293	0.298	0.144
T5	0.169	0.096	0.012	0.012	0.000
T6	0.493	0.346	0.037	0.051	0.052
T7	0.504	0.363	0.113	0.061	0.060
T8	0.285	0.192	0.114	0.111	0.054
T9	0.139	0.139	0.031	0.018	0.000
T10	0.186	0.070	0.040	0.042	0.000
T11	0.217	0.130	0.026	0.026	0.000
T12	0.000	0.000	0.000	0.000	0.000

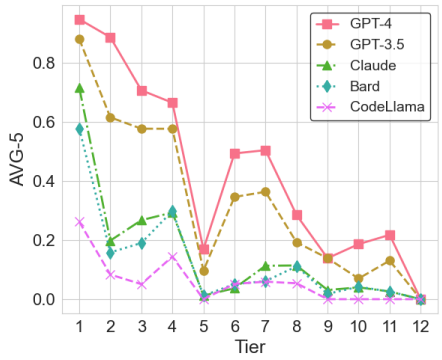


Figure 4: AVG-5 trend of the 5 best-performing LLMs.

We select 5 best-performing LLMs and compute their AVG-5 across 12 tiers, as illustrated in Table 6. The table shows that GPT-4 and GPT-3.5 consistently demonstrate superior performance across all tiers. Notably, GPT-4 achieves an impressive AVG-5 of 94.6% for tier T1, and it gradually declines with increasing tier, except tier T5. The unexpected low AVG-5 results for tier T5 are addressed in Section 5. Figure 4 emphasizes the descending trend of AVG-5 for each LLM across the 12 tiers.

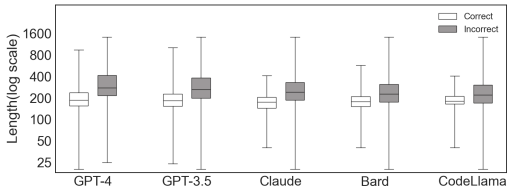


Figure 5: Distributions of correct/incorrect code lengths.

Figure 5 illustrates the distributions of code lengths for correct and incorrect predictions generated by the 5 selected LLMs. It is observed that correct predictions (present in white) tend to have shorter code lengths compared to incorrect predictions (present in grey) across all the selected LLMs. Furthermore, it can be observed that longer incorrect codes adversely impact LLMs' repair capabilities, which aligns with the prior research [62]. Statistical significance between the correct and incorrect distributions was confirmed through the Mann-Whitney-Wilcoxon (MWW) [41] test, supporting that correct predictions tend to have shorter code lengths. As illustrated in the figure, GPT-4 and GPT-3.5 exhibit better capabilities to handle lengthy prompts.

[RQ-1] Findings: *(1) In comparative analysis based on the AVG-5 metric, closed-source LLMs such as GPT-4 and GPT-3.5 demonstrate a superior performance, registering AVG-5 of 52.0% and 41.5%, respectively, in contrast to a meager 6.8% recorded by leading open-source LLMs. (2) Although CodeLlama exhibits commendable performance on the HumanEval benchmark with a TOP-1 score of 42.7%, its correctness significantly declines when evaluated on TutorCode, trailing GPT-3.5 by a substantial AVG-5 margin of 34.7%. This observation suggests that relying on small-scale benchmarks such as HumanEval may not accurately assess an LLM's repair capabilities. (3) Statistical analyses using box plots reveal that elongated code adversely impacts the repair performance of LLMs.* **Insights:** *(1) The observed disparities in repair performance between closed-source and open-source LLMs highlight the imperative for the research community to redouble efforts to advance open-source LLMs. (2) Given the marked difference in performance metrics between TutorCode and HumanEval, it is advisable for the research community to assess the repair capabilities of LLMs using large-scale, uncrawled benchmarks.*

4.2 Enhancements of Augmented Information

[Experiment Goal]: We aim to explore the enhancements of augmented information on the repair capabilities of LLMs.

[Experiment Design]: This experiment leverages 5 LLMs based on their outstanding performance in RQ-1, including GPT-4, GPT-3.5, Claude, and CodeLlama. The repair capabilities of LLMs are evaluated on TutorCode. This experiment evaluates three levels of augmented information: general hints, specific guidance, and the automated execution result. Each of these three levels contains three types of information: solution description, tutor guidance, and failing test cases, as described in Section 3.6. The prompts are structured in the format described in Figure 3. Seven different combinations of these three types of information are evaluated in this experiment, as detailed in the first column of Table 7. In the following paragraphs, we denote the combination of all three types of augmented information as **T&S&F**. Different types of augmented information are segmented into separate conversational entries within the T&S&F prompts to mitigate the adverse impacts of lengthy prompts. The repair capabilities of various combinations of augmented information are assessed in terms of two metrics: AVG-5 represents correctness, and RPSR represents patch precision. In our analysis of the enhancements of augmented information on the repair capabilities of LLMs across different tiers, we calculate AVG-5 for both the GPT-4 and GPT-3.5 across the 12 tiers for each combination of information. Subsequently, we analyze the corresponding performance enhancements and trends on tiers.

Lengthy prompts have been found to have a detrimental influence on the program repair of LLMs [10, 27, 62]. We calculate the length of input prompts with various combinations of information within TutorCode, as illustrated in Figure 6. The extended length of prompts for failing test cases in the TutorCode dataset is attributable to several factors. Unlike unit tests in other benchmarks, these test cases encompass comprehensive inputs and outputs, incorporating more data. Additionally, we present all failing test cases of the incorrect code within a single prompt. To reduce the negative impacts of lengthy prompts, we segment different types of information into separate conversational entries in this experiment.

[Experiment Result]: Table 7 presents the AVG-5 and RPSR results of the 5 selected LLMs. The table illustrates that among the three types of augmented information, tutor guidance emerges as the most effective in enhancing repair performance across all LLMs, as indicated by both AVG-5 and RPSR. Conversely, failing test cases negatively affect the repair performance of GPT-4, Claude, and Bard, as evidenced by both the AVG-5 and RPSR metrics. This deterioration in performance may be attributed to the lengthy prompt problem of failing test cases, as illustrated in Figure 6, the same as previous researchers have found [10, 27, 62]. Meanwhile, it has been demonstrated that employing LLMs solely with failing test cases in repair tasks, like ChatRepair [73], can not always surpass corresponding baselines on the TutorCode when leveraging Claude, Bard, and CodeLlama.

For cases combining two types of augmented information, the combination of tutor guidance and solution description yields the most significant AVG-5 enhancements across all LLMs except Bard. Meanwhile, combining tutor guidance and failing test cases outperforms other combinations across all LLMs. The table shows that LLMs do not always attain the highest performance enhancements by T&S&F among all 7 combinations. For example, Bard, Claude, and CodeLlama do not achieve the best AVG-5 results. CodeLlama, in particular, produces a gap of 8.0% compared to solely providing tutor guidance. All the selected LLMs provided with T&S&F do not achieve the best RPSR results compared to other combinations. These findings suggest that the issues of limited attention of LLMs persist even when all three types of augmented information are provided separately.

For any combination of augmented information, GPT-4 demonstrates a significant lead in the AVG-5 metric. The AVG-5 of other LLMs are ranked in descending order, with GPT-3.5, Claude, Bard, and CodeLlama following. Notably, CodeLlama exhibits the best RPSR result among all the selected LLMs. This finding further proves that CodeLlama achieves a higher patch precision, as discovered in RQ-1.

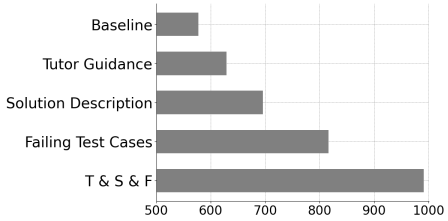
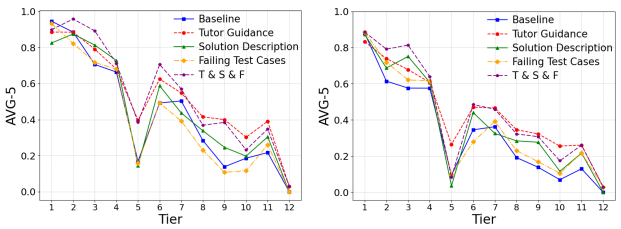


Figure 6: Average tokens of prompt types with different augmented information.

Table 7: AVG-5 and RPSR results of 5 LLMs provided with various combinations of augmented information.

Combination	GPT-4		GPT-3.5		Claude		Bard		CodeLlama	
	AVG-5	RPSR	AVG-5	RPSR	AVG-5	RPSR	AVG-5	RPSR	AVG-5	RPSR
Baseline	52.0%	3.748	41.5%	5.072	20.3%	4.083	16.8%	4.728	6.8%	3.635
Tutor Guidance (T)	61.4%	2.210	50.9%	2.950	27.1%	1.834	26.7%	1.939	16.8%	2.042
Solution Description (S)	55.6%	4.069	47.4%	5.638	21.3%	4.485	16.7%	4.393	7.7%	3.631
Failing Test Cases (F)	49.7%	4.366	42.2%	6.360	19.4%	5.285	15.1%	5.112	7.5%	4.005
T & S	61.9%	2.947	51.3%	3.069	27.6%	2.568	19.3%	2.497	14.8%	3.011
T & F	59.7%	2.736	51.3%	3.490	27.5%	2.101	26.6%	2.231	16.3%	1.980
S & F	53.2%	4.258	47.0%	6.556	24.9%	4.667	15.9%	5.098	7.9%	5.383
T & S & F	62.3%	2.778	52.4%	5.201	27.6%	3.213	24.4%	4.235	8.8%	3.376

Green cells show the best-performing combination. Gray cells are below the baseline.



(a) GPT-4 **(b) GPT-3.5**

Figure 7: AVG-5 trend of (a) GPT-4 and (b) GPT-3.5 provided with three types of augmented information across 12 tiers.

The AVG-5 results for GPT-4 and GPT-3.5 across 12 tiers are illustrated in Figure 7a and Figure 7b. Notably, for GPT-4 and GPT-3.5, the AVG-5 results for tier T5 are significantly lower than the baseline when only the solution description is provided. This finding implies potential issues with the solution description of programming problems in tier T5, further analyzed in Section 5. Furthermore, the AVG-5 for each successive tier exhibits a decreasing trend, except for tier T5.

[RQ-2] Findings: *(1) Among the three types of augmented information, tutor guidance notably outperforms the others in enhancing the repair capabilities of LLMs, showing a significant increase of over 6.8% in the AVG-5 metric across all evaluated LLMs. (2) Conversely, the simple integration of multiple types of augmented information does not consistently produce benefits over the baseline. For instance, in terms of Bard, incorporating both solution description and failing test cases results in a 0.9% AVG-5 decrease relative to the baseline.* **Insights:** *Human guidance information improves the repair capabilities of LLMs significantly; it is valuable for future researchers to explore assisting rather than replacing humans in the programming education scenarios.*

4.3 Conversational Program Repair

[Experiment Goal]: We introduce an LLM-based conversational semi-automatic repair framework Cref and investigate its repair capabilities using augmented information through automated evaluation.

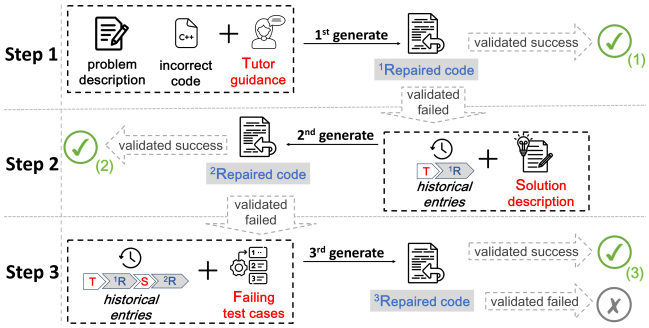


Figure 8: Overview of Cref.

[Experiment Design]: This experiment employs five top-performing LLMs identified in RQ-1, consistent with the methodology used in RQ-2. Utilizing LLMs with various types of augmented information often results in lengthy prompts, potentially leading to issues of limited attention. To counteract the detrimental effects of lengthy prompts, we introduce a repair strategy termed MultiRegenerate. This strategy partitions the repair process into three phases: tutor guidance, solution articulation, and the presentation of failing test cases. The MultiRegenerate initiates three distinct dialog sessions for each incorrect code, thereby mitigating the detrimental effects of lengthy prompts. In programming education scenarios, it is crucial to accelerate the process of repairing students' debugging requests [33]. Tutor guidance, due to its considerable impact on enhancing performance (shown in Table 7), is prioritized as the first in our approach. Subsequently, LLMs are furnished with solution descriptions, ranking second in performance improvement. Lastly, LLMs are provided with failing test cases, which yield the least performance enhancement related to lengthy prompts.

Although MultiRegenerate mitigates the adverse effects of lengthy prompts, it discards historical conversation entries. However, the historical entries could potentially aid LLMs in avoiding the same errors in subsequent generations [73] and fully utilizing conversational capabilities, which is similar to the idea of chain-of-thought (CoT) [69]. To unlock the potential conversational capabilities of LLMs, we propose a repair framework named Cref. Figure 8 illustrates the overview of Cref. In Cref, there is only one dialog session instead of three dialog sessions compared to MultiRegenerate. The Cref follows three steps: (1) Providing LLMs with programming problem description, incorrect code with corresponding tutor guidance, and generates repaired code. If the repaired code is validated successfully, then the process ends. (2) Providing LLMs with all historical entries and solution descriptions to generate a repaired code. If the repaired code is validated successfully, the process ends. (3) Providing LLMs with all historical entries with failing test cases of the last repaired code to generate the repaired code. The success of the repair process is determined by whether or not there is a correct repaired code so far.

We evaluate the repair performance of two proposed methodologies, MultiRegenerate and Cref, utilizing 5 selected LLMs on the TutorCode, in terms of AVG-5 and RPSR. We compare these results with the baseline and T&S&F. We further evaluate the AVG-5 results of Cref utilizing GPT-4 and GPT-3.5 across 12 tiers to analyze trends and enhancements compared to the baseline.

To illustrate Cref's scalability, we compared its repair capabilities with baseline, T&S&F, and MultiRegenerate, using GPT-4 and GPT-3.5, on **TutorCodePlus**. TutorCodePlus resembles TutorCode but includes a broader selection of 2,464 incorrect code submissions. These submissions, which come with tutor guidance, are created by 786 students across 45 distinct programming problems, consistently classified into 12 tiers. TutorCodePlus integrates all 35 problems from TutorCode, incorporating its original 1,239 instances alongside additional new samples. On average, each problem in TutorCodePlus includes 6.98 pairs of well-designed test cases. Notably, while TutorCode features an average of 1.89 functions per incorrect code, TutorCodePlus showcases a similar complexity with an average of 1.81 functions. TutorCodePlus does not guarantee the inclusion of codes corrected by the students, leading us to omit the RPSR metric on TutorCodePlus. Due to the company's commercial considerations, TutorCodePlus will remain proprietary, but we will update TutorCode with more samples to support the program repair research community.

Table 8: AVG-5 and RPSR results of T&S&F, MultiRegenerate, and Cref compared to the baseline.

Model	Baseline		T&S&F		MultiRegenerate		Cref	
	AVG-5	RPSR	AVG-5 (↑)	RPSR	AVG-5 (↑)	RPSR	AVG-5 (↑)	RPSR
GPT-4	52.0%	3.748	62.3% (+10.3%)	2.778	71.5% (+19.5%)	2.815	76.6% (+24.6%)	2.691
GPT-3.5	41.5%	5.072	52.5% (+11.0%)	5.201	62.6% (+21.1%)	3.438	63.8% (+22.3%)	3.454
Claude	20.3%	4.083	27.6% (+ 7.3%)	3.213	38.9% (+18.6%)	2.782	42.3% (+22.0%)	2.768
Bard	16.8%	4.728	24.4% (+ 7.6%)	4.235	35.2% (+14.9%)	2.562	35.4% (+18.6%)	2.618
CodeLlama	6.8%	3.635	8.8% (+ 2.0%)	3.376	22.5% (+15.7%)	2.781	24.0% (+17.2%)	2.880

A smaller RPSR is better, meaning the patch is more precise.

[Experiment Result]: Regarding AVG-5, Cref exhibits superior lead performance, outperforming the baseline and T&S&F by margins of 24.6% (GPT-4) and 15.2% (CodeLlama), respectively, and surpassing MultiRegenerate. Although MultiRegenerate achieves satisfactory repair performance, initiating three distinct dialog sessions for each repair requires more computational resources in practical engineering scenarios. Regarding RPSR, both MultiRegenerate and Cref exhibit significant improvements compared to the baseline and T&S&F. A lower RPSR value indicates greater precision of patches. Compared to MultiRegenerate, which modifies the initial incorrect code in a single iteration, Cref interactively repairs the initial incorrect code over multiple rounds of dialogue. Despite this, Cref maintains a comparable RPSR to MultiRegenerate while achieving a significant lead in AVG-5.

The AVG-5 trends of Cref and baseline across 12 tiers when utilizing GPT-4 and GPT-3.5 are illustrated in Figure 7a and Figure 7b, respectively. Cref exhibits a significant improvement over the baseline across all 12 tiers. Furthermore, the AVG-5 results of Cref exhibit a decreasing trend except for tier T5.

Results on Extend Benchmark: We validate the generalization of Cref's repair capability on an extensive benchmark TutorCodePlus. On TutorCodePlus benchmark, when prompt LLMs with the baseline, the AVG-5 result of GPT-4 is 49.3%, and the AVG-5 result of GPT-3.5 is 39.6%. Both are close to the corresponding results in RQ-1. When providing LLMs with T&S&F, the AVG-5 of GPT-4 reaches 60.9%, while the AVG-5 of GPT-3.5 reaches 50.1%, which is an improvement of 11.6% and 10.5% compared to the baseline respectively, and is similar to the results in RQ-2. When leveraging LLMs with MultiRegenerate strategy, the AVG-5 of GPT-4 reaches 68.3%, while the AVG-5 of GPT-3.5 reaches 60.3%. AVG-5 of Cref utilizing GPT-4 reaches **74.4%** while AVG-5 of Cref utilizing GPT-3.5 reaches 61.9%. For GPT-4 and GPT-3.5, the AVG-5 of Cref compared to MultiRegenerate is higher by **6.1%** and 1.6%, respectively, and this gap is higher than that of the results on TutorCode. These results demonstrate the scalability of our findings.

[RQ-3] Findings: *(1) In assessing the selected five LLMs, Cref shows marked improvements across both the AVG-5 and RPSR metrics when compared to the baseline and the T&S&F. Notably, every LLM in this experiment exhibits an increase in AVG-5 performance by a minimum of 17.2% over the baseline, with GPT-4 outperforming all others by achieving an impressive 24.6% improvement. (2) When compared with MultiRegenerate, Cref maintains a similar RPSR while securing a higher AVG-5. These results highlight the advantage of integrating historical conversational entries to enhance the repair capabilities of LLMs. (3) Utilizing GPT-4 and GPT-3.5, the AVG-5 results of the baseline, T&S&F, MultiRegenerate, and Cref on the TutorCodePlus support the generalizability of our earlier findings.* **Insights:** *The performance of Cref up to 76.6% of AVG-5 suggests that Cref is a useful semi-automatic repair tool in education scenarios for assisting programming tutors, and future research could further explore improving patch precision for the students.*

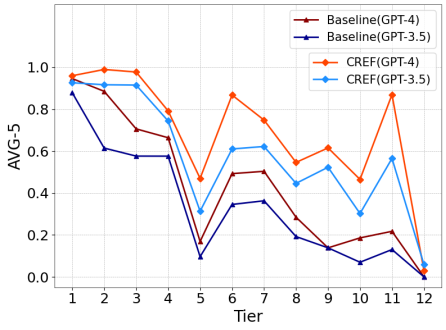


Figure 9: AVG-5 trend of Cref utilizing GPT-4/GPT-3.5.

5 Discussion

5.1 Investigation of Outlier Data

There are two programming problems in Tier 5: *Onion Girl's Flying Chess* and *Artificial Intelligence*. A subsequent investigation reveals inherent ambiguities within the problem statements and the solution description of *Onion Girl's Flying Chess*. We applied modifications to resolve these ambiguities and re-generated the results with GPT-3.5 and GPT-4 on tier Tier 5. With the same configurations, the AVG-5 result of Cref (GPT-4) increases to 76.7%,

whereas Cref (GPT-3.5) achieves an AVG-5 of 59.9%. These findings suggest that the clarity of the input information for LLMs can significantly impact LLMs' repair capabilities.

5.2 Industrial Application

On the company's programming education platform, students learn to code in C++ by working on given assignments. Novice students often require guidance to repair incorrect code and yield a correct solution. Tutors hired by the company help students debug their code, but this process can be labor-intensive and usually involves long response times. To address these challenges, we introduced Cref assisting tutors on a programming education platform to address students' debugging requests. First, tutors only need to provide preliminary guidance instead of debugging solutions, as shown in Figure 1b. Cref can automatically complete the repair process in most cases. Subsequently, tutors review Cref 's generation to formulate a response to students, ensuring precise and effective debugging assistance. Experimental results on TutorCode, where the average tutor guidance consists of just 38.25 words, show that Cref utilizing such preliminary feedback significantly increases the success rate of automated program repairs.

The deployment of Cref has demonstrated significant reductions in labor costs, decreasing the average debugging response time from 26.7 minutes to 7.7 minutes. Cref has shifted the tutors' role towards providing preliminary guidance rather than fully debugging. The labor cost reduction of 71.2% indicates that the same labor cost can address 3.5 times more debugging requirements from the same students.

The success of Cref in educational contexts suggests its potential for broader applications, such as code reviews, where it utilizes reviewer feedback to refine codes, offering developers enhanced solutions automatically. This application of Cref directly contributes to faster code reviews, highlighting its adaptability to various code-related tasks.

6 Threats to Validity

Threats to External Validity. The generalizability of our findings is influenced by two factors: the scale of benchmarks and the risk of data leakage. Consequently, we mitigate these threats by employing a large-scale uncrawled benchmark TutorCode, comprising 1,239 instances. The selection of benchmarks not only aims to cover a wide array of bugs but also mitigate the risk of data leakage, thereby enhancing both the generalizability and reliability of our study.

Threats to Internal Validity. The randomness of LLM outputs influences the reliability of our experimental results. To mitigate this influence, we have adopted a five-fold repetition strategy in our experiments, allowing for a more reliable calculation of the performance metrics TOP-5 and AVG-5. Furthermore, we have standardized the key parameters including *temperature* and *top_p* for all LLMs in our experiments to mitigate the potential influence of parameter variations on the reliability of our comparative analyses. Additionally, automated scripts were employed to extract code from LLM-generated non-compilable natural language explanations when provided with incorrect codes and contextual information to extract compilable codes, ensuring our analysis is based on usable data.

Threats to Construct Validity. The selection of evaluation metrics influences the validity of our findings. To mitigate risks associated with metric selection and ensure accurate evaluation results, we selected metrics TOP-5, AVG-5, and the RPSR. These metrics are well-known for measuring the correctness and precision of LLMs. To address concerns that these metrics potentially may lead to biased conclusions, we evaluated Cref on extensive benchmark TutorCodePlus, which includes 2,464 samples. TutorCodePlus helps us ensure our findings are solid and can be applied to diverse coding bugs.

7 Conclusion

In this paper, we introduce an extensive uncrawled benchmark TutorCode consisting of 1,239 C++ defect codes and types of associated information. Utilizing TutorCode, experiments are conducted to investigate the realistic repair performance of 12 prominent LLMs, and demonstrate the significant difference on HumanEval and TutorCode. We then investigate how augmented information improves the repair capabilities of best-performing LLMs. The experimental results show that tutor guidance improves the LLM repair performance the most, while failing test cases improve the least due to the lengthy prompts. To mitigate the negative impacts of lengthy prompts, we propose a three-round strategy MultiRegenerate to minimize the adverse effects of lengthy prompts, and we introduce a conversational semi-automatic repair framework Cref that optimizes the utilization of augmented information and conversational capabilities of LLMs. Experimental results indicate that the repair performance of Cref obtains significant lead in AVG-5 and RPSR metrics compared to the baseline, T&S&F, and MultiRegenerate. These results suggest that incorporating historical failing repairs can significantly enhance repair capabilities in LLMs by fully exploiting their conversational potential. Cref acts as an assisting tool through a three-step process: (1) tutors provide preliminary guidance of the student's buggy code, (2) based on this guidance, Cref automatically performs code debugging, and (3) tutors leverage Cref 's generation to reply to students. This approach has cut response times by 71.2% and reduced costs by 69.9%, improving the tutoring process and student learning experiences. The success of Cref highlights its potential for broader uses, such as augmenting code review processes by automatically adjusting codes based on reviewers' comments, suggesting future applications for enhancing efficiency in coding-related tasks. Although ChatGPT demonstrates superior performance, its substantial computational requirements due to massive parameters underscore the need for future research to optimize open-source LLMs' repair capabilities while maintaining efficiency.

Acknowledgement

This work has been partly supported by the National Natural Science Foundation (Grant Numbers 62273292 and 62276226), China; by the Innovation Capability Improvement Plan Project of Hebei Province (Grant Number 22567626H), China. This work has also been partly supported by the NATURAL project, which has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant No. 949014).

References

[1] Umair Z Ahmed, Zhiyu Fan, Jooyong Yi, Omar I Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified repair of programming assignments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–31.

[2] Rachith Aiyappa, Jisun An, Haewoon Kwak, and Yong-Yeol Ahn. 2023. Can we trust the evaluation on ChatGPT? *arXiv preprint arXiv:2303.12767* (2023).

[3] Gabin An, Minhyuk Kwon, Kyunghwa Choi, Jooyong Yi, and Shin Yoo. 2023. BUGSC++: A Highly Usable Real World Defect Benchmark for C/C++. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2034–2037.

[4] Anthropic. 2023. Introducing Claude. *Anthropic Blog* (2023). https://www.anthropic.com/index/introducing-claude.

[5] Amos Azaria, Rina Azoulay, and Shulamit Reches. 2023. ChatGPT is a Remarkable Tool–For Experts. *arXiv preprint arXiv:2306.03102* (2023).

[6] Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2023. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. *arXiv preprint arXiv:2306.04556* (2023).

[7] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. 2022. Constitutional AI: Harmlessness from AI Feedback. arXiv:2212.08073 [cs.CL]

[8] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-in-the-loop automatic program repair. In *2020 IEEE 13th international conference on software testing, validation and verification (ICST)*. IEEE, 274–285.

[9] 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.

[10] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. 2023. A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. arXiv:2304.08191 [cs.SE]

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[12] Yukun Dong, Meng Wu, Li Zhang, Wenjing Yin, Mengying Wu, and Haojie Li. 2020. Priority Measurement of Patches for Program Repair Based on Semantic Distance. *Symmetry* 12, 12 (2020), 2102.

[13] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

[15] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 935–947.

[16] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program repair. *arXiv preprint arXiv:2211.12787* (2022).

[17] Xiang Gao and Abhik Roychoudhury. 2020. Interactive patch generation and suggestion. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 17–18.

[18] Li Ge, Peng Xin, Wang Qianxiang, Xie Tao, Jin Zhi, Wang Ji, Ma Xiaoxing, and Li Xuandong. 2023. Challenges from LLMs as a Natural Language Based Human-machine Collaborative Tool for Software Development and Evolution. In *Journal of Software, 2023, 34(10)*. 4601–4606.

[19] Ukeje Chukwuemeriwo Goodness. 2023. What Is Claude AI and Why Should You Use It? *MakeUseOf* (2023). https://www.makeuseof.com/what-is-claude-ai-why-use-it/

[20] Claire Le Goues, Neal J. Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Software Eng.* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[21] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[22] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480.

[23] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 388–398. https://doi.org/10.1109/ASE.2019.00044

[24] Dongchen Jiang and Bo Xu. 2022. Generation of C++ Code from Isabelle/HOL Specification. *International Journal of Software Engineering and Knowledge Engineering* 32, 07 (2022), 1043–1069.

[25] 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).

[26] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[27] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler Hayes, and Christopher Kanan. 2018. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[28] YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated Program Repair from Fuzzing Perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 854–866.

[29] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis*. 165–176.

[30] 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*.

[31] 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.

[32] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic program repair. *IEEE Software* 38, 4 (2021), 22–27.

[33] Changyoon Lee, Junho Myung, Jieun Han, Jiho Jin, and Alice Oh. 2023. Learning from Teaching Assistants to Program with Subgoals: Exploring the Potential for AI Teaching Assistants. *arXiv preprint arXiv:2309.10419* (2023).

[34] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[35] Qingyuan Li, Wenkang Zhong, Chuanyi Li, Jidong Ge, and Bin Luo. 2024. Empirical Study on the Data Leakage Problem in Neural Program Repair. *Journal of Software* 35, 7 (2024), 0–0.

[36] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! (2023). arXiv:2305.06161 [cs.CL]

[37] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R Lyu. 2023. Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study. *arXiv preprint arXiv:2307.05950* (2023).

[38] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–29.

[39] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[40] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[41] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[42] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[43] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 672–683.

[44] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR* (2023).

[45] OpenAI. 2022. Introducing ChatGPT. (2022). https://openai.com/blog/chatgpt

[46] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[47] 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.

[48] Nikhil Parasaram, Earl T Barr, and Sergey Mechtaev. 2023. Rete: Learning Namespace Representation for Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1264–1276.

[49] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *arXiv preprint arXiv:2302.04662* (2023).

[50] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2023. MetaTPTrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5239–5247.

[51] Sundar Pichai. 2023. An important next step on our AI journey. *Google Technology Blog* (2023). https://blog.google/technology/ai/bard-google-ai-search-updates/.

[52] 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.

[53] Fangcheng Qiu, Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Xinyu Wang. 2021. Deep just-in-time defect localization. *IEEE Transactions on Software Engineering* 48, 12 (2021), 5068–5086.

[54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[55] Inc. Repl.it. 2023. replit-code-v1-3b. *Hugging Face Hub* (2023). https://huggingface.co/replit/replit-code-v1-3b.

[56] 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).

[57] Atsushi Shirafuji, Md Mostafizer Rahman, Md Faizul Ibne Amin, and Yutaka Watanobe. 2023. Program repair with minimal edits using codet5. *arXiv preprint arXiv:2309.14760* (2023).

[58] 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).

[59] Bjarne Stroustrup. 2013. *The C++ programming language*. Pearson Education.

[60] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (2022).

[61] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 981–992.

[62] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[63] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this change the answer to that problem? Correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[64] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[65] Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. 2023. Creating a Coding Assistant with StarCoder. *Hugging Face Blog* (2023). https://huggingface.co/blog/starchat.

[66] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2019. How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–12.

[67] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[68] 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. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[69] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[70] Emily Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, and John Woodward. 2022. Let's talk with developers, not about developers: A review of automatic program repair research. *IEEE Transactions on Software Engineering* 49, 1 (2022), 419–436.

[71] 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*.

[72] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. arXiv:2301.13246 [cs.SE]

[73] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385 [cs.SE]

[74] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F. Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-Objective Fine-Tuning for Enhanced Program Repair with LLMs. arXiv:2404.12636

[75] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*. PMLR, 10799–10808.

[76] 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.

[77] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751.

[78] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876* (2022).

[79] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2022. Automated feedback generation for competition-level code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[80] Yuwei Zhang, Zhi Jin, Ying Xing, and Ge Li. 2023. STEAM: Simulating the InTeractive BEhavior of ProgrAMmers for Automatic Bug Fixing. *arXiv preprint arXiv:2308.14460* (2023).

[81] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL]

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