When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done?

Yuxiao Chen
Institute of Software, Chinese Academy of Sciences, China
University of Chinese Academy of Sciences, China
chenyuxiao2021@iscas.ac.cn

Jingzheng Wu ✉
Institute of Software, Chinese Academy of Sciences, China
jingzheng08@iscas.ac.cn

Xiang Ling ✉
Institute of Software, Chinese Academy of Sciences, China
lingxiang@iscas.ac.cn

Changjiang Li
Stony Brook University, USA
meet.cjli@gmail.com

Zhiqing Rui
Institute of Software, Chinese Academy of Sciences, China
University of Chinese Academy of Sciences, China
zhiqing@iscas.ac.cn

Tianyue Luo
Institute of Software, Chinese Academy of Sciences, China
tianyue@iscas.ac.cn

Yanjun Wu
Institute of Software, Chinese Academy of Sciences, China
yanjun@iscas.ac.cn

ABSTRACT

In recent years, large language models (LLMs) have demonstrated substantial potential in addressing automatic program repair (APR) tasks. However, the current evaluation of these models for APR tasks focuses solely on the limited context of the single function or file where the bug is located, overlooking the valuable information in the repository-level context. This paper investigates the performance of popular LLMs in handling repository-level repair tasks. We introduce RepoBugs, a new benchmark comprising 124 typical repository-level bugs from open-source repositories. Preliminary experiments using GPT3.5 based on the function where the error is located, reveal that the repair rate on RepoBugs is only 22.58%, significantly diverging from the performance of GPT3.5 on function-level bugs in related studies. This underscores the importance of providing repository-level context when addressing bugs at this level. However, the repository-level context offered by the preliminary method often proves redundant and imprecise and easily exceeds the prompt length limit of LLMs. To solve the problem, we propose a simple and universal repository-level context extraction method (RLCE) designed to provide more precise context for repository-level code repair tasks. Evaluations of three mainstream LLMs show that RLCE significantly enhances the ability to repair repository-level bugs. The improvement reaches a maximum of 160% compared to the preliminary method. Additionally, we conduct a comprehensive analysis of the effectiveness and limitations of RLCE, along with the capacity of LLMs to address repository-level bugs, offering valuable insights for future research.

CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**; **Automated static analysis**; **Software reliability**; **Software libraries and repositories**; **Software maintenance tools**; **Error handling and recovery**.

KEYWORDS

Large language models, automatic program repair, repository-level bugs, context, static analysis

ACM Reference Format:
Yuxiao Chen, Jingzheng Wu ✉, Xiang Ling ✉, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. 2024. When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done?. In *Proceedings of 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE-Companion '24).* ACM, Lisbon, POR, 13 pages. https://doi.org/10.1145/3639478.3647633

1 INTRODUCTION

Automatic program repair (APR) is an important challenge in software engineering, helping programmers significantly reduce debugging costs. Many researchers have explored various methods for APR, including pattern-based methods like TBar [19], SketchFix [10] and ErrDoc [31], and deep learning-based methods like CoCoNuT [23] and CURE [12]. Recently, the excellent generation ability of LLMs has brought new potential solutions for APR tasks.

✉ Jingzheng Wu and Xiang Ling are the corresponding authors.

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04
https://doi.org/10.1145/3639478.3647633

Many related studies have shown that LLMs are highly competitive in processing APR tasks [5, 11, 28, 29], even surpassing previously optimal methods.

However, the current evaluation of APR tasks using LLMs relies solely on the limited context of the single function or file where the bug is located. Bugs in programs can be categorized into two groups based on the size of the context they depend on for repair: function-level and repository-level [3]. For function-level bugs, the correct repair requires only providing the function where the error is located. However, due to widespread modular programming in software engineering [27, 30], there are often complex interactions or dependencies between multiple code files. This relationship can easily result in repository-level bugs, such as interface inconsistency, incorrect error handling, global variable abuse, race conditions [24], and more. Such repair tasks often require providing a broader repository context for repair tools. The performance of LLMs remains underexplored for repository-level repair tasks.

Our work in this paper aims to explore the performance of current popular LLMs in addressing this issue. However, the most pressing challenge is the lack of a suitable dataset. Existing datasets are either not built at repository level, such as QuixBugs [18], or cannot accurately restore scenarios of repository-level bugs, such as Defects4 [13]. Furthermore, datasets created too early may pose a potential risk of data leakage if used as training data for LLMs. To address this challenge, we propose a new benchmark called RepoBugs, specifically designed for evaluating repository-level APR tasks. It is built on popular open-source repositories from GitHub and contains 124 typical repository-level bugs.

We adopt ChatGPT from current popular LLMs for preliminary experiments. The experimental results show that if the preliminary method only use the function where the error was located as context, the repair rate of repairing on RepoBugs was only 22.58%. This result was significantly different from the repair rate of ChatGPT on function-level bugs evaluated in other related studies [5, 11, 28, 29]. Figure 1 shows a simple example. When the preliminary method only provides function-level context, ChatGPT can not perform the repair correctly. However, when we provide the complete repository as context, ChatGPT provides the correct repair result. This indicates that providing repository-level context is helpful when dealing with repository-level bugs in LLMs. Figure 1 illustrates a small example. When the repository size is small, we can employ a straightforward approach by considering the entire repository as the context. However, the size of the repository can be very large, and the input prompt for LLMs has an upper limit, such as a maximum token limit of 4,096 for ChatGPT. In addition, not all code in the repository is useful for the current repair task. Most code may be redundant information that interferes with the attention of the model. Therefore, it is necessary to provide precise context for the repair task of LLMs. In the field of code generation, extracting accurate context from code repositories is also an important challenge. Currently, many studies have provided solutions to this problem [7, 22, 35]. The methods used in these studies are roughly similar, all of which first segment the repository into slices and then obtain the most relevant fragments as context based on comparative similarity. We call this method the slice-similarity method. However, the code segments obtained through this method relying on similarity are different from the code segments that programmers refer to when making corrections to errors. Therefore, we believe that the code segments acquired through the slice-similarity method may not be well-suited for APR tasks. We also demonstrate this in subsequent experiments.

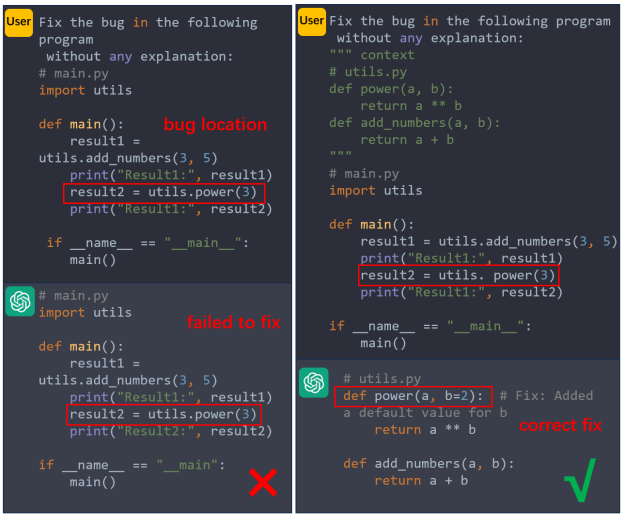

Figure 1: A simple example of using ChatGPT to handle repository-level bugs. The bug type is interface inconsistency. The number of parameter lists calling the power function is inconsistent with the function definition in *utils.py*. The left figure represents the reply to ChatGPT providing the function where the bug is located as context. The right figure represents the reply after adding the repository context.

In this paper, we propose a simple and universal repository-level context extraction method (RLCE) that can extract more precise context for repository-level code repair tasks. RLCE starts with the bug location and constructs specialized prompts for handling repository-level repair tasks for LLMs by parsing repository structures, filtering code fragments, and adding auxiliary information. We evaluate three mainstream LLMs separately, and experimental results show that compared to the preliminary method, the context provided by RLCE can significantly enhance the ability of LLMs to handle repository-level bugs, with a maximum improvement of 160%. In addition, we also conduct a comprehensive analysis of the effectiveness and limitations of RLCE, as well as the ability of LLMs to handle repository-level bugs, providing insights for future research. Our main contributions are summarized as follows:

- We initially investigate the performance of popular LLMs in addressing repository-level APR tasks. The success rates of the preliminary method in repairing errors for GPT3.5 and GPT4 are only 22.58% and 41.13%, respectively.
- We introduce a new benchmark, RepoBugs, which is built on popular open-source repositories from GitHub and contains 124 typical repository-level bugs. To the best of our knowledge, this is the first benchmark specifically designed for repository-level program repair.

- The repair rate of the preliminary method is unsatisfactory, and the length of the repository often exceeds the prompt length limit of LLMs. To address these problems, we propose a simple and universal method RLCE, which provides more precise context for APR tasks and achieves over 100% improvement in repair rates on all experimental models compared to the preliminary method.

2 BENCHMARK CONSTRUCTION

To effectively evaluate the performance of LLMs on repository-level APR tasks, the required dataset should meet the following requirements:

- Each bug needs to be based on a repository context environment;
- Bug fixing requires utilizing repository-level context;
- The creation time of the repository is later than the collection time of current popular LLMs training data.

Moreover, conventional approaches to dataset construction in the realm of computer science, such as automatic code disruption or crawling open-source repositories, encounter challenges in accurately filtering bug types and ensuring adherence to stringent dataset quality requirements. In light of these challenges, we present a novel benchmark dataset named RepoBugs. This dataset is derived from crawled open-source Python repositories and crafted through expert manual disruption. RepoBugs comprises 124 repository-level bugs specifically addressing interface inconsistency types. Although the selection of error types is circumscribed, it is highly representative, given that interface inconsistency errors constitute the most prevalent repository-level issues [37]. Subsequently, we will provide a detailed exposition of the methods employed for open-source repository collection and code disruption.

2.1 Dataset Collection

We collect repositories from open-source projects on GitHub. To minimize the risk of leakage due to the repositories being used as training data for LLMs, we set the search date condition to later than October 1, 2021. Additionally, we increase the number of crawled repositories. In this paper, we primarily focus on conceptual validation using the Python programming language. Consequently, we constrain the repositories considered in our work to those utilizing the Python language. Simultaneously, we ensure that each size of the repository does not exceed 1MB and that it has a minimum of 2,000 stars. We also filter out repositories with fewer than 4 Python files to adequately capture cross-file characteristics of repository-level bugs. In the end, we filter and obtain 11 repositories that meet these criteria, and detailed information is presented in Table 1.

2.2 Dataset Generation

The disruptive aspects of RepoBugs are manually crafted by experts with extensive programming experience. We designate the function where a call occurs as the main function and the called function as the context function, based on the characteristics of errors related to interface inconsistency. Based on existing research concerning program bugs at the repository scale [8, 16, 36], we design six typical disruption rules for main and context functions, as follows:

Index	Repo	Date	File	Sample
1	developer	2023/5/13	13	10
2	tiktoken	2022/12/1	16	11
3	gpt-migrate	2023/6/24	18	10
4	starcoder	2023/4/24	7	12
5	shell_gpt	2023/1/18	16	10
6	consistency_models	2023/2/26	23	12
7	musiclm-pytorch	2023/1/30	4	13
8	MAE-pytorch	2021/11/1	13	12
9	poe-api	2023/5/10	12	12
10	ijepa	2023/6/13	15	12
11	CommandlineConfig	2022/9/19	4	10

Table 1: Repository information in the RepoBugs dataset. *Index* represents the repository number. *Repo* represents the name of the repository. *Date* represents the date when the repository was created. *File* represents the total number of Python source files. *Sample* represents the number of test samples extracted from each repository.

- **NRV:** Inconsistency in the number of return values between the context function and the main function.
- **NP:** Inconsistency in the number of input parameters between the main function and the context function.
- **ORV:** Inconsistency in the order of return parameters between the main function and the context function.
- **OP:** Inconsistency in the order of input parameters between the main function and the context function.
- **CRV:** Inconsistency in the return from the context function and the requirements of the main function.
- **CP:** Inconsistency in the input parameters between the main function and the requirements of the context function.

During the destruction process, it is important to ensure all disruptions involve interaction between two or more functions in the repository and do not introduce syntax errors that cannot pass a Python interpreter. To simplify the process, all disruptions are completed within a single line. As a result, we have obtained a total of 124 bugs in RepoBugs.

3 PROPOSED FRAMEWORK

3.1 Overall Framework

Using LLMs to complete APR tasks at the repository level can be seen as a generation problem: the repaired code F' for F is generated based on the function F where the error is located and the context C at the repository level, which can be described as $F' = M(C, F)$, where M represents the large language model used. In this step, it is crucial to obtain the repository context C, and our goal is to provide more accurate context C for repair tasks and generate prompts for LLMs. Figure 2 shows the overview of our repository-level context extraction method (RLCE). Specifically, we first use a context retriever to retrieve repository code fragments related to repair tasks (Section 3.2). Subsequently, a comprehensive large language model prompt is generated by incorporating additional information, such as appended summaries and slices (Section 3.3).

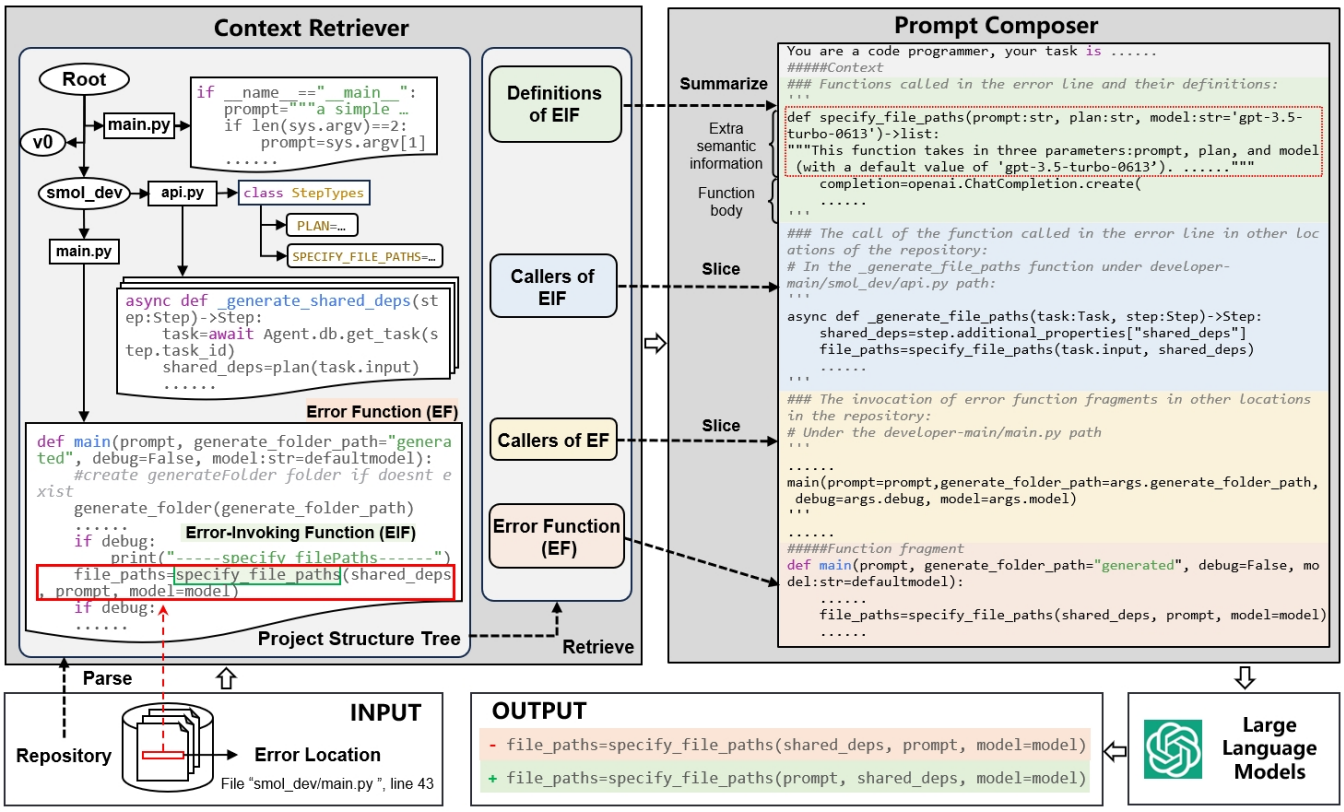

Figure 2: The overview of our repository-level context extraction method (RLCE).

3.2 Context Retriever

The core of RLCE lies in addressing the questions of where to retrieve from the repository and what kind of context to obtain. To achieve this objective, we design and implement a context retriever, which is a static code analysis tool capable of automatically parsing the repository into code segments based on its structure. It retrieves segments relevant to the repair task based on error localization information. Given that repository projects often have complex structures due to dependencies among files, our tool needs to possess the capability to analyze the structure of the repository for a clearer understanding of the relationships between its components. The overall structure of the context retriever is illustrated in Figure 2, consisting primarily of two key steps: (1) parsing repository files and constructing the project structure tree and (2) conducting retrieval in the project structure tree based on error location to obtain the required code segments.

Build project structure tree: During the construction of the project structure tree, our primary focus revolves around five types of entity nodes, namely: directories, files, classes, functions, and global variables. The connections between these nodes adhere to the original structural relationships within the repository. For example, the partial project structure tree of the *developer* repository is illustrated in Figure 2. Specifically, a project structure tree originates from a root node, with its child nodes encompassing subdirectories and files under the root directory of the repository. The child nodes of file entities include globally defined variables, classes, and functions. The leaf nodes of the project structure tree are restricted to function nodes or variable nodes, encompassing the code where functions or variables are defined. In addition to structural information, for the sake of facilitating subsequent retrieval processes, if a file calls functions, variables, or classes defined in another file, markers need to be placed on the file node for reference.

Retrieve code segments: As illustrated in Figure 2, the errors for the targeted code fix task are localized within one or several lines of code, referred to as the "error location". Before retrieval, the context retriever tool needs to analyze and extract the functions and global variables called within the error location, which we term Error-Invoking Functions (EIF). Subsequently, we define four types of context sources to determine where the retriever should extract code segments from the project structure tree as part of the context:

- **Definitions of EIF:** Retrieve code segments containing the definitions of the extracted Error-Invoking Functions within the repository scope.
- **Callers of EIF:** Search other occurrences of the Error-Invoking Function within the repository (excluding the error location) to obtain code segments containing their calling locations.
- **Error Function (EF):** The function containing the error location.

- **Callers of EF:** Examine if the Error Function is called elsewhere in the repository, and if so, retrieve code segments containing the calling locations.

A null value will be returned During the retrieval process if a particular context source is absent. Formally, the context retriever constructs a project structure tree T for the repository and gathers a collection of code segments $C_{repo} = R(EL, T)$, where C_{repo} encompasses all code segments from the four context sources.

3.3 Prompt Composer

The primary function of the prompt composer is to further process the code segments obtained by the context retriever and merge them with templates from different prompt strategies to generate the final prompt for the large language model. This can be represented as $C = P(C_{repo})$, where C represents the repository-level context in the final prompt, and P denotes the process of handling the collection of code segments. As depicted in Figure 2, for each context source, our processing approach is as follows:

Definitions of EIF: For this part, we attach extra semantic information to EIF to enhance the model's understanding of the function purposes and parameter meanings. Semantic information comprises two components: function signature and function summary. The function signature includes the function name, the type of each parameter in the parameter list, and the type of return value. The function summary provides an overview of the main functionality. Any model capable of generating code summaries and signatures can be used. For the sake of simplicity and leveraging the outstanding performance of LLMs in code summarization, our experiments choose the LLMs as the generation model.

Callers of EIF: For this part, we employ a slicing approach for processing. Since the error location contains calls to EIF, calls to EIF in other locations within the repository are likely to have valuable references for error correction. Therefore, the most useful information in code segments from Callers of EIF for the repair task is primarily concentrated around the statements where EIF are invoked. To minimize the introduction of excessive redundant information, we adopt a slicing approach, preserving the content of the statements before and after the invocation, each with a context window of five lines.

Callers of EF: For Callers of EF, providing useful contextual information about the Error Function may contribute to a better understanding of LLMs. Therefore, similar to Callers of EIF, the approach for this section also employs the same slicing method.

Error Function (EF): In the context of Error Function, no additional processing has been applied; instead, they are incorporated directly into the prompt context.

It is worth noting that, in this paper, we primarily focus on conducting our experiments using programs written in the Python language. However, the design principles of the context retriever can be extended to other programming languages. Examples of our method of repairing bugs in the Java programming language can be found in Appendix A.2.2.

4 EVALUATION SETUP

4.1 Models

We select three representative models from the currently most popular LLMs for our experiments.

GPT3.5 [26] is developed by OpenAI. It has strong language comprehension and generation capabilities through large-scale data training. The model we select in our experiment is GPT-3.5 Turbo, which supports a maximum token sequence length of 4K. The training data is up to September 2021.

PaLM2 [1] is a new generation big language model launched by Google. It has advanced reasoning ability and natural language generation ability. In our experiment, we select the text bison-001 model, which supports a maximum of 8K tokens input and 1K output, with a knowledge cutoff date of mid-2021.

GPT4 [25] stands as one of the most powerful LLMs currently released by OpenAI. In handling intricate tasks, GPT-4 exhibits greater reliability and creativity. For our experiments, we select the gpt-4-0613 model, which supports a maximum token length of 8K and was trained with data up to September 2021.

As these models are not open-source, we obtain responses through their APIs.

4.2 Prompt Generation

Due to the significant impact of prompt engineering on the performance of LLMs [21, 34], to fully evaluate their performance, we adopt the following three most commonly used prompt strategies in various tasks (for simplicity, specific prompt designs can be found in the Appendix A.1), including zero-shot, one-shot, and chain of thought (CoT).

Zero-shot [4]: This strategy does not provide any example of the model during the inference process, only natural language instructions describing the task. This method can minimize the limited prompt length and provide more contextual capacity for repair tasks, but it also faces difficulties in understanding task formats and other issues. We use two types of instructions, *Simple* and *Detail*, in the experiment. The Simple format instruction describes the task in an extremely concise language, while the Detail format is more specific, requiring the large language model to assume that it is a programmer completing a task of fixing bugs from the repository using context.

One-shot [4]: This strategy is similar to zero-shot but allows for an example other than natural language instructions that describe the task. We use the same instruction as the Detail method in the zero-shot strategy in the experiment and add a complete repair example.

CoT [32]: Previous studies have shown that the CoT strategy can significantly enhance the reasoning ability of LLMs. The process of automatically fixing bugs can be seen as an inference process. To investigate the efficacy of the CoT strategy in the field of repository-level APR, we propose a straightforward zero-shot-CoT [15] approach in this paper. This approach decomposes the repair task into three distinct logical steps: first, identifying the root cause of errors by integrating contextual information; second, devising targeted solutions based on the identified error causes; and finally, generating the comprehensive repaired code. Our instructional prompt guides the model to systematically engage in these three steps, providing

explicit error explanations, repair strategies, and the resultant fixed code, respectively.

4.3 Compared Repair Baselines

Preliminary method: A key contribution of our RLCE is the extraction of more precise repository-level context for repair tasks. To demonstrate the effectiveness of our method, we implement a preliminary method as the baseline that only provides the function itself where the error is located as the context. This simulates the existing preliminary method that leverages LLMs to address function-level APR tasks.

Slice-similarity method: To explore whether the slice-similarity method applies to the field of APR, our experiment reproduced the Retrieval Model used in the RepoCoder [35] method. Specifically, we set the slicing window to 10, use the sparse word bag model as the vector representation model, and use the Jaccard algorithm to calculate the similarity between the segment where the error is located and other segment vectors in the repository. Finally, 5 segments with the highest similarity are selected as the context.

4.4 Evaluation Metrics

Due to the high cost of running the repository and designing test cases, as well as the diversity of bug-fixing solutions, effective fixing accuracy cannot be achieved through precise matching (example can be found in the Appendix A.2.1). Therefore, to ensure the accuracy of the evaluation results, we ultimately adopt a manual evaluation method, and the evaluation results were provided by two experts who have more than 5 years of experience in Python programming. We will divide the evaluation indicators into four items to fully evaluate the return results of the large model. The specific evaluation criteria are as follows. When evaluating, if it meets the criteria, it is marked as 1; otherwise, it is marked as 0:

- **Related reply**: The return result is not empty and is related to the repair task in the instruction.
- **Correct format**: The returned result is in the expected format and the content is complete without any duplicate or redundant content.
- **Correct repair**: The returned result contains the correct fix for the error.
- **Correct explanation**: This item is specialized for the CoT prompt strategy, and the criteria are that the returned result includes a correct explanation of the cause of the error.

In the specific evaluation process, the two experts first have a detailed discussion on the evaluation criteria to ensure both reach a consistent understanding. Then, the two experts independently evaluated all the experimental results. Subsequently, the evaluation results of the two individuals are compared, and any minor discrepancies were addressed through further discussion and reassessment by both experts. This iterative process ensured the attainment of a unanimous final result.

5 RESULTS AND ANALYSIS

5.1 RLCE Outperforms the Baselines

We present the results of our experiments in Table 2, where each cell represents the proportion of samples passing the corresponding evaluation metric among all samples. Our RLCE method exhibits a significant improvement compared to the other two baselines. Regarding the relevance of model responses and the correctness of format, all methods and models generally generate responses relevant to the questions and conform to the expected format. In terms of repair rate, we observe that all models perform poorly when employing the preliminary method. For instance, even the superior GPT4 achieves only a 41.43% success rate. This indicates that even state-of-the-art LLMs struggle to accomplish repository-level repair tasks with only limited context at the function level, as they fail to provide sufficient information for repair tasks. After using the RLCE method to provide repository-level context, the repair rate improvement compared to the preliminary method has reached over 100%, with the GPT3.5 reaching the highest of 160%. This underscores the necessity of repository-level context for such code repair tasks.

Additionally, across the experiments with the three models, the repair rate of the slice-similarity method does not surpass our RLCE method. For example, in the GPT4 model, the respective improvement rates compared to the preliminary method are 20% and 100%. This suggests that the enhancement provided by the context of this method is limited. The slice-similarity method exploits code repetition in the repository, retrieving code segments similar to the error location to aid in error repair. However, relying solely on similar code makes it challenging to reconstruct the actual execution-time context before and after the error location, leading LLMs to struggle in correctly inferring the reasons for errors.

5.2 Impact of Prompt Strategies on RLCE

It is worth noting that the performance of the RLCE method is influenced by the use of different prompt strategies. As shown in Table 2, a comparison of different models reveals the most pronounced performance fluctuation in GPT3.5 due to the prompting strategy. Specifically, the one-shot strategy yield a repair rate improvement of over 37% compared to the Simple method with the zero-shot strategy. In practical repair tasks, it is desirable for the model to exhibit low sensitivity to prompting strategies. Because, in situations with similar contexts, stable responses of the model to various prompting strategies can reduce the uncertainty of repair outcomes and mitigate the exploration costs associated with finding appropriate prompting strategies. In summary, considering both response stability and accuracy, GPT4 demonstrates optimal performance.

5.3 Correct Explanation is Important for CoT

Analyzing the data in Table 2, we find that the repair rate of all models employing the CoT strategy did not meet our expectations, with a notable decrease in performance even on PaLM2. The GPT4, which exhibited the best performance overall, also experienced a slight decline in repair accuracy after adopting the CoT strategy. To investigate the reasons behind this, we categorize and statistically analyze the repair outcomes of the three models using both the Detail and CoT methods, along with the explanations generated by CoT. The results are presented in Figure 3. It is evident from the data that a significant correlation exists between the repair and explanation aspects for all models. In cases where the CoT

Model	Metric	Method					
		Preliminary	Slice-similarity	RLCE			
				Simple	Detail	One-shot	CoT
GPT3.5	Related reply	**1**	**1**	**1**	**1**	**1**	0.9919
	Correct format	**0.9597**	0.9516	0.9274	0.9516	**0.9597**	**0.9597**
	Correct repair	0.2258	0.3387	0.4113	0.5645	**0.5968**	0.5161
	Correct explanation	-	-	-	-	-	**0.5284**
PaLM2	Related reply	**0.8871**	0.8468	0.8387	0.8629	0.871	0.879
	Correct format	**0.8548**	0.7903	0.7984	0.8064	0.8387	0.5242
	Correct repair	0.2177	0.2419	**0.4272**	0.3952	0.4032	0.2742
	Correct explanation	-	-	-	-	-	**0.1774**
GPT4	Related reply	0.9677	**1**	0.9919	0.9919	0.9839	**1**
	Correct format	0.9677	0.9758	0.9839	0.9758	0.9839	**1**
	Correct repair	0.4113	0.4919	0.7742	0.7581	**0.8145**	0.75
	Correct explanation	-	-	-	-	-	**0.7742**

Table 2: Evaluation results of different models and methods on RepoBugs. The values in the cells represent the proportion of samples that passed the corresponding evaluation metrics out of the total number of samples. The cell data corresponding to the best-performing method in each row is bolded. Detailed descriptions of the *Preliminary* and *Slice-similarity* methods can be found in Section 4.3, both employing the one-shot prompting strategy.

method was successfully repaired (clusters denoted by the second uppercase letter T), the proportion of correct explanations was exceptionally high. Moreover, as the inference capabilities of the models improved, this trend became more pronounced. The statistical results for GPT4 indicate that all successfully repaired cases also had accurate explanations. When observing the three sets of TF clusters, corresponding to cases where the Detail method successfully repaired while the CoT method did not, the proportion of incorrect explanations was significantly higher than that of correct explanations. From the analysis above, we hypothesize that in the CoT method, incorrect interpretations can introduce significant interference, leading to erroneous outcomes. Using models with enhanced inference capabilities or guiding models to generate more accurate explanations may potentially improve the repair rate.

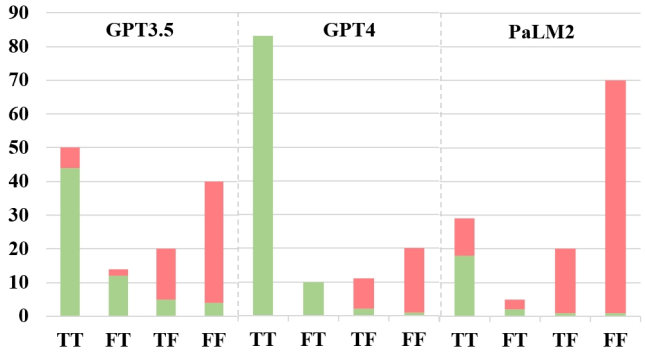

Figure 3: Statistical categorization of repair outcomes using the detail and CoT methods for three models, along with CoT-generated explanations. The horizontal axis comprises four categories, each composed of two uppercase letters, T or F, representing whether the repair results using the Detail and CoT methods are correct (e.g., FT denotes all cases where Detail repair is incorrect and CoT repair is correct). The vertical axis represents the number of cases for each category. In each category, red indicates instances where the CoT method provided incorrect explanations, while green signifies correct explanations.

5.4 Validity of Context Sources

Our research results indicate that the RLCE method can significantly enhance repository-level program repair tasks. To explore the correlation between the four context sources mentioned in Section 3.2 and extra semantic information with code repair performance, we conduct a set of ablation experiments. In these experiments, we remove three context sources and extra semantic information, excluding EF due to the inclusion of error localization. The results of the experiments can be found in Table 3. The models selected for the experiments are GPT3.5 and GPT4, both showing better overall performance, with all strategies adopting the one-shot approach.

From the results in Table 3, it is evident that both models perform well in terms of response format and relevance. In terms of accuracy, the experimental groups utilizing the complete context sources achieve the highest repair accuracy. However, the repair rate of different context sources on the repair results varies. If the context source of definitions of EIF is lacking, both models exhibit a significant decrease in repair rates, with GPT3.5 even dropping by more than half. This indicates that the definitions of EIF context source provide the most important information for the repair task. Moreover, the extra semantic information does not noticeably enhance the success rate. It suggests that the useful information extracted from it might be limited for LLMs.

Model	Context Source				Evaluation		
	Summarize	Callers of EF	Definitions of EIF	Callers of EIF	Related reply	Correct format	Correct repair
GPT3.5	✓	✓	✓	×	1	0.9597	0.5806
	-	✓	×	✓	1	0.9355	0.2742
	✓	×	✓	✓	1	0.9597	0.5565
	×	✓	✓	✓	1	0.9597	0.5403
	✓	✓	✓	✓	1	0.9597	**0.5968**
GPT4	✓	✓	✓	×	0.9839	0.9839	0.7742
	-	✓	×	✓	0.9919	0.9677	0.5000
	✓	×	✓	✓	0.9839	0.9839	0.7661
	×	✓	✓	✓	0.9839	0.9839	0.7823
	✓	✓	✓	✓	0.9839	0.9839	**0.8145**

Table 3: Comparison of the effects of different context sources in the Context.

5.5 Error Types Affect Repair Effectiveness

In Section 2.2, we present six distinct disruption rules. Errors arising from these diverse disruption rules often necessitate varying analytical approaches of LLMs to repair. To investigate the differences in the repair rate of LLMs when confronted with different error types, we categorize the results of distinct prompt strategies based on error categories and compute the correction accuracy of GPT-3.5 and GPT-4 model, as depicted in Figure 4.

Analysis of Figure 4 reveals a consistent trend in the performance of both models when addressing different error types. Both models exhibit poorer performance when faced with ORV and CRV error types. Upon examining instances of these two error types, we observe that, compared to other categories, these errors are more subtle and challenging to repair, mainly in two aspects. Firstly, ORV and CRV errors involve parameter mismatch or incorrect order between interfaces, requiring a deep understanding of the specific meaning of each parameter and the functionality of the function. This requires a deep semantic understanding. In contrast, the mismatch in the number of parameters can often be identified through simple syntax checks. Secondly, in real-world repositories, variable names in most cases align between parameters and arguments (We generally use the word 'parameter' for a variable named in the parenthesized list in a function definition, and 'argument' for the value used in a call of the function [14]), enabling the large language model to identify some obvious errors by comparing the parameter list in the function definition position with the argument list in the invocation position. In contrast, inconsistencies in the return values, as seen in ORV and CRV types, demand the large language model to comprehend the functionality of function and even the meaning of each variable during the parameter-passing process.

5.6 Long Prompt vs. Short Prompt

The prompt length of LLMs is subject to an upper limit. Does this imply that one should aim to incorporate as much information from the repository as possible within the confines of the prompt length? We classify the experimental results of GPT-3.5 and GPT-4 models using the one-shot method based on the length of the prompt. The statistical results are illustrated in Figure 5. It is observed that as the prompt length increases, both GPT3.5 and GPT4 exhibit a noticeable decline in repair accuracy. This reflects that the performance of LLMs varies when handling contexts of different scales. We posit that, for program repair tasks, a longer prompt may provide more information to the large language model, but simultaneously, it may also distract the model, increasing the difficulty of repair. Therefore, the longer length of the provided context does not lead to better results; instead, it should aim to provide precise context to enhance the density of useful information for the APR task.

6 THREATS TO VALIDITY

Internal Validity: In our experiments, we select three major LLMs (GPT3.5, GPT4, and PaLM2). However, different models usually differ in various aspects such as parameter count, training data, and fine-tuning methods. These differences may impact the repair rate of our method. For instance, if a large language model is inadequately trained on a specific programming language, its performance for that language may be suboptimal. Additionally, in our experiments, we standardize the temperature parameter across all LLMs to 0, ensuring stable outputs when facing the same prompt. Increasing the temperature parameter leads to unstable model outputs, potentially influencing the final results.

External Validity: The most significant factor influencing external validity is the choice of datasets. Firstly, if the creation time of repositories in the dataset predates the cutoff time of the training data, there is a high probability that the repository has already been learned as part of the training data, potentially impacting the final results. Secondly, an increase in the scale and complexity of repositories may escalate the cost and difficulty of context retrieval, affecting the reliability of the results. Additionally, in our experiments, we primarily focus on the error type of interface inconsistency, as it is common at the repository level and can typify the language model's ability to handle repository-level errors. However, given that RepoBugs is the only dataset known to us specifically designed for repository-level error repair, the generalization of our approach to datasets involving other error types may vary.

7 RELATED WORK

Automatic program repair (APR) is a crucial research problem in the field of software engineering, representing a significant research direction to reduce the cost of software maintenance and enhance software reliability. Since the introduction of an APR framework by Arcuri and Yao in 2008 [2], the field of APR has undergone rapid development. Early approaches to APR predominantly relied on traditional methods; for instance, GenProg [33] employed an extended form of genetic programming to evolve program variants and validated the effectiveness of repairs through test cases. TBar [19] explored template-based APR methods, assessing the effectiveness of different repair patterns through experiments. LSRepair [20] addressed program errors by conducting real-time searches for repair components within existing code repositories.

In recent years, a substantial amount of research has shifted towards leveraging machine learning techniques, particularly deep learning, for program repair. Researchers generate repair solutions by learning from extensive repair data in code repositories. For example, DeepFix [9] utilized a multi-layer sequence-to-sequence

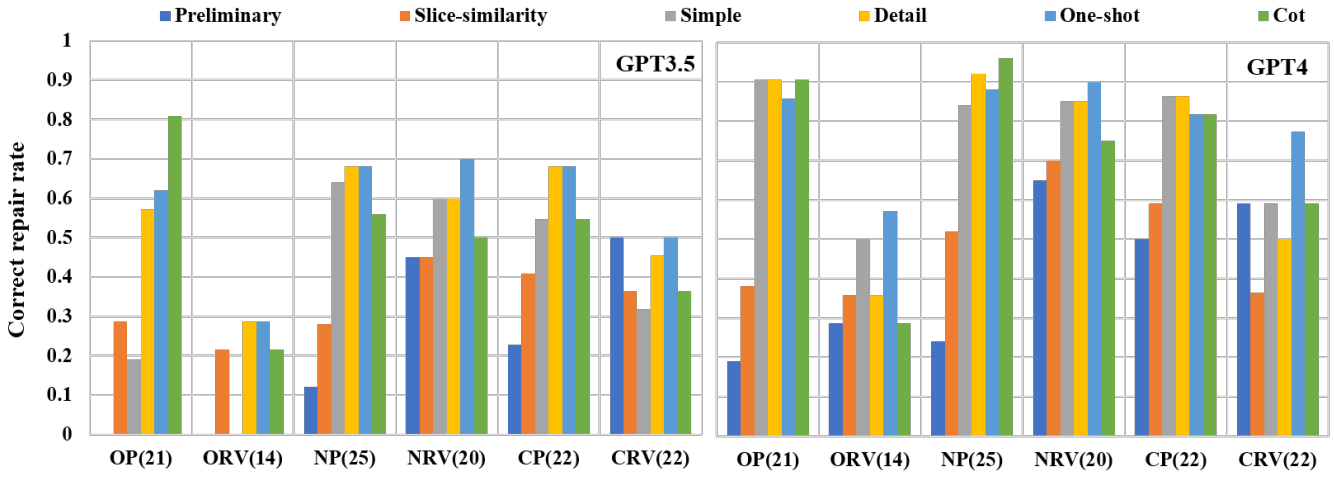

Figure 4: The results of various prompt strategies employed by GPT-3.5 and GPT-4 are statistically compiled in terms of correction accuracy based on error categories. The horizontal axis denotes the six categories of errors, with specific definitions provided in Section 2.2. The total number of samples corresponding to each category is enclosed in parentheses. Different colors of bars represent distinct methods, while the vertical axis represents the repair accuracy.

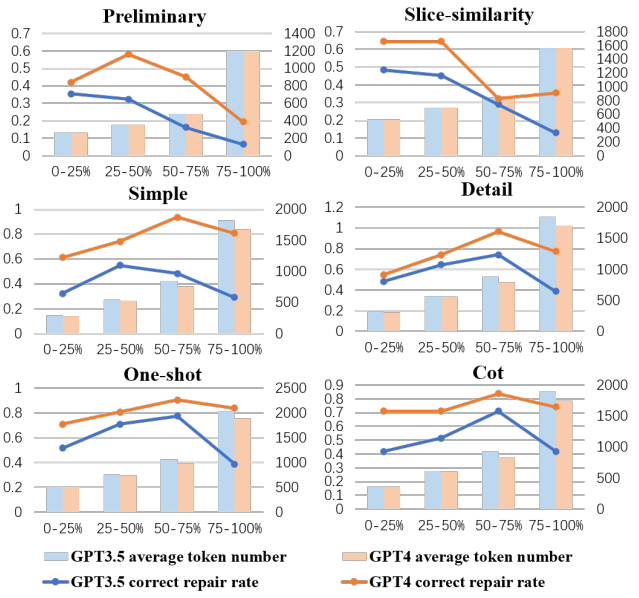

Figure 5: The relationship between different prompt lengths and repair accuracy is depicted through six subfigures. Each subfigure represents a distinct prompt strategy. All cases within each prompt strategy are evenly divided into four subsets based on prompt length (with a total dataset size of 124, resulting in 31 cases per subset). In each subfigure, bars represent the average length of tokens in the prompt, while the line graph illustrates the repair accuracy.

neural network to fix common programming errors without relying on external tools for locating or repairing. SequenceR [6] combined the encoder/decoder architecture with a copying mechanism to overcome the challenge of large vocabulary in source code. CURE [12] integrated pre-trained GPT models for programming languages with translation models, introducing a Context-aware Targeted Search strategy. SGEPR [17] uses a novel intermediate representation named sequence code property graph (SCPG) to model program semantic information. Recently, the remarkable comprehension and generation capabilities of LLMs have attracted widespread attention. Nan, Liu, and others [11] compared ten code language models and four deep learning APR techniques across four APR benchmarks. The experimental results demonstrated the competitive repair abilities of code language models.

8 CONCLUSION

In summary, we conduct a pioneering evaluation of the capabilities of major existing LLMs in handling repository-level repair tasks. We introduce a benchmark dataset, RepoBugs, along with a straightforward and versatile repository-level context extraction method, RLCE. RLCE, leveraging repository structure parsing and relevant context retrieval, offers more precise context for repository-level repair tasks. Experiments on the RepoBugs benchmark indicate that RLCE significantly enhances the performance of LLMs in repository-level program repair tasks. Furthermore, we conduct a detailed analysis of the experimental results from aspects such as context sources, error types, and prompt length, providing valuable insights for future research. RLCE has the potential to empower LLMs to offer efficient and accurate guidance for addressing errors encountered in actual development processes.

ACKNOWLEDGMENTS

This paper is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA0320401 and the National Natural Science Foundation of China under No. 62202457. This paper is supported by YuanTu Large Research Infrastructure.

A APPENDIX

A.1 Prompt design

We prepare Figure 6, illustrating the design structure of prompts in our experiments. From top to bottom, it includes the task instruction, example (one-shot method), context retrieved by the RLCE method, and the Error Function.

Prompt Structure

```
{instruction}
########Example
{example}
#####Context
{context}
#####Function fragment
{error function}
#####Response:
```

Figure 6: Prompt structure of our experiment

For each prompt strategy, the designed instructions are depicted in Figure 7.

A.2 Case Study

A.2.1 Diverse repair methods. In APR tasks, one bug often corresponds to multiple distinct repair possibilities, increasing the difficulty of assessing the accuracy of model-generated repairs. Figure 8 illustrates an example, the error involves a missing parameter in the parameters returned by the function. The original correct line in the repository utilized "_" to receive this parameter, and both the manual correction and the repair generated by the GPT3.5 model employed "mask_C", effectively repairing the error.

A.2.2 RLCE for other program languages. As described in Section 3.3, RLCE is a simple and versatile method that can be applied to different programming languages. As illustrated in Figure 9, in this example, the bug arises from an incorrect parameter order during a function call. GPT-4 successfully repairs the bug based on the repository-level context provided by our RLCE method.

Simple

```
# Please use context to fix the bug in the
'{bug_line}' line in the function fragment:
```

Detail

```
# You are a code programmer, your task is to fix
a bug in the '{bug_line}' line of a function
fragment from a code repository. Context is
potentially useful contextual information
provided to you, please make full use of it:
```

One-shot

```
# You are a code programmer, your task is to fix
a bug in a function fragment from a code
repository. Context is potentially useful
contextual information provided to you, please
make full use of it:
# Here is an example of using context to repair
a function fragment in another repository:
########Example
{{  #####Context
    # utils.py
    def square(a):
        return a ** 2
    def add_numbers(a, b):
        return a + b
    #####Function fragment
    def main():
        result1 = add_numbers(3, 5)
        print("Result1:", result1)
        result2 = square(3, 2)
        print("Result1:", result2)
    #####Response:
    'Fixed code':
    # main.py
    def main():
        result1 = add_numbers(3, 5)
        print("Result1:", result1)
        result2 = square(3)
        print("Result1:", result2)}}
# Here are the context and function fragment you
need to complete your task. The bug is located
in the {bug_line} line of the function fragment:
```

Chain of Thought

```
# You are a code programmer, your task is to fix
a bug in the '{bug_line}' line of a function
fragment from a code repository. You need to use
the information provided in the context as a
reference. Let's gradually solve this problem,
first start with the error line and analyze the
possible cause of the bug based on the function
fragment and context, then provide a solution to
the bug, and finally provide the fixed code:
```

Figure 7: Instructions for different prompt strategies

Repository Context

```
def _sample_block_mask(self, b_size: tuple,
acceptable_regions: list = None) -> tuple:
    ......
    return mask, mask_complement
```

Error Function

```
def __call__(self, batch):
    ......
    for _ in range(self.nenc):
        mask = self._sample_block_mask(e_size,
acceptable_regions=acceptable_regions)
        masks_e.append(mask)
        min_keep_enc = min(min_keep_enc,
len(mask))
    ......
```

Correct Line

```
mask, _ = self._sample_block_mask(e_size,
acceptable_regions=acceptable_regions)
```

√

LLM Return

```
mask, mask_C = self._sample_block_mask(e_size,
acceptable_regions=acceptable_regions)
```

√

Figure 8: Diversification of repair methods

Prompt

```
You are a code programmer, your task ......
////Context
/**The call of the function called in the error line
in other locations of the repository: */
    ......
    Partial newPartial = new Partial(iChronology,
newTypes, newValues);
    ......
/**Functions called in the error line and their
definitions:*/
public Partial(DateTimeFieldType[] types, int[]
values, Chronology chronology) {
        ......
    }
////Function fragment
public Partial without(DateTimeFieldType fieldType) {
    int index = indexOf(fieldType);
        ......
        System.arraycopy(iValues, index + 1,
newValues, index, newValues.length - index);
        Partial newPartial = new Partial(iChronology,
newTypes, newValues);
        iChronology.validate(newPartial, newValues);
        ......
}
////Response:
'Fixed code':
```

GPT4 Repair

```
public Partial without(DateTimeFieldType
fieldType) {
    ......
    Partial newPartial = new Partial(newTypes,
newValues, iChronology);
    ......
```

√

Figure 9: RLCE method for Java language program repair

REFERENCES

[1] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403* (2023).

[2] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. https://doi.org/10.1109/cec.2008.4630793

[3] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, et al. 2023. Codeplan: Repository-level coding using llms and planning. *arXiv preprint arXiv:2309.12499* (2023).

[4] 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.

[5] 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 preprint arXiv:2304.08191* (2023).

[6] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (Jan 2021), 1–1. https://doi.org/10.1109/tse.2019.2940179

[7] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. *arXiv preprint arXiv:2310.11248* (2023).

[8] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. 2019. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, Vol. 1. IEEE, 11–20.

[9] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2022. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* (Jun 2022). https://doi.org/10.1609/aaai.v31i1.10742

[10] 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.

[11] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[12] 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.

[13] 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.

[14] Brian W Kernighan and Dennis M Ritchie. 1988. The C programming language. *Prentice-Hall* (1988).

[15] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[16] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 241–252.

[17] Xiaoyu Li, Jingzheng Wu, Xiang Ling, Tianyue Luo, and Yanjun Wu. 2023. Automatic Program Repair via Learning Edits on Sequence Code Property Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE.

[18] 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.

[19] 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.

[20] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawende F. Bissyande. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. https://doi.org/10.1109/apsec.2018.00085

[21] 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.

[22] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[23] 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.

[24] Robert HB Netzer and Barton P Miller. 1992. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 1 (1992), 74–88.

[25] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[26] 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.

[27] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[28] Julian Aron Prenner and Romain Robbes. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922* (2021).

[29] 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).

[30] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. 2001. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes* 26, 5 (2001), 99–108.

[31] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 752–762.

[32] 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.

[33] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. https://doi.org/10.1109/icse.2009.5070536

[34] 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).

[35] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[36] Xin Zhang, Rongjie Yan, Jiwei Yan, Baoquan Cui, Jun Yan, and Jian Zhang. 2022. ExcePy: A Python Benchmark for Bugs with Python Built-in Types. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 856–866.

[37] Yangyang Zhao, Hareton Leung, Yibiao Yang, Yuming Zhou, and Baowen Xu. 2017. Towards an understanding of change types in bug fixing code. *Information and software technology* 86 (2017), 37–53.