Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs

Shengbei Jiang*
Beijing Jiaotong University
Beijing, China
21291232@bjtu.edu.cn

Jiabao Zhang*
Beijing Jiaotong University
Beijing, China
21281296@bjtu.edu.cn

Wei Chen*
Beijing Jiaotong University
Beijing, China
21281275@bjtu.edu.cn

Bo Wang†
Beijing Jiaotong University
Beijing, China
wangbo_cs@bjtu.edu.cn

Jianyi Zhou
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
zhoujianyi2@huawei.com

Jie M. Zhang
King's College London
London, United Kingdom
jie.zhang@kcl.ac.uk

ABSTRACT

Automated debugging is an emerging research field that aims to automatically find and repair bugs. In this field, Fault Localization (FL) and Automated Program Repair (APR) gain the most research efforts. Most recently, researchers have adopted pre-trained Large Language Models (LLMs) to facilitate FL and APR and their results are promising. However, the LLMs they used either vanished (such as Codex) or outdated (such as early versions of GPT). In this paper, we evaluate the performance of recent commercial closed-source general-purpose LLMs on FL and APR, i.e., ChatGPT 3.5, ERNIE Bot 3.5, and IFlytek Spark 2.0. We select three popular LLMs and evaluate them on 120 real-world Java bugs from the benchmark Defects4J. For FL and APR, we designed three kinds of prompts for each, considering different kinds of information. The results show that these LLMs could successfully locate 53.3% and correctly fix 12.5% of these bugs.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

Large Language Model, Fault Localization, Program Repair, Software Debugging

ACM Reference Format:
Shengbei Jiang, Jiabao Zhang, Wei Chen, Bo Wang, Jianyi Zhou, and Jie M. Zhang. 2024. Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs. In *Proceedings of The First International Workshop on Large Language Models for Code (LLM4Code Workshop '24).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

*These three authors contributed equally to this research.
†The corresponding author.

LLM4Code Workshop '24, Apr 20, 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Bugs are ubiquitous in modern software systems, threatening our daily lives. However, program debugging is time-consuming and challenging, consuming more than half of the developers' programming time [5]. Hence, a large body of research efforts have been dedicated to automated debugging techniques.

Among these debugging techniques, Fault Localization (FL) techniques and Automated Program Repair (APR) approaches are emerging fields and gaining traction. FL techniques aim to automatically localize buggy code elements (e.g., lines or methods) [9–11, 17], while APR techniques aim to automatically generate patches to fix buggy programs without human intervention [18–20, 23].

FL and APR techniques have adopted deep-learning models to facilitate the capture and comprehension of the context of the buggy code [10, 11, 17, 22], and deep-learning-based approaches have been recognized as the state of the art. However, their performance is still limited because their training data only contains buggy code extracted from development history.

Most recently, with the overwhelming success of Large Language Models (LLM), researchers have proposed to harness LLMs for FL and APR [3, 4, 7, 14, 16]. Currently, with ChatGPT becoming more and more popular, researchers in the field of debugging predominantly utilize various versions of the ChatGPT models. For example, Fan et al. employed the latest model of Codex in 2022, while Xia et al. employed the latest model of ChatGPT in 2023, (i.e., gpt-3.5-turbo) [16]. However, with the emergence of more and more general-purpose LLMs, the capabilities of other models in debugging have yet to be evaluated.

In this paper, we aim to evaluate the capabilities of existing closed-source general-purpose LLM in FL and APR. More specifically, we performed a preliminary case study on 120 real-world Java bugs from six projects of the benchmark Defects4J [6], by comparing three commercial closed-source LLMs, namely ChatGPT 3.5, ERNIE Bot 3.5, and IFlytek Spark 2.0. We designed three types of prompts by emphasizing different kinds of information, including source code context, failure traces, and test code context. In contrast to existing work, we have employed several other closed-source

LLMs apart from ChatGPT and considered transforming the stack traces of failure traces into prompts.

The evaluation results show that all three LLMs successfully locate more than half of all bugs (i.e., 64/120) and correctly fix 12.5% (15/120). ChatGPT 3.5 achieves the best performance and IFlytek Spark archives the second best. Contrary to our expectation that providing more information would enhance the performance of LLMs, surprisingly, the prompts that only code the source code of the buggy method achieve the best FL performance. The prompts that copy the buggy statement achieve the best APR performance.

2 STUDY DESIGN

Figure 1 illustrates the overview of our study. Given a bug from the Defects4J benchmark, we generate prompts by tailoring the predefined prompt template and the bug context information extracted from code and test failure messages. We designed different prompt templates for FL and APR, respectively. We send the prompts to the target LLMs and get their results. Note that if the length of the prompt exceeds the token limit of the LLM (e.g., ChatGPT 3.5 has a 4096 token limit), we truncate it. Then we manually compare the results with the developer patches of Defects4J, which are used as the oracle. Note that despite ChatGPT, few LLMs provide an API interface, which makes us engage in conversation with these models through their web pages.

All experiments were conducted on a Legion Y9000P PC, with 16GB memory and an Intel i9-13900H CPU.

2.1 Research Questions

In this study, we aim to investigate the following research questions:

- **RQ1: How do different LLMs perform for FL?**
- **RQ2: How do different LLMs perform for APR?**
- **RQ3: Do different sources of information help with localization and repair?**

2.2 Chosen Bugs

We directly the popular benchmark Defects4J v1.2.0 [6], which contains 395 real-world Java bugs from six open-source projects. Defects4J is considered to be the baseline benchmark in the field of FL [1, 8, 10, 11, 21] and APR [18–20, 23].

Note that some bugs of Defects4J may cover multiple methods or even multiple files, resulting in an inability to measure FL and APR performance in a single conversation with LLMs. So we skip these bugs and only keep the bugs whose patch only affects a single method. As a preliminary study, we pick the first 20 bugs of each project, and thus in total, we perform our study on 120 bugs.

2.3 Chosen Models

We use the following criteria for selecting LLMs in this study.

(1) The LLMs should be available to external users.
(2) The LLMs should support coding missions.
(3) The LLMs should be closed-source.

Currently, research on the performance of LLMs in software engineering tasks mainly focuses on open-source ones (e.g., CodeLlama) and ChatGPT. However, many novel commercial LLMs are continuously being proposed, and their code capabilities also need to be evaluated. At last, we select three LLMs, namely ChatGPT 3.5, ERNIE Bot 3.5, and IFlytek Spark 2.0. All these models are generative rather than infilling, but the parameter sizes are not officially released.

2.4 Fault Localization Experiment Settings

2.4.1 Prompt design. In our study, we directly use a zero-shot fashion single conversation to perform FL. To be more specific, we employ three types of information, namely source method code (i.e., `src`), bug-trigger assertion code (i.e., `assert`), and stack trace (i.e., `stack`). We combine them with different prompt templates as follows.

- **FL Prompt 1:** `src` + "There is a bug in the above code, please help me locate it."
- **FL Prompt 2:** `src` + `stack` + "There is a bug in the above code, please help me locate it by considering the stack trace."
- **FL Prompt 3:** `src` + `stack` + `assert` + "There is a bug in the above code, please help me locate it by considering the stack trace information and failure assertion code."

2.4.2 Evaluation metric. As mentioned before, the bugs used in our study are limited within a single method. Thus we judge the line returned by LLMs by checking it is in the lines that are affected by the corresponding developer patch, which is also a common practice in the FL community. For the bugs that have multiple hunks, we consider a bug to be successfully located if all the hunks are reported as buggy.

2.5 Automated-Program Repair Experiment Settings

2.5.1 Prompt design. In the APR experiment, we also use a zero-shot fashion single conversation. Different from our FL experiment, we only use `src` information. Additionally, to meet the generative task, we kept the code before the bug and the bug itself for LLM to fix. In our pilot study, we found that LLMs rarely correctly fix the bugs without perfect localization, so we give either the line number of the bug or directly use the buggy statement. We combine them with different prompt templates as follows, where `X` is the placeholder of the line number of the bug and `S` is the buggy statement literal.

- **APR Prompt 1:** `src` + "There is a bug in line `X` of the code, please help me fix it."
- **APR Prompt 2:** `src` + "There is a bug in `S`, please help me fix it."
- **APR Prompt 3:** `src` + "There is a bug in the last statement, please help me fix it."

2.5.2 Evaluation metric. Following existing APR common practice, we manually compare patches made by developers. If they match entirely, we consider them correct. If they don't match, we further assess whether they are semantically equivalent. Note that manually checking the semantic equivalence is a common practice in the field of APR. If the patches are semantically equal to the developers' patch, we mark it as correct, otherwise, we mark it as incorrect.

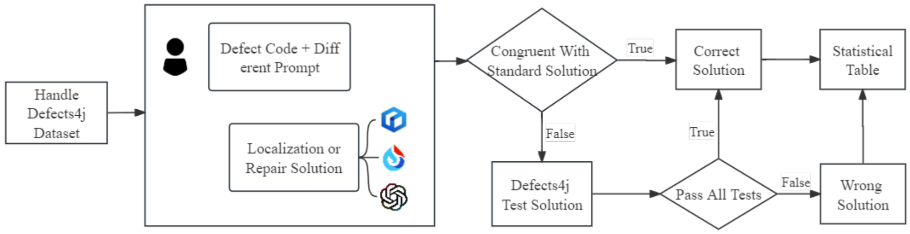

Figure 1: An overview of our study

Table 1: The number of successfully localized bugs.

Project	ChatGPT 3.5	ERNIE Bot 3.5	IFlytek Spark
Chart	11	8	9
Lang	7	1	0
Math	9	4	10
Time	6	1	3
Mockito	7	6	2
Closure	7	5	1
Total	47	25	25

Table 2: The number of successfully repaired bugs.

Project	ChatGPT 3.5	ERNIE Bot 3.5	IFlytek Spark
Chart	5	3	3
Lang	0	0	1
Math	1	1	0
Time	1	0	1
Mockito	0	1	1
Closure	2	1	2
Total	9	6	8

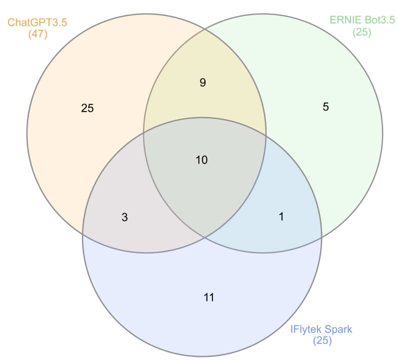

Figure 2: The intersection of successfully localized bugs of chosen LLMs.

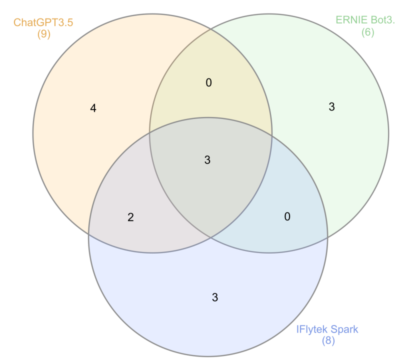

Figure 3: The intersection of successfully repaired bugs of chosen LLMs.

3 EVALUATION RESULTS

3.1 RQ1: FL Results

We merge all the results of three kinds of prompts, and Table 1 illustrates the FL performance of the LLMs. ChatGPT 3.5, which achieves the best performance, successfully locates 47 (i.e., about 39%) bugs with a single conversation. Figure 2 shows the intersection of successfully localized bugs of the three LLMs. In total, the three LLMs locate 64 out of the 120 bugs (about 53%). We can find that all three LLMs have a significant number of uniquely localized bugs, and ChatGPT 3.5 still has the most uniquely located bugs by locating 25 bugs that other LLMs could not.

3.2 RQ2: APR Results

Following the settings of the FL experiments, we merge all the repair results of three kinds of prompts. Table 2 shows the results, indicating that different from the performance of FL, there is no significant difference in the APR performance. All three LLMs only successfully fix less than 10% of the 120 bugs, and ChatGPT 3.5 achieves the best. Figure 3 shows the intersection of successfully fixed bugs of the three LLMs. In total, all three LLMs successfully fixed 15 out of the 120 bugs (i.e., 12.5%). Different from the Venn diagram of FL, there is no significant difference concerning the number of uniquely fixed bugs. In addition, the portion of the bugs that are fixed by all LLMs is large.

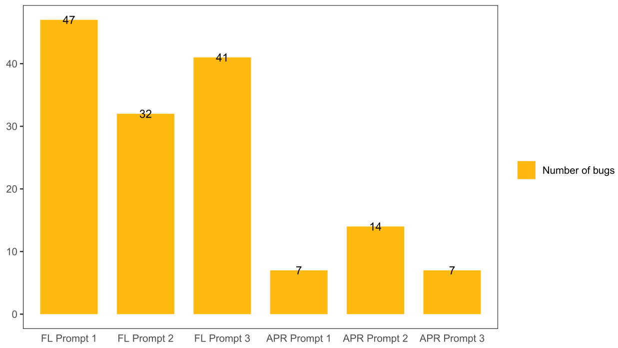

Figure 4: Comparison of different prompts.

3.3 RQ3: Performance of Different Sources of Information

Figure 4 shows the overall performance of all used prompts in FL and APR, respectively. We can find that in FL experiments, the **FL Prompt 1** achieves the best performance, though it uses the least kinds of information. In APR experiments, the **APR Prompt 2**, which copies the statement to be fixed in prompts, achieves the best performance.

4 RELATED WORK

LLMs have been widely studied in the field of automated debugging. Some LLM-based APR approaches employed open-sourced LLMs such as CodeT5 [13–15]. While the most LLM-based APR/FL approaches are built upon ChatGPT 3.5 [2, 12, 16]. Our paper investigates underutilized LLMs, serving as a complement to the evaluation of existing LLMs' capabilities.

5 CONCLUSION AND FUTURE WORK

In this paper, we perform a study to evaluate existing commercial closed-source LLMs on FL and APR. We respectively designed three kinds of prompts for the two tasks and performed a study on 120 bugs from Defects4J. The results show that existing LLMs could locate 53.3% and correctly fix 12.5% of the bugs.

In the future, we plan to first extend our study to all bugs of Defects4J. Then to reduce the threat of data leak, we will study their code capabilities on the datasets that are collected after the LLMs' training phase. In addition, concerning the performance of FL and APR, we plan to compare closed-source LLMs with open-source LLMs. At last, we plan to perform prompt engineering to find a more proper prompt.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable comments on this study. This work is supported by the National Natural Science Foundation of China under Grant No. 62202040.

REFERENCES

[1] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 907–918.

[2] 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).

[3] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).

[4] 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.

[5] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*. 1219–1219.

[6] 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.

[7] Sungmin Kang, Gabin An, and Shin Yoo. 2023. A Preliminary Evaluation of LLM-Based Fault Localization. *arXiv preprint arXiv:2308.05487* (2023).

[8] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 169–180.

[9] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[10] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[11] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*. 1169–1180.

[12] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276* (2023).

[13] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*.

[14] 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*.

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

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

[17] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. 2022. A universal data augmentation approach for fault localization. In *Proceedings of the 44th International Conference on Software Engineering*. 48–60.

[18] Yingfei Xiong and Bo Wang. 2022. L2S: A framework for synthesizing the most probable program under a specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–45.

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

[20] 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.

[21] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault localization via efficient probabilistic modeling of program semantics. In *Proceedings of the 44th International Conference on Software Engineering*. 958–969.

[22] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-Based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology* (2023).

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