Evaluating the Generalizability of LLMs in Automated Program Repair

Fengjie Li*, Jiajun Jiang*†, Jiajun Sun*, Hongyu Zhang‡
*College of Intelligence and and Computing, Tianjin University, Tianjin, China
‡School of Big Data and Software Engineering, Chongqing University, Chongqing, China
*{fengjie, jiangjiajun, sjjtianjin}@tju.edu.cn, ‡hyzhang@cqu.edu.cn

***Abstract*—LLM-based automated program repair methods have attracted significant attention for their state-of-the-art performance. However, they were primarily evaluated on a few well-known datasets like Defects4J, raising questions about their effectiveness on new datasets. In this study, we evaluate 11 top-performing LLMs on DEFECTS4J-TRANS, a new dataset derived from transforming Defects4J while maintaining the original semantics. Results from experiments on both Defects4J and DEFECTS4J-TRANS show that all studied LLMs have limited generalizability in APR tasks, with the average number of correct and plausible patches decreasing by 49.48% and 42.90%, respectively, on DEFECTS4J-TRANS. Further investigation into incorporating additional repair-relevant information in repair prompts reveals that, although this information significantly enhances the LLMs' capabilities (increasing the number of correct and plausible patches by up to 136.67% and 121.82%, respectively), performance still falls short of their original results. This indicates that prompt engineering alone is insufficient to substantially enhance LLMs' repair capabilities. Based on our study, we also offer several recommendations for future research.**

***Index Terms*—Program Repair, LLM, Generalizability of LLM**

I. INTRODUCTION

With the rapid growth of the scale and complexity of modern software systems, the number and intricacy of software bugs have also increased, resulting in significant financial losses for organizations and end-users. Fixing these bugs requires substantial consumption of time and effort from developers. As a result, Automated Program Repair (APR), which focuses on automatically fixing software bugs [1]–[6], has attracted considerable attention from academia and industry.

Recently, Large Language Models (LLMs) have demonstrated impressive performance across various software engineering (SE) tasks leading to the emergence of an increasing number of LLM-based APR methods [7]–[13]. Several existing works [7], [14], [15] have shown that LLMs, even without additional repair-relevant information, can outperform previous learning-based APR methods just using a few-shot prompting. Researchers have explored various approaches to enhance LLM-based APR by providing more repair-relevant information, such as fault localization [15], bug reports [10] and trigger test information [12]. Techniques such as Chain-of-Thought (CoT) prompting [16], multi-turn dialogues [17], and multi-agent systems [18] have been employed to better inform LLMs and improve their capabilities.

Although these LLM-based APR methods have achieved remarkable results, they have primarily only been evaluated on well-known datasets such as Defects4J [19] and QuixBugs [20], which were proposed years ago. Recent works [21]–[24] have highlighted the significant risk of memorization in LLMs when evaluated on these datasets, leading to our ***first research question (RQ1): How is the generalizability of LLMs?*** Specifically, can LLMs achieve the same impressive performance on another fresh new dataset? To answer this question, we create the DEFECTS4J-TRANS dataset by applying program transformations to Defects4J dataset, ensuring emantic equivalence while altering code content. Then, we assess the generalizability of LLMs by comparing their repair results on both Defects4J and DEFECTS4J-TRANS.

Results: The results show that the performance of LLMs significantly declined on DEFECTS4J-TRANS, with the number of correct and plausible patches decreasing by an average of **49.48%** and **42.90%**, respectively. This suggests unsatisfactory generalizability and underscores the need for more comprehensive evaluations of LLM-based APR methods.

This poor generalizability raises our ***second research question (RQ2): Can repair-relevant information enhance the generalizability of LLMs?*** To answer this question, we incorporate three types of repair-relevant information used in previous LLM-based APR methods [8], [10], [12], [13], to explore their potential to enhance the generalizability of LLMs.

Results: The results indicate that incorporating this information increased the number of correct and plausible patches by up to **136.67%** and **121.82%**, respectively. However, most LLMs still underperformed compared to their original results on Defects4J, highlighting the need to develop more effective LLM-based APR methods.

Contributions. To sum up, the contributions of this study are:

- We conducted the first extensive experiments to investigate the generalizability of 11 top-performing LLMs, using 4 types of prompts to explore the impact of different information. Our findings reveal that the selected LLMs exhibit unsatisfactory generalizability in APR tasks.
- We propose several future directions to enhance the generalizability of LLM-based methods. Our findings aim to guide the SE community in effectively utilizing LLMs for APR.

†Jiajun Jiang is the corresponding author for this work.

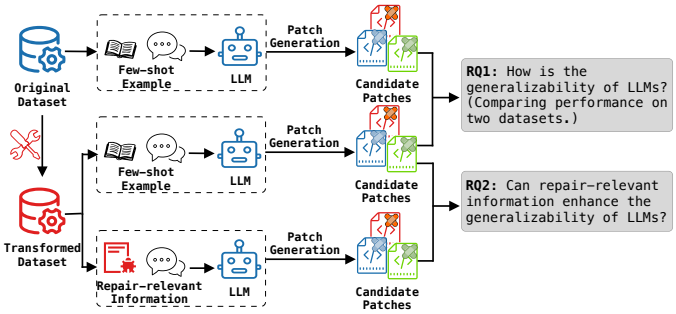

Fig. 1. Overview of study design

• We have open-sourced all code, data and results involved in our work [25] to promote replication and future research.

II. Related Work

We have not identified any studies dedicated to validating the generalizability of LLMs in the APR task in the literature, although some works aimed at addressing the risks associated with LLMs in SE tasks. For example, studies [21]–[24] have pointed out several potential issues (e.g., memorization and reproducibility) when using LLMs for SE tasks, but they lack thorough experimental validation. Additionally, new defect datasets have been proposed to address the memorization issues in LLMs, such as ConDefects [26], which collected recent bugs from the online competition platform, and HumanEval-Java [15], a mutation of the original HumanEval [27] dataset. However, these datasets consist mainly of simple algorithmic programs, where LLMs tend to perform well even when directly generating the corresponding functional code, limiting their ability to fully reflect the LLMs' coding capabilities. In contrast, Defects4J-Trans used in this study retains more characteristics of real-world projects, such as project-specific APIs, while effectively mitigating memorization issues in LLMs. Other works [28], [29] have also introduced evolving datasets for code generation tasks, which are different from our study.

III. Study Design

In this section, we detail the LLMs selected for evaluation, the construction of Defects4J-Trans, the utilized prompts, and the overall settings of the LLMs employed in this study. Figure 1 shows the overview of our study. First, we apply our code transformation tool to the Defects4J dataset to generate Defects4J-Trans, which maintains the original fault semantics but contains different code content. Then, we apply the LLMs on both Defects4J and Defects4J-Trans, comparing their performance differences to assess their generalizability. Finally, we investigate the impact of three types of repair-related information on the generalizability of LLMs.

A. Studied LLMs

Our selection of LLMs is based on the EvalPlus leaderboard [30], [31], which offers more rigorous tests for evaluating coding capabilities of LLMs. We selected LLMs that ranked in the top 20 on that leaderboard and capable of performing inference locally. Due to resource constraints, we excluded LLMs with parameter size exceeding 34B and those without open-source access. In total, we chose 11 different LLMs for our evaluation. These remaining LLMs still exhibit competitive coding capabilities. For instance, CodeQwen1.5 achieved a score of 73.8, securing the 4th position on the leaderboard. Table I provides details, including LLM name, number of parameters, and average pass@1 score on the EvalPlus benchmark. As shown, the parameter sizes of LLMs ranging from 6.7B to 34B, with scores varying from 66.5 to 73.8. Particularly, GPT-4-Turbo achieves a score of 77.5.

TABLE I
Summary of studied LLMs.

LLM	#Parameters ↑	#Average Score
WaveCoder-Ultra [32]	6.7B	66.5
DeepSeek-Coder-Instruct [33]	6.7B	68.4
OpenCodeInterpreter-DS [34]	6.7B	69.2
Magicoder-S-DS [35]	6.7B	70.2
Artigenz-Coder-DS [36]	6.7B	71.1
DeepSeek-Coder-Instruct-v1.5 [33]	7B	66.8
CodeQwen1.5 [37]	7B	73.8
StartChat2-v0.1 [38]	15B	67.9
OpenCodeInterpreter-DS [34]	33B	71.2
DeepSeek-Coder-Instruct [33]	33B	72.5
SpeechLess-CodeLlama-v2.0 [39]	34B	66.7

B. Construction of Defects4J-Trans

To ensure the semantic equivalence, we primarily performed equivalent transformations related to control flow statements in the code. Following existing code transformation works [40], [41], we designed five transformation operators and applied them by parsing and modifying abstract syntax trees (AST) using the Java Development Toolkit [42]. Due to space limits, we briefly introduce each transformation operator's functionality here. For detailed implementation, please refer to our homepage [25]:

- **T_1-Variable Renaming:** Replaces all variable names in the original code with the new names generated by StarCoder2-3B to maintain the naturalness of code.
- **T_2-Loop Transformation:** Transforms between `For-Loop` and `While-Loop` structures equivalently.
- **T_3-Switch Transformation:** Replaces `Switch Statement` with a series of `If Statement` by adding `break flag` and `fall-through flag` on demand to ensure consistent executing logic.
- **T_4-Dead Code Injection:** Inserts statements that will never be executed, such as `if (false) {...}`. To save tokens, we applied this operator once under each `Block Statement`, and injected at most three instances of dead code within a single function.
- **T_5-Boolean Transformation:** Double negating boolean predicates in `If Statements`. For example, `if (!(!(condition))) {...}`.

We sequentially apply the above five operators to all 438 single-function bugs in Defects4J (existing LLM-based APR

methods mainly focusing on repairing bugs within a single function [10], [12], [14]) to generate DEFECTS4J-TRANS. Semantic consistency between DEFECTS4J-TRANS and the original Defects4J was ensured through manual inspection and by running Defects4J's compilation and testing scripts.

```
// Provide a fix for the buggy function
{Buggy code and fixed code exmaples} or
{Repair-relevant Information}
// Provide a fix for the buggy function
// Buggy Function
{Buggy code want to fix}
//Fixed Function
```

Listing 1. The input prompt used in this study.

C. Prompt Engineering

Listing 1 shows the prompts used in this study. Following prior works [10], [14], we provide the entire buggy function as the input to the LLMs, which then generate the complete fixed function. This approach reflects a practical scenario and effectively assesses the coding capabilities of the LLMs. Specifically, the basic prompt template employs "`//Provide a fix for the buggy function`" to indicate the APR task and uses "`//Buggy Function`" and "`//Fixed Function`" to help the LLM identify buggy and fixed code. Additionally, for evaluating the impact of additional repair-relevant information, we designed three extended prompts, detailed as follows.

1) Two-shot: Using the aforementioned template, we provide two pairs of buggy and fixed code examples. One is a manually constructed toy example to help LLMs understand the APR task, and the other is the repair example with the shortest context from the same project, offering insight into the coding style. This serves as the default prompt in RQ1.

2) Two-shot$_{fl}$: This prompt further incorporates perfect line-level fault localization information into the aforementioned Two-shot prompt by manually annotating the buggy lines in the functions with `/*bug is here*/`.

3) Bug Report: This prompt replaces the two-shot examples with bug report information, and marks the report title and content with "`//Bug Report Title`" and "`//Bug Report Content`". In other words, we only provide the faulty function and the associated bug report to fit the input length limit of LLMs without presenting repair examples.

4) Trigger Test: Similarly, this prompt replace the repair examples with the trigger test and the corresponding error message, which are marked with "`//Trigger Test`" and "`//Error Message`", respectively.

D. General Settings

We used the HuggingFace Library [43] to load LLM weights and perform inference. Following Xia et al. [14], we set top-p to 0.95 and the temperature to 0.8. Each LLM was invoked 200 times with the respective prompt for each bug. A patch is considered *plausible* if it passes the test cases and *correct* if it is semantically equivalent to the developer's patch.

Our experiments were conducted on a local machine equipped with dual Intel Xeon Golden 6348 CPUs, 512GB RAM, and eight A800 GPUs, running Ubuntu 20.04.6LTS.

TABLE II
REPAIR PERFORMANCE OF DIFFERENT LLMS ON DEFECTS4J (D4J) AND DEFECTS4J-TRANS (D4J-T)

LLM	#Parameters	#Correct Fixes			#Plausible Fixes		
		#D4J	#D4J-T	Dec.(%)	#D4J	#D4J-T	Dec.(%)
WaveCoder-Ultra	6.7B	65	34	47.69↓	105	57	45.71↓
DeepSeek-Coder-Instruct	6.7B	63	36	42.86↓	114	73	35.96↓
OpenCodeInterpreter-DS	6.7B	56	27	51.79↓	91	51	43.96↓
Magicoder-S-DS	6.7B	77	38	50.65↓	120	67	44.17↓
Artigenz-Coder-DS	6.7B	75	43	42.67↓	111	71	36.04↓
CodeQwen1.5	7B	74	37	50.00↓	125	72	42.40↓
DeepSeek-Coder-Instruct-v1.5	7B	71	42	40.85↓	136	89	34.56↓
StarChat2-v0.1	15B	92	30	**67.39**↓	152	55	**63.82**↓
OpenCodeInterpreter-DS	33B	87	37	57.47↓	119	61	48.74↓
DeepSeek-Coder-Instruct	33B	94	45	52.13↓	133	76	42.86↓
Speechless-CodeLlama-v2.0	34B	107	66	38.32↓	160	108	32.50↓
Average		78.27	39.55	49.48↓	124.18	70.91	42.90↓

E. Research Questions

In this work, we study the following two questions:

- **RQ1:How is the generalizability of LLMs?** We evaluated the generalizability of 11 top-performing LLMs in program repair by comparing their repair capabilities on Defects4J and DEFECTS4J-TRANS accordingly.
- **RQ2:Can repair-relevant information enhance the generalizability of LLMs?** Based on RQ1, we selected 4 LLMs that exhibited the weakest generalizability in this RQ to optimize time efficiency. We empirically analyzed whether incorporating repair-relevant information could enhance their repair performance.

IV. PRELIMINARY EVALUATION

A. RQ1: Generalizability

Table II shows the number of correct and plausible patches generated by 11 LLMs using the Two-shot prompt, as described in Section III-C, on both Defects4J and DEFECTS4J-TRANS. We observe a decline in both correct and plausible repairs across all selected LLMs. Notably, the StarChat2 [38] experienced the largest drop, with correct and plausible repairs decreasing by 67.39% and 63.82%, respectively. The smallest decline was seen in SpeechLess-CodeLlama-v2.0 [39], with decreases of 38.32% and 32.50%. On average, the number of correct and plausible patches generated by the LLMs decreased by 49.48% and 42.90%, respectively. This results indicate that LLMs may still struggle to correctly understand the semantics of faulty programs as their repair capability highly depends on the form of code. For instance, we observed that many LLM-generated patches tend to repeatedly modify the injected dead code or transformed boolean predicates, rather than addressing the actual errors. This also motivates the experiments in RQ2.

When comparing the 6.7B and 33B parameter versions of DeepSeek-Coder-Instruct and OpenCodeInterpreter-DS, we observe a significant scaling effect. LLMs with larger parameter size generate more correct and plausible patches. Interestingly, the larger parameter size appears to correlate with weaker generalization capabilities. The two 33B models experienced decline rates of 52.13% and 57.47%, while the decline rates for the 6.7B models were 42.86% and 51.79%. We believe this is due to the larger models having a higher degree of overfitting to the original dataset.

TABLE III
REPAIR PERFORMANCE OF LLMS ON DEFECTS4J-TRANS USING DIFFERENT PROMPTS

LLM	Two-Shot		Two-Shot$_{fl}$		Trigger Test		Bug Report	
	#Correct Fixes	#Plausible Fixes	#Correct Fixes	#Plausible Fixes	#Correct Fixes	#Plausible Fixes	#Correct Fixes	#Plausible Fixes
WaveCoder-Ultra-6.7B	34	57	44(+29.41%↑)	64(+12.28%↑)	59(+73.53%↑)	94(+64.91%↑)	72(+111.76%↑)	111(+94.74%↑)
StarChat2-v0.1-15B	30	55	53(+76.67%↑)	76(+38.18%↑)	71(**+136.67%**↑)	114(+107.27%↑)	69(+130.0%↑)	122(**+121.82%**↑)
OpenCodeInterpreter-DS-6.7B	27	51	48(+77.78%↑)	75(+47.06%↑)	43(+59.26%↑)	72(+41.18%↑)	62(+129.63%↑)	100(+96.08%↑)
OpenCodeInterpreter-DS-33B	37	61	50(+35.14%↑)	67(+9.84%↑)	59(+59.46%↑)	96(+57.38%↑)	68(+83.78%↑)	110(+80.33%↑)
Average	32.00	56.00	48.75(+52.34%↑)	70.5(+25.89%↑)	58.0(+81.25%↑)	94.0(+67.86%↑)	67.75(+111.72%↑)	110.75(+97.77%↑)

B. RQ2: Impacts of Repair relevant Information

Table III shows the repair results of the four selected LLMs, which exhibited the most significant decline in RQ1 (IV-A), using different prompts. Overall, with the incorporation of additional repair-relevant information, all LLMs demonstrate improved repair capabilities, with 29.41% to 136.67% improvements on the number of correct patches and 9.84% to 121.82% improvements on the number of plausible patches.

Among the three types of repair-relevant information added, the bug report information yielded the most significant improvement for these four LLMs, with average increases of 111.72% in correct patches and 97.77% in plausible patches. We attribute this enhancement to the high quality of the bug reports in the Defects4J dataset, which involved substantial human effort in analyzing and providing information relevant to bug identification and repair, allowing the LLMs to better understand how to fix the buggy code. In contrast, fault localization information provided the least improvement for the four LLMs, with average increases of 52.34% in correct patches and 25.89% in plausible patches. This aligns with previous findings [15], suggesting that LLMs may struggle to effectively interpret fault localization information in this format. A possible reason is the limited availability of training data that include fault localization identifiers within the code.

Although incorporating repair-relevant information can improve LLMs' repair capabilities, only OpenCodeInterpreter-6.7B and WaveCoder-Ultra-6.7B showed slight improvements with bug reports across 12 experimental setups (4 LLMs $\times$ 3 prompts), compared to their performance on the original Defects4J dataset. The other 10 experimental setups still performed below their results on the original dataset. The findings suggest that enhancing the generalizability of LLM-based APR methods still requires significant progress.

V. THREATS TO VALIDITY

Manually reviewing all plausible patches to identify correct patches that are semantically consistent with the developer patches is an internal threat to the validity of our work. Following common APR practice, we perform a careful analysis of each plausible patch and have published our full set of correct and plausible patches at our homepage [25].

VI. FUTURE PLANS

Evaluate on a broader range of LLM-based APR methods and datasets. The number of LLM-based methods selected in this study is limited, and the investigation was conducted solely on Java programming language. In the future, we aim to evaluate more LLM-based methods on more datasets across different programming languages.

Explore better methods to enhance LLMs' understanding of program semantics. This study reveals that LLMs continue to face challenges in accurately grasping program semantics. Most LLM-based APR methods focus primarily on designing different ways to utilize LLMs. A promising direction lies in integrating traditional APR techniques, such as heuristic and constraint-solving based methods, with LLMs. For example, program analysis and verification could be applied to validate and refine LLM-generated patches, thereby improving their semantic understanding and repair capabilities.

Enhance the generalizability of LLMs. This study indicates that simple code transformation can significantly degrade the performance of LLMs. Therefore, we plan to explore a code normalization method to unify code format before processing by LLMs. Specifically, we may either pre-define a set of code normalization rules or use a fine-tuned LLM to ensure consistent representation of code, such as naming conventions and code structures, with the same semantics. This approach potentially allows us to filter out non-essential code features while preserving the key code semantics. Moreover, the unified representation can be used to fine-tune LLMs and enhance the application of learned knowledge from historical data. Finally, we urge developers of LLMs and LLM-based methods to share their datasets and engage in discussions about potential risks.

VII. CONCLUSION

In this study, we explore the generalizability of LLMs in the APR task and examine how various types of repair-relevant information affect the LLMs' bug-fixing capabilities. Our findings indicate that the studied 11 top-performing LLMs show limited genralizability. While incorporating repair-relevant information helps improve repair performance, challenges remain. Based on our study, we propose several promising directions for future research and have open-sourced all our experimental data to facilitate replication and further investigation [25].

VIII. ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 62202324.

REFERENCES

[1] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[2] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 31–42.

[3] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "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*, 2021, pp. 341–353.

[4] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1251–1263.

[5] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2023, pp. 1443–1455.

[6] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.

[7] C. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[8] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 522–534.

[9] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 535–547.

[10] J. Xiang, X. Xu, F. Kong, M. Wu, H. Zhang, and Y. Zhang, "How far can we go with practical function-level program repair?" *arXiv preprint arXiv:2404.12833*, 2024.

[11] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.

[12] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.

[13] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[14] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *IEEE/ACM 45th International Conference on Software Engineering*, 2023, pp. 1482–1494.

[15] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2023, pp. 1430–1442.

[16] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[17] Z. Yi, J. Ouyang, Y. Liu, T. Liao, Z. Xu, and Y. Shen, "A survey on recent advances in llm-based multi-turn dialogue systems," *arXiv preprint arXiv:2402.18013*, 2024.

[18] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou *et al.*, "The rise and potential of large language model based agents: A survey," *arXiv preprint arXiv:2309.07864*, 2023.

[19] R. Just, D. Jalali, and M. D. Ernst, "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*, 2014, pp. 437–440.

[20] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "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*, 2017, pp. 55–56.

[21] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.

[22] J. A. H. López, B. Chen, T. Sharma, and D. Varró, "On inter-dataset code duplication and data leakage in large language models," *arXiv preprint arXiv:2401.07930*, 2024.

[23] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[24] R. Xu, Z. Wang, R.-Z. Fan, and P. Liu, "Benchmarking benchmark leakage in large language models," *arXiv preprint arXiv:2404.18824*, 2024.

[25] Homepage, 2024, available at: https://zenodo.org/records/13901271.

[26] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, p. 642–646.

[27] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[28] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, "Evocodebench: An evolving code generation benchmark aligned with real-world code repositories," *arXiv preprint arXiv:2404.00599*, 2024.

[29] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv e-prints*, pp. arXiv–2403, 2024.

[30] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[31] "Evalplus leaderboard," https://evalplus.github.io/leaderboard.html, 2024, accessed: 2024-6-05.

[32] Z. Yu, X. Zhang, N. Shang, Y. Huang, C. Xu, Y. Zhao, W. Hu, and Q. Yin, "Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 5140–5153.

[33] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[34] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, "Opencodeinterpreter: Integrating code generation with execution and refinement," *arXiv preprint arXiv:2402.14658*, 2024.

[35] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," in *Forty-first International Conference on Machine Learning*, 2024.

[36] "Artigenz-coder-ds," https://huggingface.co/Artigenz/Artigenz-Coder-DS-6.7B, 2024, accessed: 2024-6-05.

[37] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[38] L. Tunstall, E. Beeching, N. Lambert, N. Rajani, S. Huang, K. Rasul, A. Bartolome, A. M. Rush, and T. Wolf, "The Alignment Handbook." [Online]. Available: https://github.com/huggingface/alignment-handbook

[39] "Speechless-codellama-v2.0," https://huggingface.co/uukuguy/speechless-codellama-34b-v2.0, 2024, accessed: 2024-6-22.

[40] H. Cheers, Y. Lin, and S. P. Smith, "Sppagiarise: A tool for generating simulated semantics-preserving plagiarism of java source code," in *2019 IEEE 10th International conference on software engineering and service science (ICSESS)*. IEEE, 2019, pp. 617–622.

[41] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 850–862.

[42] "Eclipse JDT Core," https://www.eclipse.org/jdt/core/, Eclipse Foundation, 2024, accessed: 2024-06.

[43] "Hugging face," https://huggingface.co, 2024, accessed: 2024-6-04.