Can GPT-O1 Kill All Bugs?
An Evaluation of GPT-Family LLMs on QuixBugs

Haichuan Hu[1], Ye Shang[2], Guolin Xu[3], Congqing He[4], Quanjun Zhang[2][*]

[1]Alibaba Cloud

[2]State Key Laboratory for Novel Software Technology, Nanjing University, China

[3]School of Computer Science and Technology, Chongqing University, China

[4]University Sains Malaysia

huhaichuan.hhc@alibaba-inc.com; {quanjun.zhang,201250032}@smail.nju.edu.cn;
xuguolin0626@gmail.com; hecongqing@hotmail.com;

Abstract

LLMs have long demonstrated remarkable effectiveness in automatic program repair (APR), with OpenAI's ChatGPT being one of the most widely used models in this domain. Through continuous iterations and upgrades of GPT-family models, their performance in fixing bugs has already reached state-of-the-art levels. However, there are few works comparing the effectiveness and variations of different versions of GPT-family models on APR. In this work, inspired by the recent public release of the GPT-o1 models, we conduct the first study to compare the effectiveness of different versions of the GPT-family models in APR. We evaluate the performance of the latest version of the GPT-family models (i.e., O1-preview and O1-mini), GPT-4o, and the historical version of ChatGPT on APR. We conduct an empirical study of the four GPT-family models against other LLMs and APR techniques on the QuixBugs benchmark from multiple evaluation perspectives, including repair success rate, repair cost, response length, and behavior patterns. The results demonstrate that O1's repair capability exceeds that of prior GPT-family models, successfully fixing all 40 bugs in the benchmark. Our work can serve as a foundation for further in-depth exploration of the applications of GPT-family models in APR. We have published experimental results online.

1 Introduction

Automated program repair (APR) is widely regarded as one of the most important and challenging areas in software quality assurance (Zhang et al., 2023a). In the literature, numerous APR techniques have been proposed to fix software bugs automatically. Among them, traditional template-based APR (Liu et al., 2019b) performs well for known bugs but is powerless against previously unseen bugs. In contrast, learning-based APR (Chen et al., 2022) is able to learn the bug-fixing patterns automatically from existing code repositories, thus yielding good generalization capability. Recently, as a further development of learning-based APR, large language model (LLM) based APR is receiving increasing attention from both academia and industry, delivering state-of-the-art performance (Zhang et al., 2024b).

ChatGPT (OpenAI, 2023), widely recognized as a leading application of LLMs, has been successfully integrated into various code-related tasks (Sun et al., 2023; Zhang et al., 2023b). In the APR field, ChatGPT stands as the most widely-used LLM, and has facilitated a significant amount of subsequent APR work, greatly advancing the progress in APR (Zhang et al., 2023d). Sobania et al. (Sobania et al., 2023) evaluate the bug-fixing capabilities of ChatGPT and find ChatGPT is able to fix 31 out of 40 bugs on QuixBugs (Lin et al., 2017) benchmark. Xia et al. (Xia and Zhang, 2023b) employ ChatGPT in a conversational manner to fix 114 and 48 bugs on the Defects4J-v1.2 and Defects4J-v2.0 benchmark. Zhang et al. (Zhang et al., 2023e) demonstrate that ChatGPT is able to fix 109 out of 151 buggy programming problems.

However, during the continuous advancements of GPT-family models, few studies have systematically compared the effectiveness and differences across various GPT versions in APR. Typically, upgrades in GPT models are attributed to newer versions having more parameters and larger training datasets, thus resulting in better code understanding and improved APR performance. Such a perspective may be challenged with the occurrence of the latest version of GPT, i.e., GPT-o1. Unlike its predecessors, GPT-o1 incorporates reinforcement learning (RL) and chain of thought (COT) techniques. Particularly, before answering a question, it first takes time to think, organizes a chain of thought, and then arrives at a final answer. By practice, O1 is more suitable for fields with com-

---

[*] Corresponding author.

plex logic and relatively definitive answers, such as mathematics, programming, and bug-fixing.

In this work, inspired by the fundamental differences between GPT-o1 and its predecessors, we conduct an empirical study to evaluate GPT-o1's performance in APR in a dialogue manner. We conduct experiments on the two currently available trial versions of the O1 model (i.e., O1-mini and O1-preview), as well as GPT-4o and ChatGPT, representing the most popular versions of the GPT-family models. We also include other LLMs (e.g., CodeX) and APR techniques (e.g., CIRCLE) as baselines on the QuixBugs benchmark. To this end, we design a two-step repair process. First, a basic prompt template is provided to the GPT models to directly repair the bugs. For bugs that fail the test cases, we further provide models with the error information from the dynamic execution to perform a second round of repair.

After all repairs are done, we collect data on the repair success rate, time spent on repairs, response length and behavior patterns of the model during the repair process, to further analyze O1's performance in APR. The results demonstrate that O1 outperforms ChatGPT (31/40) and GPT-4o (38/40) in APR, successfully repairing all 40 bugs in the Quixbugs benchmark. Moreover, O1's unique chain-of-thought reasoning pattern has proven effective for APR tasks. By forming a chain-of-thought, O1 can better understand the logic of the buggy code, provide repair ideas, and deliver the correct repair code. These findings are valuable and merit further attention in future research.

2 Methodology

In this section, we present the method for evaluating GPT-family models' ability to fix bugs. We use different GPT-family models to fix benchmark bugs and evaluate the repair behavior from multiple dimensions.

2.1 Two-step Fix

The process of bug fixing by GPT-family models consists of two steps. First, we use a basic repair template to ask the model whether there are bugs in the target program. After making a judgment, the model is requested to fix the bugs if they exist.

Listing 1: A basic template used to fix program bugs.

```
Does this program have a bug? How to fix
it?

[Code of the Buggy Program]
```

Listing 1 shows a basic template we use to fix program bugs. We begin the prompt with the question *Does this program have a bug? How to fix it?* to request GPT model to fix the following Python program, leaving a blank line between the question and the program.

After GPT model is asked to fix the buggy program, it will provide an initial fix answer. Previous GPT models (GPT-4o and earlier) would directly provide an answer, while the new version (ChatGPT-o1) will first spend some time thinking, develop a chain of thought, and then fix the program based on the chain of thought before presenting the complete fix code.

We verify the correctness of the answer provided by GPT models with existing test set. For buggy programs that do not pass tests, we perform a secondary fix. To help GPT models better understand the reasons for the program errors, we include the error messages in the template used for the secondary fix. Listing 2 is the prompt template used in the secondary fix step. Following *The given corrected version fails to pass the test cases, and the results are as follows:*, we provide the original error report to prompt GPT models for further repair.

Listing 2: The secondary fix template for bug fixing.

```
The given corrected version fails to
pass the test cases, and the results are
as follows:

[Error Message of Test Cases]
```

2.2 Evaluation Metrics

We evaluate the performance of GPT-o1 and prior GPT-family models in fixing program bugs from the following dimensions.

- **Repair success rate**. We utilize available test cases to identify generated patch correctness, which is the most important metric for evaluating the effectiveness of APR techniques in the literature.

- **Time for thinking**. Compared to earlier versions of GPT models, O1 has added a "thinking" phase. Thus, we record and analyze the time spent by O1-preview and O1-mini in thinking phase.

- **Response length**. We record and compare the output lengths of different models to evaluate the monetary cost of repairing.

- **Model behavior pattern**. We analyze behavior patterns of O1 and other GPT models to explore their mind when fixing bugs.

2.3 Benchmark and Baselines

We use QuixBugs (Lin et al., 2017) as the benchmark, which is widely adopted in the APR literature (Zhang et al., 2023a). For each of the 40 benchmark problems, we take the erroneous Python version. These programs have complete context and corresponding test cases, and they are relatively short, making them suitable for bug fixing through dialogue with GPT models. We compare our results with previous work (Sobania et al., 2023; Yuan et al., 2022). In prompt designing, we keep consistent with previous work (Sobania et al., 2023) and carefully review the comments in the original program to ensure that they do not reveal the solution, retaining only the relevant parts of the test cases. Thus, we integrate the baseline method into the first step of the repair method presented in Section 2.1.

3 Results

In this section, we first compare the repair results of O1, previous GPT-family models, and the baseline method on QuixBugs. We then analyze and summarize the details of the O1's repair behavior.

3.1 Comparison of O1, Previous GPT-family Models, and APR techniques

We present the main results of comparison in Table 1. In terms of baseline results, despite the significant improvement in repair effectiveness after introducing additional information, with a repair success rate reaching 31/40, it is still far from latest GPT models (GPT-4o, O1-mini, O1-preview). It can be seen that GPT-family models have made significant improvements in program repair capabilities through iterations.

Compared to the current mainstream GPT model (GPT-4o), O1 shows some improvement in program repair capabilities. Before providing test case error information, O1-mini and O1-preview can repair 37 and 38 bugs respectively, which is 2 and 3 more than GPT-4o. After providing test case error information, both O1-mini and O1-preview are able to repair all 40 bugs, whereas GPT-4o can only repair 38 bugs.

To further investigate the improvements of the O1 model compared to other GPT-family models in program repair, we also conduct a case analysis on the programs that the O1 model is able to repair but

GPT-4o and previous GPT models can not. We find that these programs are relatively complex, involving recursion and nested loops, and more relative to real-world problems. Take hanoi as instance, both O1-preview and O1-mini successfully repair the bug on the first attempt, whereas neither GPT-4o nor ChatGPT are able to fix it. ChatGPT fails four times util the correct test case answers are provided. GPT-4o incorrectly interpret the boundary condition, assuming that the source rod cannot be equal to the destination rod. In order to solve the hanoi problem, the O1 model spends 15 seconds thinking and forms a chain of thought (analyze functionality, check code logic, correct steps, optimize move steps). This chain of thought helps it correctly understand the logic of the problem, avoiding falling into incorrect logical branches. From this, we can infer that the reasoning pattern from chain of thought to solution is crucial for repairing buggy programs with complex logic.

3.2 Evaluation of O1 on Response Time, Response Length, and Behavior Pattern

In terms of response length, we conduct a token-count analysis of O1-preview, O1-mini, and GPT-4o on all benchmark programs with tiktoken[1]. Full results can be found in Table 2. We find that the average response lengths for the three models are 1450, 1086, and 654 tokens, respectively. This indicates that the O1's response length is over 50% longer compared to previous GPT models, leading to more comprehensive and complete answers, albeit at a higher cost.

In terms of response time, since the generation time is related to the length of output, we only measure the time taken by O1 for thinking. Full results can be found in Table 3. We find that the average thinking time for O1-preview reaches 19.82 seconds, which is about three times longer than that of O1-mini (7.02 s). Due to GPT-4o's ability to generate outputs directly without a "thinking" phase, Table 3 does not include GPT-4o when calculating thinking time. Considering the relationship between generation time and the output length of the model, we estimate that O1 takes over 70% more time on program repair tasks compared to previous GPT models. Thus when repairing relatively simple programs without involving too much context and complex logical relationships, considering the costs of time and money, traditional GPT models

---

[1] https://github.com/openai/tiktoken

Table 1: Results achieved by O1-preview, O1-mini, GPT-4o, ChatGPT, Codex (Prenner et al., 2022), CIRCLE (Yuan et al., 2022), and the standard APR approaches (Ye et al., 2021) on the problems from the QuixBugs benchmark. For ChatGPT, the number of successful runs are listed in brackets. The baseline method (Sobania et al., 2023) also mentions that when additional information is given, the repair effectiveness of ChatGPT improves. We highlight these results in blue. For O1-preview, O1-mini and GPT-4o, we provide the failed test case output in the prompt and ask the model to try fixing it again. We mark the results of the second fix in red.

Benchmark problem	O1-preview	O1-mini	GPT-4o	ChatGPT	Codex	CIRCLE	Standard APR
bitcount	✓	✓	✓	✗(0 / 4)✓	✓	✓	✗
breadth-first-search	✓	✓	✓	✓(2 / 4)	✗	✓	✗
bucketsort	✓	✓	✓	✓(4 / 4)	✓	✓	✗
depth-first-search	✓	✓	✓	✗(0 / 4)✓	✓	✗	✗
detect-cycle	✓	✓	✓	✗(0 / 4)✓	✗	✓	✓
find-first-in-sorted	✓	✓	✓	✓(2 / 4)	✓	✓	✗
find-in-sorted	✓	✓	✓	✓(3 / 4)	✗	✓	✗
flatten	✓	✓	✓	✓(4 / 4)	✓	✓	✗
gcd	✓	✓	✗✓	✗(0 / 4)✓	✓	✓	✗
get-factors	✓	✓	✓	✓(1 / 4)	✓	✓	✗
hanoi	✓	✓	✗✗	✗(0 / 4)✓	✓	✓	✗
is-valid-parenthesization	✓	✓	✓	✓(2 / 4)	✓	✗	✗
kheapsort	✓	✓	✓	✗(0 / 4)✗	✓	✗	✗
knapsack	✓	✓	✓	✓(1 / 4)	✓	✓	✓
kth	✓	✓	✓	✗(0 / 4)✓	✗	✗	✗
lcs-length	✓	✓	✓	✗(0 / 4)✗	✗	✓	✗
levenshtein	✓	✓	✓	✗(0 / 4)✓	✗	✓	✓
lis	✗✓	✗✓	✓	✗(0 / 4)✗	✗	✗	✓
longest-common-subsequence	✓	✓	✓	✗(0 / 4)✗	✓	✗	✗
max-sublist-sum	✓	✗✓	✗✓	✗(0 / 4)✓	✓	✗	✗
mergesort	✓	✓	✓	✓(1 / 4)	✗	✓	✓
minimum-spanning-tree	✓	✓	✓	✗(0 / 4)✓	✗	✓	✗
next-palindrome	✓	✓	✓	✓(1 / 4)	✗	✓	✗
next-permutation	✓	✓	✓	✗(0 / 4)✓	✗	✓	✗
pascal	✓	✓	✓	✓(1 / 4)	✗	✓	✗
possible-change	✓	✓	✓	✓(1 / 4)	✓	✗	✗
powerset	✓	✓	✓	✗(0 / 4)✓	✓	✓	✗
quicksort	✓	✓	✓	✓(1 / 4)	✓	✓	✓
reverse-linked-list	✓	✓	✓	✓(2 / 4)	✓	✗	✗
rpn-eval	✓	✓	✓	✗(0 / 4)✗	✗	✗	✓
shortest-path-length	✗✓	✗✓	✗✓	✓(1 / 4)	✗	✗	✗
shortest-path-lengths	✓	✓	✓	✗(0 / 4)✗	✗	✗	✗
shortest-paths	✓	✓	✓	✓(1 / 4)	✗	✗	✗
shunting-yard	✓	✓	✓	✓(2 / 4)	✗	✗	✗
sieve	✓	✓	✓	✗(0 / 4)✓	✓	✓	✗
sqrt	✓	✓	✓	✓(1 / 4)	✓	✗	✗
subsequences	✓	✓	✓	✓(1 / 4)	✗	✓	✗
to-base	✓	✓	✓	✗(0 / 4)✗	✓	✗	✗
topological-ordering	✓	✓	✓	✗(0 / 4)✗	✗	✓	✗
wrap	✓	✓	✗✗	✗(0 / 4)✗	✓	✗	✗
Σ (Solved)	**38(40)**	**37(40)**	**35(38)**	**19(31)**	**21**	**23**	**7**

seem to be more practical compared to O1.

In terms of model behavior patterns, we observe that O1 typically begins by performing logical analysis on the buggy program, generating a solution, and then gradually proceeding with repairs and testing before providing the complete fixed code. In contrast, GPT-4o tends to provide the repaired code first, followed by an explanation of the code.

4 Related Works

Traditional APR methods can be mainly categorized into search-based (Jiang et al., 2018; Wong et al., 2021; Xin and Reiss, 2019), constraint-based (Gao et al., 2021; Xuan et al., 2017; Le et al.,

Table 2: The token length output by O1 and GPT-4o when fixing each benchmark problem.

Benchmark Problem	O1-preview	O1-mini	GPT-4o
bitcount	1846	1172	647
breadth-first-search	1179	1015	780
bucketsort	1540	710	669
depth-first-search	1620	738	667
detect-cycle	1518	719	757
find-first-in-sorted	2099	1406	820
find-in-sorted	2159	912	785
flatten	987	856	487
gcd	1393	822	456
get-factors	1109	932	553
hanoi	1826	1262	726
is-valid-parenthesization	878	578	770
kheapsort	1282	1203	1078
knapsack	1722	1029	904
kth	1848	1171	472
lcs-length	1345	927	589
levenshtein	1485	914	727
lis	1612	896	597
longest-common-subsequence	1299	875	710
max-sublist-sum	1423	1370	485
mergesort	575	1017	1438
minimum-spanning-tree	1386	1346	903
next-palindrome	1215	1057	1658
next-permutation	1642	931	817
pascal	1778	798	692
possible-change	940	860	798
powerset	1642	1128	563
quicksort	760	691	281
reverse-linked-list	1062	1005	342
rpn-eval	1638	1097	415
shortest-path-length	814	1097	768
shortest-path-lengths	2035	1263	432
shortest-paths	1130	1244	495
shunting-yard	2104	1329	573
sieve	1432	1067	511
sqrt	931	812	268
subsequences	1557	653	363
to-base	968	1015	281
topological-ordering	1497	2755	390
wrap	1374	752	383
Average	1450.07	1086.90	654.07

Table 3: The thinking time spent on repairing each benchmark problem using the O1 model. When thinking time is less than five seconds, we count it as five seconds.

Benchmark problem	O1-preview	O1-mini	GPT-4o
bitcount	17s	< 5s	~
breadth-first-search	27s	< 5s	~
bucketsort	18s	< 5s	~
depth-first-search	9s	< 5s	~
detect-cycle	23s	< 5s	~
find-first-in-sorted	31s	11s	~
find-in-sorted	18s	< 5s	~
flatten	19s	< 5s	~
gcd	8s	< 5s	~
get-factors	16s	< 5s	~
hanoi	15s	7s	~
is-valid-parenthesization	10s	< 5s	~
kheapsort	30s	6s	~
knapsack	38s	5s	~
kth	28s	10s	~
lcs-length	12s	< 5s	~
levenshtein	18s	< 5s	~
lis	51s	38s	~
longest-common-subsequence	15s	42s	~
max-sublist-sum	21s	6s	~
mergesort	17s	< 5s	~
minimum-spanning-tree	28s	10s	~
next-palindrome	53s	8s	~
next-permutation	23s	7s	~
pascal	33s	8s	~
possible-change	14s	< 5s	~
powerset	23s	< 5s	~
quicksort	11s	< 5s	~
reverse-linked-list	14s	< 5s	~
rpn-eval	16s	5s	~
shortest-path-length	16s	7s	~
shortest-path-lengths	13s	< 5s	~
shortest-paths	21s	< 5s	~
shunting-yard	17s	< 5s	~
sieve	15s	8s	~
sqrt	10s	5s	~
subsequences	47s	< 5s	~
to-base	20s	< 5s	~
topological-ordering	18s	6s	~
wrap	21s	< 5s	~
Average	19.82s	7.02s	~

2017), and template-based (Liu et al., 2019a,c; Ghanbari and Zhang, 2019) approaches. These repair methods, although effective, struggle to achieve a balance between performance and effectiveness.

Learning-based (Fu et al., 2022; Chen et al., 2021; Li et al., 2022; Zhang et al., 2023c) APR leverages prior knowledge from pre-trained models to automatically fix program defects, demonstrating wide-ranging effectiveness. It is capable of cross-language repairs and can correct previously unseen defects. Building on this foundation, researchers (Xia et al., 2023; Xia and Zhang, 2023a; Zhang et al., 2024a) further utilize LLMs to further enhance the repair effects.

This paper conducts extensive research on GPT for APR. Based on previous work (Sobania et al., 2023; Zhang et al., 2023e) and in conjunction with the newly released GPT model O1, we provide a comprehensive analysis and comparison of the performance and differences of representative models within the GPT-family in the context of APR.

5 Conclusion

This paper conducts the first empirical study to evaluate the capabilities of the latest GPT-o1 model in automated program repair. The experimental results on the QuixBugs benchmark demonstrate the advancements of GPT-o1 against previous GPT-family models and repair techniques. We also provide a comprehensive analysis of GPT-o1 from four

dimensions, like thinking time and response length. Besides, we conduct a case analysis and conclude the advantages of GPT-o1 in repairing complex bugs due to chain of though. Our findings offer a reference for future in-depth research on utilizing GPT-o1 and GPT models for program repair.

6 Limitations

Currently, O1 is in the trial phase, with usage limits and high API costs. Understanding of O1 is still quite insufficient. The benchmark used in our study is relatively small, and further research is needed with larger datasets to explore the O1 model's capabilities in APR more comprehensively.

References

Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. Vulrepair: a t5-based automated software vulnerability repair. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, pages 935–947.

Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Trans. Softw. Eng. Methodol.*, 30(2):14:1–14:27.

Ali Ghanbari and Lingming Zhang. 2019. Prapr: Practical program repair via bytecode mutation. In *34th International Conference on Automated Software Engineering, ASE*, pages 1118–1121.

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. ISSTA.

Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 593–604.

Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. 2022.

Transrepair: Context-aware program repair for compilation errors. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 1–13.

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*, pages 55–56.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019a. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*, pages 456–467.

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

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019c. Tbar: revisiting template-based automated program repair. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 31–42.

Chatgpt OpenAI. 2023. Optimizing language models for dialogue, 2022. *URL: https://openai.com/blog/chatgpt*.

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

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30. IEEE.

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865*.

Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. Varfix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 354–366.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the

era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494.

Chunqiu Steven Xia and Lingming Zhang. 2023a. Conversational automated program repair. *Preprint*, arXiv:2301.13246.

Chunqiu Steven Xia and Lingming Zhang. 2023b. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*.

Qi Xin and Steven Reiss. 2019. Better code search and reuse for better program repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*, pages 10–17.

Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Software Eng.*, 43(1):34–55.

He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, 171:110825.

Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. Circle: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 678–690.

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023a. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69.

Quanjun Zhang, Chunrong Fang, Ye Shang, Tongke Zhang, Shengcheng Yu, and Zhenyu Chen. 2024a. No man is an island: Towards fully automatic programming by code search, code generation and program repair. *arXiv preprint arXiv:2409.03267*.

Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024b. A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466*.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023b. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223*.

Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023c. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing*.

Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023d. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 535–547. IEEE.

Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023e. A critical review of large language model on software engineering: An example from ChatGPT and automated program repair. *arXiv preprint arXiv:2310.08879*.