DePro: Understanding the Role of LLMs in Debugging Competitive Programming Code

Nabiha Parvez
Military Institute of Science And Technology
Dhaka, Bangladesh
nabihaparvez11@gmail.com

Tanvin Sarkar Pallab
Military Institute of Science And Technology
Dhaka, Bangladesh
tanvin.pallab2442002@gmail.com

Mia Mohammad Imran
Missouri University of Science and Technology
Rolla, Missouri, USA
imranm@mst.edu

Tarannum Shaila Zaman
University of Maryland Baltimore County
Baltimore, Maryland, USA
zamant@umbc.edu

Abstract

Debugging consumes a substantial portion of the software development lifecycle, yet the effectiveness of Large Language Models (LLMs) in this task is not well understood. Competitive programming offers a rich benchmark for such evaluation, given its diverse problem domains and strict efficiency requirements. We present an empirical study of LLM-based debugging on competitive programming problems and introduce DePro, a test-case–driven approach that assists programmers by correcting existing code rather than generating new solutions. DePro combines brute-force reference generation, stress testing, and iterative LLM-guided refinement to identify and resolve errors efficiently. Experiments on 13 faulty user submissions from *Codeforces* demonstrate that DePro consistently produces correct solutions, reducing debugging attempts by up to 64% and debugging time by an average of 7.6 minutes per problem compared to human programmers and zero-shot LLM debugging.

Keywords

Large Language Models, Competitive Programming, Debugging, Manual Study, Empirical Study

ACM Reference Format:
Nabiha Parvez, Tanvin Sarkar Pallab, Mia Mohammad Imran, and Tarannum Shaila Zaman. 2026. DePro: Understanding the Role of LLMs in Debugging Competitive Programming Code. In *Companion Proceedings of the 34th ACM Symposium on the Foundations of Software Engineering (FSE '26), June 5–9, 2026, Montreal, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Debugging, the process of identifying and fixing software defects, is a multi-step procedure involving bug identification, localization, reproduction, root cause analysis, and resolution [22]. It is estimated that debugging consumes nearly 50% of the software development life cycle [20]. Effective debugging requires not only familiarity with the source code but also reasoning about root causes; incomplete or superficial fixes often introduce additional errors [19].

Large Language Models (LLMs) [21] have recently transformed Natural Language Processing (NLP) [18] and code-related tasks such as algorithmic problem solving and code generation [6, 12]. Their growing adoption in software engineering raises an important question: *to what extent can LLMs assist in debugging, particularly in complex, real-world scenarios?* Despite progress in LLM-based development tools (e.g., Copilot [5], AlphaCode [1]), systematic evaluation of their debugging capabilities remains limited.

Competitive programming provides a rich and rigorous testbed for evaluating debugging techniques. These problems span diverse domains, impose strict computational constraints, and demand both efficiency and correctness, making them well-suited for assessing the debugging capabilities of LLMs [12].

Recent work has analyzed LLMs' ability to solve competitive programming problems [12]. Other studies have investigated LLM-assisted debugging in broader contexts, including interactive debugging [17], scientific debugging frameworks [15], and multi-agent collaboration for code repair [8]. However, research on debugging within competitive programming remains limited, and systematic evaluations of LLM performance in this context are lacking. Given the diversity and rigor of competitive programming challenges, analyzing LLMs' zero-shot debugging effectiveness offers a meaningful measure of their overall debugging capabilities.

In this paper, we first present an empirical and manual study on 10 different problems from Codeforces [3]. For each problem, we selected 10 users who had repeatedly received incorrect answers. For each user, we took their faulty code and asked ChatGPT-5 [4] to debug it—not by generating new code, but by correcting the existing code. We then analyzed the results and found that the LLM generally required fewer attempts than human users. However, in some cases, the LLM needed many attempts; for instance, it took 8 attempts to fix a single user's fault. We manually investigated the causes behind these high-attempt cases and also observed differences in the LLM's code-solving strategies. Our manual study summary is as follows: LLMs are effective at reproducing standard solution patterns and applying localized fixes, but they lack the flexibility and strategic reasoning that human programmers employ, particularly for problems with multiple valid solutions or unconventional approaches.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We propose an automated approach, DePro, based on the observation that LLMs debug more effectively when provided with specific problem details, particularly failing test cases. DePro first generates a brute-force reference solution using ChatGPT-5, then stress-tests both the user's code and the reference on random and edge-case inputs to identify failures. The LLM is iteratively prompted with the failing input, expected output, and user code to suggest fixes, with each modification retested for up to eight iterations. This process enables the LLM to refine code, identify errors, and improve debugging efficiency.

We evaluate DePro on 13 cases drawn from 6 faulty Codeforces problems [3], each submitted by a different user, and find that it can effectively debug all 13 problems, producing correct solutions in an average of 6.5 minutes per problem.

1.1 Empirical Study

1.1.1 Data Collection. We curate a dataset of 10 problems, each with 10 user codes, resulting in a total of 100 user codes. We take these codes from the popular competitive programming platform Codeforces [3]. We select the problems based on user attempts, focusing on cases where users submit multiple wrong answers, struggle to debug their solutions, but ultimately solve the problems after several tries. An attempt is defined as a single modified submission intended to correct the previous error. The problem set includes constructive algorithms (4), math (2), greedy (3) and implementation (1). Their difficulty ranges from easy (3) to intermediate (7).

We also select user codes based on the number of attempts required to solve the problem. In particular, we include codes that fail on the first submission. To ensure variety, we choose codes with different failure points across test cases (e.g., TC-2, TC-8, TC-15) rather than selecting codes that fail at the same point. This approach prevents the dataset from biasing toward a single debugging pattern. We test GPT-5's reasoning model by providing the LLM with the problem statement, time complexity, memory complexity, input, and output before giving it the user's code. We then use Prompt 1 to extract the debugging feedback generated by the LLM.

Prompt 1: Debugging Feedback Extractor Prompt

Problem Statement: You are given an array a1,a2,...
Time Limit: 1 second **Memory Limit:** 256 megabytes
Input: Each test contains multiple test cases...
Output: For each test case, output a single integer...
Sample Input: 8 15 22 30 **Sample Output:** 7
Task:Consider the given problem description. Here is my code for this problem. I got wrong answer. Can you debug my code?

1.1.2 Data Preprocessing: We remove unnecessary spacing and newlines, then decompose each problem into distinct components, such as, problem statement, time constraint, memory constraint, input, output, and initial prompt to improve clarity. We clarify and simplify mathematical notations before presenting the problems to the LLM.

1.1.3 Empirical Study Result: We analyze the LLM's responses after administering Prompt 1. Our primary objective is to compare the number of attempts the LLM requires to solve a problem with the number of attempts made by human problem solvers. The first three columns of Table 1 present the problem name, difficulty level, and problem type. Columns four and five report the average number of attempts made by the problem solvers and by the LLM, respectively. For each problem, we collect data from ten problem solvers. On average, human problem solvers require 3.68 attempts to solve a problem, whereas the LLM requires 1.8 attempts.

Table 1: Summary of the empirical study on LLM performance

Name	Dif	Type	N_U	N_L	Sim.
Flip Bits	Intrm	Con. Algo.	4.2	1.3	3.8
Equal with mod	Intrm	Con. Algo.	6.7	4.2	1.9
Alternating	Easy	Implement	4	2	4.4
Stable Groups	Intrm	Implement	3.8	1	5
Good Start	Intrm	Implement	3.7	2.6	4.3
Cherry Bomb	Intrm	Implement	4	1.3	3.8
T-primes	Intrm	Implement	3.2	1.2	4.7
Bobritto Banditto	Intrm	Implement	2.6	1.9	2.7
Letter Home	Intrm	Implement	2.2	1.5	4
Brightness Begins	Intrm	Implement	2.4	1	3

$\#Dif$ = Difficulty. N_U = Average number of manual attempts by the user to solve a problem. N_L. = Average number of attempts by the LLM to solve a problem. Sim.= Similarity Score.

1.2 Manual Study

Two competitive programmers analyze our dataset and compare the problem-solving approaches of human participants and the LLM. For the manual study, they first construct a table similar to Table 2. Drawing on their experience, they describe the human and LLM debugging approaches, which appear in Columns 3 and 4 of Table 2. They then assign a similarity score from 1 to 5 to indicate how closely the two approaches align and record it in the final column. We compute the similarity score by averaging the two programmers' ratings. The final column of Table 1 reports the average similarity score for each problem. Through this manual study, we address three research questions.

RQ1: *Is the LLM's response relevant to the user's solution?*

When humans and the LLM converge on similar approaches, the resulting solutions typically follow canonical or well-known strategies, indicating that the LLM effectively reproduces documented solution patterns. For purely mathematical problems, the LLM often uses different formulas than human solvers. In contrast, when humans rely on unconventional heuristics, shortcuts, or domain-specific tricks, the LLM tends to diverge, producing alternative solutions that are syntactically correct but sometimes less efficient, more verbose, or structurally different. We also observe that the LLM struggles with problems that admit multiple valid solutions, as it usually converges on a single reasoning template rather than exploring diverse approaches.

RQ2: *How do the LLM's answers differ from users' code in terms of approach?*

The LLM's debugging behavior differs from humans'. Humans locate errors by testing boundary cases, reasoning backward, or restructuring their approach. The LLM tends to make localized edits around failing test cases, adjusting loops, conditions, or data structures based on learned patterns. While this often fixes issues quickly, it can repeat ineffective fixes or reintroduce past errors. This highlights a key distinction: humans rely on flexible reasoning

Table 2: Summary of the manual study on LLM performance

Name	User	Approach (User)	Approach (LLM)	Sim.
Cherry Bomb	u1	Collect known pairs $b_i \geq 0$, form $s_i = a_i + b_i$. If multiple distinct $s_i \to 0$; if one x verify $0 \leq x - a_i \leq k$ for missing b_i; if none count integers in $[\max_i a_i,\ \min_i(a_i + k)]$.	Enforce single forced x (or none), validate only missing-b_i indices, and use count = $\max(0, \min_i(a_i + k) - \max_i a_i + 1)$.	5
	u2	Set first forced $x = a_i + b_i$ but later overwrote it; used global min/max for all-−1 case.	Keep-first-x (or second-pass verify all known sums); clamp negative counts to 0.	5
Bobritto Bandito	u1	Shrank only right by $(n - m)$ (assumes unilateral growth), may exclude 0.	Split rewind: take_left = $\min(n - m, -l)$, take_right = $(n - m)$ − take_left; $l' = l$ + take_left, $r' = r$ − take_right.	3

and strategic exploration, whereas the LLM primarily uses pattern-based refinements.

RQ3: *How can we improve the performance of LLM?*

We evaluate an LLM's performance by its ability to transform buggy code into an accepted solution when provided with failing test cases. The LLM is first prompted to debug the user's code (Prompt 1), and the generated solution is submitted to Codeforces. If the submission fails, the corresponding failing test case is provided along with the code. We find that supplying failing test cases significantly improves debugging performance compared to re-prompting without additional information. This observation motivates DePro, which leverages failing test cases to enhance LLM-assisted debugging.

2 Methodology and Evaluation

Based on our manual observations, we find that LLMs perform more effectively in debugging when prompted with specific details of the problem, particularly the failing test case. Motivated by this finding, we develop an automated technique, DePro, that leverages LLM-assisted debugging and evaluate it on 13 real-world problems.

2.1 DePro Architecture:

Figure 1 illustrates the workflow of DePro, which consists of three major steps: (1) initial code generation, (2) stress testing and identification of the failure-inducing test case, and (3) debugging with the LLM.

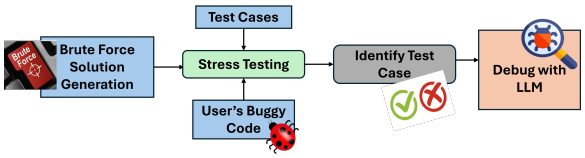

Figure 1: Overview of DePro Methodology

Initial Code Generation: The process begins by prompting ChatGPT-5's reasoning model with a problem template that we construct, which includes the problem description, constraints, and input-output specifications. We instruct the LLM to generate a brute-force solution that prioritizes passing all sample test cases. This solution serves as a reference for subsequent debugging steps and provides a benchmark to compare against the user's original code.

Stress Testing and Failure Detection: After generating the brute-force solution we conduct stress testing with the user's original code. We execute both programs on a wide range of generated test cases, including edge cases and boundary values. A separate

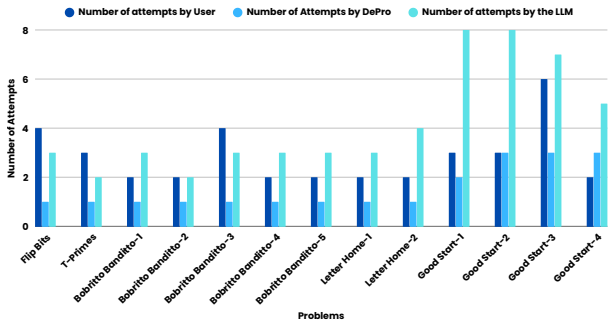

Figure 2: Attempts required by DePro, manual methods, and zero-shot LLMs

script generates the inputs, with ranges according to the problem statement. The test case generation helps to identify any mismatch between the brute-force solution and the user code. The stress testing loop terminates when it encounters a failing test case. For each failing case, we extract and store both the input and the resulting output for use in the subsequent debugging phase.

Iterative Debugging with LLM: After identifying a failing test case, we prompt the LLM with the failing input, the expected output, and the user's original code. The model is instructed to analyze and debug the code, as illustrated in Prompt 2. Once the LLM suggests modifications, we subject the updated code to another round of stress testing. This process creates a feedback loop in which debugging is guided primarily by failing test cases. The loop of code modification and stress testing continues for up to eight iterations. In each iteration, the model refines the code's logic, learns from previous failures, and identifies potential sources of error.

Prompt 2: DePro Debugging Prompt

This is my code. It failed in the test case:
Input: 1 2 3 4 Output: 2
Expected Output: 4
Can you debug my code?

2.2 Evaluation

We select 13 faulty user codes from 6 problems, for which the LLM requires an average of 4 attempts to debug, with a maximum of 8 attempts and a minimum of 2 attempts. We evaluate the effectiveness and efficiency of DePro.

We implemented a prototype of DePro using ChatGPT-5's reasoning model [4]. The prototype consisted of three main C++ files: (i) a brute-force solution, (ii) a user-provided solution, and (iii) a random test generator. These components were orchestrated and

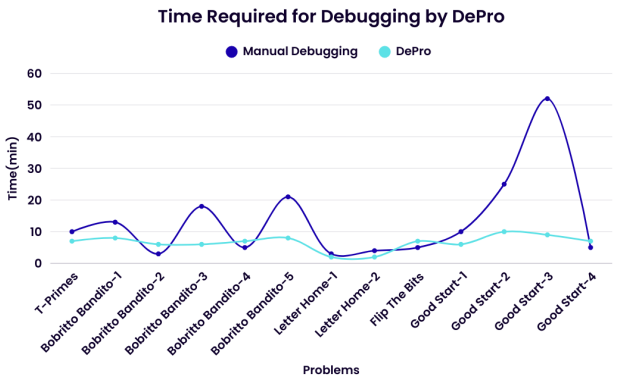

Figure 3: Total time in Debugging

executed together via a batch script [2]. During execution, if the output of the user's solution differed from that of the brute-force solution, the corresponding failing test case was automatically displayed in the terminal. We conducted the experiments on a single workstation with an 11th-Gen Intel® Core™ i7-1165G7 @ 2.80 GHz, 16 GB RAM, a graphics device with 2 GB VRAM.

Figure 2 presents a comparison of the total number of attempts required by human problem solvers, zero-shot LLMs, and DePro for each problem. On average, DePro reduces the number of attempts by 64% compared to the zero-shot LLM approach described in Section 2, with a maximum reduction of 75% and a minimum of 50%. Across all 13 faulty user codes, DePro consistently requires fewer attempts than both human solvers and zero-shot LLMs, successfully producing the correct solution after debugging the user's code. These results demonstrate that DePro is highly effective at debugging a variety of problem types.

DePro reduces the overall time required to debug competitive programming problems, taking an average of 6.5 minutes per problem. **Figure 3** compares the debugging times of DePro and human problem solvers across our 13 user codes. On average, DePro reduces the debugging time by 7.6 minutes per problem compared to manual methods. These results demonstrate that DePro is highly efficient in debugging competitive programming solutions.

3 Related Work

LLM-based debugging has gained traction, with studies exploring strategies from agent-based collaboration to runtime feedback.

Interactive Debugging. Systems have been developed that integrate LLMs with debuggers for natural-language interaction. ChatDBG [17] connects LLMs to GDB/LLDB, allowing queries about program state. AutoSD [15] combines LLMs with hypothesis generation and testing, and DebugBench [23] introduces a benchmark for evaluating LLM debugging across multiple error types.

Runtime Feedback. Several approaches incorporate execution feedback into the debugging process. LDB [8] uses runtime traces to validate program blocks. Ledex [13] applies execution-guided refinement to train LLMs to self-debug and provide explanations. RTLFixer [24] addresses hardware description languages by using compilation feedback to repair RTL syntax errors.

Agent-Based Debugging. Multi-agent frameworks assign different roles to the debugging process. FixAgent [16] separates tasks such as fault localization and patch generation. RGD [14] and COAST [27] explore role-based collaboration and data synthesis. AGDebugger [10] introduces user-driven steering mechanisms for multi-agent debugging systems.

Automated Program Repair. Recent advances in LLM-based automated program repair have shown promising results in fixing real-world bugs. AutoCodeRover [29] combines LLMs with program structure-aware code search to automatically resolve Github issues, achieving 19% efficacy on SWE-bench. SWE-agent [26] discusses agent-computer interfaces for automated software engineering through iterative tool invocation. InspectCoder [25] enables LLM's to conduct dynamic analysis through interactive debugger control. While these approaches show remarkable results in general program repair, they mostly focus on production codebases rather than the algorithmic complexity and efficiency constraints of competitive programming.

Although prior studies have advanced LLM-assisted debugging, they generally overlook dynamic, test-case–driven feedback. We bridge this gap by integrating brute-force generation, stress testing, and iterative refinement.

4 Threats to Validity

In this section, we discuss four types of threats, similar to prior research related to LLM[7, 12, 21].

Internal Validity: Data contamination poses a significant threat to the internal validity of our study, as some of the problems we evaluate may already exist in the training data of the LLM.

External Validity: In this work, we use a dataset of 100 code submissions. Although the dataset is relatively small, we ensured generalizability by collecting code from 10 different users for each problem, resulting in 100 unique user submissions. This diversity reduces threats to external validity by incorporating variations in coding styles across different programmers. Additionally, we use the latest version of ChatGPT for our evaluation.

Construct Validity: All experimental procedures are documented, and the prompts for data collection and solution evaluation are publicly available to promote reproducibility and transparency.

Conclusion Validity: To reduce bias, all evaluations are conducted using a consistent prompt. Furthermore, manual data annotations are validated through independent review by two separate programmers.

5 Discussion & Future Scope

Our study shows that LLMs can effectively assist in debugging, but their performance relies on execution feedback.

While we focus on competitive programming, DePro has broader applications in software engineering. Real-world systems rarely provide failing test cases, requiring developers to manually identify edge cases. Automating test case generation through stress testing can reveal these cases, and DePro demonstrates how combining generated test cases with LLM-based debugging can support software maintenance, program repair, and automated debugging

pipelines. A key limitation is its dependence on a correct brute-force solution and effective test case generation, which may not scale to large input spaces or complex problems.

In the future, DePro could handle more complex programming contexts by integrating runtime debuggers and symbolic execution tools, such as GDB [11] and KLEE [9], providing richer execution feedback and exposing hidden edge cases. Adaptive prompting could reformulate failed debugging attempts with new constraints. DePro could also integrate into educational platforms as a teaching assistant or into professional IDEs and version control systems, offering real-time analysis, fix suggestions, and automated refactoring. These extensions would expand DePro's impact across research, learning, and professional programming.

6 Conclusion

In this paper, we conduct an initial manual and empirical study on LLMs' performance in debugging competitive problems using zero-shot prompting. Based on this study, we introduce DePro, a novel debugging approach that combines the capabilities of Large Language Models (LLMs) with an iterative, test-case–driven method. DePro integrates an iterative feedback loop guided by failing test cases and enhances the efficiency and effectiveness of debugging. Our study shows that this approach expands debugging techniques and simplifies the process compared to manual methods. DePro holds strong potential to serve as a debugging assistant tool for developers.

7 Acknowledgment

This work was supported in part by NSF grants CCF-2348277 and CCF-2518445 [28].

References

[1] [n. d.]. AlphaCode Attention Visualization. https://alphacode.deepmind.com/
[2] [n. d.]. Batch Script Tutorial. https://www.tutorialspoint.com/batch_script/index.htm
[3] [n. d.]. Codeforces. https://codeforces.com/
[4] [n. d.]. GPT-5 is here. https://openai.com/gpt-5/
[5] [n. d.]. Microsoft Copilot: Your AI companion. https://copilot.microsoft.com/
[6] 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374
[7] Alif Al Hasan, Subarna Saha, Mia Mohammad Imran, and Tarannum Shaila Zaman. 2025. LLPut: Investigating Large Language Models for Bug Report-Based Input Generation. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 1652–1659. https://doi.org/10.1145/3696630.3728701
[8] Nazmus Ashrafi, Salah Bouktif, and Mohammed Mediani. 2025. Enhancing LLM Code Generation: A Systematic Evaluation of Multi-Agent Collaboration and Runtime Debugging for Improved Accuracy, Reliability, and Latency. arXiv:2505.02133 [cs.SE] https://arxiv.org/abs/2505.02133
[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[10] Will Epperson, Amanda Fang, Kaixuan Zhang, Saleema Amershi, Daniel S. Weld, and Ece Kamar. 2025. Interactive Debugging and Steering of Multi-Agent AI Systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, 1–13. https://doi.org/10.1145/3613904.3642700
[11] GNU Project. 2026. *Debugging with GDB*. https://www.gnu.org/software/gdb/documentation/ Accessed: 2026-03-19.
[12] Md Sifat Hossain, Anika Tabassum, Md. Fahim Arefin, and Tarannum Shaila Zaman. 2025. LLM-ProS: Analyzing Large Language Models' Performance in Competitive Problem Solving. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. 80–87. https://doi.org/10.1109/LLM4Code66737.2025.00015
[13] Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya B Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Ledex: Training LLMs to better self-debug and explain code. *Advances in Neural Information Processing Systems* 37 (2024), 35517–35543.
[14] Haolin Jin, Zechao Sun, and Huaming Chen. 2024. Rgd: Multi-llm based agent debugger via refinement and generation guidance. In *2024 IEEE International Conference on Agents (ICA)*. IEEE, 136–141.
[15] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. arXiv:2304.02195 [cs.SE] https://arxiv.org/abs/2304.02195
[16] Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jen tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R. Lyu. 2024. A Unified Debugging Approach via LLM-Based Multi-Agent Synergy. arXiv:2404.17153 [cs.SE] https://arxiv.org/abs/2404.17153
[17] Kyla H. Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. 2025. ChatDBG: Augmenting Debugging with Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE085 (June 2025), 22 pages. https://doi.org/10.1145/3729355
[18] Lutfun Nahar Lota, Tarannum Shaila Zaman, Mirza Mohammad Azwad, Labiba Farah, Abrar Chowdhury, Zaarin Anjum, Chadni Islam, and Abu Raihan Mostofa Kamal. [n. d.]. Recent Trends and Challenges in Using Nlp Techniques in Software Debugging: A Systematic Literature Review. *Available at SSRN 5060080* ([n. d.]).
[19] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGOPS Oper. Syst. Rev.* 42, 2 (March 2008), 329–339. https://doi.org/10.1145/1353535.1346323
[20] Steve McConnell. 2004. *Code Complete*. Microsoft Press.
[21] Shireesh Reddy Pyreddy and Tarannum Shaila Zaman. 2025. EmoXpt: Analyzing Emotional Variances in Human Comments and LLM-Generated Responses. In *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*. 00088–00094. https://doi.org/10.1109/CCWC62904.2025.10903889
[22] Margaret Rouse. 2017, Dec 12. Debugging. http://searchsoftwarequality.techtarget.com/ definition/debugging.
[23] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, et al. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).
[24] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*. ACM/IEEE, 1–6. https://doi.org/10.1145/3649329.3656906
[25] Yunkun Wang, Yue Zhang, Guochang Li, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. 2025. InspectCoder: Dynamic Analysis-Enabled Self Repair through interactive LLM-Debugger Collaboration. arXiv:2510.18327 [cs.SE] https://arxiv.org/abs/2510.18327
[26] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793
[27] Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. 2025. COAST: Enhancing the Code Debugging Ability of LLMs through Communicative Agent Based Data Synthesis. In *Findings of the Association for Computational Linguistics: NAACL 2025*. 2570–2585.
[28] Tarannum Shaila Zaman. 2024. CRII: SHF: An Automated and User-centered Framework for Reproducing System-level Concurrency Bugs by Analyzing Bug Reports. NSF Award Number 2348277. Directorate for Computer and Information Science and Engineering, Division of Computing and Communication Foundations. 2024.. , 48277 pages.
[29] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE] https://arxiv.org/abs/2404.05427