Guiding Novice Programmers: LLM-Enhanced IDEs for Learning and Debugging

Gregory Charles K. Tiong[1,*,†]

[1] De La Salle University Manila 2401 Taft Avenue, Manila

Abstract

Debugging is an essential skill for every programmer, but is overlooked, time-consuming, and prone to error tasks. While various debugging tools like IDE debuggers and Automatic Program Repair (APR) aim to streamline the process, these are frequently challenging for experienced developers and novices alike to navigate. The rise of LLMs and introduction of ChatGPT in 2022 marked a significant shift in this field, with researchers finding that it performs comparably or even surpasses APR tools. In addition, its conversational interface is user-friendly and intuitive, making it a preferred alternative to more traditional interfaces. However, the accuracy and performacne of LLMs tend to deteriorate as the project grows, making them an unsustainable option in the long term. This research will focus on leveraging LLMs to support novices by teaching them debugging skills and how to navigate debuggers effectively, acknowledging that humans still hold a clear advantage over automatic debugging tools in complex projects.

Keywords

Large Language Model, Debugging Tool, Novice Programmers

1. Introduction

As we advance toward a more technologically oriented society, computer programming has become a crucial skill that extends beyond traditional computer science roles and even into fields such as engineering and mathematics [1]. This invaluable skill is essential for developing cutting-edge technologies and creating solutions across a vast array of industries and disciplines. However, the dynamic and continuously evolving nature of technologies and software systems inevitably result in the emergence of bugs due to varying factors such as the deprecation of old features, addition of new functionalities, and code refactoring. When left undetected, these can have severe consequences and may even lead to global system failure for companies. The longer these defects remain undiscovered in the software lifecycle, the more effort and money are required to patch these [2]. These shed light on one of the most important and fundamental components of programming: debugging, which is a necessary and critical skill for all programmers.

Debugging refers to the process of localizing, understanding, and repairing any defects found in codes [3]. It is essential in ensuring that various softwares are maintained, reliable and functioning accordingly. However, it is a complex and time-consuming task that may take up to 30 to 90% of the software development process [4]. Moreover, it often causes struggle, frustration, and negative emotions which leads to many developers to view it as a tedious and daunting activity [5]. One reason for this is that it requires developers to be competent in various areas such as having a strong ability to logically comprehend code, track bugs, and implement fixes which makes it especially demanding for those with limited experience [1]. In addition to this, Whalley et al. [5] also stated that students and novices, in particular, face difficulties when debugging due to their fragile understanding of the subject. Additionally, programmers also encounter difficulties with debugging due to the absence of a systematic and methodical approach, sometimes over-relying instead on ineffective methods such as

CHIRP 2025: Transforming HCI Research in the Philippines Workshop, May 08, 2025, Baguio, Benguet, Philippines

*Corresponding author.

†These authors contributed equally.

gregory\protect\TU_charles\protect\TU_tiong@dlsu.edu.ph (G. C. K. Tiong)

trial and error or excessive use of print statements [6][5]. Despite these challenges, debugging remains an overlooked and neglected aspect of programming education, with inexperienced programmers often expected to develop this skill solely as a byproduct of learning to code. This issue is further exacerbated by a notable absence of formal learning models and a lack of clarity on effective ways to teach debugging [5]. To address these challenges, some developers rely on debugging tools to enhance efficiency and organization.

Debugging tools have evolved significantly through various generations of increasing complexity and functionality [7]. Today, some of the most widely used debugging tools are integrated into IDEs like Eclipse[1], Visual Studio Code[2], and IntelliJ IDEA[3], providing features such as step-by-step executions and breakpoints to streamline the debugging process and allow them to gain a better overall understanding of the code [8]. However, developers—especially students and novice programmers—often avoid using IDE debuggers due to usability issues and a lack of formal training [6][9]. For instance, Ko et al. (2023) [9] discovered that the biggest hindrance in the usage of a debugger is the effort required to learn and understand proper utilization of the tool. Moreover, participants noted that the usage of debuggers can increase their efficiency especially in more complex projects. Similarly, Böttcher et al. (2016) [10] also conducted a study in which students attempted to follow a structured debugging process but struggled with using the debugger, prompting researchers to provide additional guidance. These findings suggest that debuggers can be highly effective once users develop proficiency in using them. However, there is a notable gap in the literature when it comes to understanding the specific factors that make novice programmers feel overwhelmed by debuggers.

Another prominent debugging tool are automatic program repair (APR) tools and have become a prominent research area and choice for a debugging tool, with extensive studies conducted on their capabilities and performance. APR tools aim to fix bugs or errors automatically without human input. Its process involves firstly identifying suspicious code, generating possible patches, and validating them using a test suite. The final output is a repair report, which comprises of two parts: the log and patch output. The former shows the suspicious code and list of attempted patches used while the patch output shows the correct patch if it exists [11]. In addition, major corporations such as Meta and Fujitsu are actively investigating the practical applications of APR in real world environments and systems [12]. Similar to IDE-debuggers, many experienced developers and novices often avoid or struggle with these debugging tools due to their complex and confusing nature. This is supported by a study by Zhang et al. [11] that highlighted significant limitations, such as unintuitive user interfaces that hinder navigation and integration into debugging workflows. Moreover, the repair reports produced by APRs were often unclear, inaccurate, and difficult for users to interpret.

In 2022, there has been a growing exploration of Large Language Model (LLM)-based error repair methods, given that modern LLMs have shown remarkable capabilities and performance on debugging tasks [13]. Among these, OpenAI's GPT models stands out as one of the most renowned and researched LLMs and ChatGPT having significantly influenced various fields with its advanced conversational and generative AI capabilities [14]. Furthermore, these conversational LLM-based chatbots are user-friendly and easy-to-use tools that possess the capability to debug code. Although LLMs may perform comparably to or even surpass other APR techniques, both types of tools often exhibit low accuracy, making sole reliance on them ineffective for resolving all bugs [15][16][17]. Additionally, As LLMs continue to evolve and new versions are introduced, existing datasets used to evaluate their debugging performance have become less reliable due to data leakage. This occurs when evaluation problems are already included in the models' training data, which impacts performance results. As a result, there is a growing need for the development of new, carefully curated datasets to fairly and

[1]https://www.eclipse.org/
[2]https://code.visualstudio.com/
[3]https://www.jetbrains.com/idea/

accurately assess the capabilities of newer LLMs.

With these issues, this paper aims to create a debugging tool that employs an LLM to engage in conversational interactions with students, guiding and teaching them through the process of systematic debugging while also helping them navigate and understand the use of IDE-integrated debuggers. Additionally, the paper aims to create a new dataset that addresses issues in existing datasets, such as data leakage. Various LLMs will be evaluated on their debugging performance using this dataset, and the LLM with the best debugging performance will be selected for the tool based on these evaluations. It also explores the usability challenges that contribute to developers' difficulties in using traditional debugging interfaces. Overall, the paper aims to empower novices to become more effective debuggers, emphasizing the importance of human reasoning in debugging—especially as APR and LLM-based tools may yield inconsistent or inaccurate results especially with complex projects.

2. Review of Related Literature

2.1. Debugging Tools Experience

As discussed in section 1, debugging tools have evolved through various generations, with today's most common being IDE debuggers and APR tools. Despite their shared goal of aiding programmers to accelerate the debugging process, both manual and automated tools face common challenges: they are difficult to use and navigate due to their complexity and lack of formal education in utilizing these tools.

This is supported by numerous studies such as by Beller[6] wherein their research addressed a gap in understanding different developers' debugging behaviors wherein they firstly conducted a survey which revealed that 81.3% of 143 respondents use IDE debuggers, with the breakpoint feature being the most utilized. However, a field study using WATCHDOG 2.0 to monitor the overall coding behavior in Eclipse and IntelliJ showed a significant discrepancy, as 71.2% of 458 developers did not use the IDE debuggers. Additionally, interviews revealed that developers prefer printf debugging due to its universality across programming languages. Participants also noted that debuggers are difficult to use, not user-friendly, hinder self-exploration, and there is lack of education on how to operate these, as many developers are self-taught. Similarly, Zayour and Hamdar [18] conducted a qualitative study on IDE debugging, finding that developers favored features such as step execution and breakpoints. However, they also expressed a desire for more advanced debugging tools, noting that line-of-code based debugging is becoming less effective due to the growing complexity of systems and code. Shifting our focus toward novices' experiences, Böttcher et al. (2016) [10] conducted an experiment involving self-learning materials and a subsequent lecture on debugging for students. During the initial run, students were overwhelmed by the IDE debugger. Thus, the lesson and lecture were revised to include formal instruction and guidance on using the debugger, leading to improved outcomes.

Regarding APR tools, Zhang et al. (2022) [11] conducted a comparative study evaluating manual program repair without the usage of tools against three APR tools: PraPR (template-based), SimFix (heuristic-based), and ACS (constraint-based). The study found that APR tools can reduce debugging time for developers. Participants noted that APRs effectively provide fault localization, usable patches, and can accelerate the debugging process. However, the study also highlighted significant drawbacks, including non-user-friendly interfaces that complicate navigation and integration into existing debugging processes. Additionally, the repair reports generated by these tools were often confusing, inaccurate, and difficult to understand. In a similar study, Tao et al. (2014) [19] collected feedback from developers on using APRs as debugging aids. The qualitative data revealed that APRs can speed up the initial debugging process, simplify problem recognition, and offer valuable suggestions for developers to address issues independently. However, negative feedback indicated that APRs can be misleading, incomplete, and may either over analyze or generalize the debugging problems, leading to potential confusion.

Aside from these, AI programming assistants such as Copilot have already been employed and are continuously going through updates as time passes [20]. The researchers conducted a study to explore the reasons why developers choose to utilize or avoid AI-powered coding assistants. The findings revealed that the primary motivation for using these tools is efficiency, particularly through features like autocompletion and leveraging LLMs as a search engine. Conversely, the main reason for not using them is concerns over code accuracy. Additionally, participants suggested various improvements, including better handling of programs with cross-file dependencies, enhanced personalization and feedback mechanisms, as well as more advanced code analysis capabilities. Studies found that AI coding agents are primarily utilized for code completion, debugging, and code generation[21]. However, participants also reported several challenges and expectations, including poor performance in complex tasks, low accuracy in debugging, accessibility and costs.

2.2. Large Language Models (LLMs)

Large Language Models (LLMs) are advanced systems that have the ability to understand and generate human-like languages and texts using AI. These models are trained on a large amount of data and undergo numerous pre-processing steps [22]. LLMs can be categorized into three types: encoder-only models like BERT, decoder-only models like OpenAI's GPT models and LLama, and encoder-decoder models like CodeT5 which results in the research utilizing different LLM types [11]. Models that leverage an encoder-only architecture excel at tasks involving natural language understanding (NLU), while those with a decoder-only architecture are best suited for natural language generation (NLG). On the other hand, encoder-decoder models are ideal for sequence-to-sequence (Seq2Seq) tasks. Aside from this, the method of how the model is utilized in a zero-shot, one-shot, or a few-shot setting, can significantly impact the results. In zero-shot scenarios, the model is tasked with no prior examples, while in one-shot and few shot it receives one or multiple examples are provided to the model respectively. Furthermore, utilization of leveraging prompt engineering and techniques is a powerful strategy to optimize the model's performance [23]. Nevertheless, LLMs such as OpenAI's GPT models tends to struggle more and their performance deteriorates with complex tasks [24]. Another important point to note is the non-deterministic nature of LLMs in code generation, which, while introducing some variability, also fosters greater creativity [25].

Usability is a major concern in debugging tools, and because of this, ChatGPT's conversational interface is often preferred over traditional ones due to its more intuitive, human-like interactions. Studies have shown that users favor ChatGPT for its user-friendly design, its ability to explain concepts effectively, and its structured, natural dialogue outputs [26][27]. These strengths support the integration of ChatGPT into educational environments. However, a drawback of this tool is the accuracy and quality of its outputs depend heavily on how prompts are formulated. A common theme across multiple studies is the critical role of prompt design, which significantly influences the model's performance, especially in technical domains like programming and debugging[27] [16] [28].

Initial research highlights the diverse capabilities of GPT, such as personalized learning, concept explanation, essay writing, code generation, and more [14][29]. Several studies have also evaluated its effectiveness in debugging by developing various debugging datasets and benchmarks, which were used to assess the performance of GPT and other LLMs in this area. Several studies have highlighted the performance of GPT-based models in APR tasks. Sobania et al. (2023) [16] evaluated the GPT model from January 9, 2023, against two learning-based APR tools, Codex and CoCoNut, along with ten traditional APR tools, using the QuixBugs benchmark. GPT solved 19 out of 40 problems, closely matching Codex (21) and CoCoNut (19), while outperforming all traditional APR tools, which solved only 7. Interestingly, its performance improved to 31 out of 40 when human-provided hints were added, emphasizing the importance of human interaction. Similarly, Xia and Zhang (2023) [17] introduced ChatRepair, a gpt-3.5-turbo-0301-based APR system that outperformed other learning-based APR

methods across multiple benchmarks (Defects4j 1.2, 2.0, and QuixBugs). The tool resolved significantly more defects than CodexRepair and AlphaRepair, showing GPT's strong potential in automated debugging. These findings collectively underscore GPT models' robust debugging capabilities, especially when combined with human input.

While prior studies highlighted strong performance by GPT-based models on existing benchmarks (Defects4j & QuixBugs), evidence from a study by Zhang et al. (2023) [15] suggested that GPT was already familiar with various datasets and may be relying on its training data in debugging. To address this, the researchers introduced a new dataset, EvalGPTFix, which includes 151 problems. Notably, its performance improved further by solving 9 additional defects when detailed prompts were utilized which supports the importance of prompt formulaiton. Similarly, Tian et al. (2024) [30] introduced the DebugBench benchmark to address limitations found in existing datasets—such as data leakage and a lack of bug diversity—including those present in EvalGPTFix. The studied evaluated multiple open-source and closed-source models, including ChatGPT, LLaMA, and others, comparing their performance against developers with a minimum of four years of experience across 18 distinct bug types. The results indicated that experienced human debugging remained the most effective, with closed-source models performing slightly worse and open-source models achieving the lowest accuracy.

2.3. Teaching Debugging

While there is a lack of attention towards debugging education, there have been a number of studies that sought to address this through the teaching of created processes and frameworks. For instance, Böttcher et al. (2016) [10] proposed a structured approach to systematically teach debugging to students. Their method focused on a divide-and-conquer strategy for bug localization, combined with instruction on effective debugger usage. However, results revealed that majority of them decided to not utilize the approach despite being told to do so. Another example is the study by Michaeli and Roimeike (2019) [31], which employed a pre-post control group design to evaluate the impact of a systematic debugging process as an intervention for students. They discovered that the utilization of a structured approach has a positive impact on the the self-efficacy and performance of students in debugging. Similarly, Li et al. (2019) [32] connected studies on debugging based on the framework created by Jonassen & Hung (2006) [33] which is as follows:

- **Domain Knowledge:** Refers to the core theories and principles that a system's design is based on. In debugging, this corresponds to the knowledge and understanding of the programming language.
- **System/Device Knowledge:**
  - **Topological Knowledge:** Involves the visual representation of the system's structure and components, equivalent to understanding the package hierarchy, directory structure, or the structure of methods/functions within a program.
  - **Functional Knowledge:** Refers to understanding the function of system components and the behavior of their interactions, corresponds to understanding how the program functions as a whole, including the relationships between different parts or programs.
- **Procedural Knowledge:** Refers to the understanding of how to perform specific actions or tasks, such as setting up a test suite or utilizing IDE-debugger features.
- **Strategic Knowledge:**
  - **Global Strategies:** Strategies that can be applied to any type of system/program, such as forward/backward tracing or a breadth-first approach to debugging.
  - **Local Strategies:** Strategies specific to a system/program, such as printf debugging, using meaningful variable names, and adding comments.

- **Previous Experience:** Crucial factor in expertise for troubleshooting and debugging, as it enables the recognition of recurring patterns or common bugs.

Aside from this, the researchers also outlined the debugging process based on the troubleshooting model which is as follows:

1. **Construction of Mental Model:** refers to understanding the structure, components, intended logic and actual behavior of the program
2. **Identification of Discrepancies:** refers to recognizing the difference between the intended and actual behavior of the code
3. **Interpretation of Discrepancies:** refers to the creation of hypotheses on possible locations of the fault
4. **Generation and Validation of Solution:** Refers to the identification and resolution of the bug, ensuring the program's correctness, and checking for any further issues or bugs.

3. Research Questions

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

- RQ1: What usability challenges and learning barriers do novice programmers encounter when debugging code and interacting with IDE-integrated debuggers?
- RQ2: How do the latest LLMs perform on a newly constructed debugging dataset designed to address the limitations of previous datasets?
- RQ3: How can LLMs be used to teach novices debugging and how to navigate debuggers?

4. Methodology

4.1. Interviews

Interviews will be conducted either in person or online to gather a more detailed and thorough insights into participants' debugging processes and experiences with debugging and debugging tools. The interviews will also examine their interactions with LLMs for programming and debugging purposes. The responses from the interviews will undergo thematic analysis to systematically identify recurring themes and patterns. By analyzing participants' experiences and preferences through thematical analysis, the study aims to uncover key challenges and areas where a LLM-based tool can provide meaningful improvements. Additionally, the interviews could uncover important roles, insights or lessons where LLMs can provide support or teach.

4.2. Dataset Construction

To address the limitations identified in previous research, the dataset will be carefully curated to overcome these challenges. It will consist of programming problems with buggy submitted solutions of varying difficulty levels sourced from AtCoder[4], a competitive programming platform. To ensure that the problems and buggy codes are not present in the training data of the language models, only problems from contests that occurred after the models' training cutoffs will be utilized. Problems and buggy solutions will be scraped using a custom Python script leveraging and compiled into a csv file. Additionally, the dataset will be designed to cover a wide range of bug types to ensure diversity to better test the LLM's capability.

[4] https://atcoder.jp/

4.3. LLM Evaluation

After the construction of the debugging dataset, a variety of LLMs will be evaluated using it, including DeepSeek-V3, GPT-4o, Llama-4-Maverick, Quasar-Alpha, and Gemini-Exp-1206. These LLMs were selected for their top performance on the BigCodeBench dataset [34]. Based on preliminary surveys and interviews, the various LLMs will be tested using different prompting strategies: zero-shot, one-shot, or few-shot, depending on how users typically interact with them. To ensure consistency and comparability among models, the same prompts will be used for each LLM under the same evaluation condition. The output of each LLM for each problem in the dataset would then be submitted in AtCoder and will evaluate its correctness.

4.4. Prototype and Development

After the best performing LLM is determined, the framework created by Li et al. (2019) [32] will serve as one of the foundations of this tool wherein the LLM will be utilized in order to teach novices the debugging process and different types of knowledge mentioned in section 2.3 in order to effectively enhance the debugging skills of novices. In addition, the insights gained from the thematic analysis of the interviews will also inform and guide the iterative prototyping process. The process will involve creating both low- and high-fidelity prototypes using Figma. After each iteration, usability testing will be conducted online on around 5 participants in order to observe user behavior and identify any issues or areas of confusion. There will be a total of 3 iterations but this number may vary depending on the feedback and insights gathered from each round of usability testing. Additionally, a brief interview will be held post-testing to gather user feedback and insights wherein thematic analysis will be performed in order to gather common themes. This approach aims to ensure that the design and usability of the interface align with users' needs and expectations.

Once the design is finalized, the research will advance to the development phase of the tool. LangChain will be employed to seamlessly integrate different LLMs into the system. During this phase, thorough testing will be carried out to detect and fix bugs, verify core functionalities, and ensure the tool behaves as intended across various scenarios. Any issues identified during this process will be addressed to refine and optimize the tool to ensure that the final version is reliable and effective that supports the intended debugging tasks.

4.5. Pre-Post Between-Subjects Experiment

Upon completion of the tool, a pre-post between-subjects design will be employed to evaluate the tool's effectiveness in improving debugging skills. Participants will be randomly assigned to either the experimental group, which will have access to the debugging tool, or the control group, which will not have access. The study will begin with a pre-test to establish a baseline of participants' debugging skills. Both the experimental and control groups will be given the same set of C programming debugging tasks that simulate real-world coding problems. After the pre-test, the experimental group will engage with the debugging tool for a one-week intervention period, although the duration may be extended based on user feedback. In the post-test, participants will be given a set of C programming tasks of similar difficulty to the pre-test to assess their performance and determine the effectiveness of the tool in improving debugging skills. The debugging tasks will cover a wide range of bugs, ensuring a thorough evaluation of both the participants' performance and the tool's impact on enhancing their debugging abilities. In addition, the sample size for the experiment will be determined based on a Power Analysis to ensure sufficient statistical power to detect meaningful differences between the experimental and control groups.

To thoroughly evaluate the effectiveness of the tool, several metrics will be employed:

1. **Time Taken to Resolve Defects:** The amount of time each participant takes to identify and

fix each bug will be recorded which will allow the study to measure the student's debugging efficiency with and without the tool
2. **Number of Bugs Resolved:** The total number of bugs each participant successfully resolves will be counted to determine the tool's effectiveness.
3. **UMUX-lite:** After completing the tests, participants will complete a 2-item questionnaire to provide qualitative feedback on their experience with the tool.
4. **Bloom's Taxonomy:** The participants' performance will be analyzed according to Bloom's Taxonomy, which evaluates cognitive skills across six levels: remembering, understanding, applying, analyzing, evaluating, and creating. This framework will assess how well the tool facilitates the development of higher-order thinking skills, particularly in debugging tasks, and how it supports users in progressing through different levels of comprehension and problem-solving.

These metrics will be crucial for evaluating the tool's impact on the debugging process, providing a comprehensive understanding of its effectiveness in improving both efficiency and accuracy in debugging tasks.

4.6. Post Test Interview

A concluding semi-structured interview session will be conducted with each participant from the experimental group to evaluate the overall effectiveness of the debugging tool. Participants will be asked to share their experiences, insights, and feedback on various aspects of the tool, such as its usability, functionality, and impact on their debugging process and learning. Thematic analysis will also be used to examine the interview data. This final evaluation will provide valuable information on the tool's performance and will be the basis of areas for further improvement.

4.7. Scope and Limitations

This research will specifically target freshman novice programmers as the main participants of the experiment since they are the ones who struggle the most with understanding complexities of debugging as well as the usage of debugging tools. The participants must be first-year undergraduate students with no more than one year of experience in programming, ensuring that their debugging skills are in the early stages of development. Given their limited experience, they will proceed with the experiment with a fresh perspective when navigating the tool thus allowing their interactions with it to be unimpeded by any preconceived notions transferred from using other tools. The different requirements and features of the tool will be primarily based on novice's struggle and needs when debugging. As a result, experienced and seasoned developers might perceive the tool differently, given that it is designed primarily for novices.

In assessing the tool's effectiveness, this research will concentrate on C programming debugging exercises, given that C is a widely used language where novices often encounter difficulties when programming and debugging [35].

5. Significance of the Research

This study aims to contribute to the emerging field of Human-Centered AI (HCAI) and the integration of LLMs into IDEs. By addressing the challenges programmers face during debugging, this research seeks to provide valuable insights that can inform the creation of more effective tools designed to enhance programmers' learning. Additionally, it will explore in depth on how users interact with LLMs, particularly in debugging scenarios, to gain a deeper understanding of their usage patterns.

As this is a relatively new research area and AI-assisted debugging—existing literature remains limited, with much of the focus centered on ChatGPT. To bridge this gap, this study aims to evaluate

the latest top performing LLMs, assessing their effectiveness in debugging tasks. Furthermore, many existing debugging datasets present challenges such as data leakage, where LLMs may have already been trained on or exposed to these datasets. To address this issue, this research will develop a new dataset specifically designed to ensure that current LLMs do not have prior knowledge of its contents.

Overall, this study will assess how AI can teach novice programmers, enhance their learning, and improve their debugging processes. Beyond benefiting novices, the findings will also serve as a foundation for the future development of AI-driven debugging tools.

References

[1] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, C. Zander, Debugging from the student perspective, IEEE Transactions on Education 53 (2010) 390–396. doi:10.1109/TE.2009.2025266.

[2] W. E. Wong, X. Li, P. A. Laplante, Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures, Journal of Systems and Software 133 (2017) 68–94. doi:10.1016/j.jss.2017.06.069.

[3] C. Parnin, A. Orso, Are automated debugging techniques actually helping programmers?, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 199–209. doi:10.1145/2001420.2001445.

[4] A. Ang, A. Perez, A. Van Deursen, R. Abreu, Revisiting the practical use of automated software fault localization techniques, in: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2017, pp. 175–182. doi:10.1109/ISSREW.2017.68.

[5] J. Whalley, A. Settle, A. Luxton-Reilly, Novice reflections on debugging, in: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, 2021, pp. 73–79. doi:10.1145/3408877.3432374.

[6] M. Beller, N. Spruit, D. Spinellis, A. Zaidman, On the dichotomy of debugging behavior among programmers, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 572–583.

[7] R. Law, An overview of debugging tools, ACM SIGSOFT Software Engineering Notes 22 (1997) 43–47. doi:10.1145/251880.251926.

[8] A. Afzal, C. L. Goues, A study on the use of ide features for debugging, in: Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 114–117. doi:10.1145/3196398.3196468.

[9] M. Ko, D. B. Bose, H. A. Chowdhury, M. Seyam, C. Brown, Exploring the barriers and factors that influence debugger usage for students, in: 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2023, pp. 168–172. URL: https://chbrown13.github.io/papers/vlhcc23_debug.pdf. doi:10.1109/VL-HCC57772.2023.00027.

[10] A. Böttcher, V. Thurner, K. Schlierkamp, D. Zehetmeier, Debugging students' debugging process, in: 2016 IEEE Frontiers in Education Conference (FIE), IEEE, 2016, pp. 1–7. doi:10.1109/FIE.2016.7757447.

[11] Q. Zhang, Y. Zhao, W. Sun, C. Fang, Z. Wang, L. Zhang, H. ... Yang, Program repair via semantic search: A large-scale empirical study, IEEE Transactions on Software Engineering 48 (2022) 3876–3895. doi:10.1109/TSE.2021.3084157.

[12] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Y. Z. Chen, A systematic literature review on large language models for automated program repair, arXiv preprint arXiv:2405.01466 (2024). doi:10.48550/arXiv.2405.01466.

[13] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, Y. Zhang, A survey on automated program repair techniques, arXiv preprint arXiv:2303.18184 (2023). doi:10.48550/arXiv.2303.18184.

[14] M. M. Rahman, Y. Watanobe, Chatgpt for education and research: Opportunities, threats, and strategies, Applied Sciences 13 (2023) 5783. doi:10.3390/app13095783.

[15] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, 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). URL: https://doi.org/10.48550/arXiv.2310.08879. doi:10.48550/arXiv.2310.08879.

[16] D. Sobania, M. Briesch, C. Hanna, J. Petke, An analysis of the automatic bug fixing performance of chatgpt, in: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), IEEE, 2023, pp. 23–30. doi:10.1109/APR59189.2023.00012.

[17] C. S. Xia, L. Zhang, Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt, arXiv preprint arXiv:2304.00385 (2023). doi:10.48550/arXiv.2304.00385.

[18] I. Zayour, A. Hamdar, A qualitative study on debugging under an enterprise ide, Information and Software Technology 70 (2016) 130–139. doi:10.1016/j.infsof.2015.10.010.

[19] Y. Tao, J. Kim, S. Kim, C. Xu, Automatically generated patches as debugging aids: A human study, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 64–74. doi:10.1145/2635868.2635873.

[20] J. T. Liang, C. Yang, B. A. Myers, A large-scale survey on the usability of ai programming assistants: Successes and challenges, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13. doi:10.1145/3597503.3608128.

[21] X. Tan, X. Long, X. Ni, Y. Zhu, J. Jiang, L. Zhang, How far are ai-powered programming assistants from meeting developers' needs?, arXiv preprint arXiv:2404.12000 (2024). URL: https://arxiv.org/pdf/2404.12000.

[22] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, A. ... Mian, A comprehensive overview of large language models, arXiv preprint arXiv:2307.06435 (2023). URL: https://arxiv.org/pdf/2307.06435.

[23] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, D. C. ... Schmidt, A prompt pattern catalog to enhance prompt engineering with chatgpt, arXiv preprint arXiv:2302.11382 (2023). URL: https://file.mixpaper.cn/paper_store/2023/681177f8-cd15-4e0f-a23b-997c6b9f9dd2.pdf.

[24] J. Kocoń, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydło, J. Baran, P. ... Kazienko, Chatgpt: Jack of all trades, master of none, Information Fusion 99 (2023) 101861. doi:10.1016/j.inffus.2023.101861.

[25] S. Ouyang, J. M. Zhang, M. Harman, M. Wang, An empirical study of the non-determinism of chatgpt in code generation, ACM Transactions on Software Engineering and Methodology 34 (2025) 1–28. doi:10.1145/3697010.

[26] T. Sakirin, R. B. Said, User preferences for chatgpt-powered conversational interfaces versus traditional methods, Mesopotamian Journal of Computer Science (2023) 22–28. URL: https://mesopotamian.press/journals/index.php/cs/article/download/45/70.

[27] A. Shoufan, Exploring students' perceptions of chatgpt: Thematic analysis and follow-up survey, IEEE Access 11 (2023) 38805–38818. doi:10.1109/ACCESS.2023.3268224.

[28] Q. Zhang, C. Fang, Y. Ma, W. Sun, Z. Chen, A survey of learning-based automated program repair, ACM Transactions on Software Engineering and Methodology 33 (2023) 1–69. doi:10.1145/3631974.

[29] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, G. ... Kasneci, Chatgpt for good? on opportunities and challenges of large language models for education, Learning and Individual Differences 103 (2023) 102274. doi:10.1016/j.lindif.2023.102274.

[30] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, M. ... Sun, Debugbench: Evaluating debugging capability of large language models, arXiv preprint arXiv:2401.04621 (2024). URL: https://arxiv.org/pdf/2401.04621.

[31] T. Michaeli, R. Romeike, Improving debugging skills in the classroom: The effects of teaching a systematic debugging process, in: Proceedings of the 14th Workshop in Primary and Secondary Computing Education (WiPSCE), ACM, 2019, pp. 1–7. URL: https://computingeducation.de/pub/2019_Michaeli-Romeike_WIPSCE19.pdf. doi:10.1145/3361721.3362121.

[32] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, E. Tempero, Towards a framework for teaching debugging, in: Proceedings of the Twenty-First Australasian Computing Education Conference, 2019, pp. 79–86. doi:10.1145/3286960.3286970.

[33] D. H. Jonassen, W.-C. Hung, Learning to troubleshoot: A new theory-based design architecture, Educational Psychology Review 18 (2006) 77–114. doi:10.1007/s10648-006-9001-8.

[34] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, et al., Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, arXiv preprint arXiv:2406.15877 (2024).

[35] M. Heinsen Egan, C. McDonald, An evaluation of seec: a tool designed to assist novice c programmers with program understanding and debugging, Computer Science Education 31 (2021) 340–373. doi:10.1080/08993408.2020.1777034.