International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 09 | Sept - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

A Review on AI-Powered Code Debugger and Explainer

Poornima K.M.[1], Madan H K[2], Mizbah Kounain[3], Mohamad Saif[4], Mohammed Sufyan[5]
Department of CS&E
JNN College of Engineering
Shivamogga, Karnataka, India
madanhk18@gmail.com

Abstract - The AI-powered code debugging and explainer has emerged as a promising approach to assist programmers beyond traditional syntax error detection. Prior studies explore conversational debuggers, prompt-based logical error correction, neural machine translation for bug fixing, and educational tools that diagnose common student errors. However, limitations such as dependency on third-party APIs, lack of offline functionality, poor explainability of fixes, and restricted language support remain significant challenges. An AI-powered Code Debugger and Explainer aims to identify both syntax and logical errors across multiple programming languages while providing clear, human-readable explanations. By emphasizing interpretability, accessibility, and educational support, the system seeks to improve debugging efficiency and enhance conceptual understanding for students, educators, and software developers.

Key Words: logical error identification, syntax and logic analysis, large language models (LLM), AI-powered debugging, code explanation, programming education.

1. INTRODUCTION

Modern software development requires tools that go beyond syntax checking to identify and explain logical errors in code. Traditional compilers and IDEs often miss deeper issues that affect program correctness, especially for beginners. To address this, an intelligent web-based debugging and explanation platform has been developed, supporting languages like C, Java, and Python. It features a user-friendly interface with a code editor, language selector, input field, and two main functions: Run and AI Debug. While Run handles compilation and basic error reporting, AI Debug uses artificial intelligence to detect, explain, and correct logical flaws. The tool enhances learning by offering clear explanations, encouraging better coding habits, and supporting collaborative features like session sharing. Over time, it can evolve into a predictive assistant that helps users write cleaner, more efficient code.

2. LITERATURE REVIEW

The literature survey includes various studies related to AI-driven code debugging and explanation. These below works explore tools that assist developers by detecting, correcting, and explaining code errors. The insights gathered form the foundation for building an intelligent web-based debugging system.

Reference [1] introduces an approach called ChatDBG that integrates large language models into well-known debuggers such as GDB, LLDB, and Pdb to enhance debugging with natural language interaction. By allowing developers to ask questions about runtime behavior in everyday language, the debugging process becomes more intuitive and efficient, particularly for Python developers. This model helps to improves convenience for
beginner developers. However, it depends upon on third-party APIs, which results in issues related to privacy, data transmission delays, and also unpredictable performance due to its dependency on external services. Along with that, these limitations suggest the need for local LLM deployments for enterprise usage.

As shown in Ref. [2], AutoSD is introduced to simulate human-style scientific reasoning during debugging. It guides the debugging process through steps like identifying issues, hypothesizing causes, performing tests, and generating explanations all facilitated by LLMs. This approach results in deeper comprehension of the code along with it, making it especially useful in educational platforms. The transparent workflow allows developers to learn as they debug, fostering better programming habits. However, one challenge is that this technique is computationally expensive and its success heavily relies on prompt formulation, which could make it less effective for users who are unfamiliar with LLM prompt engineering.

The technique presented in [3] includes Abstract Syntax Trees (ASTs) with transformer models in order to generate detailed and human-readable error explanations. The usage of ASTs results in enabling the model to understand the code's structural and syntactic context, improving the accuracy of explanations. This helps the model distinguish between surface-level errors and deeper logical issues. Even this method performs very well compared to other traditional debugging models, these may require access to vast datasets, which may not be readily available for all programming languages. Future work could focus on making these models more data-efficient or leveraging unsupervised learning techniques to overcome dataset limitations.

In Ref. [4], a neural machine translation (NMT) approach is explored to learn bug-fixing patches from commit data on GitHub. By training on real-world code changes, the model learns how developers typically fix bugs and uses that knowledge to propose repairs for new code. This method is particularly effective for Java code and focuses on method-level changes. However, it lacks flexibility and struggles with scaling across programming languages or larger codebases. Additionally, this technique assumes clean and consistent commit histories, which may not always be the case in open-source repositories, potentially limiting training data quality.

The empirical study in [5] takes a user-centric view by analyzing how developers interact with automated fault localization (AFL)

tools. It highlights major barriers such as poor usability, weak integration with existing development tools, and skepticism about the tools' accuracy. Although AFL tools have the potential to save time and reduce cognitive load, developers may abandon them if the interface or feedback is not clear and actionable. The study emphasizes the importance of intuitive design and robust testing to improve user trust and long-term adoption. In future the systems could get benefits from this personalized error explanations which is based on their behavior or their preferences.

The reference in [6] introduces a hybrid model which iteratively corrects the logical errors by combining a correct code generator and a code editing predictor. It is particularly effective in structured educational environments like programming courses, where the types of errors are often predictable. The model's iterative nature allows it to refine its fixes over multiple passes, increasing accuracy. Despite this, the model has limited ability to generalize beyond such environments and faces difficulties when applied to more dynamic, real-world codebases. Future research could explore ways to expand this approach.

In Ref. [7], a prompt-based debugging method is implemented using CodeBERT, a pre-trained language model fine-tuned on programming data. This method, called LecPrompt, formulates prompts to guide the model in locating and fixing logical errors. It works well on artificial datasets and in controlled environments where the pattern of bugs is predictable. However, in practical scenarios involving unstructured or poorly written code, the model's effectiveness declines. Moreover, the effectiveness of this method heavily depends on the quality of the prompts, indicating a need for better prompt generation strategies or prompt-tuning frameworks to enhance real-world usability.

The study in [8] presents a debugging tool tailored for students and novice programmers. By analyzing control flow graphs and matching common error patterns, it identifies and explains frequent mistakes. This is especially helpful for learners as it provides clear explanations that provides conceptual programming knowledge. The tool has demonstrated success in academic settings but requires continuous manual updates to its rule base to remain effective. As complexity of the code increases or learners move to advanced topics, maintaining the relevance of such rule- based systems becomes a challenge. Future systems might integrate AI components to automatically update and learn the new patterns from student feedbacks.

In Ref. [9], a survey of professional developers explores their expectations and concerns regarding automated debugging tools. The findings reveal a strong interest in tools that provide intelligent suggestions and integrate smoothly into existing development environments. However, many developers also expressed distrust, citing fears that such tools may introduce incorrect fixes or obscure the debugging process. This feedback points to a crucial need for explainability and transparency in debugging tools. Tools which can not only fix bugs but can also explain the reasoning behind the fix in a developer-friendly manner are more likely to gain acceptance and be used consistently in real-world development.

A thorough review in [10] evaluates around 100 deep learning based automated program repair (APR) techniques. It categorizes the models based on their architecture, code-level granularity (for e.g., token, line or method), and the datasets they use. While the review is thorough from a research perspective, it lacks insights into the challenges of deploying these models in production environments. Many APR techniques have been validated only in controlled benchmarks and lack evaluations under real-world conditions. Bridging the gap between academic prototypes and production-ready systems remains an essential direction for future work in this area.
In Ref. [11], a tool called Bugsplainer is introduced that generates real- time bug explanations using a tuned CodeT5 model combined with AST features. This combination allows the model to produce more accurate and contextual bug descriptions, especially for Python code. It outperforms traditional models by producing clear and meaningful explanations that developers can easily comprehend. However, its limitation lies in being restricted to only certain programming languages. Future iterations of the tool could include support for additional programming languages and cross-language bug analysis to enhance versatility and utility in diverse development environments.

In Ref. [12], researchers developed a self-debugging mechanism for LLMs, where the models are capable of detecting and correcting their own output errors without additional training. This feedback loop improves the quality of generated code and reduces reliance on external validators. Such mechanisms can make AI tools more autonomous and reliable. However, the explanations these models provide for their corrections are not always clear, which can confuse users. Enhancing the interpretability of self-debugging LLMs and ensuring that their corrections align with best coding practices are important areas for future development.

The tool in [13] helps identify logical errors in student code by comparing it with inferred intentions. It uses a rule-based system to detect where the student's implementation diverges from what they likely meant to do. This proves highly beneficial in educational settings by assisting students in grasping not just the errors, but also the reasons behind them. The approach is insightful but difficult to scale, as it requires manually defining error patterns and associated intentions for every task. Using machine learning to automate the creation of these rules could enhance the system's scalability and effectiveness across various programming fields.

In Ref. [14], a user study on automated program repair (APR) tools like SimFix, TBar, and Recoder reveals that although these tools generate patches, users often find them confusing or incomplete. Developers prefer tools that offer not only repair suggestions but also clear and insightful explanations supporting those recommendations. In its absence, confidence in deploying automated patches to live code is compromised. This reveals a considerable failing in current APR tools, which tend to prioritize accuracy over user proficiency. Future tools must prioritize transparency and clarity to bridge this trust gap .

According to [15], Getafix is a machine learning tool that examines version control histories to identify common bug-fix patterns. These patterns are then delivered as live templates in IDEs, streamlining and accelerating the debugging workflow. Although this method works well for recurring bugs, it tends to be less effective when developers encounter new problems that are not documented in the version control system. Consequently, the advancement of these systems may depend on hybrid approaches that dynamically choose between the template-based and generative AI methods depending on nature of the bug.

In Ref. [16], a hierarchical approach to debugging is explored, where code is broken into smaller functional units that are analyzed independently using LLMs. This approach enhances debugging by breaking it into smaller, more manageable parts, which increases accuracy by focusing each analysis on a specific context. Nonetheless, this leads to higher computational expenses and longer processing times, particularly in large-scale projects. Ensuring a balance between performance and scalability is crucial for making these models viable in real-world production settings. Optimizations like targeted analysis and caching commonly used components may help reduce this limitation.

The method presented in [17] utilizes symbolic evaluation in combination with traditional rule-based reasoning to analyze LISP code, making it especially effective for structured educational purposes. By breaking down code behavior into symbolic steps, beginners can develop a clearer understanding of how programs execute internally. However, its scope is confined mainly to specific languages like LISP and lacks adaptability to the broader spectrum of modern multi-paradigm programming languages such as Python, Java, and JavaScript. This limitation restricts its scalability in real-world applications where diverse frameworks and syntaxes coexist. A potential enhancement would be the integration of symbolic reasoning with LLMs, thereby creating hybrid educational tools capable of blending the precision of rule-based methods with the adaptability of statistical learning. Such integration could also enable personalized learning experiences, where the debugger adjusts explanations according to the learner's level of expertise.

DeepBugs [18] is an innovative tool that leverages machine learning to detect bugs by learning semantic relationships between variable names. Its effectiveness largely depends on the quality of variable naming, performing exceptionally well when identifiers are clear and descriptive. This feature highlights the critical role of coding standards, naming conventions, and documentation in ensuring higher debugging accuracy. However, in real-world scenarios where naming conventions may be inconsistent or legacy codebases use ambiguous identifiers, the tool's performance tends to degrade. Addressing this drawback could involve combining DeepBugs with static and dynamic analysis methods to provide a more context-aware debugging framework. Additionally, augmenting it with LLM-driven semantic analysis could help the tool infer intent even when variable names are not optimally chosen, thus making it more resilient and adaptable in practical development environments.

The multi-agent system described in [19] employs multiple LLMs that collaboratively analyze code while incorporating runtime feedback to enhance bug detection and resolution. This cooperative mechanism mimics a team-based debugging process where different agents specialize in unique tasks, leading to more thorough evaluations. The system's ability to adapt to evolving code makes it particularly suitable for large-scale software projects with frequent updates. Nevertheless, such multi-agent collaboration introduces challenges, including communication overhead, synchronization issues, and the risk of producing conflicting suggestions. These drawbacks can result in longer processing times and occasional inconsistencies in recommendations. To overcome these challenges, future research should focus on optimizing inter-agent communication protocols, reducing latency, and establishing a consensus-driven mechanism for decision-making. Such improvements would make multi-agent systems not only powerful but also practical for everyday software development workflows.

AGDebugger [20] represents a significant advancement in interactive debugging by allowing users to directly guide and monitor multi-agent LLMs during the debugging process. Its visual feedback and control mechanisms enable developers to influence the decision-making process, thereby maintaining transparency and trust in AI-driven systems. This interactive approach empowers developers to strike a balance between automated suggestions and human judgment. However, its current design demands substantial technical knowledge, making it less accessible to beginners and non-experts. To broaden its usability, AGDebugger could be enhanced with simplified user interfaces, step-by-step guidance modules, and customizable difficulty levels for explanations. Such enhancements would democratize access to advanced AI debugging systems, making them beneficial not only to professionals but also to students, educators, and self-learners aiming to improve their programming skills.

The key findings from the above literature review are summarized in Table 1.

TABLE-1 : LITERATURE SURVEY SUMMARY

Authors	Title	Methodology	Remarks
K. H. Levin et al. [1] 2024	ChatDBG: Augmenting Debugging with Large Language Models	Integrating LLMs into traditional debuggers (GDB, LLDB, Pdb) for conversational debugging	High effectiveness in debugging Python; limited by LLM dependency and privacy concerns
S. Kang et al [2], 2024	Explainable Automated Debugging via Scientific Debugging (AutoSD)	Uses LLMs to simulate human debugging with explainable patch suggestions	Improves transparency; computationally expensive and dependent on prompt engineering
S. Chakraborty, B. Ray. [3], 2022	Explaining Software Bugs Leveraging Code Structures	Uses AST-aware transformer models for bug explanation in natural language	Outperforms prior models; requires large datasets for training
M. Tufano et al. [4] 2019	Learning Bug-Fixing Patches in the Wild via NMT	Trains NMT model on GitHub commits to learn bug-fix pairs	Good results; limited to Java and method-level granularity
C. Parnin, A. Orso. [5] 2011	Are Automated Debugging Techniques Actually Helping Programmers?	Empirical study of developer interaction with fault localization tools	Reveals usability gaps; small participant sample
T. Matsumoto et al. [6] 2021	Iterative Trials for Correcting Logic Errors in Source Code	Combines Correct Code Model and Editing Operation Predictor for iterative logic correction	Strong educational value; limited scalability beyond structured tasks
Z. Xu, V. Sheng. [7], 2024	LecPrompt: Logical Error Correction with CodeBERT	Prompt-based logic bug repair using MLM with CodeBERT	Efficient and accurate; limited to synthetic datasets
A. M .Zin et al. [8], 2000	A Knowledge-Based Automated Debugger in Learning Systems	Matches student code with bug patterns using knowledge base and flow graph parsing	Strong pedagogical intent; limited generalization
P. S. Kochhar et al. [9] 2016	Practitioners' Expectations on Automated Fault Localization	Global survey of expectations and concerns about AFL adoption in industry	Highlights real-world tool gaps; excludes post-2015 literature

Q. Zhang et al. [10], 2023	A Survey of Learning-based Automated Program Repair	Reviews 112 studies on APR using deep learning	Excellent summary of APR research; lacks practical deployment discussion

P. Mahbub et al. [11] 2023	Bugsplainer: Code Structures + NMT for Bug Explanation	Generates bug explanations using fine-tuned CodeT5 and ASTs	Real-time utility; currently supports only Python
X. Chen et al. [12], 2023	Teaching Large Language Models to Self-Debug	LLMs explain and revise their own code outputs through self-feedback loops	No training required; sensitive to explanation quality
W. L. Johnson, E Soloway. [13], 1984	Intention-Based Diagnosis of Programming Errors (PROUST)	Diagnoses student logic errors based on inferred intentions	High educational value; complex rule creation
H. Eladawy et al. [14] 2024	APR: What Is It Good For?	User study of three APR tools (Recoder, SimFix, TBar) in debugging Java projects	Strong validity; participants debug unfamiliar code
J. Bader et al. [15], 2019	Getafix: Learning to Fix Bugs Automatically	Learns bug-fix patterns from VCS and applies them in real-time	Suitable for IDEs; struggles with novel cases
Y. Shi et al. [16], 2024	From Code to Correctness: Hierarchical Debugging with LLMs	Breaks code into subfunctions for isolated testing and AI-based debugging	Modular and precise; high computational overhead
W. R. Murray. [17], 1984	Heuristic and Formal Methods in Automatic Program Debugging	Uses symbolic evaluation and heuristics for LISP- based student code debugging	Useful in education; language-specific (LISP)
M. Pradel, K. Sen. [18] 2018	DeepBugs: Name-based Bug Detection with Learning	Uses embeddings to detect naming-based semantic bugs in JavaScript	Learns from names; fails if variables are poorly named
N. Ashrafi et al. [19] 2025	Enhancing LLM Code Generation with Multi-Agent + Runtime Debugging	Combines multi-agent LLM collaboration with runtime feedback	Comprehensive; long execution time and inconsistent performance across models
W. Epperson et al. [20] 2025	Interactive Debugging and Steering of Multi-Agent AI Systems	AGDebugger allows visualization and intervention in multi-agent LLM workflows	Innovative interaction; requires deep user understanding and may produce non-deterministic results

3. CONCLUSIONS

The AI-Powered Code Debugger and Explainer offers several advantages for improving the coding experience, particularly in educational and development environments. Key benefits include enhanced error detection, clear logical error identification, interactive learning through code explanations, real-time correction suggestions, and support for multiple programming languages. The system promotes efficiency, reduces debugging time, aids conceptual clarity, and serves as an intelligent assistant for both novice and experienced programmers.

REFERENCES

[1] *K. H. Levin, N. Van Kempen, E. D. Berger, and S. N. Freund, "ChatDBG: Augmenting Debugging with Large Language Models," arXiv preprint arXiv:2403.16354, Mar. 2024. [Online].Available: https://arxiv.org/abs/2403.16354*

[2] *S.Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable Automated Debugging via Large Language Model-driven Scientific Debugging," Empirical Software Engineering, vol. 29, no.1,Article 5, 2024. [Online].Available: https://link.springer.com/article/10.1007/s10664-024-10594-x*

[3] *S. Chakraborty and B. Ray, "Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation," in Proc. 44th Int. Conf. on Software Engineering (ICSE), 2022, pp. 1466–1478. doi: 10.1145/3510003.3510051*

[4] *M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," ACM Trans. Softw. Eng. Methodol., vol. 28, no.4, Art. no. 19, Sep. 2019. doi: 10.1145/3340544*

[5] *C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?," in Proc. Int. Symp. on Software Testing and Analysis (ISSTA), Toronto, ON, Canada, Jul. 2011, pp. 199–209. doi: 10.1145/2001420.2001445*

[6] *T. Matsumoto, Y. Watanobe, and K. Nakamura, "A Model with Iterative Trials for Correcting Logic Errors in Source Code," Appl. Sci., vol. 11, no. 11, p. 4755, May 2021. doi: 10.3390/app11114755*

[7] *Z. Xu and V. S. Sheng, "LecPrompt: A Prompt-Based Approach for Logical Error Correction with CodeBERT," arXiv preprint arXiv:2410.08241, Oct. 2024. [Online]. Available: https://arxiv.org/abs/2410.08241*

[8] *A. M. Zin, S. A. Aljunid, Z. Shukur, and M. J. Nordin, "A Knowledge-Based Automated Debugger in Learning System," in Proc. Int. Workshop on Automated and Algorithmic Debugging (AADEBUG), 2000, Universiti Kebangsaan Malaysia.*

[9] *P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in Proc. 25th ACM Int. Symp. Software Testing and Analysis (ISSTA), Saarbrücken, Germany, Jul. 2016, pp. 165–176. [Online].Available: https://dl.acm.org/doi/10.1145/2931037.2931051*

[10] *Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A Survey of Learning-based Automated Program Repair," ACM Trans. Softw. Eng. Methodol., vol. 32, no. 3,Article 1, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/2931037.2931051*

[11] *P. Mahbub, M. M. Rahman, O. Shuvo, and A. Gopal," Bugsplainer: Leveraging Code Structures to Explain Software Bugs with Neural Machine Translation," in Proc. 2023 IEEE Int. Conf. Software Maintenance*

[12] *X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching Large Language Models to Self-Debug," arXiv preprint arXiv:2304.05128, Apr. 2023. [Online]. Available: https://arxiv.org/abs/2304.05128*

[13] *W. L. Johnson and E. Soloway, "Intention-Based Diagnosis of Programming Errors," in Proc. AAAI-84: National Conference on Artificial Intelligence, Austin, TX, USA, Aug. 1984, pp. 162–168. [Online]. Available: https://aaai.org/papers/00162-aaai84-002-intention-based-diagnosis-of-programming-errors/*

[14] *H. Eladawy, C. Le Goues, and Y. Brun, "Automated Program Repair, What Is It Good For? Not Absolutely Nothing!," in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE), Lisbon, Portugal, Apr.2024, 1–13. [Online]. Available: https://doi.org/10.1145/3597503.3639095*

[15] *J. Bader, A. Scott, M. Pradel, and S. Chandra, , "Getafix: Learning to Fix Bugs Automatically," Proceedings of the ACM on Programming Languages, vol. 3, no. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3360585*

[16] *Y. Shi, S. Wang, C. Wan, and X. Gu, "From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging," arXiv preprint, arXiv:2410.01215, Oct. 2024. [Online]. Available: https://arxiv.org/abs/2410.01215*

[17] *W. R. Murray, "Heuristic and Formal Methods in Automatic Program Debugging," Department of Computer Sciences, University of Texas at Austin, Technical Report, 1984.*

[18] *M. Pradel and K. Sen, "DeepBugs: A Learning Approach to Name-based Bug Detection," arXiv preprint, arXiv:1805.11683, May 2018.[Online].Available: https://arxiv.org/abs/1805.11683*

[19] *N. Ashrafi, S. Bouktif, and M. Mediani, "Enhancing LLM Code Generation: A Systematic Evaluation of Multi-Agent Collaboration and Runtime Debugging for Improved Accuracy, Reliability, and Latency," arXiv preprint, arXiv:2505.02133, May 2025. [Online]. Available: https://arxiv.org/abs/2505.02133*

[20] *W. Epperson, G. Bansal, V. Dibia, A. Fourney, J. Gerrits, E. Zhu, and S. Amershi, "Interactive Debugging and Steering of Multi- Agent AI Systems," in Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '25), Yokohama, Japan, Apr.–May 2025, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3706598.3713581*