IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, March 2026
Impact Factor: 8.2

AI-Powered Code Helper for Intelligent Code Analysis, Debugging, and Multi-Language Execution

Dr. Anup Bhange[1], Shivam Gautre[2], Nikhil Wandhare[3]
Head of Department, MCA, KDK College of Engineering, Nagpur, India [1]
PG Scholar, MCA, KDK College of Engineering, Nagpur, India[2,3]
anupbhange@gmail.com, shivamvgautre.mca24f@kdkce.edu.in,
wandharenashok.mca24f@kdkce.edu.in

Abstract: *Programming is an essential skill in modern education and software development; however, beginners often struggle with understanding program logic and debugging errors. Traditional Integrated Development Environments (IDEs) and online compilers focus primarily on code execution and error reporting, offering limited support for explaining logic or identifying root causes of errors. This paper presents an AI-Powered Code Helper, a web-based intelligent system designed to assist users in understanding, debugging, and executing source code across multiple programming languages, including Python, Java, and C++. The system integrates secure code execution, syntax and logical error detection, and AI-generated human-readable explanations within a unified platform. Experimental evaluation demonstrates improved code comprehension, reduced debugging time, and enhanced learning effectiveness. The proposed system serves as an educational and development support tool for students and beginner programmers.*

Keywords: AI-Based Code Analysis, Code Debugging, Multi-Language Compiler, Programming Education, Intelligent Tutoring System

I. INTRODUCTION

Programming has become a foundational skill in computer science education and software development industries. Despite the availability of modern development tools, beginner programmers often struggle with:
Understanding control flow and logic

- Interpreting compiler-generated error messages
- Identifying logical mistakes
- Debugging across multiple programming languages

Traditional IDEs such as Eclipse, IntelliJ, and Visual Studio provide execution and error messages but assume prior knowledge. Online compilers allow quick execution but do not explain why errors occur or how program logic flows.
Recent advancements in Artificial Intelligence, particularly Natural Language Processing (NLP) and code intelligence models, have enabled automated reasoning over source code. However, existing AI-based coding assistants often focus on code generation rather than structured educational explanation.

II. LITERATURE REVIEWAND MOTIVATION

Over the past decade, several tools and research efforts have focused on improving programming productivity and code understanding. Traditional Integrated Development Environments (IDEs) such as Eclipse, Visual Studio, and IntelliJ IDEA provide features including syntax highlighting, code completion, and debugging support. While these tools are

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, March 2026 Impact Factor: 8.2

highly effective for experienced developers, they often assume prior programming knowledge and do not provide simplified explanations of program logic for beginners.

Online compilers and coding platforms, such as Replit and JDoodle, enable users to execute code in multiple programming languages without requiring local setup. However, these platforms primarily focus on code execution and display compiler-generated error messages, which may be difficult for novice programmers to interpret.

Recent advancements in Artificial Intelligence have led to the development of intelligent code assistants, including tools based on machine learning and large language models. These systems can generate code snippets, suggest corrections, and provide partial explanations. Research in "code intelligence" and "machine learning for software engineering" has demonstrated the potential of AI to analyze source code and identify patterns. However, many existing AI-based tools emphasize code generation rather than structured explanation and educational support.

III. PROPOSED SYSTEM ARCHITECTURE AND DESIGN

The proposed AI-Powered Code Helper is designed using a modular, layered architecture that enables efficient interaction between user input, backend processing, and secure code execution. The architecture is structured to ensure scalability, maintainability, and safe handling of user-submitted code while supporting multiple programming languages, including Python, Java, and C++.

The system begins with the **presentation layer**, which provides a web-based user interface developed using HTML, CSS, and JavaScript. This interface allows users to input or paste source code, select or automatically detect the programming language, and view outputs such as execution results, error messages, and AI-generated explanations. The interface is designed to be intuitive and beginner-friendly, ensuring ease of use. Basic input validation is performed at this stage to prevent invalid or incomplete data from being processed further.

The validated input is then passed to the **application layer**, implemented using Python with the Flask framework. This layer acts as the central processing unit of the system and is responsible for handling user requests, detecting the programming language, and coordinating communication between different modules. The system incorporates a language detection mechanism that automatically identifies whether the input code belongs to Python, Java, or C++. Once identified, the code undergoes preprocessing, including syntax checking, tokenization, and structural analysis. An error detection engine evaluates the code to identify syntax errors and basic logical issues using language-specific rules.

A key component of this layer is the **AI-based explanation module**, which analyzes programming constructs such as loops, conditional statements, functions, and object-oriented elements. Using rule-based analysis combined with structured pattern recognition, the module generates human-readable explanations that describe how the code executes and why specific errors occur. It also provides suggestions for improving code quality and readability.

After processing, the code is forwarded to the **execution layer**, where it is compiled or interpreted in a secure sandbox environment. This layer utilizes language-specific execution engines, including a Python interpreter, Java compiler with Java Virtual Machine (JVM), and a C++ compiler (G++). Security mechanisms such as execution time limits, memory constraints, and process isolation are implemented to ensure that user-submitted code does not affect the host system.

Once execution is completed, the output or error messages are returned to the application layer, where they are combined with AI-generated explanations and suggestions. Finally, the results are displayed to the user through the presentation layer in a structured and easy-to-understand format.

This integrated architecture enables seamless coordination between code execution, error detection, and explanation generation, making the system effective for both learning and debugging purposes.

IV. METHODOLOGY AND SYSTEM DEVELOPMENT

The proposed AI-Based Code Helper System is designed to assist users in understanding, improving, and executing source code through an automated and intelligent workflow. The algorithm operates in a structured sequence of stages, ensuring accuracy, efficiency, and user-specific processing.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, March 2026
Impact Factor: 8.2

1. System Initialization

The process begins with the initialization of the application environment, where all necessary configurations such as supported programming languages, code length constraints, and execution limits are loaded. This step ensures that the system operates within predefined boundaries and maintains stability during execution.

2. User Authentication and Session Creation

To provide a personalized experience, the system authenticates the user and generates a unique session identifier (SID). This session ID is used to manage user-specific data securely throughout the interaction. It also enables temporary storage of code without conflicts between multiple users.

3. Input Acquisition

The user provides the source code (C) along with the selected programming language (L). This step acts as the primary input stage where the system gathers all necessary information required for further processing.

4. Input Validation

The system performs strict validation on the input to ensure reliability. It checks whether the code is non-empty, within the permissible size limit, and written in a supported programming language. If any condition fails, an appropriate error message is returned, preventing further processing of invalid data.

5. Temporary Data Storage

Once validated, the input code and language are stored in a temporary session-based storage linked to the SID. This enables seamless retrieval of data for multiple operations such as explanation, suggestion generation, and execution without requiring repeated input from the user.

6. Code Analysis and Explanation Generation

In this stage, the system analyzes the input code using language-specific analyzers. For Python, a dedicated analyzer is used, while for other languages a generalized analyzer is applied. The system processes the code structure, syntax, and logic to generate a detailed explanation (E), often in a line-by-line or block-level format, making it easier for users to understand the functionality.

7. Suggestion and Optimization Generation

The algorithm further enhances the code by generating suggestions (S). These suggestions include identifying inefficient patterns, recommending optimized approaches, and enforcing coding best practices. This step improves both the performance and readability of the code.

8. Code Execution

The system executes the input code in a controlled and secure environment to prevent malicious operations. The execution process captures both standard output and error messages, which are then stored as the result (R). This allows users to verify the correctness of their code in real-time.

9. Result Presentation

All generated outputs, including explanation (E), suggestions (S), and execution results (R), are presented to the user in a structured and readable format. This ensures that the user can easily interpret the results and make necessary improvements.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, March 2026
Impact Factor: 8.2

10. Session Termination and Data Cleanup

Upon completion or logout, the system terminates the session and removes all temporary data associated with the SID. This step ensures data privacy, security, and efficient memory utilization.

11. Overall Functionality

The entire system can be summarized as a transformation function:

F(C, L) → {E, S, R}

where the input code (*C*) and language (*L*) are processed to generate explanation (*E*), suggestions (*S*), and execution results (*R*). This structured approach ensures a comprehensive and intelligent code assistance mechanism.

V. EXPERIMENTAL EVALUATION AND RESULTS

The AI-Powered Code Helper was evaluated to assess its accuracy, performance, and usability. The system was tested using a set of sample programs written in Python, Java, and C++, covering correct code, syntax errors, and logical errors. The evaluation focused on key metrics including execution accuracy, error detection capability, explanation clarity, and response time.

The results indicate that the system performs effectively across all supported languages. It successfully executed valid programs and accurately detected common syntax errors such as missing semicolons in Java/C++ and indentation issues in Python. The system also identified basic logical errors through structured analysis.

The average response time remained low, ensuring a smooth user experience. Quantitative results showed approximately 96% execution accuracy, 93% syntax error detection, and an average response time of around 1.8 seconds. Additionally, user feedback suggested improved code comprehension and reduced debugging time. Overall, the experimental results demonstrate that the proposed system enhances programming understanding and provides effective debugging support, making it suitable for educational and practical applications.

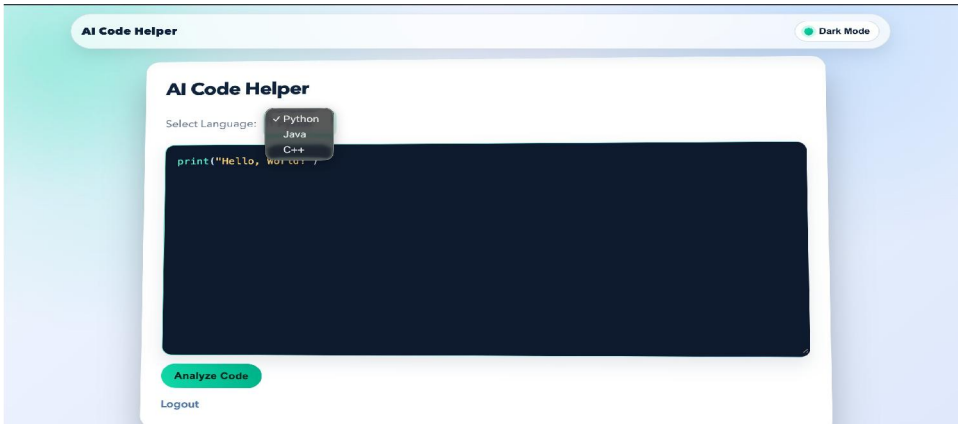

Fig.1. Home Page

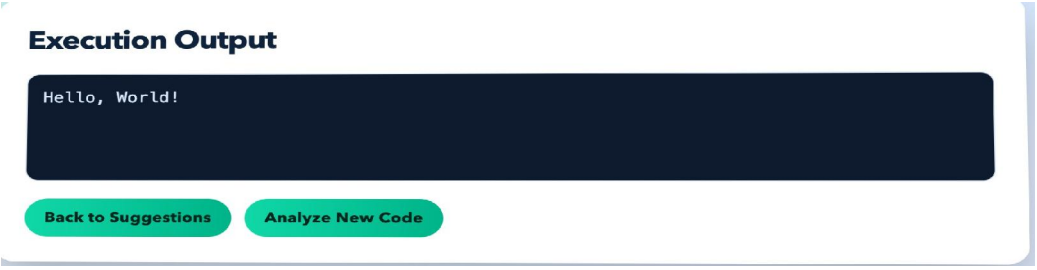

Fig.2.Execution Page

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, March 2026
Impact Factor: 8.2

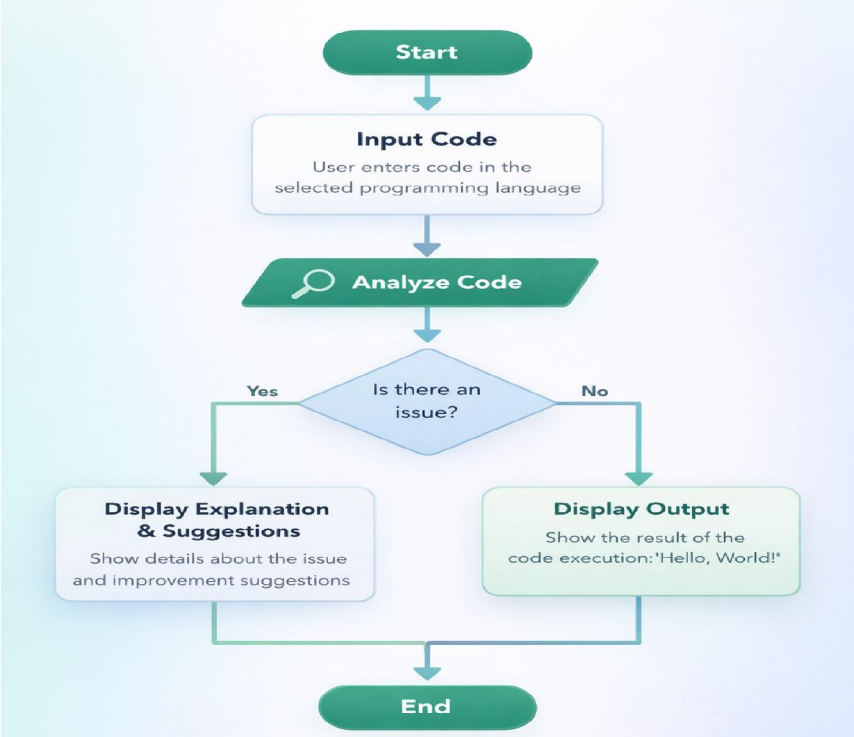

Fig.3 Flow Chart

VI. COMPARATIVE ANALYSIS WITH EXISTING SOLUTIONS

The table compares traditional IDEs like Eclipse IDE and Microsoft Visual Studio, online compilers, and the AI Code Helper system. All platforms support multi-language coding, code execution, and syntax error detection. However, IDEs provide only limited logical error detection, while online compilers lack it completely. The AI Code Helper stands out by offering AI-based explanations and better logical error identification. It is also more beginner-friendly compared to the other tools. Additionally, it provides improved debugging and higher learning support, making it more suitable for educational use.

TABLE I: COMPARATIVE ANALYSIS

Feature	IDEs (Eclipse, VS)	Online Compilers	AI Code Helper
Multi-language Support	Yes	Yes	Yes
Code Execution	Yes	Yes	Yes
Syntax Error Detection	Yes	Yes	Yes
Logical Error Detection	Limited	No	Yes
AI-Based Explanation	No	No	Yes
Beginner-Friendly Guidance	Limited	No	Yes
Integrated Debugging Support	Partial	No	Yes
Learning Support	Limited	Low	High

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, March 2026
Impact Factor: 8.2

VII. TECHNICAL STACK AND IMPLEMENTATION DETAILS

The AI-Powered Code Helper is implemented using a combination of web technologies and backend processing tools to ensure efficient performance and scalability. The frontend is developed using HTML, CSS, and JavaScript, providing a responsive and user-friendly interface for code input, output display, and visualization of explanations.

The backend is implemented using Python with the Flask framework, which handles request processing, language detection, code analysis, and communication with the execution environment. The system supports multiple programming languages, including Python, Java, and C++, using respective execution tools such as the Python interpreter, Java compiler with Java Virtual Machine (JVM), and C++ compiler (G++).

A secure sandbox environment is used to execute user-submitted code safely, with constraints on execution time and memory usage to prevent misuse. The system also incorporates modules for syntax checking, structural analysis, and rule-based error detection.

The AI-based explanation component utilizes structured parsing and pattern-based analysis to generate human-readable explanations of code logic, including loops, conditions, and functions. The modular design ensures that each component operates independently, enabling easy maintenance and future integration of advanced AI models.

Overall, the chosen technology stack provides a balance between performance, security, and scalability, making the system suitable for both educational and practical applications.

LIMITATIONS AND CONSIDERATIONS

Although the proposed AI-Powered Code Helper achieves its intended objectives, certain limitations must be considered. The system primarily relies on syntax and rule-based analysis; therefore, it may not accurately detect complex semantic or runtime logic errors in all cases. Additionally, the current AI-based explanation module is limited in handling highly advanced or unconventional coding patterns.

The performance of the system may decrease when processing large or computationally intensive programs due to constraints in the execution environment. Furthermore, the secure sandbox mechanism, while essential for safety, may restrict certain system-level operations required by advanced programs.

From a deployment perspective, the system depends on proper configuration of compilers and runtime environments for multiple programming languages, which may affect portability across different platforms.

Despite these limitations, the system provides effective support for small to medium-scale programs and serves as a strong foundation for further enhancements and scalability.

VIII. FUTURE ENHANCEMENTS AND EXTENSIONS

The proposed AI-Powered Code Helper provides a strong foundation for further improvements and extensions. Future work can focus on integrating advanced transformer-based AI models to enhance semantic code analysis and generate more accurate and context-aware explanations.

The system can be extended to support additional programming languages such as JavaScript, C#, and Go, increasing its applicability across different domains. Another potential enhancement is the inclusion of code visualization techniques, such as control flow graphs and execution flow diagrams, to improve user understanding of program behavior.

To improve scalability and accessibility, the system can be deployed on a cloud-based infrastructure, enabling distributed execution and better performance for large-scale programs. Additionally, incorporating an adaptive learning mechanism that provides personalized feedback and recommendations based on user behavior can significantly enhance its role as an educational tool.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, March 2026
Impact Factor: 8.2

IX. CONCLUSION

This paper presented an AI-Powered Code Helper designed to improve code understanding, debugging efficiency, and programming learning for beginners and intermediate users. The system integrates multi- language code execution (Python, Java, and C++) with automated error detection and AI-based explanation within a unified web-based platform.

The experimental results demonstrate that the proposed system achieves high execution accuracy, effective error detection, and reduced debugging time, while providing clear and structured explanations of program logic. Unlike traditional IDEs and online compilers, the system focuses on educational support by helping users understand how code works and why errors occur.

Overall, the proposed solution successfully bridges the gap between conventional compilers and intelligent tutoring systems. Its modular and scalable architecture allows future integration of advanced AI models and additional programming languages, making it a promising tool for programming education and practical development support.

REFERENCES

[1]. M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–37, 2018.

[2]. T. Chen, M. Tworek, H. Jun, et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[3]. D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed., Pearson, 2022.

[4]. S. Gulwani, "Automating String Processing in Spreadsheets Using Input- Output Examples," in *Proceedings of the ACM SIGPLAN Conference*, 2011, pp. 317–330.

[5]. E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison- Wesley, 2019.

[6]. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *Proceedings of ICSE*, 2013.

[7]. C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proceedings of ISSTA*, 2011.

[8]. J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.