

CompileX: An Intelligent Online Compiler with AI Error Resolution

Ms. Kimaya Ingale¹ Ms. Renu Meshram² Ms. Pooja Patil³ Prof. Aritri Sen⁴

^{1,2,3,4}Department of Artificial Intelligence

^{1,2,3,4}Usha Mittal Institute of Technology, SNDT Women's University, Mumbai, India

Abstract — Online compilers have reduced the installation barriers associated with programming education, but conventional compiler interfaces generally expose technical diagnostics without contextual guidance for novice programmers. This paper presents CompileX, an intelligent online compiler that integrates Artificial Intelligence (AI) for error resolution, contextual code explanation, conversational programming assistance, and cross-language code translation. CompileX implements four primary AI modules: (i) an AI Error Resolution Engine that analyzes compiler and runtime errors to generate simplified explanations and debugging guidance, (ii) an AI Chat Panel that provides context-aware conversational assistance using source-code context, (iii) an AI Code Translator supporting translation across five programming languages, and (iv) an Explain This Code module that provides selection-based explanations of source-code segments. The system uses a React.js frontend with Vite, a Flask backend, GCC, G++, JDK, Python, and Node.js runtimes, and an LLM API for AI-assisted code understanding. Evaluation across 500 compilation scenarios reports an overall error explanation accuracy of 91.1%, a 64.7% reduction in average debugging time compared with a compiler-only baseline, and a mean AI response latency of 2.83 seconds. A user evaluation with 45 participants also reports positive scores for explanation clarity, debugging helpfulness, and overall satisfaction. The results indicate that integrating AI assistance directly into a multi-language compiler environment can support error comprehension and reduce debugging effort while preserving conventional compilation and execution workflows.

Keywords: Artificial Intelligence, Online Compiler, Error Resolution, Large Language Models, Programming Education, Web-Based IDE, Code Translation, Debugging Assistance

I. INTRODUCTION

The digital transformation of software development has made programming an essential skill in all disciplines, from engineering and science to business and humanities [1]. As introductory programming courses experience unprecedented enrollment growth, educators and tool developers face the dual challenge of providing scalable support while maintaining pedagogical effectiveness. Online compilers have emerged as critical infrastructure for programming education, eliminating the installation barriers that historically prevented widespread participation in computing courses [2]. Platforms such as OnlineGDB, Ideone, and Replit have enabled millions of students to write, compile, and execute code directly within web browsers, fundamentally democratizing access to programming environments. However, a critical limitation persists in existing online compiler architectures: they faithfully reproduce compiler and runtime error messages without providing interpretive assistance [3]. For novice programmers, these error messages represent a formidable barrier. Cryptic syntax

error notifications, obscure runtime exceptions, and technical jargon-filled diagnostics frequently overwhelm beginners who lack the experience to map these messages to actionable debugging steps.

The remainder of this paper is organized as follows: Section II presents the research gap identified from existing programming and AI-assisted compiler systems. Section III presents the research objectives of the proposed CompileX system. Section IV provides the literature review of existing approaches related to AI-assisted programming, debugging, and compiler support. Section V describes the proposed methodology, including the system architecture and workflow. Section VI presents the implementation details, including the frontend, backend, compiler integration, AI service integration, and AI-assisted programming modules. Section VII presents the results and discussion, including the experimental evaluation of error explanation accuracy, debugging time reduction, user experience, and AI response latency. Section VIII concludes the paper by summarizing the proposed system, its key findings, and its potential for further development and evaluation.

II. RESEARCH GAP

Analysis of existing literature reveals several specific research gaps that CompileX addresses:

- 1) **Language Scope Limitation:** The reviewed compiler-integrated AI literature includes systems evaluated in specific language environments, including C++-focused compiler assistance [13], while other approaches are primarily evaluated in CS1 programming contexts [1], [2], [10]. CompileX extends AI error resolution across five programming languages: Python, Java, C, C++, and JavaScript.
- 2) **Feature Fragmentation:** Existing AI-assisted programming approaches address capabilities such as compiler-error explanation, conversational debugging, and programming assistance separately [1], [2], [10], [11]. CompileX combines four complementary AI modules within the same compiler workflow.
- 3) **Selection-Based Explanation Gap:** The reviewed literature does not identify an online compiler implementation that combines compiler execution with context-aware explanation of user-selected code portions. CompileX provides selection-based explanations intended to support targeted understanding of specific code segments.
- 4) **Cross-Language Translation Integration:** Code translation between programming languages has been explored as an AI-assisted capability, but the reviewed compiler-focused approaches do not combine this functionality with the multi-language compilation and debugging workflow described in CompileX.
- 5) **Context-Aware Chat Integration:** Existing compiler-integrated conversational systems demonstrate the value of providing programming context during debugging [1],

[10]. CompileX implements source-code context injection for chat queries so that conversational assistance remains tied to the program being developed.

III. OBJECTIVES

The primary objectives of this research are:

- 1) To design and implement an intelligent online compiler with deeply integrated AI capabilities across multiple programming languages.
- 2) To develop an AI Error Resolution Engine that parses compiler and runtime errors to generate beginner-friendly explanations, root cause analysis, and debugging guidance.
- 3) To implement a context-aware AI Chat Panel that automatically injects source code context to enable conversational programming assistance.
- 4) To create an AI Code Translator supporting translation between Python, Java, C, C++, and JavaScript while attempting to preserve program logic and functionality.
- 5) To implement an Explain This Code module providing selection-based line-by-line explanations for user-selected code portions.
- 6) To evaluate the system's effectiveness in improving debugging efficiency, error comprehension, and learning outcomes for novice programmers.
- 7) To demonstrate that integrated AI-assisted compilation environments can significantly reduce barriers for beginning programmers while maintaining pedagogical effectiveness.

IV. LITERATURE REVIEW

Several existing studies have explored Artificial Intelligence for programming education, debugging assistance, compiler error explanation, and intelligent programming support. Table I summarizes the selected studies relevant to CompileX.

Author/Year	Study and Key Contribution	Limitation
SyntaxMentor Team, 2025	SyntaxMentor: An AI-Augmented Debugging Assistant for Novice Programmers. Provides AI-based syntax error explanations, debugging guidance, and beginner-friendly programming assistance.	Limited programming environments and lacks broader multi-language integration.
D. Kohen-Vacs, M. Usher, and M. Jansen, 2025	Integrating Generative AI into Programming Education. Explores generative AI for code generation, debugging, concept explanation, and interactive learning.	Excessive AI dependency may reduce independent problem-solving abilities.
M. Hoq et al., 2025	Automated Identification of Logical Errors in Programs. Uses intelligent techniques to identify logical errors by analyzing program behavior and execution patterns.	Limited support for syntax and runtime error explanation.
J. Renzella, A. Vassar, L. L. Solano, and A. Taylor, 2025	Compiler-Integrated, Conversational AI for Debugging CS1 Programs. Integrates conversational AI with compilers to assist beginners in debugging and understanding compiler errors.	Limited evaluation across programming languages and restricted scalability.
G. Akçapınar and E. Sidan, 2024	AI Chatbots in Programming Education: Guiding Success or Encouraging Plagiarism. Examines AI chatbots for guidance, code explanation, and debugging assistance.	Raises concerns about academic integrity and over-dependence on AI-generated solutions.
E. A. Santos and B. A. Becker, 2024	Not the Silver Bullet: LLM-Enhanced Programming Error Messages are Ineffective in Practice. Evaluates AI tools for explaining compiler errors and supporting beginner debugging.	Performance varies with the complexity and type of compiler errors.
A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, 2006	Compilers: Principles, Techniques, and Tools. Provides the theoretical foundation of compiler construction, including analysis, parsing, code generation, optimization, and error handling.	Focuses on traditional compiler theory and does not address AI-based debugging assistance.

Table I: Literature Review of Ai-Based Programming and Compiler Assistance

V. COMPILEX METHODOLOGY

A. System Architecture

CompileX employs a three-tier architecture comprising a presentation layer, application layer, and service layer, as illustrated in Fig. 1. The Presentation Layer comprises the React.js frontend with TypeScript, implementing a component-based UI architecture. The frontend includes the Monaco-based code editor, language selector, execution controls, output console, AI Explanation Panel, AI Chat

Panel, and code translation interface. The Application Layer consists of the Flask backend server implementing RESTful API endpoints.

This layer orchestrates code execution, error detection, AI service communication, and response aggregation. The Flask application exposes endpoints including /compile for code execution, /ai/chat for conversational assistance, /ai/translate for code translation, and /ai/explain for selection-based explanation. The Service Layer encompasses two primary service categories:

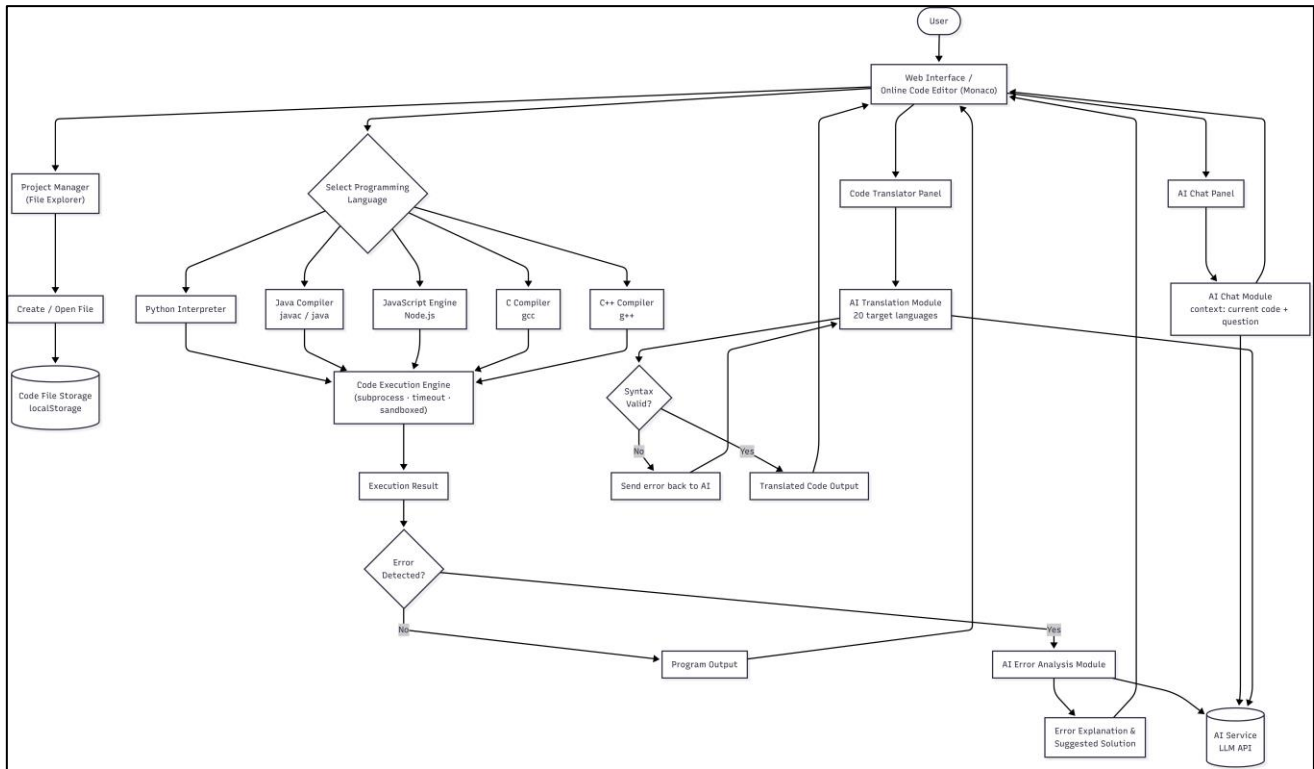


Fig. 1: CompileX system architecture diagram.

- 1) **Compiler Services** providing multi-language code execution via GCC, G++, JDK, Python, and Node.js runtimes within secure execution sandboxes, and (ii) **AI Services** implementing the Large Language Model API integration with context-aware prompt engineering for error resolution, code explanation, translation, and conversational assistance.
 - 7) **Error Detection and Analysis:** The execution result is analyzed to determine whether the program completed successfully or encountered an error. Detected errors are associated with the relevant source-code context.
 - 8) **AI Error Resolution:** When an error is detected, the system provides the source code, error information, programming language, and relevant execution context to the AI module. The AI analyzes the information and generates a simplified explanation and debugging suggestions.
 - 9) **Response Aggregation:** The system combines the execution result, error information, AI explanation, and debugging guidance into a unified response.
 - 10) **Frontend Rendering:** The React frontend displays the program output, compiler or runtime errors, and AI-generated assistance through the corresponding interface panels.
- B. System Workflow**
- The overall workflow of CompileX, illustrated in Fig. 2, follows a structured pipeline from source-code input to program execution and AI-assisted output generation. The workflow consists of the following stages:
- 1) **Code Input:** The user writes or pastes source code into the browser-based editor and selects the required programming language.
 - 2) **Compilation Request:** When the user selects the Run Code option, the frontend sends a request to the Flask backend containing the source code and selected language.
 - 3) **Request Validation and Context Preparation:** The backend validates the received request and prepares the source code, selected language, execution parameters, and relevant context required for compilation and subsequent AI-assisted analysis.
 - 4) **Language Detection and Routing:** The backend identifies the selected programming language and routes the request to the corresponding compiler or interpreter.
 - 5) **Program Execution:** The submitted code is compiled and executed using the configured compiler or runtime environment. Execution controls and timeout handling are applied to manage program execution.
 - 6) **Output Capture:** The backend captures the program's standard output along with compilation errors, runtime errors, and execution status.

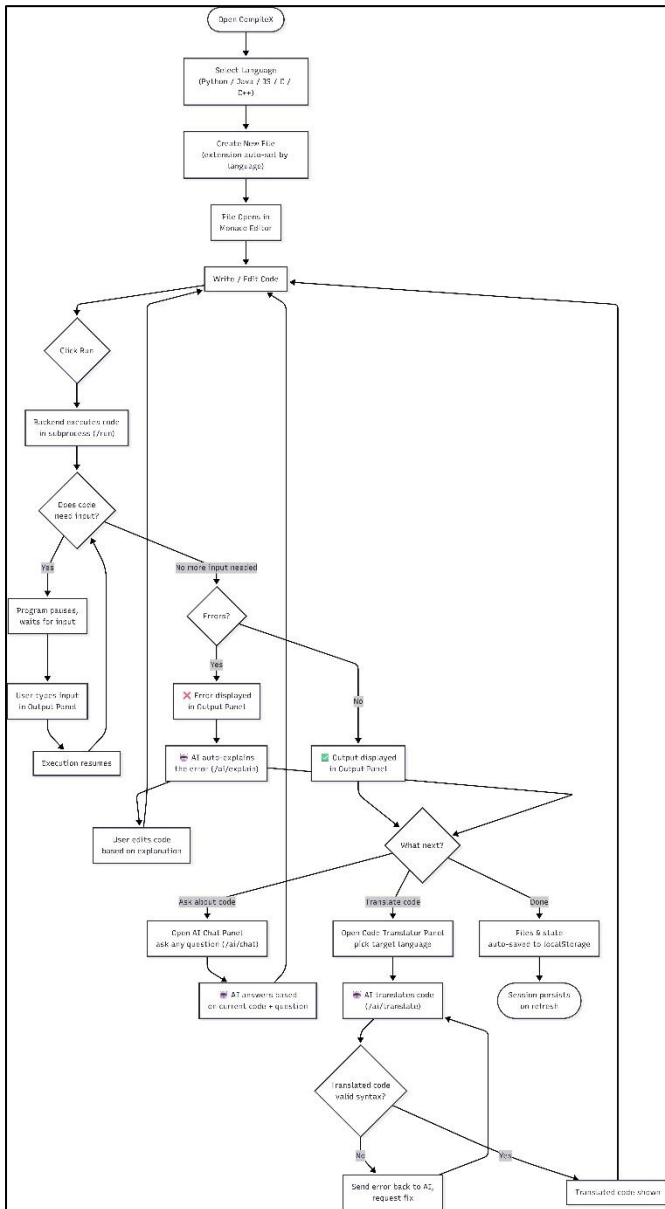


Fig. 2: CompileX system workflow from source-code input to AI-assisted output generation.

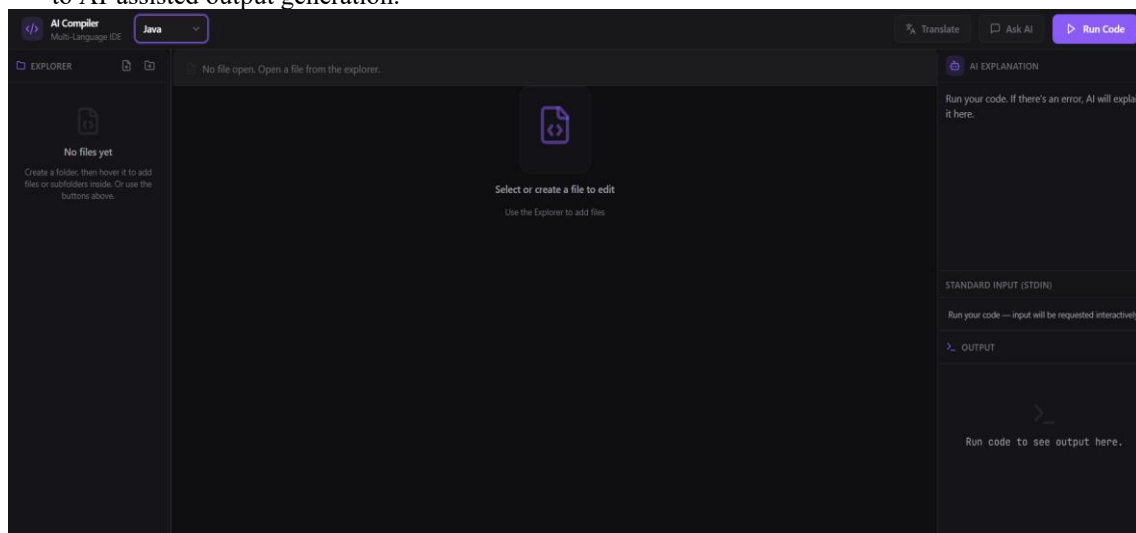


Fig. 3: Initial CompileX multi-language IDE interface.

VI. IMPLEMENTATION

The CompileX system is implemented as a web-based programming environment that integrates code editing, multi-language compilation, program execution, and AI-assisted programming features. The implementation consists of a React-based frontend, Flask backend, compiler and runtime integration, AI service integration, and modules for error resolution, code explanation, conversational assistance, code translation, code optimization, and real-time code analysis.

A. Frontend Implementation

The frontend is developed using React.js with Vite build tooling and provides an interactive browser-based coding environment. It includes a code editor, file explorer, language-selection dropdown, Run Code control, output display, AI explanation panel, AI Chat Assistant, Code Translator, and AI Code Editor with real-time suggestions. The frontend communicates with the Flask backend through REST API requests.

Language	Compiler/Interpreter
C	GCC
C++	G++
Java	Java Development Kit (JDK)
Python	Python Interpreter
JavaScript	Node.js Runtime

Table II: compiler and runtime integration

B. Backend Implementation

The backend is implemented using Python and Flask. It receives source code and language information from the frontend, determines the appropriate compiler or interpreter, executes the submitted program, captures the generated output and errors, and returns the results to the frontend. The backend also handles requests related to AI chat, code translation, code explanation, error resolution, code optimization, and real-time code analysis.

C. AI Service Integration

The AI functionality is integrated with the backend through an AI service interface. The Flask backend prepares requests containing the source code, selected programming language,

error information, or user query and provides the relevant context to the AI model. The generated response is processed by the backend and returned to the frontend for display in the corresponding AI feature. This integration enables context-aware error explanation, code assistance, code explanation, code translation, and code optimization within the CompileX environment.

D. Compiler Integration

CompileX integrates GCC for C, G++ for C++, the Java Development Kit (JDK) for Java, the Python interpreter for Python, and Node.js for JavaScript. The backend selects the corresponding compiler or runtime according to the programming language selected by the user.

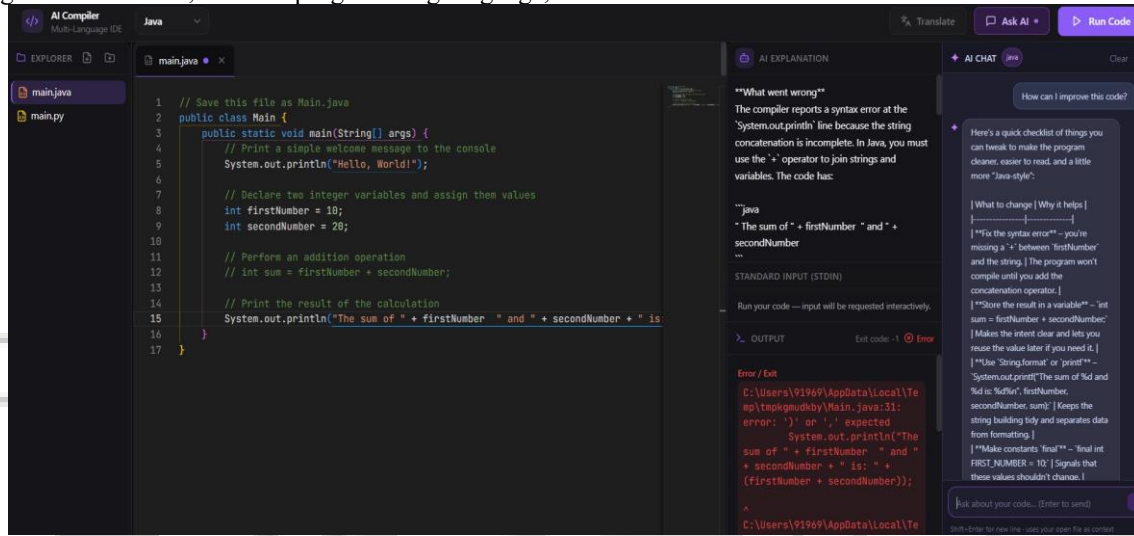


Fig. 4. AI-based error explanation and AI Chat Assistant

E. Program Execution and Output

When the user executes a program, the backend creates the required temporary execution resources and invokes the corresponding compiler or interpreter. The system captures standard output, compilation errors, runtime errors, and execution status. The processed result is returned to the frontend and displayed through the output interface.

F. AI Error Explanation Implementation

When a compilation or runtime error occurs, the error information is combined with the relevant source-code context and processed by the AI module. The system generates a simplified explanation of the error, identifies its probable cause, and provides debugging suggestions. This allows users to understand technical compiler messages and assists them in identifying and correcting programming errors.

G. AI Chat Assistant Implementation

The AI Chat Assistant provides contextual programming assistance related to source code, programming concepts, syntax, functions, algorithms, and debugging. The current

code context can be provided to the AI so that responses remain relevant to the program being developed. Users can interact with the assistant to obtain explanations and guidance during the coding process.

H. AI Code Translation Implementation

The AI Code Translator converts source programs from one supported programming language to another. The user selects the target language, and the AI generates a translated version while attempting to preserve the original program logic and functionality. The translated code is displayed through the Code Translator interface.

I. AI Code Optimization Implementation

The AI Code Optimization feature analyzes the submitted source code and identifies possible improvements in code efficiency, readability, and performance. The AI examines the program structure and suggests optimized implementations while attempting to preserve the original functionality and logic of the program. The generated optimization suggestions are presented to the user so that they can review the proposed changes before applying them to the source code.

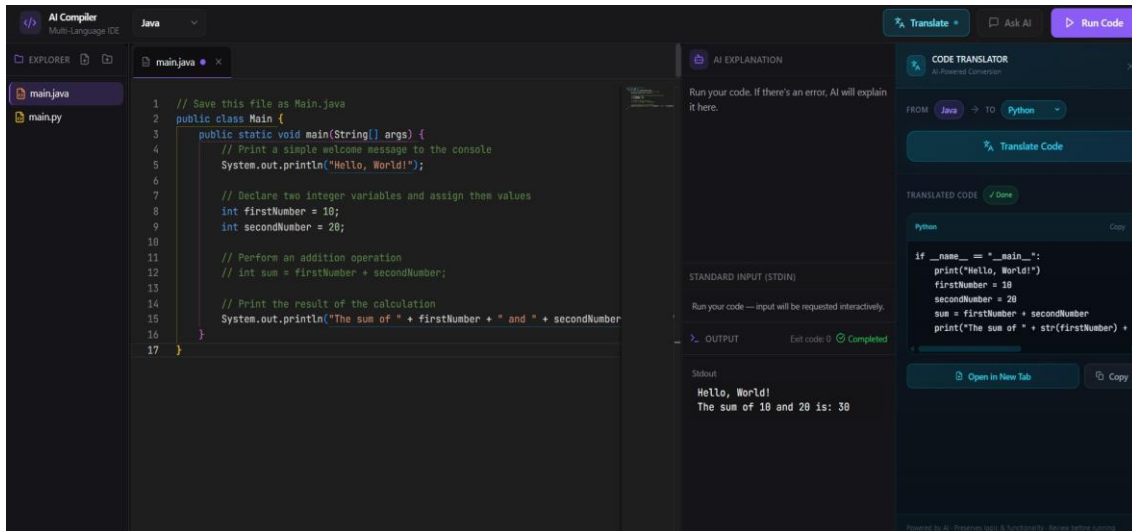


Fig. 5. AI Code Translator interface.

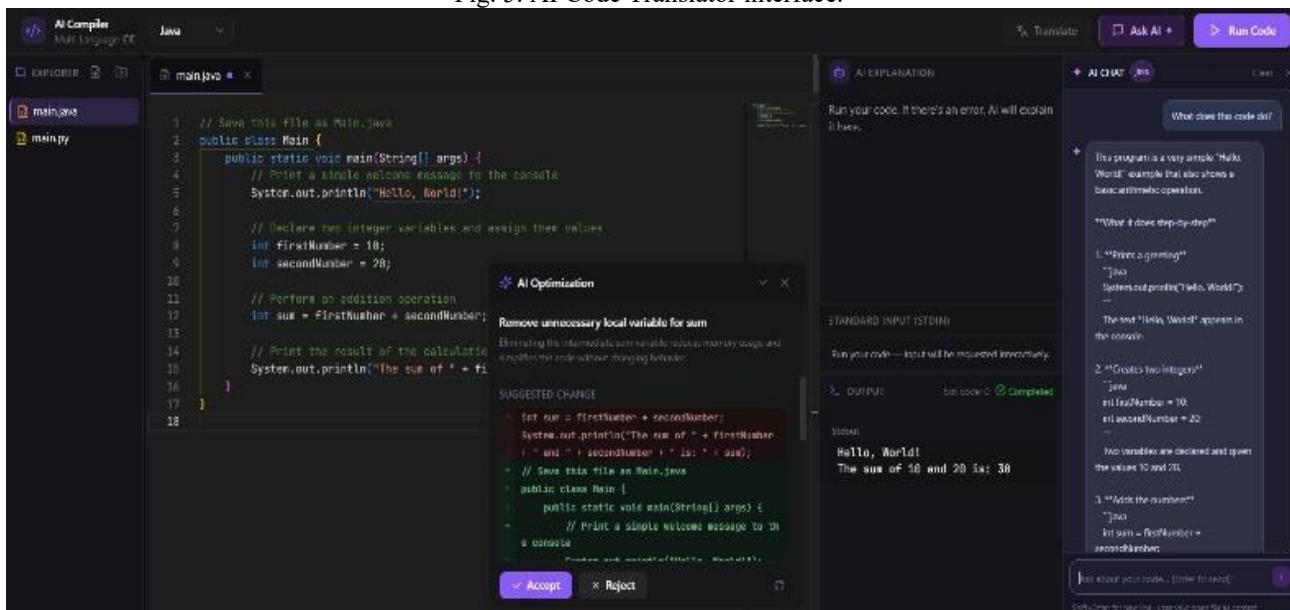


Fig. 6. AI Code Optimization interface.

J. AI Code Explanation Implementation

The Explain This Code feature analyzes selected source code and generates simplified explanations of its variables, parameters, functions, logic, and operations. This feature helps users understand the structure and working of a program without requiring them to manually analyze every component of the source code.

K. Real-Time Code Analysis Implementation

The AI Code Editor performs real-time analysis of the source code during development. Potential errors and improvement opportunities are identified from the current code context and presented to the user while editing. Users can review the generated suggestions and either accept them to update the code or reject them and continue editing manually.

L. File Management and Local Storage

CompileX provides file management functionality through the browser-based file explorer. User-created files and relevant code data can be maintained using browser Local Storage, allowing users to retain their work without requiring

an external database. The file management interface enables users to create, select, and edit source files within the CompileX environment.

M. Execution and Resource Management

The backend manages program execution through temporary execution resources and timeout controls. Input validation and execution management are applied to handle submitted programs and prevent continuously running processes from affecting the system. Temporary execution resources are removed after processing.

N. Overall Integration

The final implementation integrates browser-based code editing, file management, multi-language compilation, program execution, AI-assisted error resolution, conversational assistance, code translation, code explanation, code optimization, and real-time code analysis. These components work together to provide an integrated programming environment in which conventional compiler functionality and AI-based programming assistance are available within the same workflow. The integration enables

users to write, execute, understand, translate, optimize, and improve their programs through a unified browser-based environment.

VII. RESULTS AND DISCUSSION

A. Experimental Setup

CompileX was evaluated using a prototype implementation across 500 compilation scenarios covering the five supported programming languages: Python, Java, C, C++, and JavaScript. The evaluation considered error explanation accuracy, debugging time reduction, user comprehension, and AI response latency. For the debugging-time comparison, 50 error cases per language were evaluated with novice programmers using identical error scenarios for the compiler-only baseline and CompileX. User experience was assessed with a 5-point Likert-scale questionnaire administered after debugging tasks, and AI response latency was measured across 500 API calls. These metrics were selected to evaluate both the effectiveness of AI-assisted debugging and the usability and responsiveness of the proposed system.

B. Error Explanation Accuracy

The error explanation module was evaluated across syntax, runtime, and semantic error categories. As shown in Table III, the overall average explanation accuracy was 91.1%. Syntax errors achieved the highest average accuracy of 93.7%, followed by runtime errors at 90.9% and semantic errors at 88.8%. The results indicate that the system provides consistent error explanations across the supported programming languages, with relatively lower accuracy for errors requiring greater contextual or logical understanding. The explanation accuracy was calculated using:

$$Accuracy = \frac{N_{correct}}{N_{total}} \times 100 \quad (1)$$

where $N_{correct}$ represents the number of evaluated error cases classified as correctly explained and N_{total} represents the total number of evaluated error cases.

C. Debugging Time Reduction

The debugging performance of CompileX was compared with a compiler-only baseline using identical error scenarios. The evaluation involved 50 error cases per language and novice programmers. The average debugging time decreased from 285.9 seconds for the baseline to 100.8 seconds with CompileX,

Language	Syntax	Runtime	Semantic	Overall
Python	94.2%	91.8%	89.5%	91.8%
Java	93.7%	90.4%	88.2%	90.8%
C	92.1%	89.7%	87.3%	89.7%
C++	93.5%	90.1%	88.8%	90.8%
JavaScript	95.1%	92.3%	90.1%	92.5%
Average	93.7%	90.9%	88.8%	91.1%

Table III: Error Explanation Accuracy by Language and Error Category

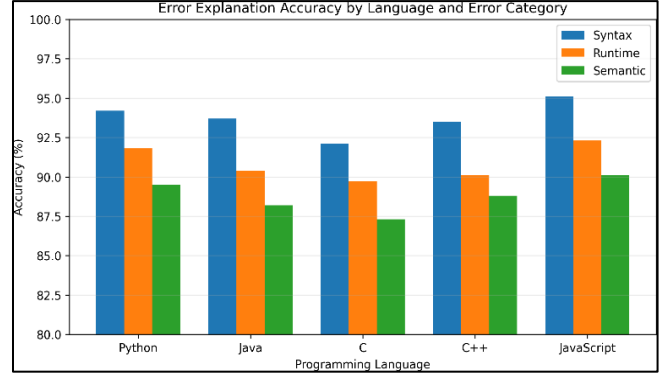


Fig. 7: Error explanation accuracy by language and error category (grouped comparison).

Error Category	Baseline (s)	CompileX (s)	Reduction
Syntax Errors	187.3	52.4	72.0%
Runtime Errors	245.8	89.6	63.5%
Semantic Errors	312.4	118.7	62.0%
Logic Errors	398.2	142.3	64.3%
Average	285.9	100.8	64.7%

Table IV: Debugging Time Comparison

representing a 64.7% reduction. The results show that AI-generated explanations and debugging suggestions can reduce the time required to interpret and resolve programming errors.

The percentage reduction in debugging time was calculated using:

$$Reduction = \frac{T_{baseline} - T_{CompileX}}{T_{baseline}} \times 100 \quad (2)$$

where $T_{baseline}$ represents the average debugging time using the compiler-only baseline and $T_{CompileX}$ represents the average debugging time using CompileX.

D. User Comprehension and Experience

User experience was evaluated using a 5-point Likert-scale questionnaire administered to 45 participants after completing debugging tasks. Explanation clarity obtained a mean score of 4.52/5, while overall satisfaction was 4.45/5. Helpfulness for debugging, ease of understanding, and perceived learning value also received scores above 4.0, indicating positive user responses toward the AI-assisted programming environment.

The mean score for each evaluation metric was calculated as:

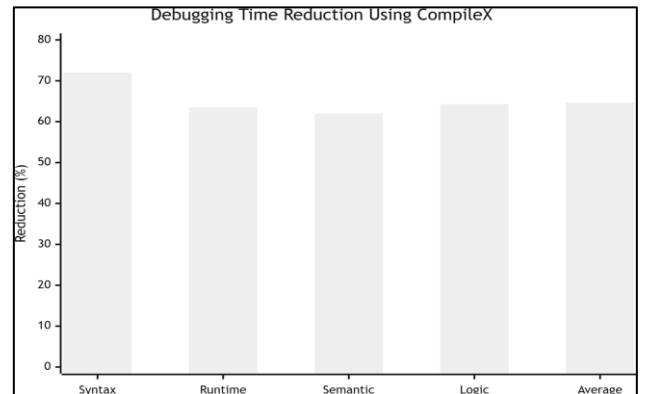


Fig. 8: Percentage reduction in debugging time using CompileX.

Metric	Mean	SD	Interpretation
Explanation Clarity	4.52	0.68	Excellent
Helpfulness for Debugging	4.41	0.72	Excellent
Ease of Understanding	4.38	0.75	Excellent
Learning Value	4.29	0.81	Very Good
Overall Satisfaction	4.45	0.71	Excellent

Table V: User Experience Evaluation

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i \quad (3)$$

where X_i represents the score assigned by participant i and N represents the total number of participants.

E. AI Response Latency

System performance was evaluated across 500 AI API calls. The measured mean response latency was 2.83 seconds, with a median of 2.61 seconds. The 90th-percentile and 99th-percentile latencies were 4.12 and 6.87 seconds, respectively. These measurements provide an indication of the response time of the AI assistance component during system operation.

The mean AI response latency was calculated as:

$$\bar{L} = \frac{1}{N} \sum_{i=1}^N L_i \quad (4)$$

where L_i represents the response latency of the i -th AI API call and N represents the total number of evaluated API calls. The percentile latency values were used to further evaluate the response-time distribution and identify the performance experienced by slower API requests.

VIII. CONCLUSION

This paper presented CompileX, an intelligent online compiler integrating AI-assisted error resolution, contextual code explanation, conversational programming assistance, and cross-language code translation. The implemented platform supports C, C++, Java, Python, and JavaScript through a browser-based development environment. Experimental evaluation across 500 compilation scenarios reported an average error explanation accuracy of 91.1%, a 64.7% reduction in debugging time relative to a compiler-only baseline, and a mean AI response latency of 2.83 seconds. User evaluation with 45 participants reported mean scores of 4.52/5 for explanation clarity, 4.41/5 for debugging helpfulness, and 4.45/5 for overall satisfaction. The lower accuracy observed for semantic errors indicates an area for further improvement, particularly for cases requiring deeper program-context and logical reasoning. The current evaluation is limited to the five supported languages and the reported prototype scenarios; broader datasets, additional programming environments, and larger user studies can provide further validation. Overall, the implementation and evaluation provide a basis for continued development of integrated AI- assisted compiler environments

Metric	Value
Mean Latency	2.83 seconds
Median Latency	2.61 seconds
90th Percentile	4.12 seconds
99th Percentile	6.87 seconds
Standard Deviation	1.24 seconds

Table VI: AI Response Latency

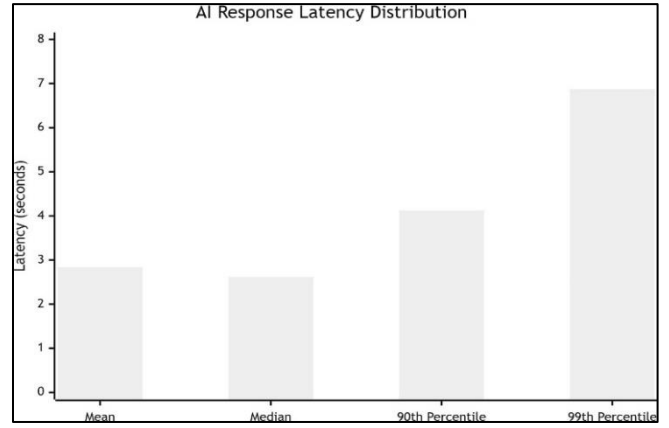


Fig. 9. AI response latency metrics.

ACKNOWLEDGMENT

The authors thank their project guide, Prof. Aritri Sen, and co-guide, Dr Anita Morey, for their valuable guidance and continuous support. The authors also thank the Department of Artificial Intelligence, Usha Mittal Institute of Technology, SNDT Women's University, for providing the necessary resources and facilities.

REFERENCES

- [1] J. Renzella, A. Vassar, L. L. Solano, and A. Taylor, "Compiler-integrated, conversational AI for debugging CS1 programs," in Proc. 56th ACM Technical Symposium on Computer Science Education, Pittsburgh, PA, USA, Feb. 2025, pp. 994–1000, doi: 10.1145/3641554.3701827.
- [2] A. Taylor, A. Vassar, J. Renzella, and H. Pearce, "dcc – help: Transforming the role of the compiler by generating context-aware error explanations with large language models," in Proc. 55th ACM Technical Symposium on Computer Science Education, Portland, OR, USA, Mar. 2024, pp. 1314–1320, doi: 10.1145/3626252.3630822.
- [3] M. Hoq et al., "Automated identification of logical errors in programs: Advancing scalable analysis of student misconceptions," in Proc. AAAI Conf. Artificial Intelligence, 2025.
- [4] SyntaxMentor Team, "SyntaxMentor: An AI-augmented debugging assistant for novice programmers," Int. J. Engineering Development and Research, vol. 13, no. 4, Dec. 2025.
- [5] G. Akçapınar and E. Sidan, "AI chatbots in programming education: Guiding success or encouraging plagiarism," in Proc. ACM Conference on Innovation and Technology in Computer Science Education, 2024.
- [6] D. Kohen-Vacs, M. Usher, and M. Jansen, "Integrating generative AI into programming education: Student perceptions and the challenge of correcting AI errors," Int. J. Artificial Intelligence in Education, Springer, 2025.
- [7] E. A. Santos and B. A. Becker, "Not the silver bullet: LLM-enhanced programming error messages are ineffective in practice," in Proc. UKICER, 2024, doi: 10.1145/3689535.3689554.

- [8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Pearson Education, 2006.
- [9] A. Taylor et al., "Scaling CS1 support with compiler-integrated conversational AI," arXiv preprint arXiv:2408.02378, 2024.
- [10] L. L. Solano, J. Renzella, and A. Vassar, "DCC Sidekick: Helping novices solve programming errors through a conversational explanation interface," in *Proc. 55th ACM Technical Symposium on Computer Science Education*, Portland, OR, USA, Mar. 2024, pp. 1714–1715, doi: 10.1145/3626253.3635483.
- [11] J. Leinonen, A. Hellas, S. Sarsa, B. N. Reeves, P. Denny, J. Prather, and B. Becker, "Using large language models to enhance programming error messages," in *Proc. 54th ACM Technical Symposium on Computer Science Education*, 2023, pp. 563–569, doi: 10.1145/3545945.3569770.
- [12] M. Pankiewicz and R. S. Baker, "Navigating compiler errors with AI assistance—A study of GPT hints in an introductory programming course," in *Proc. 29th Annual ACM Conference on Innovation and Technology in Computer Science Education*, Milan, Italy, 2024, doi: 10.1145/3649217.3653608.
- [13] X. Dong and G. Lin, "Design and application of a C++ compiler error solution query platform integrating large language models and human-in-the-loop support," *Proc. ACM Hum.-Comput. Interact.*, 2025, doi: 10.1145/3775073.3775185.

