AI-Based Logical Error Detection and Explainable Code Analysis Using Graph Neural Networks and Large Language Models

Vaishnavi Satish Deshmukh[1], Samruddhi Shankar Mote[2], Suraj Suresh Divate[3], Mr. Jagtap Kiran Prakash[4]

[1,2,3]*Student at Department of Computer Science and Engineering, Yashoda Technical Campus, Faculty of Engineering, Satara, Maharashtra, India*

[4]*Asst. Professor at Department of Computer Science and Engineering, Yashoda Technical Campus, Faculty of Engineering, Satara, Maharashtra, India*

Abstract*—S *This paper introduces a Hybrid AI System that tackles logical error detection and explainable code analysis to make debugging smarter and help developers actually understand their code. The setup uses both Graph Neural Networks (GNNs) and Large Language Models (LLMs), which work together to dig into Python code. Here's how it goes: first, the system breaks down the code into Abstract Syntax Trees (ASTs) and logical flow graphs. That way, it grabs not just the structure, but how things play out when the program runs.

A GNN, trained on piles of code—both correct and flawed—hunts for sketchy patterns and logical slip-ups. When it finds something odd, the LLM jumps in, offering explanations you can actually read, plus suggestions and corrected code. There's more: the system builds interactive flowcharts and graph visuals, with those buggy parts marked bright red so you don't waste time hunting them down.

All the debugging action happens inside a clean, web-based interface. It's not just about catching errors—it helps you follow the code's logic, pin down where things went wrong, and see it visually. With all these layers working together, debugging gets faster, code insights deeper, and the whole process more transparent. It's a hands-on, AI-powered approach for program analysis and spotting logical errors right when they show up.

***Index Terms*—Abstract Syntax Tree (AST), Automated Debugging, Control Flow Graph (CFG), Explainable Artificial Intelligence (XAI), Graph Neural Networks (GNN), Large Language Models (LLM), Logical Error Detection, Program Analysis, Source Code Analysis.**

I. INTRODUCTION

Software debugging and finding logical errors are still major challenges in software engineering as logical errors usually do not result in error messages from the compiler and are difficult to locate manually. Debugging techniques and static analysis tools currently mainly focus on syntactic issues and rule-based detection methods, thus limiting their ability to understand complex semantic and structural relationships in source code. Recent advances in Artificial Intelligence (AI), especially Graph Neural Networks (GNNs), have opened new avenues for automated program analysis and intelligent debugging.

Z. Xu, K. Zhang, and V. S. Sheng proposed "Logic Error Localization in Student Programming Assignments Using Pseudocode and Graph Neural Networks" [1]. They used Graph Neural Networks to find logical errors by utilizing pseudocode-based program representations. Their work showed that graph-based learning can effectively capture structural dependencies for locating logic errors. Similarly, M. N. Rafi, D. J. Kim, A. R. Chen, T.-H. P. Chen, and S. Wang suggested better code representation methods for GNN-based fault localization in "Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation" [2]. Their method improved fault localization accuracy with structural graph representations and scalable graph learning techniques.

Also, graph-based learning has reported promising results in software defect prediction and bug localisation. L. Šikić, A. S. Kurdija, K. Vladimir and M. Šilić [3] proposed a Graph Convolutional Neural Network model for source code defect prediction based on Abstract Syntax Trees (ASTs) as graph representations of source code. Their work demonstrated that graph-based neural networks can effectively learn from the syntactic and semantic structures of source code. Furthermore, L. Yousofvand et al. [4] proposed a framework based on graph for accurate bug localisation by structural program analysis using Graph Neural Networks. Their approach is based on node classification and graph analysis for automated bug localisation.

Recent studies have also been conducted on integrated graph learning models for software quality assessment and explainable AI systems. P. Dai, H. Zhu, J. Wu and H. He proposed an integrated Graph Neural Network model based on Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs) for software defect prediction and code quality assessment to capture multi-level program semantics [5]. In addition, L. Yousofvand, S. Soleimani, V. Rafe and A. Nikanjam underlined the utility of graph representations for automated bug localisation and structural program analysis in current debugging systems [6]. A. Ragno, M. Plantevit, and C. Robardet proposed "On Logic-based Self-Explainable Graph Neural Networks" [7], where logic-based explainability was integrated into Graph Neural Networks to enhance interpretability and transparency in AI-based systems.

Taking inspiration from these developments, a novel Hybrid AI-Based Logical Error Detection and Code Explainability System is proposed in this study using Graph Neural Network and Large Language Models (LLMs) for intelligent software program debugging. In the proposed hybrid system, the Python source code is converted into graphical representations, and suspicious code blocks or logical errors are detected using structural learning through GNNs. Furthermore, the embedded LLM offers human-readable explanations, bug fixes, and flow-based visualization that helps in efficient debugging. Additionally, the proposed system offers interactive visualization of graphs and flowcharts with highlights of error-containing nodes, which assist in understanding the flow of the code.

1) Login Page:

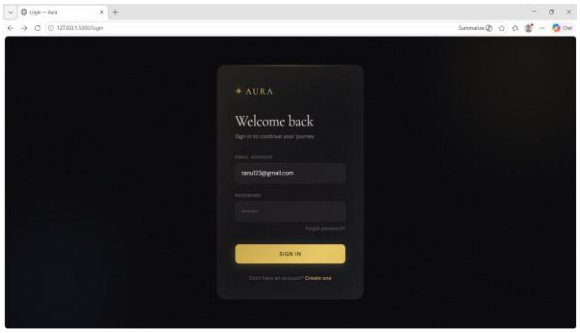

Fig. 1. Login interface of CodeX

2) Registration Page:

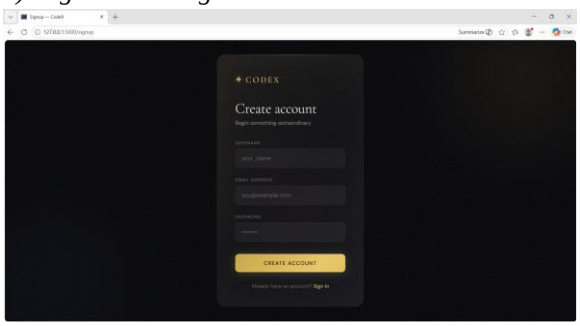

Fig. 2. Registration interface of CodeX

3) Dashboard Page:

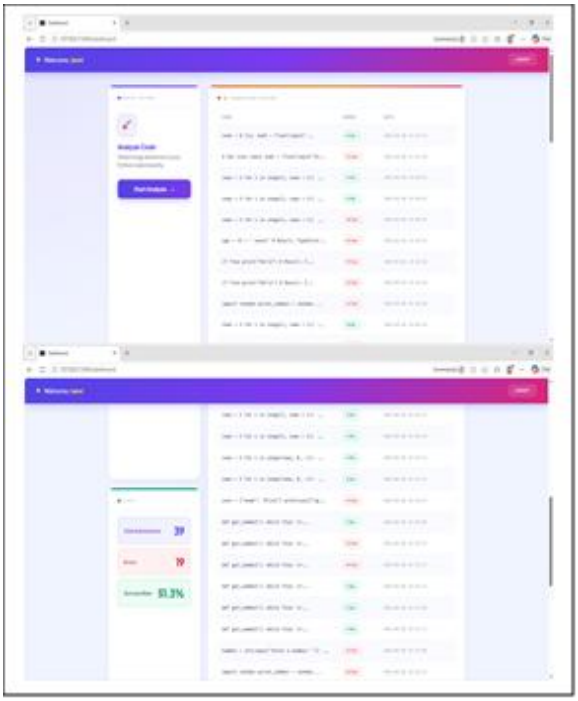

Fig. 3. Dashboard interface of CodeX

4) Analysis Page:

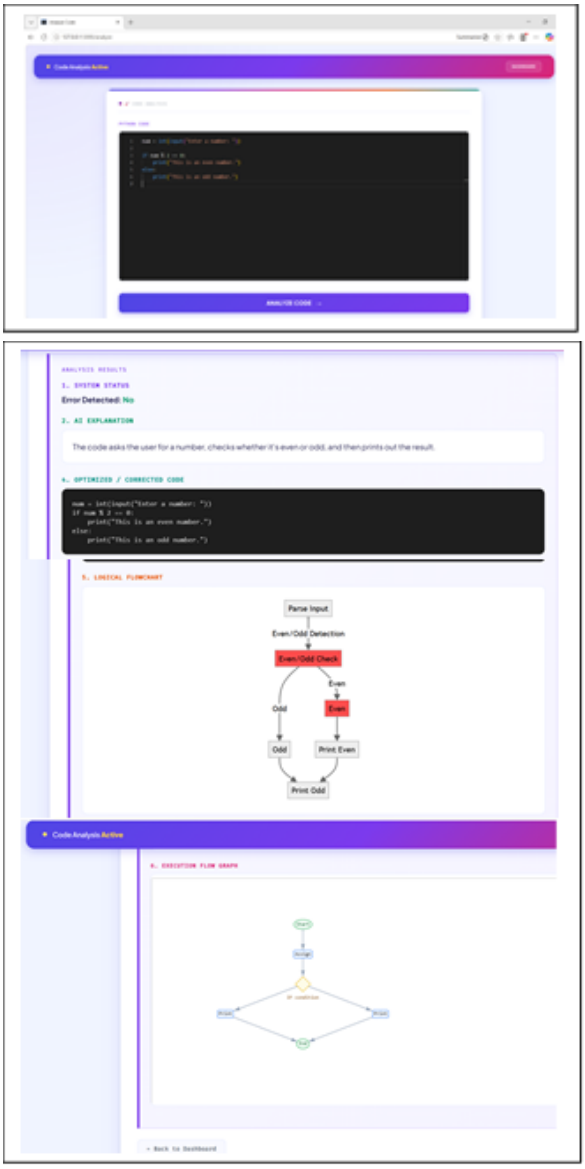

Fig. 4. Analysis interface of CodeX

II. METHODOLOGY

The Hybrid AI-Based Logical Error Detection and Explainable Code Analysis System comprises Graph Neural Networks (GNNs), Large Language Models (LLMs), and graph-based program analysis and is modular and explainable. The implementation of the system involves use of client-server architecture made up of Flask (backend), MySQL (database), and HTML, CSS, and JavaScript (frontend). The framework aims at analyzing Python source code, identifying errors logically or structurally, creating explainable flow visualization, and providing suggestions on how to fix errors via AI. The users will use the framework by submitting the code to be analyzed via a web interface.

A. Architecture of the System

CodeX Architecture consists of three levels:

1) Frontend Level:

This level is built on HTML, CSS, and JavaScript, which serve as interfaces to interact with the application. Users initially register themselves using the interface illustrated in Fig. 2 and then log in using Fig. 1. Once logged in, users can access the dashboard displayed in Fig. 3 and conduct code analysis using the analysis interface demonstrated in Fig. 4.

2) Backend Level:

In the backend level, the Flask framework manages the process of authentication, routing, code processing, analysis via GNN, and integrating LLM via the Groq API. Based on the submitted source code, it produces flowcharts, graphs, explanations, and suggestions for fixing the code.

3) Database Level:

The MySQL database saves user details, code submissions, code analysis, graphs produced by code analysis, time stamps, etc.

B. Data handling and preprocessing:

Using the analysis portal displayed in Fig. 4, users can provide their Python source code for analysis. Each time a piece of code is received, the back-end performs its validation through user-ID labeling and timestamping before storing them into the MySQL database. The preprocessing steps of syntax checking, text cleaning, AST creation, graph generation, and nodes extraction ensure logical error detection and accurate code analysis for Graph Neural Network application. The dashboard displayed in Fig. 3 provides analysis history along with the results of analysis.

C. Real-Time Skill Analysis Using the Gemini API:

Rather than just using static and rule-based code analysis techniques, the proposed framework of CodeX employs Groq API with a Large Language Model (LLM) for dynamic and real-time code analysis. Groq API will take input as processed source code and GNN analysis findings and will produce a set

of structured outputs including logical explanations of errors, corrected source code, code optimization tips, and even flow charts of the program. All these AI-produced analysis outputs are then recorded in MySQL databases with time stamps for later references. The process can be seen in Fig. 4 below.

D. Assigning Tasks and Tracking Progress:

Python scripts are submitted by users using the provided analysis interface as depicted in Fig. 4. The code is analyzed using Graph Neural Networks (GNNs) as well as the Groq API in order to identify logic errors, develop explanations, draw flow diagrams, and suggest corrections to the code. All analysis output, graphs, and timestamps are saved into the MySQL database that keeps track of the information on the dashboard interface seen in Fig.

E. System Evaluation:

The performance of CodeX is assessed on various parameters like accuracy of detection of logical errors, response time, efficiency of graphs, and quality of AI explanations. The user interface, along with the dashboard and analytics module as illustrated in Figs. 3–4, is also subjected to evaluation for assessing usability, responsiveness, visualizations, and debugging.

F. System Architecture:

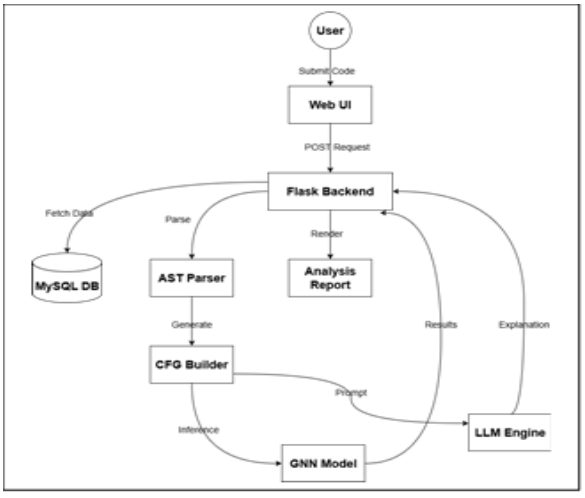

Fig 5. System Architecture of CodeX Analysis loop

III. HARDWARE AND SOFTWARE USED

A. Hardware Requirements

The CodeX is a lightweight artificial intelligence-based code analysis system that does not necessitate high-end hardware for development and testing purposes. The minimal hardware requirements include:

- Processor – Dual-core CPU (Intel i3 or similar)
- RAM – 4 GB (8 GB is recommended for smooth AI processing)
- Storage Space – 2 GB (for saving project files, datasets, database, and dependencies)
- System – Windows 10/11, Linux, or Mac
- GPU – optional NVIDIA GPU for quick GNN model training
- Network – stable Internet connection for Groq API requests

B. Software Requirements

The following software stack is used for developing and implementing CodeX:

1. Backend Technologies:

- Flask for routing, authentication, API interactions, and other backend functionalities
- Python 3.x for the development of backend logic, AI process, and graph creation

2. Frontend Technologies:

- HTML5, CSS3, and JavaScript for designing the dashboards, code editor, and UI components

3. Database Technologies:

- MySQL for storing user data, code submissions, report generation, graphs, and timestamps

4. AI/Analysis Technologies

- Graph Neural Networks (GNN) for detecting logical errors and conducting code analysis
- Groq API (LLM) for providing explanations for AI actions, code reasoning, corrected code generation, and debugging
- AST parser to transform source code into Abstract Syntax Trees
- NetworkX and PyVis for creating and visualizing graphs
- Mermaid.js for generating and visualizing logical flowcharts

5. Development/Deployment:

- VS Code for writing and debugging code and developing the app
- MySQL Workbench for working with the MySQL

database
- Git and GitHub for maintaining repositories

IV. DATABASE

1) Level 0 DFD

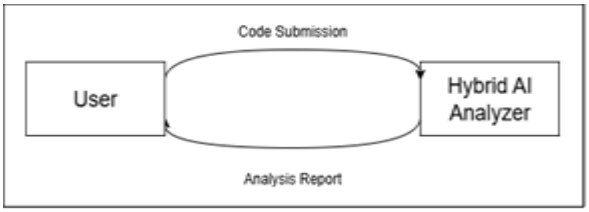

Fig 6. Interaction model between user and Analyzer.

2 Level 2 DFD

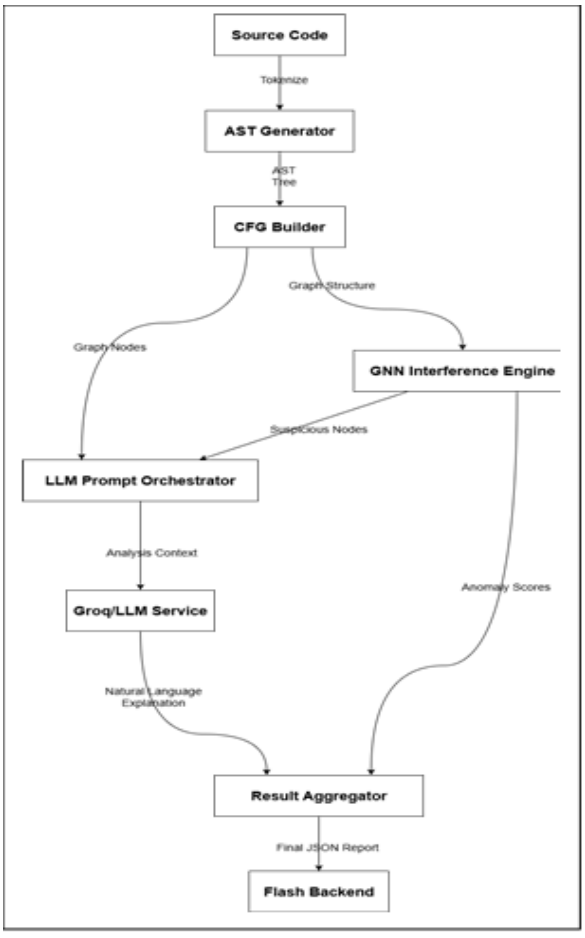

Fig 7. Code analysis Process Flow.

V. FUTURE SCOPE

In addition, CodeX can be enhanced by supporting other programming languages, including Java, C++, and JavaScript. More data could also increase GNN performance in detecting logical errors and software defects [1]– [5].

Moreover, future versions might incorporate sophisticated explainable AI technology, real-time debugging, cloud computing, and Visual Studio Code integration for better usability and debugging efficacy [7]. Other advanced techniques for vulnerability detection and semantic code analysis like Devign [8] can also enhance software security analysis.

Finally, the proposed system can be implemented in EduMentorAI as an AI code analysis and debugging mode for students, offering automated code analysis, explanation of logical errors, and flowchart learning.

Finally, future research could explore combining Graph Neural Networks with transformer language models to boost semantic analysis and intelligent code repair [1]– [8].

VI. LITERATURE REVIEW

Logical error localization, defect prediction, and automated bug localization using Graph Neural Networks (GNNs) and Graph-based program analysis have attracted considerable attention recently. The authors of [1] presented a logic error localization method based on pseudocode and Graph Neural Networks for programming assignments. They used Graph Neural Networks along with graph structural learning to detect logic errors and achieved high accuracy levels. However, the suggested approach is mainly intended for logical error localization and lacks interactive debugging features and flowcharts.

In [2], enhanced code representations for fault localization using Graph Neural Networks are provided. Specifically, authors suggest DepGraph for accurate fault localization via structural graph representations and scalable graph learning. Although this approach helps to improve structure-aware fault localization, it still lacks interactive debugging visualization and explainable AI reasoning.

Authors of [3] proposed a graph model for predicting defects in software source code based on Abstract Syntax Trees (ASTs). In their paper, the authors prove that graph-based neural networks help to analyze syntactic and semantic structures of source code. However, this approach does not incorporate visualization, explainability, or real-time code correction.

Authors of [4] proposed a Graph Neural Network

model for bug localization through structural program analysis. This approach uses node classification to identify bugs. Despite this method's efficiency in terms of structural bug detection, the suggested technique lacks explainable reasoning mechanisms and interactive debugging visualization.

The authors of [5] proposed an integrated model of Graph Neural Networks for source code defect prediction and code quality assessment based on Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). The dual-branch architecture allows for efficient code quality analysis and accurate defect prediction; however, this system is focused more on prediction performance rather than debugging.

In [6], authors focused on graph-based structural analysis and node classification to develop an automated bug localization system based on Graph Neural Networks. Authors' results demonstrate that graph representation helps with software debugging. However, the proposed approach lacks explainable AI reasoning techniques and interactive debugging capabilities.

Finally, the authors of [7] proposed logic-based self-explainable Graph Neural Networks for improved explainability and transparency of AI systems. Although this approach provides explainable graph learning, it cannot be applied for source code debugging applications.

The existing literature mostly revolves around prediction accuracy, structural learning, and bug localization. In stark contrast to all of the above, what sets apart CodeX is its uniqueness in offering a novel AI-based debugging approach that encompasses Graph Neural Networks, Large Language Models, visualization of graphs, creation of flowcharts, suggested correct code snippets, and reasoning by Explainable Artificial Intelligence (AI). It is pertinent to note that while existing tools aim at “software defect detection,” CodeX also pays attention to “explanation, visualization, and correction of logical bugs.”

VII. LIMITATIONS

While CodeX offers intelligent logical error detection and explainable code analysis using Graph Neural Networks (GNNs) and Large Language Models (LLMs), the application has certain limitations.

Currently, CodeX can analyze source code files written in the Python programming language, but cannot perform full code analysis of multiple programming languages. As in any machine learning algorithm, the success rate of detecting logic errors using GNN models relies on high-quality datasets with diversified training samples. Lack of sufficient information for training could lead to false-positive and false-negative predictions.

In order to generate explanations, corrected versions of code files, and diagrams, CodeX uses external Groq API services. Availability and response time of external APIs may impact application speed and availability; additionally, large source code files may slow down generation and readability of generated graphs.

ACKNOWLEDGMENT

We would like to express our gratitude to everyone who helped us finish this mini project successfully. We are incredibly appreciative of our mentor, Mr. Jagtap Kiran Prakash, for his unwavering support, direction, and inspiration during the project. We also thank Dr. S. V. Balshetwar, Head of the Department of Computer Science and Engineering at YSPM's YTC, Satara, for her invaluable assistance. We also thank Dr. Vikram S. Patil, our Honourable Principal, for creating a great learning environment and opportunities. Finally, we would like to sincerely thank our friends and family for their unwavering support, encouragement, and assistance.

REFERENCES

[1] Z. Xu, K. Zhang, and V. S. Sheng, “Logic Error Localization in Student Programming Assignments Using Pseudocode and Graph Neural Networks,” *arXiv preprint arXiv:2410.21282*, 2024.

[2] M. N. Rafi, D. J. Kim, A. R. Chen, T.-H. P. Chen, and S. Wang, “Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, Article 86, pp. 1–23, 2024.

[3] L. Šikić, A. S. Kurdija, K. Vladimir, and M. Šilić, “Graph Neural Network for Source Code Defect Prediction,” *IEEE Access*, vol. 10, pp. 10402–10416, 2022.

[4] L. Yousofvand, S. Soleimani, V. Rafe, and A.

Nikanjam, “Graph Neural Networks for Precise Bug Localization through Structural Program Analysis,” *Automated Software Engineering*, vol. 33, no. 17, 2026.

[5] P. Dai, H. Zhu, J. Wu, and H. He, “An Integrated Graph Neural Network Model for Joint Software Defect Prediction and Code Quality Assessment,” *Scientific Reports*, vol. 16, Article 1677, 2026.

[6] L. Yousofvand, S. Soleimani, V. Rafe, and A. Nikanjam, “Graph Neural Networks for Precise Bug Localization through Structural Program Analysis,” *Automated Software Engineering*, vol. 33, no. 17, pp. 1–28, 2026.

[7] Ragno, M. Plantevit, and C. Robardet, “On Logic-based Self-Explainable Graph Neural Networks,” in *Proc. 39th Conf. on Neural Information Processing Systems (NeurIPS)*, 2025, pp. 1–29.