Multi Agent AI-Powered Codebase Debugger & Analysis System

[1]Binish Moosa, [2]Nishad Joglekar, [3]Samar Khan, [4]Kashif Shaikh
[1234]Students
[1234]Department of Computer Engineering,
[1234]Rizvi College of Engineering, Mumbai, India
[1]binishmoosa12@eng.rizvi.edu.in, [2]nishad@eng.rizvi.edu.in,
[3]samarkhan66554@eng.rizvi.edu.in, [4]sheikhkashif0406@eng.rizvi.edu.in

Prof. Shiburaj Pappu
Department of Computer Engineering.
Rizvi College of Engineering, Mumbai, India
shiburaj@eng.rizvi.edu.in

***Abstract*— The growing size and complexity of modern software systems make it hard for developers to understand, debug, and maintain large codebases effectively. Traditional static analysis tools are helpful but often struggle to explain the intent and behavior of code in simple language. To tackle this issue, this project presents an AI-powered multi-agent system that analyzes, interprets, and explains source code in various programming languages. The system combines static code analysis, machine learning embeddings, and local large language model (LLM) reasoning. This approach gives developers an intelligent assistant that can improve understanding, debugging speed, and overall productivity.**

The system's structure relies on a multi-agent framework, where each agent focuses on a specific part of code comprehension. The Architect Agent handles structural analysis using tools like Tree-sitter and networkx. The Vulnerability Agent seeks out potential weaknesses with static analysis tools like Semgrep. Meanwhile, the Explainer Agent uses a locally hosted Code Llama model to generate human-readable descriptions of code behavior. All agents connect through a shared FAISS-based vector database.

A notable feature of the proposed system is its fully offline operation; this ensures data confidentiality and removes dependence on external cloud-based APIs. The modular design permits flexibility and scalability. The entire system is created in Python.

***Index Terms*— Artificial Intelligence, Multi-Agent System, Code Analysis, Local LLM, Software Debugging.**

I. Introduction (Heading 1)

The understanding, debugging, and maintenance of large-scale codebases have become increasingly complex with the exponential growth of modern software systems. Developers spend a significant amount of their time in code analysis and understanding, identifying dependencies, tracing logic, explaining existing code, etc. Software maintenance and debugging take up more than 50% of the time spent by developers due to the lack of automatic code comprehension and analysis tools. Traditional static analyzers are usually language-specific and have limited semantic understandings. So, the absence of integrated tools that explain, visualize, and analyze code behavior across multiple programming languages has formed a major bottleneck in software engineering productivity.

Addressing the shortcomings, this project introduces an AI-based Codebase Analysis and Explanation System, powered by locally hosted Large Language Models. The system is designed for code understanding, the detection of vulnerabilities, and providing human-like explanations based on a multi-agent architecture with no dependency on any external APIs.

The primary objectives of this research are:

- To develop an AI-powered multi-agent system capable of understanding and explaining large codebases.
- To integrate static code analysis, machine learning embeddings, and local LLM reasoning for cross-language comprehension.
- To design agents specialized in architecture mapping, security vulnerability detection, and code explanation.

- To ensure complete offline execution using open-source, self-hosted components to maintain data privacy.

II. Related Work (Heading 1)

Over the past few years, the area of software engineering has witnessed considerable research contributions related to the application of LLMs in different activities of the development life cycle. The current paper reviews the relevant literature in three areas: the use of LLMs in code comprehension, in-IDE code-specific analysis tools, and multi-agent systems applied to code tasks.

A. LLMs Applied for Source Code Analysis (Heading 2)

Recent surveys have provided a comprehensive overview of the application of LLMs to source code analysis. Review the landscape of code comprehension and explore how different LLM architectures are used to comprehend and model code [4]. Provide a systematic review of LLMs for source code analysis in order to categorize the existing models, applications, and datasets [6]. Both reviews conclude that while LLMs show immense promise, context window limitations, model hallucinations, and the need for fine-tuning on domain-specific data remain challenges.

B. Code Summarization and Explanation Frameworks (Heading 2)

A few notable frameworks have been developed to help developers gain a better understanding of code. The PROCONSUL system, through which Formal Program Analysis is integrated with LLMs to construct the project-level context concisely [1]. Although strong in grounding the model on the formal analysis side and hence reducing hallucinations, its framework is heavily focused on callee functions and retains certain language dependencies.

Addressing in-IDE assistance, GILT, a VS Code plugin using GPT-3.5-turbo to provide context-sensitive explanations for highlighted code [2]. A user study demonstrates significant improvement over state-of-the-art web searches. However, this tool and most of its variants depend on external, cloud-based APIs. This makes them unsuitable for environments with strict data privacy policies or those requiring offline capabilities, a key gap that our project intends to fill.

C. Multi-Agent Systems in Software Engineering (Heading 2)

The use of multi-agent mechanisms for code generation and debugging is a relatively new area of research. Compared the different multi-agent roles that may be used-namely, Analyst, Coder, and Tester-along with runtime debugging, to improve the accuracy and reliability in code generated using LLM [3]. Their results showed that certain combinations of these methods can be more effective; at the same time, increasing the number of agents is clearly linked to increased latency. Similarly, the Codesim framework [5] uses collaborative multi-agent code generation and problem solving via simulation-driven planning and debugging. However, while powerful in their own regard, these systems [3, 5] mainly focus on new code generation rather than analysis and debugging of any existing large codebases.

D. Identified Research Gap (Heading 2)

A look through the literature shows a clear trend towards employing LLMs for code summarization [1] and API-based in-IDE assistance [2]. Multi-agent systems also showed promise in code generation [3,5]. However, there is still an apparent gap where such integrated systems combine a multi-agent framework with locally hosted LLMs for comprehensive codebase analysis, explanation, and debugging in a completely offline environment. Our project directly addresses this gap by proposing a modular, multi-agent system based on local LLMs in order to provide an efficient, secure analytical tool for developers.

III. Proposed System (Heading 1)

The proposed system is designed to enhance the understanding, debugging, and visualization of large-scale software projects. It overcomes limitations of existing tools through its use of multiple specialized AI agents that collaborate in the analysis, visualization, and explanation of code behavior. The system combines traditional static analysis with modern AI frameworks; mainly LLMs and multi-agent reasoning architectures.

A. System Architecture (Heading 2)

The system architecture consists of four key layers:

- Input Layer: This layer is responsible for the ingestion of source code. It utilizes the Tree-sitter parser to quickly generate Abstract Syntax Trees and extract structural metadata across a wide array of languages.

- Embedding Layer: The embedding layer uses pre-trained open-source models to create contextual embeddings from code snippets. These vector representations are kept in a FAISS vector database that allows for fast similarity searches supporting a Retrieval-Augmented Generation pipeline.

- Agent Layer: It is the core of the system, implementing various collaborative agents orchestrated using LangGraph. Among the specialized agents involved are:

  – Architect Agent: Performs dependency and architectural mapping. It utilizes NetworkX to create these dependency graphs that show the relationships that different modules and functions have, thus explaining the overall architecture of the project.

  – Vulnerability Agent: Performs static security analysis. Through the use of pattern-based static analysis tools like Semgrep, this tool will analyze source code for potential weaknesses and bugs without actually executing it.

  – Explanatory Agent: This system provides natural language explanations of code logic. Utilizing a locally hosted Code Llama model, it produces summaries in human-readable form-what the code does, how logic flows, and intended functionality.

- Visualization Layer: This layer provides an interactive front-end dashboard developed in React.js/D3.js. It is used to visualize analysis results, code structures, and the interactive dependency graphs.

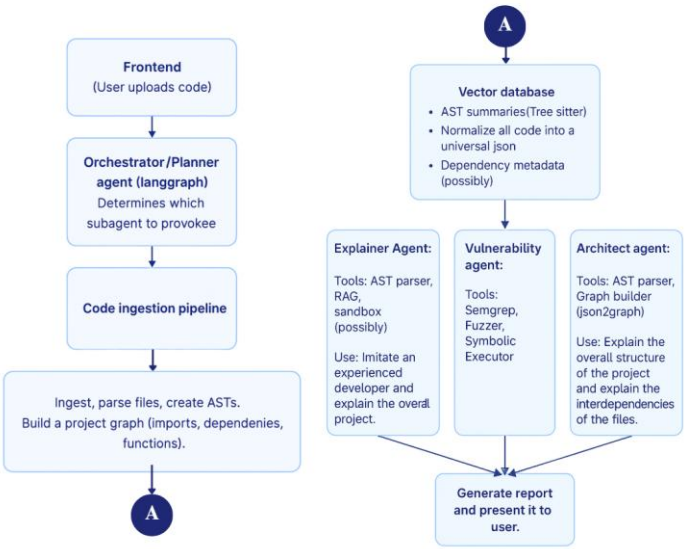

Fig 1 System Architecture

The whole system is designed to run fully offline, ensuring data confidentiality and eliminating dependencies on any external APIs.

B. Core Algorithms and Data Flow (Heading 2)

Data flow through the system starts when a user uploads source code. The Code Ingestion Pipeline parses the files to build ASTs and generates code embeddings, which are then stored into the FAISS vector database.

The Orchestrator/Planner Agent receives the request for analysis and decomposes the problem into subtasks by using a multi-agent task allocation algorithm. Then, it routes these tasks to the specialized agents: Architect, Vulnerability, Explainer.

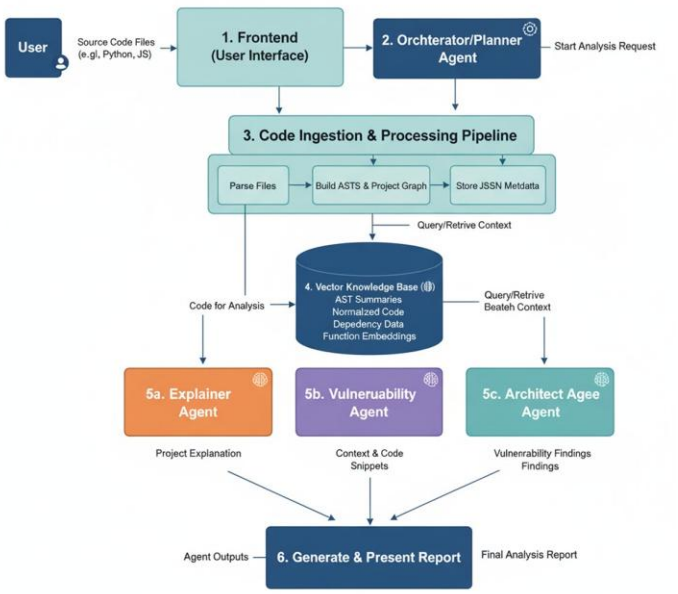

Fig. 2 System Overview: Data Flow Diagram

Each agent queries the vector database for relevant context: embeddings and AST summaries. The agents perform their specialized analysis:

- AST Generation represents the syntactic structure of code.
- Code Embedding Generation means converting code snippets into numerical vectors for semantic comparison.
- Dependency Graph Construction models the codebase as a directed graph G = (V, E) to visualize module relationships.
- Vulnerability Detection employs static pattern matching against a pre-defined rule set R.
- The LLM generates a Natural Language Explanation by predicting an explanation sequence Y based on code Ci and context X.

Finally, the output of all agents is combined into one final analysis report, which is then presented to the user through the dashboard.

IV. Implementation and Results (Heading 1)

The system is currently being developed in two main phases. In Phase 1, the work done consisted of research, architecture design, and technology selection. More specifically, the selection was among Tree-sitter for parsing, FAISS for vector search, NetworkX for graph analysis, and Semgrep for vulnerability detection. Among the important decisions was the selection of Code Llama for offline inference so that its capabilities would meet the project's requirements with respect to code understanding without making calls to any APIs.

Currently ongoing is Phase 2, which involves system development and integration. Python is used, with FastAPI serving to create high-performance, asynchronous APIs for communication between agents; the frontend dashboard is being built in React.js with Tailwind CSS for a responsive component-based user interface.

LangGraph will orchestrate the multi-agent framework, defining the collaborative workflows as a directed graph. Within this framework, it will manage the Planner Agent, Architect Agent, Vulnerability Agent, and Explainer Agent.

Currently, the core backend architecture is under implementation, and the development of a visualization dashboard is in progress. The integration of these will be done in further steps, followed by performance benchmarking that will quantify the impact of the proposed system on debugging efficiency and comprehension accuracy.

V. Conclusion (Heading 1)

The proposed system is poised to take a leap in software debugging, integrating multi-agent systems, retrieval-augmented generation, and local large language models into an innovative format. The successful realization of this offline-capable framework will meet the most critical challenges present in security-sensitive environments where connectivity to the internet is not practical or prohibited.

The core innovation is the specialized multi-agent architecture; the collaborative framework of Planner, Architect, Vulnerability, and Explainer Agents will enable comprehensive, multi-perspective debugging. Such an approach is likely to be more comprehensive and contextually aware than what traditional methods could have allowed. The RAG pipeline will ensure that responses are accurately grounded in the provided code context. Once developed, performance testing will be carried out to verify the efficiency of the system. The proposed system is supposed to set a standard for intelligent code analysis tools, proving that such advanced AI-powered debugging is effective and secure even in a completely offline environment.

VI. Acknowledgment

The authors would like to thank Prof. Shiburaj Pappu for his invaluable guidance and support throughout this project.

References

[1] V. Lomshakov, A. Podivilov, S. Savin, O. Baryshnikov, A. Lisevych, and S. Nikolenko, "Proconsul: Project context for code summarization with llms," Journal of Systems and Software, vol. 7, no. 4, pp. 325–339, 2024.

[2] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), Apr. 2024, pp. 1–13.

[3] 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, May 2025.

[4] J. Cui, Y. Zhao, C. Yu, J. Huang, Y. Wu, and Y. Zhao, "Code comprehension: Review and large language models exploration," in Proceedings of the 2024 IEEE 4th International Conference on Software Engineering and Artificial Intelligence (SEAI), IEEE, Jun. 2024, pp. 183–187.

[5] Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging, arXiv preprint, 2025.

[6] H. Jelodar, M. Meymani, and R. Razavi-Far, Large language models (llms) for source code analysis: Applications, models, and datasets, arXiv preprint, Mar. 2025.