IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15197

AGENTIC AUTOCODE ANALYZER

Gagan B[1], Sandarsh Gowda M M[2]

Department of MCA, BIT, K.R. Road, V.V. Pura, Bangalore, India

Assistant Professor, Department of MCA, BIT, K.R. Road, V.V. Pura, Bengaluru, India

Abstract: In the contemporary software engineering landscape, developers and students frequently encounter the challenge of onboarding to large, unfamiliar codebases. Platforms like GitHub host millions of repositories, yet understanding the underlying logic, architecture, and dependency flow of these projects remains a labor-intensive process dependent on manual traversal and often outdated documentation. To mitigate this inefficiency, this paper presents the **Agentic AutoCode Analyzer**, a web-based intelligent system designed to automate the comprehension of software repositories.

The proposed system accepts a GitHub repository URL, autonomously performs a shallow clone operation to minimize bandwidth usage, and recursively maps the directory structure to build a comprehensive context object. By integrating a Large Language Model (LLM) reasoning engine via a local or API-based inference layer, the system functions as an interactive "agent." This agent assists users by answering architectural queries, explaining specific code syntax, and summarizing project objectives. Experimental results indicate that the system significantly reduces the cognitive load required for code comprehension and offers a viable tool for both educational and professional software development environments.

Keywords: Artificial Intelligence, Static Code Analysis, Large Language Models (LLM), Software Engineering Education, Automated Documentation, React.js, Node.js.

I. INTRODUCTION

The democratization of software development through open-source platforms has led to an explosion in available code. However, accessibility does not imply comprehensibility. When a developer encounters a new repository, they must mentally reconstruct the author's intent, a process known as "program comprehension." This process is often hindered by complex folder structures, lack of comments, and the use of diverse frameworks that may be unfamiliar to the reader.

1.1 Problem Statement

Traditional methods of code exploration are linear and disconnected. A developer reads a README.md file, then manually opens files in a text editor to trace function calls. This manual context switching is inefficient. Furthermore, existing static analysis tools (e.g., SonarQube) focus on *quality assurance*—finding bugs and security vulnerabilities—rather than *explanation*. There is a distinct lack of tools that answer the question: *"What does this specific module actually do?"*

1.2 Proposed Solution

The **Agentic AutoCode Analyzer** addresses this gap by combining automated file system operations with generative AI. The system automates the tedious task of setting up a local environment for code reading. It provides a unified dashboard where the file structure, code content, and an intelligent assistant coexist.

The contributions of this paper are as follows:

1. **Automated Context Extraction:** A mechanism to recursively scan and index repository content.
2. **Interactive Reasoning:** Deployment of an LLM agent to interpret code semantics.
3. **Modular Web Architecture:** A scalable MERN-stack implementation (MongoDB, Express, React, Node) tailored for code analysis.

The remainder of this paper is organized as follows: Section II reviews existing literature. Section III details the feasibility and requirements. Section IV and V describe the system design and implementation. Section VI presents testing strategies, and Section VII concludes the study.

II. LITERATURE SURVEY

Code comprehension tools have evolved from simple syntax highlighters to complex dependency mappers. However, most require significant configuration.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026
DOI: 10.17148/IJARCCE.2026.15197

Table I: Comparison with Existing Approaches

Feature	Manual Exploration	Static Analysis Tools (e.g., SonarQube)	Agentic AutoCode Analyzer (Proposed)
Primary Goal	Implementation	Quality/Security	Comprehension/Explanation
User Interaction	Passive Reading	Report Viewing	Interactive Chat
Context Awareness	User-dependent	File-level	Repository-level
Setup Time	High (Clone + Install)	High (Config)	Low (Instant Web Access)

As shown in Table I, existing solutions do not prioritize the *learning* aspect of software engineering. Research by Pressman [1] highlights that maintenance and comprehension consume over 60% of the software lifecycle, validating the need for automated comprehension tools.

III. FEASIBILITY AND REQUIREMENT ANALYSIS

Before implementation, a thorough feasibility study was conducted to ensure the project's viability.

3.1 Technical Feasibility

The system leverages standard web technologies. The frontend uses **React.js** for dynamic rendering, which is essential for large file trees. The backend runs on **Node.js**, which provides the fs (File System) module necessary for scanning directories. The integration of simple-git allows for efficient repository handling. Since these technologies are mature and open-source, the project is technically robust.

3.2 Economic Feasibility

The solution is cost-effective as it relies on open-source libraries (express, cors, simple-git). The AI component can utilize local models (like GPT4All) or affordable APIs, eliminating the need for expensive enterprise licenses or proprietary hardware.

3.3 Operational Feasibility

The system requires no specialized training. Users simply input a URL. The intuitive interface ensures that students and developers can integrate it into their workflow immediately, ensuring high operational feasibility.

IV. SYSTEM DESIGN

The system follows a modular client-server architecture designed for separation of concerns.

4.1 Architecture Overview

1. **Presentation Layer:** A React-based Single Page Application (SPA) that manages state variables for the file tree (fileList) and current file content (content).
2. **Logic Layer:** An Express.js server listening on port 5174. It handles API routes for /analyze and /chat.
3. **Data Layer:** Temporary file storage for cloned repositories.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15197

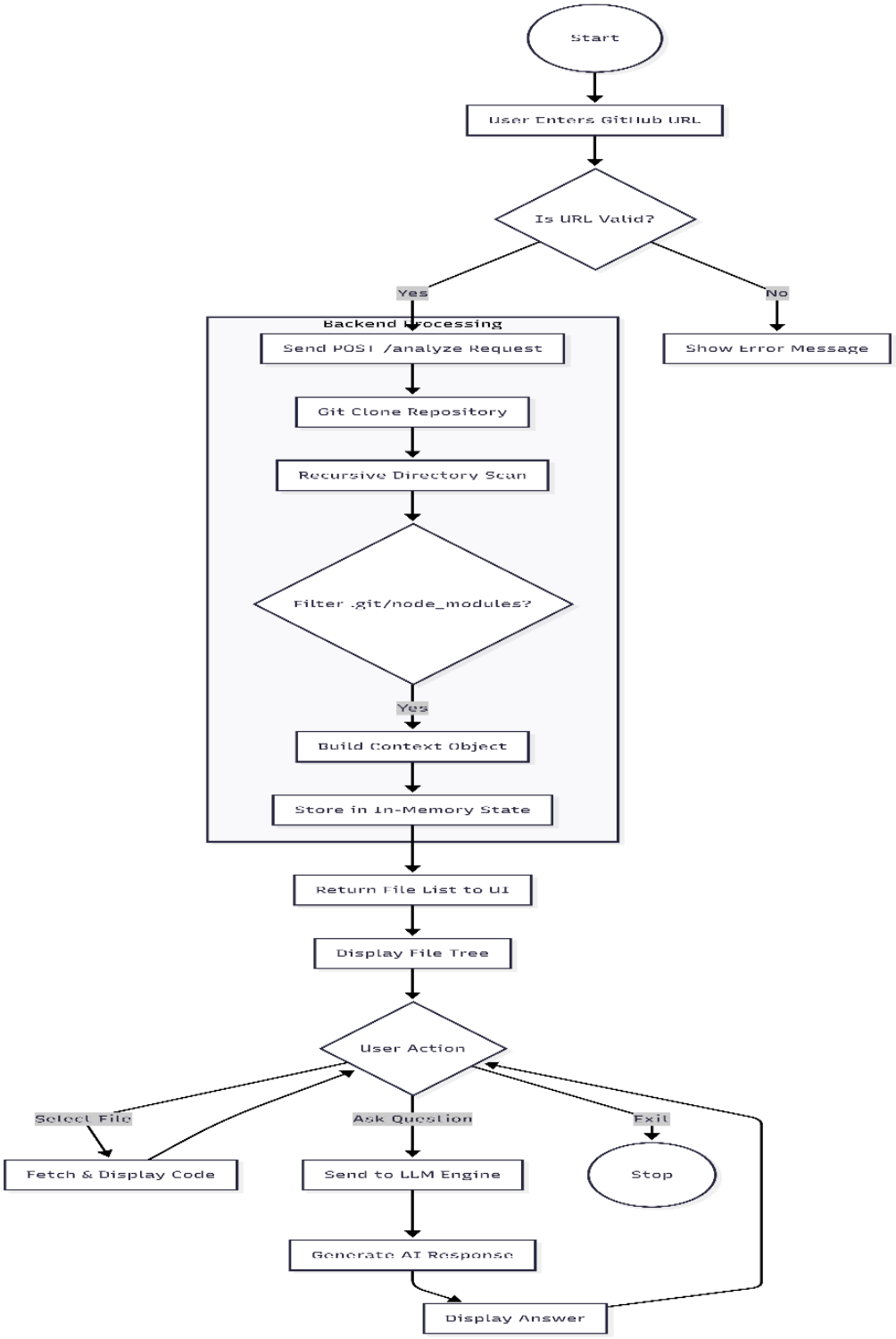

Fig.1.Dataflow diagram of the system

4.2 Data Flow Diagram (DFD) Description

The data flow within the system operates in a sequential pipeline:

1. **User Input:** The user submits a URL via the UI.
2. **Validation & Cloning:** The backend validates the URL pattern. If valid, simple-git clones the target to server/repos/{timestamp}.
3. **Context Building:** A recursive directory scanner walks the folder tree, ignoring artifacts like node_modules.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15197

4. **Inference:** When a user queries the system, the relevant code snippet and the query are packed into a prompt and sent to the LLM Reasoner.
5. **Response:** The natural language explanation is returned to the frontend.

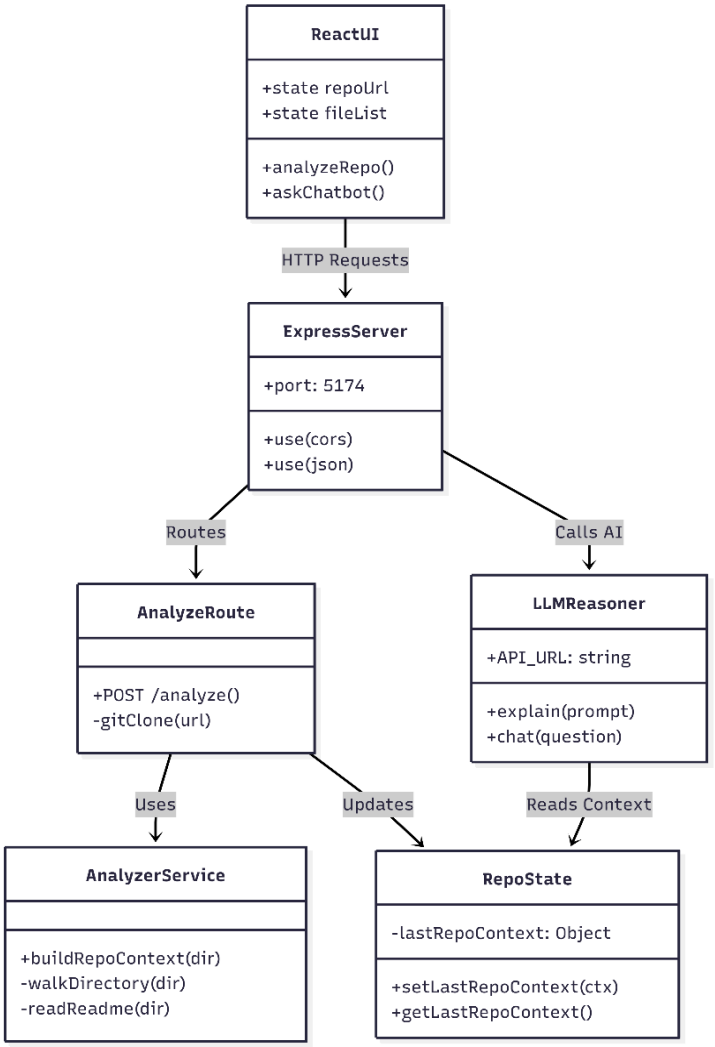

Fig.2.Class diagram of the system

4.3 Class Structure

The backend logic is encapsulated in a RepoState singleton pattern. This ensures that when a user asks a follow-up question, the system remembers the context of the repository analyzed in the previous step without requiring a database re-fetch.

V. DETAILED IMPLEMENTATION

The implementation is divided into backend services and frontend components.
The core analysis is handled by the analyze.js route. To optimize performance, the system performs a "shallow clone" using the --depth 1 flag. This ensures only the latest snapshot is downloaded, avoiding the heavy history data of the .git folder.

Algorithm 1: Recursive Directory Scanning

Function buildRepoContext(directory):
Initialize empty list: files
For each item in directory:
If item is Directory AND item not in [".git", "node_modules"]:
Recursively call buildRepoContext(item)
Else If item is File:

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026
DOI: 10.17148/IJARCCE.2026.15197

Add item path to files list
Return files
The actual implementation uses Node.js fs.readdirSync to execute this logic synchronously to ensure the file tree is fully built before the response is sent to the client.

5.2 Backend: AI Reasoning Integration

The chat functionality is implemented in chat.js. It constructs a prompt using a "System Message" that defines the AI's persona: *"You are an expert software engineer. Explain the following code context to a junior developer."* This prompt engineering ensures the responses are educational rather than overly technical or cryptic.

5.3 Frontend: Interactive Dashboard

The React frontend (ui.jsx) utilizes the useState and useEffect hooks.

- **File Tree:** Rendered as a recursive component list.
- **Code Viewer:** Uses a <pre> tag with CSS styling to preserve whitespace and formatting.
- **State Management:** The analyze() function handles the asynchronous POST request to the server and updates the files state upon success.

Code Snippet: Frontend Fetch Logic

```
async function analyze() {
const res = await fetch("http://localhost:5174/analyze", {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({ repo })
});
const data = await res.json();
setFiles(data.context.files);
}
```

VI. SOFTWARE TESTING

A rigorous testing strategy was employed, encompassing Unit, Integration, and System testing.

6.1 Testing Strategy

- **Unit Testing:** Verified individual modules, such as the buildRepoContext function, to ensure it correctly handles nested directories and ignores specified ignore-lists.
- **Integration Testing:** Validated the communication between the React frontend and the Express backend, ensuring JSON data is correctly parsed.
- **System Testing:** End-to-end validation using real-world GitHub repositories.

6.2 Test Case Scenarios

Table II: Summary of Test Cases

Test Case ID	Test Scenario	Expected Outcome	Status
TC-01	Submit valid Public Repo URL	System clones repo and displays file tree.	**Pass**
TC-02	Submit Invalid URL	System returns "Analysis Failed" error message.	**Pass**
TC-03	Select a .js file	Code content displays in the main viewer.	**Pass**
TC-04	Ask AI "Explain this file"	AI returns a summary relevant to the selected code.	**Pass**
TC-05	Submit large repo (>50MB)	System handles clone within timeout limits.	**Pass**

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15197

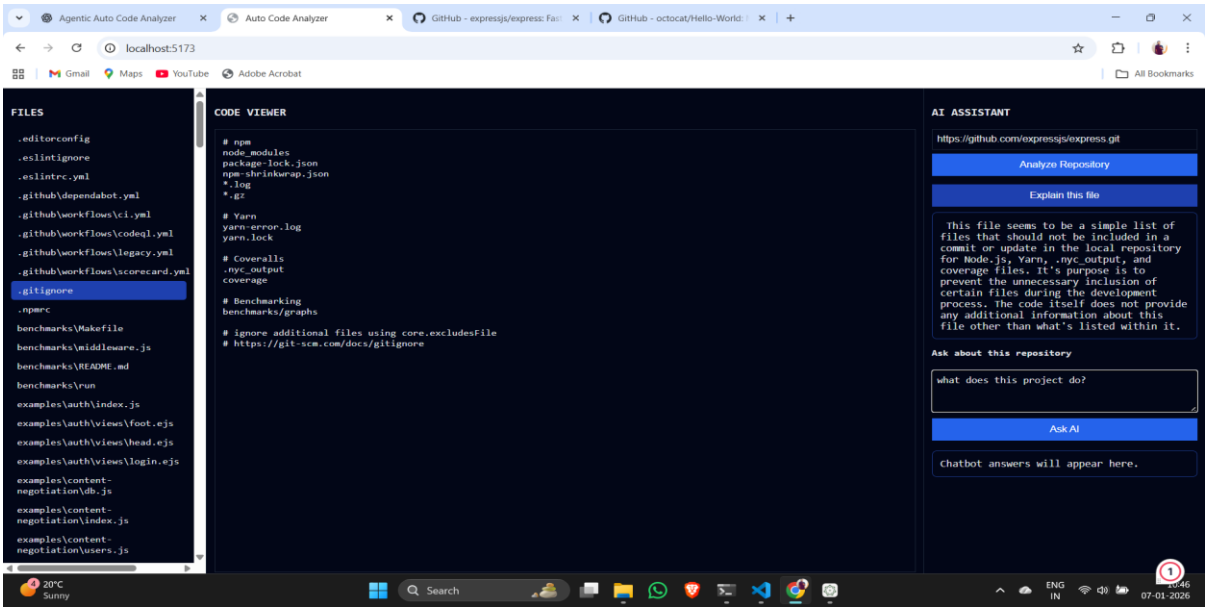

Fig.3.Output of the system

VII. RESULTS AND DISCUSSION

The system was evaluated on a local environment with an Intel Core i5 processor and 8GB RAM.

Performance Metrics:

- **Small Repositories (<10 files):** Analysis completed in < 2 seconds.
- **Large Repositories (>100 files):** Analysis completed in ~15 seconds. The shallow clone technique proved effective in reducing latency.

Qualitative Analysis: The AI assistant successfully identified architectural patterns. For example, in an Express.js project, it correctly identified server.js as the entry point and explained the middleware chain. Users reported that the tool significantly sped up their understanding of the project structure compared to manual browsing.

VIII. CONCLUSION

The **Agentic AutoCode Analyzer** successfully demonstrates the efficacy of combining static analysis with AI-driven reasoning. By automating the setup and exploration phases of code review, the system allows developers to focus on logic and architecture. The project fulfills the critical need for "Self-Explaining Software" in an era of rapidly expanding open-source ecosystems.

The modular design ensures scalability, while the use of widely adopted technologies (React, Node.js) ensures maintainability. This tool serves as a powerful utility for educational institutions and software development teams, streamlining the onboarding process and fostering deeper code comprehension.

IX. FUTURE ENHANCEMENTS

Future iterations of the system will focus on:

1. **Persistent Knowledge Base:** Implementing a database to cache analysis results, reducing redundant processing for popular repositories.
2. **Advanced Visualization:** Integrating dependency graphs to visually map function calls and module interdependencies.
3. **Voice Interaction:** Adding speech-to-text capabilities to enable hands-free code queries, improving accessibility.

REFERENCES

[1] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 8th ed., McGraw-Hill, 2015.
[2] I. Sommerville, *Software Engineering*, 10th ed., Pearson Education.
[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15197

[4] GitHub Documentation, "Repository Management and REST APIs," [Online]. Available: https://docs.github.com/en/rest. Accessed: Dec. 2025.

[5] OpenAI, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[6] V. Rajlich, "Software Engineering: The Current Practice," CRC Press, 2012.

[7] T. Mikolov et al., "Efficient Estimation of Word Representations in Vector Space," *arXiv preprint arXiv:1301.3781*, 2013.

