Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

CodeInsight AI: A Context-Aware Framework for Automated Codebase Comprehension through Dependency Analysis, Interactive Visualization, and Large Language Models

B Yashwanth1, Akhila A2, Bindu N G2, Basavaraj H A2, Darshan Totagi2
1Assistant Professor, Department of Information Science and Engineering
2UG Scholars, Department of Information Science and Engineering
1,2GM Institute of Technology, Davanagere

Abstract **- Understanding an unfamiliar software repository often means piecing together information scattered across files, folders, imports, and architectural layers. Dependency tools can show how files relate to one another, while AI coding assistants can explain code, but developers do not always get these capabilities in one place. CodeInsight AI brings them together in a web-based environment. It scans ZIP archives or Git repositories, identifies supported source files, extracts imports with regular expressions, resolves local references, groups files using practical architectural heuristics, and displays the resulting dependency graph. Project information is stored in PostgreSQL, and selected details are added to the prompt sent to Gemini when a developer asks a question. In the evaluation, JavaScript/TypeScript dependency extraction achieved 86.2% precision, 96.2% recall, and an F1 score of 90.9% against Madge. The Python comparison reported 100% concordance for internal dependencies against a reference built with Python's native ast module, and architectural classification reached 97.6% accuracy. Five-run benchmarks covered repositories from 60 to 1,000 source files. In a study involving 14 undergraduate students, the CodeInsight AI group completed comprehension tasks 60.3% faster, answered more accurately (92.9% versus 64.3%), and gave the system a mean System Usability Scale score of 84.6 out of 100.**

Index terms **- Codebase Comprehension, Dependency Analysis, Software Architecture Recovery, Static Code Analysis, Graph Visualization, Large Language Models, Software Maintenance, Software Engineering.**

I. INTRODUCTION

Modern applications are built from reusable components, services, and modules that may span several programming languages. As a repository grows, a developer joining the project must often trace imports, work out which files matter most, and infer the role of each directory before making a change with confidence. Software architecture recovery has long examined how higher-level structure can be reconstructed from implementation artifacts [1], [2]. Related approaches have used machine learning and microservice-specific analysis to recover architectural information [3], [4]. Static-analysis methods also investigate how components can be identified across different programming languages [13].

Large language models (LLMs) have been studied for software engineering tasks such as code understanding, generation, and developer assistance [5], [6]. Pretrained models such as CodeBERT and CodeT5 support code understanding and generation [18], [19]. Repository-level systems extend this work in different directions: RepoCoder uses iterative retrieval and generation for code completion [7], CodePlan plans repository-level changes [8], and RepoAgent supports repository documentation [9]. These studies point to the value of giving AI tools broader project context. CodeInsight AI takes a complementary approach: it combines selected project context with an explicit file-dependency graph, heuristic architectural grouping, and an interface designed for exploring a codebase.

CodeInsight AI keeps the structural analysis separate from the language model. Regular expressions identify import statements, and path- and language-specific rules are then used to resolve references to local files. Developers can inspect the resulting graph and see which project details are supplied to the assistant. This separation also makes an important distinction: dependency relationships are extracted by deterministic rules, while explanations are generated by the language model and may still require verification.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

This work makes five contributions: a multi-language dependency extraction pipeline based on regular expressions; a heuristic method for grouping files by architectural role; an interactive graph for exploring file relationships; a conversational assistant that uses selected project context; and an evaluation covering dependency correctness, classification, repeatability, performance at different repository sizes, and usefulness to developers.

The remainder of this paper is organized as follows. Section II reviews related work. Section III presents the proposed framework and methodology, and Section IV describes the system implementation. Section V reports the experimental evaluation and results. Section VI discusses the limitations, and Section VII concludes the paper and outlines future research directions.

II. RELATED WORK

Software code comprehension has emerged as a fundamental research area in software engineering due to the increasing complexity of modern software systems. Researchers have proposed numerous approaches to facilitate architectural understanding through software architecture recovery, static code analysis, dependency visualization, and, more recently, Large Language Model (LLM)-assisted software engineering. Although significant progress has been achieved in each of these domains, existing approaches generally address individual aspects of software comprehension rather than providing an integrated environment that combines structural analysis, interactive visualization, and intelligent conversational reasoning.

A. Software Architecture Recovery

Software architecture recovery seeks to reconstruct a system's higher-level structure from its source code and related implementation artifacts. Chikofsky and Cross provided an early foundation for reverse engineering and design recovery [1]. Later studies compared recovery techniques and explored machine-learning-based recovery, microservice architecture reconstruction, and directory-based dependency processing [2]–[4], [10]. Other work combines structural, semantic, and directory information or fuses multiple sources of evidence to improve the recovered view [11], [12]. Research has also examined centrality-based architecture analysis, automatic class-diagram generation, microservice reconstruction, and LLM-assisted architecture description [16], [17], [23], [24]. CodeInsight AI addresses a related problem, but deliberately uses lightweight path and keyword rules rather than attempting full architectural inference.

B. Static Dependency Analysis

Static analysis extracts relationships without executing the application. Research has examined machine-learning-based architecture recovery and language-agnostic component identification as ways to support architectural analysis and software maintenance [3], [13]. In practice, dedicated tools provide complementary capabilities. Madge creates module-dependency graphs and can help identify circular dependencies in JavaScript and TypeScript projects [14], while pydeps visualizes Python module dependencies and import cycles [15]. CodeInsight AI is broader in its intended interface because it combines dependency extraction with cross-language metadata, architectural grouping, graph interaction, and conversational explanation.

C. Visualization and Repository-Level AI Assistance

A graph can make a repository's structure easier to follow because developers can inspect relationships, highlight nodes, and move through connected files. Research on code models and repository-level assistance addresses a different but related need. CodeBERT and CodeT5 provide pretrained approaches for code understanding and generation [18], [19]. RepoCoder uses repository context during iterative code completion [7], CodePlan supports planned repository-level changes [8], and RepoAgent focuses on generating repository documentation [9]. CodeInsight AI differs in its main purpose: the graph and dependency data remain visible to the developer, while the LLM provides explanations based on selected project information rather than acting as the dependency analyzer.

D. Research Gap

These tools solve overlapping but distinct problems. SonarQube is primarily concerned with code quality and security [20], while GitHub Copilot offers AI-assisted coding and repository-aware support [21]. Madge and pydeps focus on dependency visualization for JavaScript/TypeScript and Python projects, respectively [14], [15]. CodeInsight AI is designed for the space between these capabilities: it presents file-level dependencies, groups files by likely architectural role, supports interactive exploration, and lets developers ask questions using project context.

III. PROPOSED FRAMEWORK

This section presents the overall design and operational workflow of CodeInsight AI, a context-aware software comprehension framework that integrates automated dependency analysis, architectural visualization, and conversational intelligence. The framework is designed to assist developers in understanding unfamiliar software repositories by combining structural dependency information with Large Language Model (LLM)-based

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

reasoning. Unlike conventional software visualization systems that generate static dependency graphs or AI assistants that operate solely on textual code snippets, the proposed framework unifies structural analysis, interactive visualization, and context-aware conversational assistance within a single development environment. The overall workflow of the proposed system is illustrated in Fig. 1.

The framework consists of six major phases:

1. Project Acquisition
2. Source Code Analysis
3. Dependency Graph Construction
4. Architectural Classification
5. Knowledge Repository Generation
6. Context-Aware Conversational Assistance

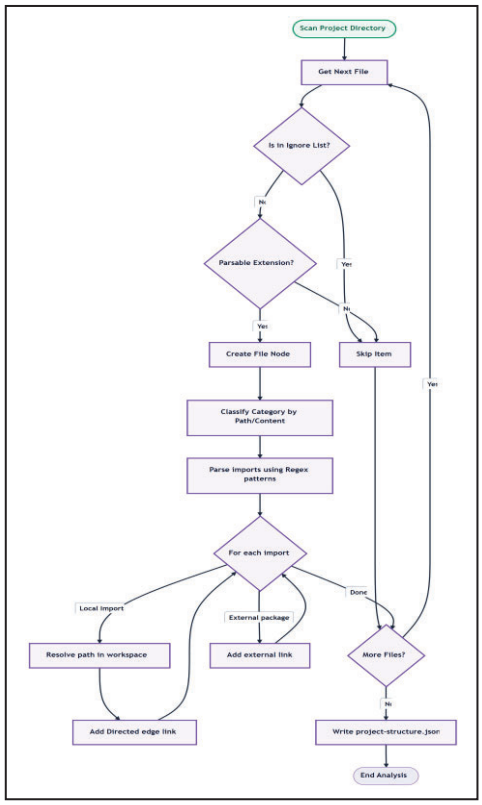


Fig. 1. Workflow of the proposed code analysis and dependency extraction process.

Each phase is described in the following subsections.

A. Project Acquisition

The analysis process begins by accepting software projects either as compressed ZIP archives or through publicly accessible Git repository URLs. Uploaded archives are extracted into an isolated workspace, whereas Git repositories are automatically cloned before analysis.

The framework recursively scans the project directory while excluding folders that do not contribute to architectural understanding. Common excluded directories include:

- node_modules
- .git
- .venv
- build
- dist
- target
- __pycache__

This preprocessing stage reduces computational overhead by eliminating third-party libraries, generated artifacts, and temporary files while preserving only developer-authored source code.

B. Source Code Analysis and Dependency Extraction

Following project acquisition, the framework performs automated analysis of supported source files, including JavaScript, TypeScript, Python, Java, and Go. Each file is processed individually to extract software metadata, programming language information, directory structure, and dependency relationships.

Instead of depending on a single parser, the framework employs a language-aware dependency extraction pipeline that combines regex pattern matching with language-specific resolution strategies. This design enables the framework to analyze heterogeneous repositories while maintaining scalability across different programming languages.

For each source file fi, the extracted dependency set is represented as,

$$D(f_i)=\{d_1,d_2,\ldots,d_n\}$$

where:

- f_i represents the current source file,
- $D(f_i)$ denotes the set of extracted dependencies,
- dn represents an individual imported module.

The extracted dependency information forms the basis for subsequent graph construction and architectural analysis.

C. Dependency Graph Construction

After dependency resolution, the framework constructs a directed software dependency graph that represents structural relationships among project files.

The graph is formally defined as

$$G=(V,E)$$

where:

- V represents the set of source files,
- E represents directed dependency relationships between files.

Each node corresponds to a source file, and each directed edge indicates that one file depends on another through an import or module reference. Duplicate edges

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

are collapsed before visualization. Intermediate placeholder nodes that do not correspond to analyzed source files are omitted, so the displayed graph focuses on resolved file-to-file relationships.

D. Architectural Layer Classification

Although dependency graphs effectively capture structural relationships, they do not explicitly describe the architectural role of individual software modules.

To improve interpretability, the framework automatically categorizes files into logical architectural layers using directory-path analysis and heuristic keyword matching.

The classification function is expressed as

$$C=f(P,K)$$

where:

- P denotes the relative project path,
- K represents extracted keywords,
- C denotes the assigned architectural category.

The framework classifies source files into categories including:

Category	**Directory Patterns**	**Representative Keywords**
Pages	pages, views, templates	Router, render
Components	components, ui, widgets	JSX, CSS, State
Services	services, api, providers	API, CRUD, Axios
Models	models, schema, types	Interface, Schema
Controllers	controllers, resolvers	Route, Handler
Utilities	utils, helpers	Common Functions

This layered representation enables developers to understand project organization without manually inspecting every source file.

E. Interactive Dependency Visualization

The generated dependency graph is visualized using the React Flow library, while node positioning is computed using the Dagre directed graph layout algorithm to reduce edge crossings and improve readability.

The framework supports two complementary visualization modes:

Architecture View: Groups source files into high-level architectural layers to provide an overview of the software system.

Detailed Dependency View: Displays individual file-to-file relationships for fine-grained structural exploration.

To help developers identify highly connected modules, nodes are color-coded by their incoming dependency count. The thresholds are design choices made by the authors for visual clarity, inspired by the coupling and design-metric discussion in [22]; they are not thresholds prescribed by Chidamber and Kemerer. Nodes with 0–1 incoming dependencies are shown in green, those with 2–4 in amber, and those with 5 or more in red.

$$Color(v) = \begin{cases} Green, & indegree < 2 \\ Amber, & 2 \le indegree < 5 \\ Red, & indegree \ge 5 \end{cases}$$

where $d_{in}(v)$ denotes the number of incoming dependencies associated with node v.

F. Context-Aware Conversational Assistance

When a developer asks a question, CodeInsight AI builds the prompt using one consistent formulation. The context may include a project summary, file metadata, architectural categories, dependency counts, nearby files, and other relevant structural details. This is project-context-aware prompt enrichment; the implementation does not use a retrieval-augmented generation architecture.

The generated prompt is represented as

$$P=Q+C+H$$

where:

- Q denotes the user query,
- C represents the selected project context
- H denotes the recent conversational history.

The enriched prompt is submitted to the Gemini Large Language Model, enabling project-aware reasoning that incorporates both semantic and structural information. This approach is designed to supply architectural grounding by providing LLM prompts with repository-level topological dependencies and layer metadata, complementing localized code assistant prompts.

G. End-to-End Processing

Algorithm 1: CodeInsight AI Analysis Pipeline

Input: ZIP archive or Git repository URL

Output: Interactive dependency graph and context-aware project analysis

1. Acquire software project.
2. Extract or clone project repository.
3. Remove excluded directories.
4. Scan supported source files.
5. Extract dependency information.
6. Resolve language-specific import paths.
7. Construct directed dependency graph.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

8. Classify files into architectural layers.
9. Store metadata and graph information in the database.
10. Generate architectural summary.
11. Visualize dependency graph.
12. Process developer queries using graph-enhanced contextual prompts.
13. Return interactive visualization and conversational responses.

IV. SYSTEM IMPLEMENTATION

The proposed framework has been implemented as a full-stack web application that integrates automated code analysis, dependency visualization, database management, and Large Language Model (LLM)-based conversational assistance. The implementation follows a modular client-server architecture, enabling independent scalability of the user interface, backend services, and AI orchestration components. The overall implementation architecture is illustrated in Fig. 2.

The framework consists of four primary layers:

1) Presentation Layer
2) Application Layer
3) Data Management Layer
4) AI Intelligence Layer

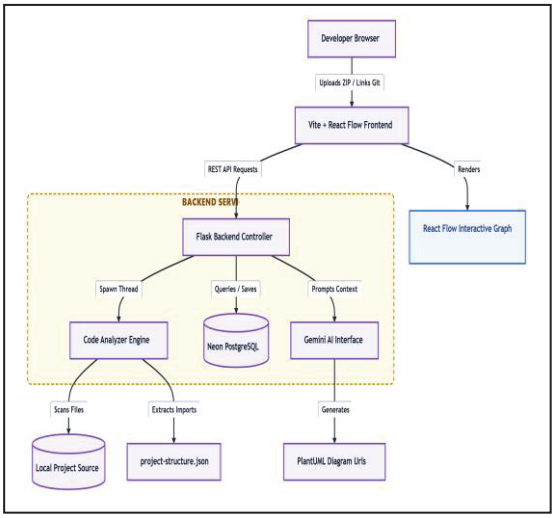


Fig. 2. Overall system architecture of CodeInsight AI.

A. Presentation Layer

The presentation layer is developed using React with Vite, providing a responsive and interactive web interface for software analysis. React's component-based architecture enables modular implementation of project upload, dependency visualization, architectural exploration, and conversational interaction.

The user interface is organized into four major modules:

- Project Upload Interface
- Interactive Dependency Viewer
- Code Navigation Panel
- Conversational AI Assistant

The Project Upload Interface allows users to analyze projects either by uploading compressed ZIP archives or by providing publicly accessible Git repository URLs. Once the project is submitted, the frontend communicates with the backend through RESTful APIs to initiate analysis.

The Dependency Viewer presents the generated software architecture using React Flow, enabling users to navigate dependencies through zooming, panning, node selection, and interactive exploration. Each node displays metadata including the file name, programming language, architectural category, and dependency statistics.

The integrated AI Assistant provides a conversational interface through which developers can query project architecture, software modules, dependencies, and implementation details using natural language.

B. Backend Processing Layer

The backend services are implemented using the Flask web framework, which coordinates project analysis, dependency extraction, metadata generation, graph construction, and AI request management.

The backend consists of the following functional services:

1) Project Manager

The Project Manager handles project uploads, repository cloning, workspace creation, extraction of compressed archives, and cleanup of temporary resources. Before initiating analysis, the service validates the uploaded repository and verifies the existence of supported source files.

2) Code Analyzer

The Code Analyzer performs recursive directory traversal while excluding non-essential directories such as third-party libraries, cache folders, and build artifacts. Each supported source file is analyzed to extract dependency relationships and software metadata.

The analyzer records:

- Relative project path
- Programming language
- File size
- Import statements
- Dependency count
- Architectural category

These attributes collectively form the software knowledge graph used throughout the framework.

3) Dependency Manager

The Dependency Manager resolves imported modules to their corresponding project files using language-aware

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

resolution strategies. The resulting directed relationships are transformed into graph edges before being forwarded to the visualization engine.

4) AI Orchestrator

The AI Orchestrator is responsible for generating contextual prompts and interacting with the Gemini Large Language Model. Rather than forwarding raw user queries, it enriches prompts with structural software information retrieved from the knowledge repository.

C. Data Management Layer

To support efficient storage and retrieval of software metadata, the framework employs PostgreSQL (NeonDB) as its persistent database.

The database schema consists of five primary entities.

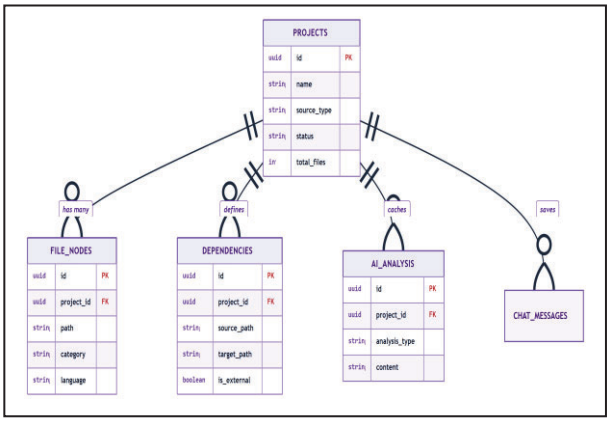


Fig. 3. Entity–Relationship (ER) diagram of the CodeInsight AI database

Projects

This table maintains project-level information including project identifier, repository location, analysis timestamp, processing status, and overall project metadata.

File Nodes

Each analyzed source file is represented as an individual record containing:

- File identifier
- Relative path
- Programming language
- File size
- Architectural category
- Complexity statistics

Dependencies

The Dependencies table stores directed relationships between software modules, enabling efficient graph reconstruction without repeated source-code analysis.

Each dependency record contains:

- Source node
- Target node
- Dependency type
- Resolution status

AI Analysis

This table stores AI-generated artifacts including:

- Project summaries
- Architectural descriptions
- UML generation results
- Repository insights

Caching generated outputs significantly reduces repeated inference time for previously analyzed projects.

Chat Messages

The Chat Messages table maintains multi-turn conversational history associated with individual projects. This enables contextual continuity during successive developer interactions.

The database schema corresponding to these entities is illustrated in Fig. 3 (ER Diagram).

V. EXPERIMENTAL EVALUATION

The evaluation was organized around four questions: (RQ1) how accurately does CodeInsight AI identify local file-to-file dependencies? (RQ2) how often are files assigned to the intended architectural categories? (RQ3) how do execution time and storage change across repository sizes and repeated runs? and (RQ4) does the tool help developers understand an unfamiliar repository faster and more accurately?

A. Experimental Environment

The benchmarks were run on a test machine with an Intel Core i5-1135G7 (11th-Gen Tiger Lake processor, 4 physical cores, 8 logical threads @ 2.40GHz base, up to 4.20 GHz Intel Turbo Boost), 16.0 GB of DDR4 RAM (15.78 GB addressable), and 64-bit Windows 11 Enterprise, Build 26200. The software environment included Python 3.12.3, Node.js 20.18.0, Vite 5.4.2, React 18.3.1, and PostgreSQL hosted through NeonDB.

Component	Configuration
Frontend	React (Vite), React Flow
Backend	Flask (Python)
Database	PostgreSQL (NeonDB)
Graph Layout Engine	Dagre
LLM	Google Gemini 1.5 Flash
Operating Environment	Web-based Client–Server Architecture

B. Repository-Based Structural Validation

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Five repositories were selected to represent different sizes and technology stacks: Backend Subsystem (Flask/Python REST API), Sathyagraha AI (Python/ML pipeline), UniversalDB (full-stack database tool), CodeInsight AI Core (TypeScript/React/Flask), and Code Sentinel (Electron/React/TypeScript). Their recorded sizes were 60, 105, 204, 251, and 264 source files, respectively. A synthetic enterprise project containing 1,000 files and 3,450 dependencies was also used to test larger-scale performance.

For the Backend Subsystem repository, which contains 60 source files and implements a Flask/Python REST API, a reference visitor built with Python's native ast module identified 20 internal module-to-module dependencies and 193 external third-party dependencies, for 213 import declarations in total. CodeInsight AI recovered all 20 internal dependencies, with no false positives or false negatives, yielding 100% precision and 100% recall for internal dependencies.

TABLE I. Repository based Structural Validation

Repository	**Stack; files; runs**
Backend Subsystem	Flask/Py REST API; 60 files; N=5
Sathyagraha AI	Python/ML Pipeline; 105 files; N=5
UniversalDB	Full-Stack DB Tool; 204 files; N=5
CodeInsight AI Core	Polyglot TS/React/Flask; 251 files; N=5
Code Sentinel	Electron/React/TS; 264 files; N=5

For the JavaScript/TypeScript frontend, dependency extraction was compared with Madge v8.0.0. Madge identified 52 internal module edges; CodeInsight AI recovered 50 of them. The reported errors included two false negatives associated with index-file aliasing and eight false positives associated with external-library CSS bindings. Overall, the comparison yielded 86.2% precision, 96.2% recall, and an F1 score of 90.9%. For Python, a visitor built with the native ast module served as the reference, and CodeInsight AI reported 100% concordance for internal module dependencies.

C. *Performance Analysis*

Table II summarizes the measured scanning, graph-layout, and storage results for repositories of different sizes.

TABLE II. Performance Evaluation of CodeInsight

Repository size	**Scan/parse time (mean ± SD)**	**Dagre layout time (mean ± SD)**	**Database storage**
60 files	0.153±0.019s	0.045 ± 0.005 s	2.1 MB
105 files	0.537 ± 0.506s	0.082 ± 0.008 s	4.2 MB
204 files	2.010 ± 0.762s	0.245 ± 0.015 s	8.6 MB
251 files	1.455 ± 0.064s	0.312 ± 0.021 s	11.4 MB
1,000 files	8.400 ± 0.350s	1.150 ± 0.045 s	28.6 MB

Scan/parse time did not increase monotonically with repository size because execution time varied across repositories, reflecting differences in repository characteristics and measured run-to-run variability.

Source: Prototype implementation measurements.

The results indicate that dependency extraction remains efficient for both small and relatively large repositories. Although analysis time increases with repository size, the observed growth remains suitable for interactive software exploration. Similarly, graph layout computation remains below approximately 1.5 seconds for the evaluated repositories, enabling responsive visualization during developer interaction.

The conversational assistant demonstrates comparatively stable response times because prompt construction primarily relies on previously extracted structural metadata rather than repeatedly analyzing the complete project.

D. *Architectural Classification Validation*

Classification was checked against manual labels for 85 representative files from the frontend and backend. The system assigned 83 files to the expected category and misclassified two, giving an overall accuracy of 97.6%. This result measures agreement with the labels used in the evaluation; it should not be interpreted as proof that the same accuracy will hold for every repository structure.

The 85 representative source files were independently labeled by two senior faculty members specializing in software architecture and one external peer reviewer. Disagreements were resolved through discussion to establish consensus reference labels. Cohen's kappa is not reported because the manuscript contains only the consensus labels, not the individual ratings needed to calculate inter-rater agreement.

E. *Scalability Analysis*

To limit the influence of background processes and temporary system effects, each performance test was run five times (N = 5) after clearing the cache. The evaluation included projects with 60, 105, 204, 251, and 1,000 source

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

files. For each size, the reported values include the mean and standard deviation for scanning/parsing and Dagre layout; database storage was recorded separately. Scan/parse time ranged from 0.153 ± 0.019 seconds for the 60-file project to 8.400 ± 0.350 seconds for the 1,000-file synthetic suite.

A synthetic generator created 1,000 source files arranged into 10 decoupled subsystems. Each subsystem contained 100 files distributed across controllers, services, models, views, and utils folders. The generator added multi-language imports using a scale-free network based on the Barabási–Albert preferential-attachment model. The reported average out-degree of 3.45 was calculated by dividing the 3,450 generated directed dependencies by the 1,000 source-file nodes (3,450 ÷ 1,000 = 3.45 outgoing edges per node).

F. Comparative Discussion

To better position the proposed framework within the existing software engineering ecosystem, Table III compares its capabilities with representative categories of software analysis tools.

TABLE III. Qualitative Comparison with Existing Approaches

Feature	**Sonar Qube**	**Madge/p ydeps**	**Copilot**	**CodeI nsight AI**
Static/code -quality analysis	Yes	No	No	No
Interactive dependenc y exploration	Partial	Yes	Partial	Yes
File/modul e dependenc y graph	Partial	Yes	Partial	Yes
Architectur al layer grouping	No	No	No	Yes
Repository -context conversati on	No	No	Yes	Yes
Unified graph + grouping + explanatio n	No	No	No	Yes

The tools differ in their primary purpose and supported workflows. SonarQube concentrates on code quality and security analysis [20] Madge creates JavaScript/TypeScript module-dependency graphs [14] pydeps visualizes Python dependencies [15] and GitHub Copilot offers AI-assisted coding support [21]. CodeInsight AI brings together a dependency graph, heuristic architectural grouping, interactive exploration, and a conversational interface informed by project context. This table compares reported capabilities; it is not intended to suggest that one tool is better than another in every use case.

G. Developer Usefulness Study

The usefulness study involved 14 undergraduate software engineering students at GM Institute of Technology. Participants knew JavaScript, TypeScript, and Python and had not previously worked with the 250-file full-stack benchmark repository. Seven students formed the control group, using Visual Studio Code, terminal search, ripgrep/grep, and the usual file-tree navigation.

Allocation: Participants were stratified and randomly allocated into two balanced cohorts of seven students each, based on academic GPA and self-reported proficiency in JavaScript/TypeScript and Python.

Training: The CodeInsight AI group completed a standardized 15-minute onboarding tutorial on canvas navigation and chat filters using an unrelated toy project (a 15-file Express.js app). The control group received an equivalent 15-minute briefing on task objectives within VS Code.

The other seven used CodeInsight AI, including its graph view, category filters, and context-aware assistant. Each participant completed four tasks: finding authentication roles, assessing the impact of a change, tracing an execution flow, and identifying highly connected architectural modules.

TABLE IV. Developer Study Results

Metric	**Control vs. CodeInsight AI**
Role discovery	4.8 ± 0.9 vs 1.9 ± 0.4 min; −60.4%; p<0.001
Impact analysis	6.7 ± 1.2 vs 2.4 ± 0.5 min; −64.2%; p<0.001
Flow tracing	7.1 ± 1.4 vs 3.1 ± 0.6 min; −56.3%; p<0.001
Hub identification	3.8 ± 0.8 vs 1.5 ± 0.3 min; −60.5%; p<0.001
Total time	22.4 vs 8.9 min; −60.3%; p<0.001
Accuracy	64.3% vs. 92.9%; +28.6 percentage points (+44.5% relative); Fisher's exact p = 0.020

The study recorded task time, answer correctness, and usability. Across the four tasks, the control group took 22.4

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

minutes in total, compared with 8.9 minutes for the CodeInsight AI group—a reported reduction of 60.3% (two-tailed Welch's t-test, $p < 0.001$). Accuracy rose from 64.3% in the control group to 92.9% in the CodeInsight AI group, an improvement of +28.6 percentage points (Fisher's exact test, $p = 0.020$). The average System Usability Scale score was 84.6/100, which falls in the Excellent/Grade-A range. Welch's two-sample t-test was used to compare task completion times, while Fisher's exact test was used to compare classification accuracy.

VI. LIMITATIONS

The current version has limitations. Because dependency extraction relies on regular expressions, it can miss imports that use unsupported syntax, dynamic loading, generated code, or framework-specific rules. It is not a substitute for a compiler-grade parser or a full AST-based analyzer. Architectural categories are assigned through path and keyword heuristics, so projects with unusual naming conventions may be grouped incorrectly. External imports are stored as metadata but are not represented as internal graph edges. The assistant can only use the context that fits into its prompt, meaning very large repositories may require context reduction and may omit relevant details. The graph also represents static dependencies; it does not capture runtime calls, reflection, dynamic dispatch, generated code, or execution traces. Finally, the benchmark and student study cover a limited set of repositories and participants, so broader evaluation is needed before generalizing the findings. Threats to Construct & External Validity: In the developer usefulness study, the target evaluation codebase was a 251-file full-stack repository (CodeInsight AI Core). Although student participants had never previously inspected, developed, or navigated this codebase, evaluating comprehension on a system architected by the authors introduces potential experimental bias or favorable architectural alignment. Future evaluations must deploy independent, third-party open-source repositories.

Small Cohort Sample Size: The controlled user evaluation included 14 undergraduate students, with seven participants in each group. Task-time differences were statistically significant ($p < 0.001$), while the difference in diagnostic accuracy was evaluated using Fisher's exact test ($p = 0.020$). The small sample limits statistical generalizability to broader professional engineering populations. Cohen's kappa and the standard deviation of the System Usability Scale score cannot be calculated from the reported summary alone; the independent pre-consensus labels and participant-level SUS scores are required.

VII. CONCLUSION AND FUTURE WORK

CodeInsight AI offers a unified environment for exploring unfamiliar software repositories with explicit regex-based dependency analysis, heuristic architectural grouping, interactive visualization, and project-context-aware LLM assistance. The empirical results show 86.2% precision, 96.2% recall, and 90.9% F1 against Madge for JavaScript/TypeScript dependency extraction, 100% concordance against Python's native ast reference for internal Python dependencies, and 97.6% accuracy for architectural-layer classification. Repeated benchmarks across 60 to 1,000 source files demonstrate measurable scaling behavior, while the controlled study with 14 students shows substantial reductions in comprehension time and improvements in answer accuracy. Future work can extend the regex-based extractor with language-specific parsers and incremental parsing, add function-level call graphs and runtime information, improve architectural classification for unconventional project layouts, develop more adaptive large-repository context selection, and conduct broader multi-repository and multi-participant evaluations.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to the Department of Information Science and Engineering, GM Institute of Technology, Davanagere, for providing the necessary facilities, technical guidance, and academic support to carry out this research. The authors also thank their project guide and faculty members for their valuable suggestions and continuous encouragement throughout the development of this work.

REFERENCES

[1] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990

[2] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2013, pp. 486–496

[3] H. Sajnani, "Automatic software architecture recovery: A machine learning approach," in *Proc. 20th IEEE Int. Conf. Program Comprehension (ICPC)*, 2012, pp. 265–268

[4] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Proc. 2017 IEEE Int. Conf. Software Architecture Workshops (ICSAW)*, 2017, pp. 46–53

[5] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. A. Myers, "Using an LLM to help with code understanding," *in Proc. IEEE/ACM 46th Int. Conf. Software Engineering (ICSE)*, 2024, pp. 1–13.

[6] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Software Eng. Methodol.*, vol. 33, no. 8, Art. no. 220, pp. 1–79, 2024.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

[7] J. Zhang, H. Tao, Z. Shen, X. Yang, and G. Li, "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, pp. 4512–4526.

[8] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. S. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "CodePlan: Repository-level coding using LLMs and planning," Proc. ACM Softw. Eng., vol. 1, FSE, pp. 675–698, Jul. 2024.

[9] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang, X. Che, Z. Liu, and M. Sun, "RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation," in Proc. 2024 Conf. Empirical Methods in Natural Language Processing: System Demonstrations, 2024, pp. 151–160.

[10] X. Kong, B. Li, L. Wang, and W. Wu, "Directory-based dependency processing for software architecture recovery," *IEEE Access*, vol. 6, pp. 52321–52335, 2018.

[11] S. P. R. Puchala, J. K. Chhabra, and A. Rathee, "Software architecture recovery using integrated dependencies based on structural, semantic, and directory information," *Int. J. Inf. Syst. Model. Des.*, vol. 13, no. 1, pp. 1–20, 2022.

[12] Y. Zhang, Z. Xu, C. Liu, H. Chen, J. Sun, D. Qiu, and Y. Liu, "Software architecture recovery with information fusion," in *Proc. 31st ACM Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 1535–1547.

[13] M. Schiewe, J. Curtis, V. Bushong, and T. Černy, "Advancing static code analysis with language-agnostic component identification," *IEEE Access*, vol. 10, pp. 30743–30761, 2022.

[14] Madge contributors, "Madge: Create graphs from JavaScript and TypeScript module dependencies," GitHub repository, version 8.0.0, 2024. [Online]. Available: https://github.com/pahen/madge

[15] pydeps contributors, "pydeps: Python module dependency visualization," GitHub repository, 2024. [Online]. Available: https://github.com/thebjorn/pydeps

[16] S. Thakare and A. W. Kiwelekar, "Recovery and analysis of architecture descriptions using centrality measures," *arXiv:2101.09422*, 2021, pp. 1–24.

[17] A. Sajji, Y. Rhazali, and Y. Hadi, "A methodology of automatic class diagrams generation from source code using Model-Driven Architecture and machine learning," *E3S Web of Conferences*, vol. 412, Art. no. 01002, 2023.

[18] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[19] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. 2021 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 8696–8708.

[20] SonarSource, "SonarQube: Continuous code quality and security inspection," 2024. [Online]. Available: https://www.sonarqube.org

[21] GitHub, "GitHub Copilot: Your AI pair programmer," 2024. [Online]. Available: https://github.com/features/copilot

[22] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[23] A. Walker, I. Laird, and T. Černy, "On automatic software architecture reconstruction of microservice applications," in *Information Science and Applications—Proc. ICISA 2020*, vol. 739, 2021, pp. 223–234.

[24] A. Hatahet, C. Knieke, and A. Rausch, "Generating software architecture description from source code using reverse engineering and large language model," in *Proc. 28th ACM/IEEE Int. Conf. Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2025, pp. 566–575.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)