DAYANANDA SAGAR UNIVERSITY

Devarakaggalahalli, Harohalli
Kanakapura Road, Ramanagara - 562112, Karnataka, India

SCHOOL OF ENGINEERING

Bachelor of Technology
in
COMPUTER SCIENCE AND ENGINEERING

Minor Project Report

CodeSense: Combining Static Analysis and LLms for Code Review

Batch: 125

By
S Rayyan Ahmed – ENG22CS0424
Suraj S - ENG22CS0425
Sahil K - ENG22CS0429
Shreyas Bharadwaj - ENG22CS0459

Under the supervision of
Prof. Sowmya H.D
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING
DAYANANDA SAGAR UNIVERSITY

(2024-2025)

SCHOOL OF ENGINEERING

Department of Computer Science & Engineering

Devarakaggalahalli, Harohalli, Kanakapura Road, Ramanagara - 562112
Karnataka, India

CERTIFICATE

This is to certify that the minor project work titled **"CODESENSE: COMBINING STATIC ANALYSIS AND LLMS FOR CODE REVIEW"** is carried out by **S Rayyan Ahmed (ENG22CS0424), Suraj S (ENG22CS0425), Sahil K (ENG22CS0429), Shreyas Bharadwaj (ENG22CS0459),** bonafide students of Bachelor of Technology in Computer Science and Engineering at the School of Engineering, Dayananda Sagar University, Bangalore in partial fulfillment for the award of degree in Bachelor of Technology in Computer Science and Engineering, during the year **2024-2025**.

Prof. Sowmya H.D	**Dr. Girisha G S**	**Dr. Udaya Kumar Reddy K R**
Assistant Professor Dept. of CS&E, School of Engineering Dayananda Sagar University	Chairman CSE School of Engineering Dayananda Sagar University	Dean School of Engineering Dayananda Sagar University
Date:	Date:	Date:

Name of the Examiner	**Signature of Examiner**
1.	
2.	

DECLARATION

We, **S Rayyan Ahmed (ENG22CS0424), Suraj S (ENG22CS0425), Sahil K (ENG22CS0429), Shreyas Bharadwaj (ENG22CS0459),** are students of sixth semester B. Tech in **Computer Science and Engineering**, at School of Engineering, **Dayananda Sagar University**, hereby declare that the Minor Project titled **"CodeSense: Combining Static Analysis and LLMs for Code Review"** has been carried out by us and submitted in partial fulfilment for the award of degree in **Bachelor of Technology in Computer Science and Engineering** during the academic year **2024-2025**.

Student **Signature**

Name: S Rayyan Ahmed

USN : ENG22CS0424

Name: Suraj S

USN : ENG22CS0425

Name: Sahil K

USN : ENG22CS0429

Name: Shreyas Bharadwaj

USN : ENG22CS0459

Place : Bangalore

Date :

ACKNOWLEDGEMENT

It is a great pleasure for us to acknowledge the assistance and support of many individuals who have been responsible for the successful completion of this project work.

First, we take this opportunity to express our sincere gratitude to School of Engineering & Technology, Dayananda Sagar University for providing us with a great opportunity to pursue our Bachelor's degree in this institution.

We would like to thank ***Dr. Udaya Kumar Reddy K R, Dean, School of Engineering, Dayananda Sagar University*** *for his constant encouragement and expert advice.*

It is a matter of immense pleasure to express our sincere thanks to ***Dr. Girisha G S, Department Chairman****,* ***Computer Science and Engineering****,* ***Dayananda Sagar University,*** *for providing right academic guidance that made our task possible.*

We would like to thank our guide Prof. Sowmya H.D, ***Assistant Professor****,* ***Dept. of Computer Science and Engineering****,* ***Dayananda Sagar University****, for sparing her valuable time to extend help in every step of our project work, which paved the way for smooth progress and fruitful culmination of the project.*

We would like to thank our ***Project Coordinators Dr. Gousia*** *and* ***Dr. Damodharan*** *as well as all the staff members of Computer Science and Engineering for their support.*

We are also grateful to our family and friends who provided us with every requirement throughout the course.

We would like to thank one and all who directly or indirectly helped us in the Project work.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

LLM	Large Language Model
IDE	Integrated Development Environment
VS	Visual Studio
AI	Artificial Intelligence
API	Application Programming Interface

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Code quality is a decisive element of software maintainability and developer productivity. Conventional static analysis tools such as PyLint can detect syntax errors, code smells, and violations of coding conventions but tend to be lacking in contextual knowledge while proposing resolutions. To overcome this limitation, CodeSense has been designed, a hybrid code analysis tool that integrates static analysis with the reasoning abilities of a Large Language Model's (LLM). CodeSense is a Visual Studio Code extension that leverages PyLint for real-time linting and enhances the diagnostic output through passing the code and linter output to an LLM model. The model presents developers with a more cognitive and intuitive coding experience by generating contextual suggestions, quick fixes, and code improvement recommendations. An evaluation of the tool's performance was conducted on a number of Python projects, demonstrating improved coding efficiency, reduced manual debugging effort, and improved code quality. The system architecture, integration pipeline, and a qualitative performance analysis are discussed in this paper. Results indicate how deterministic and generative methods can be used together to create development environments that are smarter and more responsive.

CHAPTER 1: INTRODUCTION

1.1. OVERVIEW:

Modern software projects require high code quality and maintainability, yet achieving this is challenging. Developers often rely on static analysis tools like PyLint to find syntax errors, code smells, and style issues. However, these tools typically provide lengthy, rule-based diagnostics that must be manually interpreted, offering little context or guidance on how to fix problems. This can slow development and lead to overlooked issues or technical debt accumulation.

CodeSense addresses these challenges by combining static analysis with advanced language modeling. Built as an extension for Visual Studio Code, CodeSense uses PyLint for real-time linting and then enriches the results using a locally running CodeLlama model. By sending the code and lint output to the LLM, CodeSense generates more contextual and intuitive suggestions, including quick fixes and code improvements. This hybrid approach helps developers receive actionable feedback and maintain higher code quality without leaving their IDE.

1.1.1 OBJECTIVES:

- Automate real-time code analysis by integrating a static analyzer (PyLint) into the development workflow.
- Leverage an LLM (CodeLlama) to enrich linting output with context-aware suggestions and explanations.
- Enhance productivity and code quality by reducing manual debugging and offering intuitive guidance.

1.1.2 SCOPE:

- Integrates tightly with Visual Studio Code as an IDE extension for seamless user experience.
- Focuses on Python programming language using PyLint for static analysis and CodeLlama for reasoning.
- Operates locally, processing code and linter output to generate contextual suggestions on the developer's machine.
- Provides instant feedback on code issues when files are saved or changed, fitting smoothly into the development workflow.

CHAPTER 2: PROBLEM DEFINITION

2.1. PROBLEM STATEMENT

• Static analysis tools produce lengthy diagnostics that require manual interpretation without context or guidance on fixes.
• Developers often waste time reading and deciphering these reports instead of receiving actionable recommendations.
• Code improvement suggestions and quick fixes are not automatically provided, making debugging and refactoring slower.
• Conventional IDE assistants focus on code generation rather than analyzing and explaining existing code issues.
• This gap in real-time, context-aware feedback can lead to reduced productivity and more technical debt.

2.2. PROPOSED SOLUTION

• Develop the CodeSense VS Code extension to combine static linting (PyLint) with LLM reasoning (CodeLlama).
• Trigger code analysis automatically on file save, capturing syntax and style issues via PyLint.
• Feed the code context and linter messages into the LLM to generate more intuitive, contextaware feedback.
• Present the LLM's suggestions and quick fixes directly in the editor, enabling one-click application.
• This hybrid solution offers developers actionable recommendations and explanations, reducing manual effort in code reviews.

CHAPTER 3: LITERATURE REVIEW

Code analysis has come a long way with the transition from classical static analysis tools to AI-assisted code checkers with the use of large language models (LLMs). Static checkers such as PyLint are extensively used to find syntax errors, code smells, and stylistic issues. These are extremely effective in enforcing coding standards and detecting faults early during the development cycle. But they usually generate platitudinous warnings that have no contextual knowledge behind them, so they are difficult for programmers to understand and respond to appropriately [6], [5].

In an attempt to overcome the weakness of traditional static analysis, current research has been concerned with combining AI and machine learning into the code analysis process. Machine learning-based tools like GitHub Copilot and Code Llama have become popular due to their capacity to create, rework, and auto-complete code based on deep learning models that are trained on enormous corpora of code [7], [9]. Though powerful, these models are mainly tuned for code synthesis and completion and not static analysis. Therefore, they cannot be expected to interpret and answer the diagnostic output of conventional tools directly [4].

Emerging hybrid approaches aim to bridge this gap by combining the deterministic logic of static analyzers with the contextual reasoning capabilities of LLMs. Notably, tools such as SkipAnalyzer [1], E&V [2], and LLift [3] have demonstrated the effectiveness of this integration. SkipAnalyzer uses LLMs to eliminate false positives and produce valid bug patches with a precision of up to 93.88% in eliminating false-positive warnings and a 97.30% patch generation success rate [1], [5]. E&V also uses LLMs for pseudo-code execution and verification, minimizing human effort and maximizing accuracy in triaging Linux kernel bugs—up to 81.2% accuracy in identifying blamed functions [2], [5]. LLift improves static analysis with the integration of symbolic reasoning into LLMs to detect difficult-to-spot UBI (Use-Before-Initialization) bugs in the Linux kernel with higher accuracy and consistency [3], [5].

A number of experiments highlight that integrating static analysis with LLMs results in real-world gains in productivity among developers and bug detection. For example, Panichella et al. [5] demonstrated that static analysis tools are beneficial during code reviews, particularly when set to filter out noise and report on what matters most. Moreover, in learning environments, the integration of abstract syntax trees (ASTs) with

AI models has been demonstrated to improve accuracy in code evaluation as well as student learning outcomes [5].

Systems like Pythia [4], based on AI, further reflect the increasing application of LLMs for smart code assistance. Pythia was seen to provide 92% top-5 accuracy in code completion tasks with LSTM models, trained on ASTs, and providing context-sensitive suggestions inside Visual Studio Code [4]. Similarly, LLMs such as StarCoder [13] and GPT-4 [14] are being investigated for different developer support tasks ranging from code summarization to patch suggestions.

The increasing popularity of program repair datasets such as QuixBugs [15] points towards the requirement for LLM-augmented tools that are capable of detecting as well as proposing actionable repairs for actual bugs. This is along the lines of how tools such as CodeSense are heading, which integrate LLMs directly inside IDEs like Visual Studio Code [11], [12], allowing for real-time, context-aware diagnostics and code correction proposals. CodeSense expands the hybrid paradigm by combining PyLint's diagnostic output with LLM reasoning (from models such as Code Llama through Ollama [8]) to generate human-readable fix suggestions targeted at the particular issues highlighted.

Lastly, the technical debt metaphor [10] puts the focus on long-term expenditures of code quality problems. Both syntactic and semantic defects can be tackled by hybrid code analysis systems that can be a significant aid in controlling and minimizing this debt through early, meaningful suggestions for improvement. As LLM-based systems evolve, integrating them with static analysis tools will become the norm for sound, developer-focused tooling in contemporary software engineering.

CHAPTER 4: REQUIREMENTS

4.1. FUNCTIONAL REQUIREMENTS

• **Real-time code analysis:** Perform automatic linting of Python code in VS Code upon file save.
• **Contextual suggestions:** Generate intuitive explanations and fix recommendations for identified issues using LLM reasoning.
• **IDE integration:** Seamlessly display diagnostics and suggestions within the VS Code editor interface.
• **Language support:** Analyze Python source code using PyLint as the static analysis engine.
• **User interaction:** Allow developers to trigger or apply suggested fixes with minimal effort.

4.2. NON-FUNCTIONAL REQUIREMENTS

• **Performance:** Provide fast, real-time feedback to avoid interrupting the development workflow.
• **Accuracy:** Maintain high precision in issue detection and reliability of suggested fixes.
• **Usability:** Deliver clear, understandable suggestions in a user-friendly interface.
• **Extensibility:** Design modular components to allow future extension to new rules or languages.
• **Privacy:** Run analysis locally (using a local LLM) to protect sensitive code and data.

4.3. SYSTEM REQUIREMENTS

• **Visual Studio Code:** The extension must run in VS Code on Windows, macOS, or Linux.
• **Python Environment:** Requires a Python interpreter and the PyLint package installed.
• **LLM Runtime:** Needs a locally hosted LLM (CodeLlama) accessible via an interface like Ollama.
• **Development Stack:** Uses VS Code Extension API and JavaScript for implementation.

CHAPTER 5: DESIGN & METHODOLOGY

5.1. SYSTEM ARCHITECTURE

CodeSense follows a modular architecture with three main components:

- **VS Code Extension Interface:** Continuously monitors code edits and file saves within the IDE. It triggers the static analysis process automatically and renders the results (errors, warnings, and suggestions) as diagnostics directly in the editor for seamless developer feedback.
- **Static Analysis Engine:** Utilizes PyLint to perform a thorough analysis of the Python code. It detects common issues, code smells, and potential bugs, producing structured diagnostics (including error codes, messages, and locations) for further processing.
- **LLM Reasoning Module:** Employs a locally running CodeLlama model to interpret the code context and PyLint diagnostics. It generates intelligent suggestions, explanations, and one-click quick fixes aimed at improving code quality, readability, and maintainability.

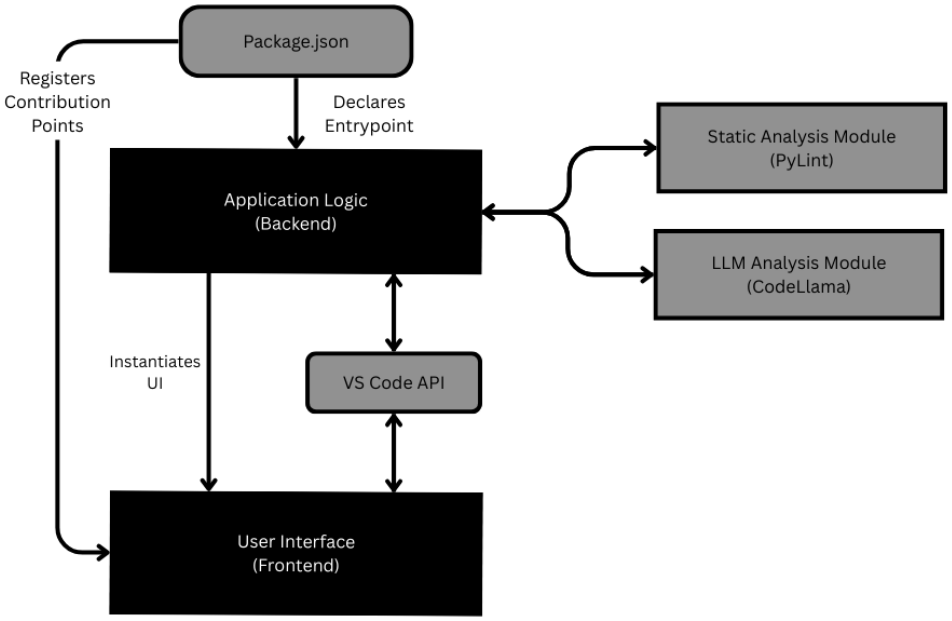

Figure 5.1 CodeSense System architecture

5.2. WORKFLOW

The overall process for CodeSense can be outlined in the following steps:

1. **Code Save:** Developer writes or edits Python code in VS Code and saves the file.
2. **Static Analysis:** The extension invokes PyLint to perform static analysis on the saved file.
3. **Diagnostics Collection:** PyLint identifies issues (syntax errors, code smells, etc.) and returns diagnostics.
4. **Prompt Construction:** The extension constructs a prompt combining the relevant code snippet and the diagnostics.
5. **LLM Reasoning:** The prompt is sent to the local LLM (CodeLlama) which returns context-aware suggestions and fixes.
6. **Result Presentation:** CodeSense displays these suggestions in the editor, allowing developers to apply or review the recommendations.

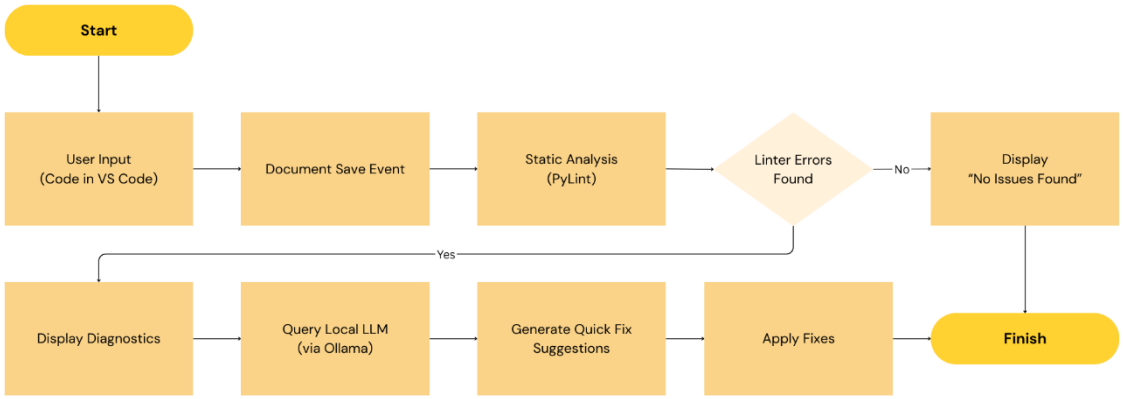

Figure 5.2 CodeSense Data Flow Diagram, illustrating the sequence from file save to suggestion display.

CHAPTER 6: RESULTS AND DISCUSSION

We evaluated CodeSense using a set of Python programs with known bugs, focusing on its ability to detect issues and suggest correct fixes. Tables below summarize the performance metrics of CodeSense in our tests.

• *Table 6.1: Issue Detection Performance on QuixBugs*

Metric	Value
Precision	0.75
Recall	1.00
F1 Score	0.86

• **Issue Detection:** CodeSense achieved precision of 0.75 and recall of 1.00 (F1 score 0.86), indicating reliable identification of all true issues with moderate false positives.
• **Fix Suggestion Accuracy:** Out of 40 attempted fixes on identified bugs, 30 were correct, yielding a fix accuracy of 75% as shown in Table 2.
• **Keystroke Savings:** Successful fixes saved an estimated 10 keystrokes each on average (total ~300 strokes), suggesting significant manual effort reduction.
• **Productivity Impact:** These results demonstrate that combining static analysis with LLM reasoning can effectively improve developer productivity by automating error detection and correction.

- *Table 6.2: Fix Suggestion Evaluation*

Metric	Value
Fix Attempts	40
Valid Fixes	30
Fix Accuracy	0.75

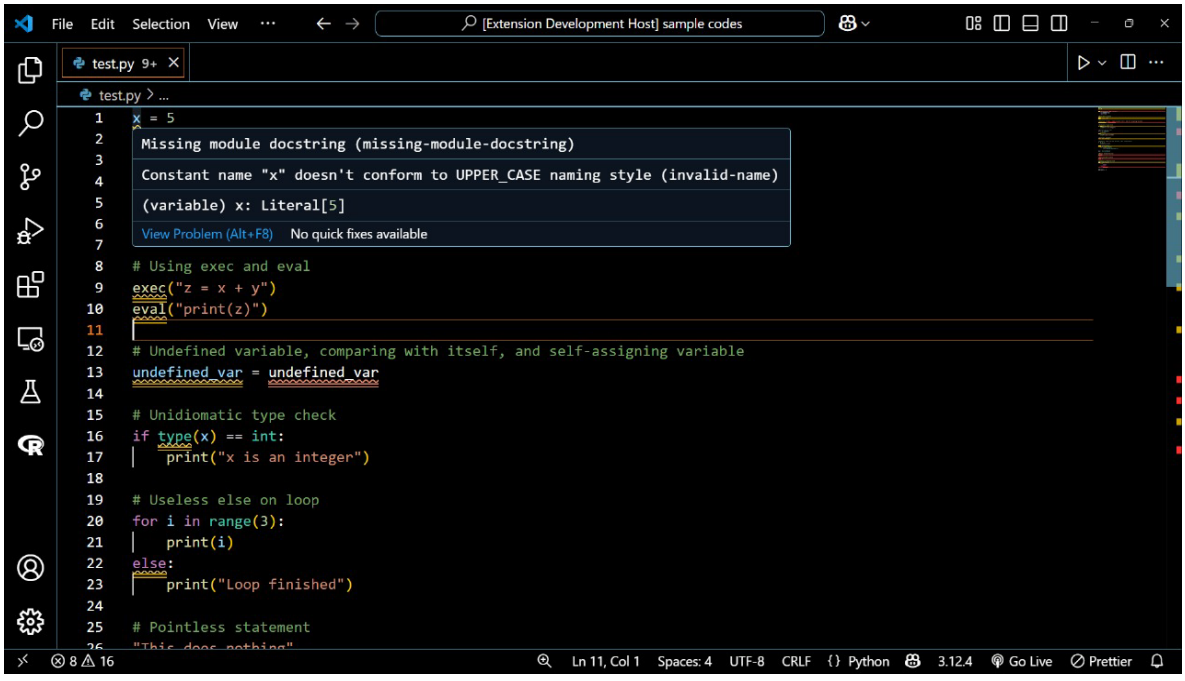

Figure 6.1 Static analysis: Red and yellow squiggles with hover-to-view error details.

CHAPTER 7: CONCLUSION AND FUTURE WORK

7.1. CONCLUSION

CodeSense integrates deterministic static analysis with generative language modeling to provide a smarter code review assistant. By combining PyLint diagnostics with contextual suggestions from a CodeLlama model, developers receive more intuitive guidance and automated fixes directly in VS Code. Evaluation on real Python projects showed that CodeSense could detect bugs with high recall and suggest correct fixes for the majority of cases, while saving developer effort in terms of typing. Overall, the hybrid approach of CodeSense demonstrates the potential of merging traditional and AI-driven methods to enhance code quality and development efficiency.

7.2. FUTURE WORK

- **Multi-language Support**: Extend CodeSense to handle additional programming languages by integrating corresponding linters and static analysis tools (e.g., ESLint for JavaScript, Flake8 for Python, etc.).
- **Advanced LLM Prompt Engineering**: Refine prompt strategies to enable the LLM to better address complex, multi-line, or context-dependent code issues with higher accuracy.
- **Performance and Scalability Improvements**: Optimize local LLM inference for speed and efficiency, and explore hybrid models that offload heavy reasoning tasks to scalable cloud-based AI services.
- **User-Centric Evaluation**: Conduct systematic user studies to measure productivity improvements, identify friction points, and gather qualitative developer feedback for iterative refinement.
- **Adaptive Learning from Feedback**: Implement a feedback mechanism where user-accepted or corrected suggestions are logged to fine-tune or retrain the LLM for project-specific or organization-specific coding patterns.

REFERENCES

[1] M. M. Mohajer *et al.* (2023). "SkipAnalyzer: A Tool for Static Code Analysis with Large Language Models," *arXiv preprint*, arXiv:2310.18532. Available: https://arxiv.org/abs/2310.18532

[2] Y. Hao, W. Chen, Z. Zhou, and W. Cui (2023). "E&V: Prompting Large Language Models to Perform Static Analysis by Pseudo-code Execution and Verification," *arXiv preprint*, arXiv:2312.08477. Available: https://arxiv.org/abs/2312.08477

[3] H. Li, Y. Hao, Y. Zhai, and Z. Qian (2024). "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, Apr. 2024. doi: 10.1145/3649828.

[4] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan (2019). "Pythia: AI-assisted Code Completion System," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, New York, NY, USA: ACM, Jul. 2019. Available: https://doi.org/10.1145/3292500.3330699

[5] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol (2015). "Would static analysis tools help developers with code reviews?," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 161–170, IEEE, Mar. 2015. Available: https://doi.org/10.1109/saner.2015.7081826

[6] PyLint Documentation (2025). *Pylint 3.3.4 documentation*, Accessed: Mar. 22, 2025. Available: https://pylint.readthedocs.io/en/stable/

[7] Meta AI (2025). "Introducing Code Llama, a state-of-the-art large language model for coding," Accessed: Mar. 23, 2025. Available: https://ai.meta.com/blog/code-llama-large-language-model-coding/

[8] Ollama (2025). *Ollama*, Accessed: Mar. 23, 2025. Available: https://ollama.com/

[9] GitHub (2025). *GitHub Copilot*, Accessed: Mar. 25, 2025. Available: https://github.com/copilot

[10] P. Kruchten, R. L. Nord, and I. Ozkaya (2012). "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov. 2012. doi: 10.1109/ms.2012.167.

[11] Microsoft (2025). *Visual Studio Code*, Accessed: Mar. 22, 2025. Available: https://code.visualstudio.com/

[12] Microsoft (2025). *Visual Studio Code API Reference*, Accessed: Mar. 25, 2025. Available: https://code.visualstudio.com/api/references/vscode-api

[13] Hugging Face (2025). "StarCoder: A State-of-the-Art LLM for Code," Accessed: Apr. 10, 2025. Available: https://huggingface.co/blog/starcoder

[14] OpenAI (2025). "GPT-4," Accessed: Apr. 10, 2025. Available: https://openai.com/index/gpt-4/

[15] jkoppel (2025). "GitHub - jkoppel/QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge," GitHub. Accessed: May 06, 2025. Available: https://github.com/jkoppel/QuixBugs