CodeSense: Combining Static Analysis and LLMs for Code Review

by S Rayyan Ahmed

CodeSense: Combining Static Analysis and LLMs for Code Review

[1] S Rayyan Ahmed
Computer Science and Engineering
Dayananda Sagar University
Bengaluru, India
eng22cs0424@dsu.edu.in

[1] Suraj S
Computer Science and Engineering
Dayananda Sagar University
Bengaluru, India
eng22cs0425@dsu.edu.in

[1] Sahil K
Computer Science and Engineering
Dayananda Sagar University
Bengaluru, India
eng22cs0429@dsu.edu.in

[1] Shreyas Bharadwaj
Computer Science and Engineering
Dayananda Sagar University
Bengaluru, India
eng22cs0459@dsu.edu.in

[6] Sowmya H.D
Assistant Professor
Computer Science and Engineering
Dayananda Sagar University
Bengaluru, India
sowmya.hd-cse@dsu.edu.in

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, we have designed CodeSense, 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. We evaluate the tool's performance 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. Our results indicate how deterministic and generative methods can be used together to create development environments that are smarter and more responsive.**

I. INTRODUCTION

The growing complexity of contemporary software systems requires strong tools to guarantee code quality, readability, and maintainability. Programmers must battle the challenge of producing functionally correct code while keeping to the best practices and reducing technical debt, especially during high-speed development cycles or pedagogy where experience and time are usually scarce. Static analysis tools like PyLint [6] are important by pointing out syntax errors, style faults, and possible bugs. Their rule-based approach normally resulting in lengthy diagnostics that need to be interpreted manually, and leading to unintended issues and decreased productivity.

Integrated development environments (IDEs) such as Visual Studio Code (VS Code) are at the core of modern workflows because they are extensible, allowing for smooth incorporation of linting, formatting, and version control tools. Notwithstanding these developments, solutions currently lack in offering real-time, context-aware guidance that closes the gap between static analysis and actionable code enhancements. Large language models (LLMs), such as CodeLlama [7], have been promising in code refactoring and code generation, as seen in tools such as GitHub Copilot [9]. Nevertheless, these tools are predominantly aimed at predictive code completion, with little integration with diagnostic feedback or structural code improvement support in the IDE.

To bridge these gaps, we introduce CodeSense, a VS Code extension that combines PyLint's static analysis with LLM-informed reasoning in order to provide smart, context-aware suggestions for code improvement. CodeSense activates PyLint on file saves to create diagnostics, which are then processed into structured prompts for a locally running CodeLlama model through Ollama [8]. The LLM produces accurate fix suggestions and refactorings presented directly in the VS Code editor.

This paper makes the following contributions:

- A hybrid VS Code extension combining static analysis with LLM-reasoning for improved code quality.
- A prompt engineering solution that makes use of PyLint diagnostics to inform LLM-produced fixes.
- A comparative study of CodeSense performance on benchmark Python datasets, measuring accuracy, latency, and developer productivity.
- Comparative study of LLM performance in producing actionable code fixes from static analysis results.

CodeSense is designed to automate the code review process, minimize manual debugging effort, and enable developers to write higher-quality code more quickly and effectively.

II. BACKGROUND

A. Code Quality

Code quality is the pillar of contemporary software programming, summarizing traits like readability, maintainability, efficiency, and correctness. Quality code is easy to comprehend,

easy to adapt, and robust to faults, thus lowering long-term maintenance expenses and improving program extendibility. By contrast, ill-designed code risks technical debt—the total expense of extra effort needed as a result of less-than-ideal design decisions [10]. Technical debt occurs when programmers focus on making rapid changes rather than refactoring or ignoring coding standards, making the software harder and harder to maintain.

Measuring code quality encompasses both qualitative reviews (e.g., code readability) and quantitative measures, such as cyclomatic complexity, code duplication, test coverage, and compliance with style guides (e.g., PEP 8 for Python). Automated analysis tools, like static analyzers, are essential for measuring these values and detecting problems early in the development process, hence avoiding technical debt and enhancing software reliability.

B. Static Analysis and PyLint

Static analysis reviews source code without running the code to identify bugs, style checks, and maintainability problems. Static analysis tools improve code quality and cut down on debugging time by enforcing coding standards and flagging probable errors early. PyLint, a popular static analyzer for Python, serves this purpose best [6]. It analyzes Python code for syntax errors, unused variables, too complex functions, and conformity with PEP 8 standards and assigns each problem a severity (e.g., error, warning) and a unique code (e.g., E0602 for undefined variables).

While strong, PyLint output is sometimes verbose and difficult to understand, especially for new programmers. As an illustration, dozens of diagnostics may result from processing a single file, demanding that individual items be manually prioritized and resolved. In addition, PyLint does not offer automated fix or context-specific improvement suggestions, something that CodeSense corrects by coupling large language model (LLM)-based reasoning with PyLint diagnostics.

C. Visual Studio Code

Visual Studio Code (VS Code), created by Microsoft, is a light-weight, cross-platform source code editor that is known for its flexibility and customizability [11]. VS Code supports a variety of programming languages and has features like syntax highlighting, integrated debugging, smart code completion, and an integrated terminal. It has gained such popularity because it is open source, has an active community, and a large marketplace of extensions, which allow the developers to tailor their environment to their needs.

Developed on Electron and web technologies, VS Code has an adaptive user interface and native integration with tools such as Git for version control. Its telemetry APIs and remote development features enhance its popularity, making it a developer favorite among academic and professional programmers alike.

D. VS Code Extensions

VS Code extensibility is propelled by its Extension API, which enables developers to develop plugins that extend the editor's functionality [12]. Extensions are written in JavaScript or TypeScript and may add language support, debuggers, linters, formatters, or integrations with external services. The API opens up editor events (e.g., save events), UI manipulations (e.g., displaying diagnostics), and command registrations, making it possible for modular and reusable extensions.

For sophisticated language features, VS Code's Extension API provides functionalities such as autocompletion, go-to-definition, and real-time diagnostics even for domain-specific languages [12]. The Extension API is being used in CodeSense to bring the static analysis of PyLint and LLM-suggested improvements together, facilitating smooth diagnosis delivery and code enhancement suggestions through the editor.

E. Large Language Models in Code Analysis

Large language models (LLMs) like CodeLlama [7], GPT-4 [14], and StarCoder [13] have transformed software development with sophisticated code completion, error identification, and refactoring. With immense code and natural language training data, these models are adept at understanding programming idioms and providing contextually appropriate recommendations. For example, CodeLlama, which is tuned for code-specific tasks, is capable of refactoring Python functions to make them more readable or proposing patches for common issues.

In IDEs, LLMs improve productivity through the automation of mundane activities like boilerplate code writing or document generation. But some issues include hallucination (producing plausible but wrong code), sparse contextual understanding within a big codebase, and how hard it is to integrate into current projects. Privacy is also a problem with cloud-hosted LLMs, as proprietary code gets exposed. CodeSense avoids such problems by locally hosting a CodeLlama model through Ollama [8], anchoring LLM responses on PyLint diagnostics to provide focused and trustable suggestions.

F. Static Code Analysis with Large Language Models

Recent work has investigated the use of LLMs in static code analysis, taking advantage of their reasoning strength to complement classical tools. SkipAnalyzer [1] integrates LLMs with static analysis to supress false positives and produce accurate patches, with 93.88% precision on Python bug fixes. LLift [3] uses LLMs to identify uses of uninitialized variables, with 100% recall over known bugs and detection of new problems at 50% precision. E&V [2] applies structured prompts to enhance LLM performance in static analysis tasks with 81.18% accuracy on actual bugs.

These investigations emphasize the capacity of LLMs to supplement rule-based analyzers through contextual knowledge and intelligent fixes. Yet the high computational expense and prompt engineering necessary to produce LLM outputs tailored to particular analysis objectives remain challenging. CodeSense extends these developments by combining the diagnostics of PyLint with CodeLlama's reasoning, providing an efficient, IDE-native option for Python programmers.
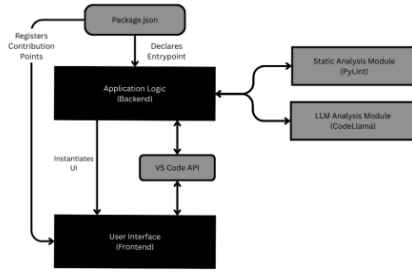
Fig. 1. CodeSense architecture.

III. System Architecture

CodeSense is a Visual Studio Code (VS Code) extension that aims to improve code quality through the integration of PyLint's static analysis and large language model (LLM)-based suggestions. The architecture, as depicted in Figure 1, consists of three main parts: the VS Code Extension Interface, Static Analysis Engine (PyLint), and the LLM-Based Reasoning Module (CodeLlama via Ollama). These elements communicate with each other through a data flow structure to examine source code, interpret problems, and offer live recommendations. Figure 2 also illustrates the system workflow and data flow, emphasizing the smooth integration of static analysis and LLM-driven examination in the VS Code platform.

A. VS Code Extension Interface

The CodeSense extension, developed in JavaScript based on the Visual Studio Code (VS Code) Extension API [12], is the main interface for providing real-time code analysis and smart fix suggestions to the developer's integrated development environment (IDE). It tracks file changes, handles diagnostics, enables user interaction, and displays actionable results via VS Code's native interfaces so that it feels like an integral part of the typical developer workflows. Its modularity guarantees compatibility with its existing feature set and allows easy future additions, for example, for sophisticated code quality visualizations or for version control system integration like Git.

The extension's main functionality starts with live monitoring of file changes through the `onDidSaveTextDocument` event triggered by the VS Code API, which is used to identify save events on Python files to automatically initiate analysis. This follows the natural developer practice of saving files as a rational place for code validation. Diagnostics are shown in the editor with the `createDiagnosticCollection` VS Code API function appearing as inline annotations—red underlines for errors, yellow for warnings, and blue for informational messages. Every annotation has extensive metadata, including a human-readable message, severity level, and exact

source location (line and column), with "CodeSense" as a label to differentiate them from other extensions or VS Code's integrated linters.

CodeSense offers fix proposals via a separate command, `CodeSense: Generate Fixed Code`, activated via the VS Code Command Palette or a user-configurable keybinding. When called, this command initiates the creation of fixed code from the response of the LLM (from CodeLlama 7B hosted by Ollama). The extension sends a request with the code snippet, description of the issue, and a request for a fixed version, and retrieves and interprets the output of the LLM. The fixed code is simply copied into the currently active editor, overwriting the appropriate section or file where needed.

B. Static Analysis Engine (PyLint)

The CodeSense extension's Static Analysis Engine (PyLint) module uses PyLint [6] to run full code checking, automatically invoked on each file save. PyLint checks Python source code against PEP 8 guidelines and user-definable configurations and flags issues like:

- Code convention rule breakages (e.g., no docstrings, C0116)
- Unused imports or variables (e.g., W0611)
- Complexity and performance issues (e.g., high cyclomatic complexity, R0912)
- Possibly erroneous constructs (e.g., undefined variables, E1101)

PyLint's output contains information such as line numbers, message IDs, and severity levels. The extension interprets this output and maps it to VS Code diagnostics via the `createDiagnosticCollection` method of the VS Code API, providing inline annotations (e.g., red underlines for errors, yellow for warnings) with hover-over tooltips for concise, developer-oriented feedback. These diagnostics are stored which can be used as the basis upon which the LLM module can produce fix suggestions through the `CodeSense: Generate Fixed Code` command. The modularity facilitates future integration of other linters, and error handling that is strong ensures reliability for malformed outputs. Configuration choices and exhaustive documentation enable developers to tailor PyLint's rules to project-specific standards so that effective issue detection at an early stage can be achieved in the VS Code ecosystem.

C. LLM-Based Reasoning Module (CodeLlama via Ollama)

The CodeSense extension's LLM-Based Reasoning Module, fuelled by a locally deployed CodeLlama 7B model through Ollama's REST API [8], augments static analysis with smart, context-based code suggestions. As opposed to the conventional linting utilities that follow predefined rules, CodeLlama utilizes its reasoning power to supply context-specific advice derived from the codebase's context. The primary features are:

- **Interpreting Diagnostics:** CodeLlama interprets PyLint's output, adding contextual information to understand fully the range and type of issues reported.

- **Producing Suggestions:** It generates actionable solutions, like refactored code or optimized algorithms, provided as corrected code fragments when the `CodeSense: Generate Fixed Code` command is called. For example, for a high cyclomatic complexity warning (R1260), it could recommend splitting an intricate function into simpler, module-like functions with meaningful names.
- **Code Quality Evaluation:** CodeLlama goes beyond error fixing by evaluating naming conventions and logical coherence and suggesting enhancements such as more descriptive variable names or better algorithms to increase productivity.
- **Prompt Generation:** The extension builds structured prompts from PyLint diagnostics such as the diagnostic code (e.g., W0611), the involved code snippet, and a particular request (e.g., "fix unused variable" or "simplify this function"). These prompts ensure accurate and consistent messaging with CodeLlama.

Prompts are sent to the Ollama server, and the output fixed code is copied into the editor. Local deployment of CodeLlama provides low-latency responses and maintains privacy by hosting code on the developer's machine. The modular architecture allows for extensibility for upcoming LLMs or improved reasoning capabilities. Configuration settings enable developers to fine-tune prompt templates or model settings, and complete documentation provides simplicity of use, giving the module its strength as an addition to static analysis in the context of VS Code.

D. Workflow and Data Flow

The Workflow and Data Flow of the CodeSense extension, as illustrated in Figure 2, adopts a streamlined process to reason about Python code and offer smart fix suggestions in VS Code. The workflow starts with user input and ends in applying fixed code, ensuring a smooth blending of static analysis and LLM-based reasoning. The process is described as follows:

1) **File Save:** The programmer saves a Python file in VS Code, which causes the extension through the `onDidSaveTextDocument` event to run analysis.
2) **Static Analysis:** PyLint analyzes the file against PEP 8 guidelines and custom settings and produces diagnostics for violations like code convention, unused variables, complexity warnings, and error-prone constructs.
3) **Display Diagnostics:** If no linting errors are detected, a "No Issues Found" message is presented. Otherwise, PyLint's diagnostics are mapped to VS Code diagnostics and presented as inline annotations (e.g., red underlines for errors) with hover-over tooltips for rich feedback.
4) **Query Local LLL:** When the user triggers the command `CodeSense: Generate Fixed Code`, the extension transforms PyLint's diagnostics into structured queries, such as the code snippet, issue information, and a call for fixed code, and the fixed code is inserted into the editor.

IV. METHODOLOGY

This subsection delivers an in-depth description of the methodology used for the development, implementation, and testing of CodeSense extension, a productivity tool aiming to augment developer productivity by combining static analysis and large language model (LLM)-backed code fix proposals in Visual Studio Code (VS Code). The approach includes the choice and preprocessing of benchmarking datasets, conversion of the output of static analysis to actionable input for the LLM, generation and validation of proposed fixes, deployment of an offline LLM for maximizing privacy and performance, development pipeline for the extension, and the evaluation metrics employed to gauge its efficacy. Every component is explained in detail in order to give a clear idea of the technical solution and the reasons behind it.

A. Data Collection

In order to carefully test the performance of CodeSense, we used the QuixBugs dataset [15], a reputable benchmark for automated program repair and code analysis. QuixBugs is comprised of 40 traditional algorithmic problems implemented in Python, each presented in two forms:

- A **buggy version**, with a mix of errors such as syntactic problems (e.g., incorrect syntax), semantic mistakes (e.g., logical bugs leading to wrong outputs), and stylistic infractions (e.g., non-adherence to PEP 8 coding norms).
- A **corrected version**, which is the ground truth, and corresponds to the perfect implementation of the algorithm without bugs.

The QuixBugs dataset was selected for some key reasons:

- **Variety of Errors**: The data set has a variety of error types, facilitating exhaustive testing of CodeSense's capacity to identify and correct various categories of errors.
- **Real-World Validity**: The algorithmic issues (e.g., sorting, graph walk, dynamic programming) reflect typical programming work, which makes the data set representative of actual coding environments.
- **Controlled Environment**: The buggy and corrected pairs enable accurate measurement of issue detection precision and fix quality through comparison of the outputs of CodeSense with the ground truth.

Each project within the dataset is treated as its own independent Python module, mirroring a common single-file developer workflow. By doing so, CodeSense is tested within a context typical of how developers work with code in an integrated development environment (IDE). To make the dataset even more relevant, we augmented it with metadata, like problem summaries and unit tests (where they existed), to enable behavioral testing of fixes when evaluating.

B. Linting Output Processing

CodeSense leverages PyLint [6], a widely used static analysis tool for Python, as its primary engine for detecting code issues. PyLint was selected due to its comprehensive rule set, which covers syntactic, semantic, and stylistic issues, and its ability to
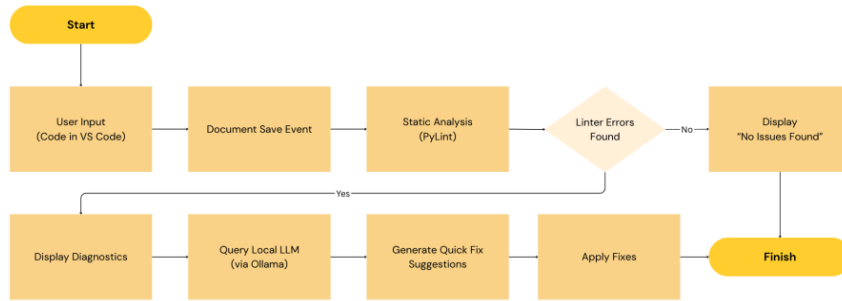
Fig. 2. CodeSense Data Flow Diagram, illustrating the sequence from file save to suggestion display.

produce structured output suitable for automation. The linting process is triggered automatically upon each file save within the VS Code environment, ensuring real-time feedback for developers.

PyLint's output is converted to JSON format, which provides a machine-readable structure for further processing and displaying diagnostics. The JSON output includes the following key fields for each detected issue:

- **line**: The source file line number at which the problem arises, allowing for precise identification of the problem.
- **column**: The character position in the line indicating the start of the errant code block, reserved for exact inline marking.
- **message**: A string description of the problem, informing developers about the kind of problem and directing LLM-based fix generation.
- **severity**: PyLint-reported severity level (e.g., `error`, `warning`, `convention`, `refactor`), which controls prioritization and UI display inside the extension.

The linting output is parsed using a custom JSON parser integrated into the CodeSense extension. The parsed diagnostics are stored within the extension, allowing for efficient retrieval during prompt generation and user interaction. Additionally, the diagnostics are displayed directly in the VS Code editor as annotations (e.g., squiggly lines with hover-over messages), providing immediate visual feedback to developers. This dual use of diagnostics—supporting both user-facing feedback and LLM input—ensures a seamless integration of static analysis and AI-driven assistance.

C. Prompt Generation for LLM

To allow the LLM to produce a good fix suggestion in terms of accuracy and context relevance, CodeSense converts PyLint diagnostics into naturally phrased, structured prompts. The generation of prompts is formulated to achieve clarity, consistency, and informativeness in order to enable the LLM

to understand the problem and provide a high-quality fix. A prompt consists of the following elements:

- **Code Snippet**: A portion of the source code around the issue to offer enough context.
- **Normalized Issue Description**: A short, standardized explanation of the issue using the PyLint `message` and `symbol`. Normalization minimizes variation in wording, which otherwise may result in inconsistent LLM replies.
- **Instruction for Fix**: A precise directive telling the LLM to return a fix for the faulty code.

Prompts are written in JSON objects to enable structured dialogue with the LLM through the Ollama API. The JSON format consists of fields for the code snippet, issue description, and instruction, ensuring efficient transmission of all required details.

The process of generating prompts also considers possible ambiguities of PyLint diagnostics. For example, some stylistic problems (e.g., naming conventions of variables) might need more context to provide sensible fixes. When that happens, CodeSense augments the prompt with additional metadata, like project coding style rules (if identified) or PEP 8 standards references, to lead the LLM to make contextually sensible recommendations.

D. Fix Suggestion Generation

The CodeSense fix suggestion engine is driven by the CodeLlama 7B model [7], a language model that has been specifically fine-tuned for code completion and code transformation tasks. CodeLlama was chosen due to its high performance on code repair evaluations and ability to run relatively smoothly in local environments through Ollama, providing real-time interaction without subjecting user code to cloud APIs.

Once a prompt has been built from PyLint diagnostics, it is passed to the local Ollama server running the CodeLlama model. The model runs the prompt and returns an altered version of the initial code with changes intended to fix the diagnosed problem(s). Different from methods that produce patch-style

diffs, CodeLlama in CodeSense is instructed to return the entire corrected code block, which is then substituted for the equivalent section in the editor in-place.

This solution is placed into the editor window in-place, so its effects are immediately apparent to the user. No automatic steps of validation—like syntax checking, re-linting, or test running—are done by the extension at this point. Rather, the developer is given the task of reviewing changes and ensuring the correctness of the proposed fix. This choice of design keeps responsiveness and simplicity ahead of potential overheads, with feedback being provided instantaneously without interference.

Where the LLM response is malformed, unparseable, or obviously incorrect (e.g., truncated code, missing syntax), the response is ignored, and no correction is made. Instead, the original PyLint diagnostic is still displayed, and the user can try a manual correction or re-solicit the LLM. This fallback mechanism prevents CodeSense from becoming disabled by or intrusive on imperfect model outputs.

E. Offline LLM Deployment

In order to alleviate developer privacy concerns, minimize inference latency, and end reliance on third-party services, CodeSense hosts the CodeLlama 7B model locally through Ollama [8], an open-source system for running LLMs on commodity hardware. The use of a local host was motivated by a variety of factors:

- **Privacy**: Processing code and prompts locally, CodeSense prevents sensitive source code from escaping the developer's machine, a top priority for enterprise and individual developers on proprietary projects.
- **Latency**: Local inference removes network round-trip times of cloud-based APIs, which results in faster response time.
- **Reliability**: Offline deployment ensures CodeSense functionality in areas with poor or no internet connectivity, for example, when traveling or in secure development environments.

The Ollama server presents a light-weight REST API that CodeSense queries to post JSON-formatted prompts and obtain streamed answers. The server is designed to be executed as a background process on the developer's machine, with resource consumption tailored to have an optimal level of impact on system performance.

F. Development Approach

The implementation of CodeSense proceeded by pragmatic and modular design, with an emphasis on incremental feature introduction and conformance to the current strengths of the Visual Studio Code extension API. The extension is written in JavaScript and uses VS Code's event-driven nature to track file changes and react to user input in real time. Static linting integration was the starting point for development with PyLint and was expanded incrementally to encompass LLM-based fix generation and inline diagnostics.

The design was modular, with fundamental features such as linting execution, diagnostic generation, prompt building,

and interaction with the LLM being encapsulated in separate functions. This modularity makes testing, debugging, and future enrichment of features like support for more linters or model types easier.

Static analysis is performed using PyLint, which is invoked through the Node.js `child_process` module. PyLint is invoked on the contents of the currently opened file via standard input, enabling CodeSense to analyze changes and produce real-time feedback. The raw output from PyLint is parsed line-by-line using regular expressions and translated into VS Code diagnostic objects that are presented in the editor's UI through the `DiagnosticCollection` API.

To enable smart code suggestions, the extension interacts with a locally served CodeLlama model through the `ollama` client library. When a user triggers the command to fix code, the extension builds an elaborate prompt encompassing the contents of the current file and PyLint messages. This prompt is forwarded to CodeLlama, which responds with the refactored code and suggested fixes. The code returned is then opened in a new editor tab so that the developer can manually review and apply changes. This keeps latency to the minimum while providing complete control over integrating suggested fixes.

In the initial development, focus was on the development of a responsive and usable baseline system that would provide real-time static analysis and minimal LLM-assisted corrections independently of cloud infrastructure.

Version control was established using Git, with development compartmentalized into feature-specific branches and regular integration testing. Ad-hoc testing was done throughout development on a variety of Python files, including those in the QuixBugs dataset, to verify that it worked with various coding styles and types of errors.

G. Evaluation Metrics

The analysis of CodeSense centers on both its technical efficacy and developer productivity impact. The effectiveness of the extension is determined by the following metrics, with values calculated from the QuixBugs dataset and actual usage logs:

- **Precision**: The ratio of bugs reported by CodeSense that are accurately classified as actual faults (i.e., true positives over true positives and false positives). High precision guarantees that developers are not inundated with irrelevant or spurious diagnostics.
- **Recall**: The ratio of actual problems in the code actually detected by CodeSense (i.e., true positives to true positives plus false negatives). High recall means that the extension misses few errors, offering extensive coverage.
- **F1 Score**: The harmonic mean of precision and recall that offers a balanced estimate of detection accuracy. F1 score is specifically helpful for assessing performance on imbalanced datasets, where the quantity of true issues could be quite diverse.
- **Fix Accuracy**: The ratio of LLM-proposed fix suggestions that correctly fix the reported defect without introducing new defects. This is calculated by applying each fix to

the buggy code, and comparing the output with the fixed version in QuixBugs.

- **Keystroke Savings**: The decrease in manual edits to code resulted by using AI fixes, quantified by the difference in the number of keystrokes it takes to manually correct a problem compared to implementing the proposed fix.

These measurements are calculated by executing CodeSense on the QuixBugs dataset and comparing its outputs (diagnostics and fixes) with the ground-truth corrected versions. For fix accuracy, we also handle scenarios where there are multiple valid fixes, with any solution being accepted that fixes the problem and passes validation. To be robust, evaluations are performed over multiple runs in order to take into account variability in LLM output, with results averaged to give mean and standard deviation statistics.

Aside from quantitative measures, we also perform qualitative analysis via developer commentary, paying attention to usability features like the readability of fix suggestions, the reactivity of the UI, and how it integrates overall with the VS Code workflow. This commentary is utilized to iteratively enhance the extension so that it stays relevant to its intended users.

V. Results

In this section, we discuss the results of testing CodeSense on the QuixBugs benchmark, which consists of 40 small-scale Python programs containing a single bug. The test centers on two essential dimensions of CodeSense's performance: whether it can effectively identify problems within the buggy code through static analysis and whether it is effective in coming up with insightful fix suggestions. Interestingly, in this data set, every program's fault was isolated to a single line of code, although in more general cases, faults would cross several lines, perhaps adding to the detection and fix complexity. The results also demonstrate CodeSense's strengths in both detecting and fixing defects as well as its potential to increase developer productivity by limiting manual debugging effort.

A. Detection Accuracy

CodeSense's detection accuracy was measured using its built-in static analysis linter, which identifies syntactic bugs efficiently and accurately without the need to execute the program. This method provides quick problem detection and is most useful for identifying syntactic and structural flaws early in the code development process. Table I shows the precision, recall, and F1 score, calculated in comparison to the ground truth established by the corrected versions of each program in QuixBugs. Precision is the ratio of reported issues that were bugs, recall the ratio of bugs caught, and the F1 score gives an equal-weighted measure of detection correctness.

These results show how strong CodeSense's performance in issue detection is. The recall of 1.00 confirms that the tool accurately detected all true bugs among the 40 programs, with complete coverage. The accuracy of 0.75 demonstrates that 75% of the reported issues were real, keeping false positives low to avoid misguiding developers. The F1 score of 0.86 depicts

TABLE I
ISSUE DETECTION PERFORMANCE ON QUIXBUGS

Metric	Value
Precision	0.75
Recall	1.00
F1 Score	0.86

an excellent balance between accuracy and recall, highlighting CodeSense's dependability. The application of a static analysis linter worked well in this case, because it allowed the tool to identify syntactic errors in one-line bugs quickly and correctly.

B. Fix Suggestion Accuracy

The accuracy of fix suggestions of CodeSense was measured by comparing how often its suggested fixes successfully fixed the found problems without causing new defects. In the QuixBugs benchmark, in which every bug was localized to a single line, the fix suggestions were optimized to cover these compact problems. The assessment involved comparing tool output to the known correct version of every program. Table II aggregates the findings, reporting the total number of fix attempts, the number of correct fixes, and the resulting fix accuracy.

TABLE II
FIX SUGGESTION EVALUATION

Metric	Value
Fix Attempts	40
Valid Fixes (Corrected)	30
Fix Accuracy	0.75

The outcomes indicate that CodeSense tried fixes for all 40 bugs they found, fixing 30 successfully, and had a fix accuracy of 0.75. This means that 75% of the suggested fixes were accurate and preserved the desired program's functionality. The high fix accuracy is a testament to the capability of CodeSense in suggesting actionable fixes, especially for single-line bugs, while its architecture is conducive to fixing more complex, multi-line defects in other applications, possibly needing more advanced fix techniques.

C. Keystroke Savings

The keystroke savings measure the saving in manual editing work provided by the use of CodeSense's AI-based fixes. For the 30 programs that were successfully fixed, the overall keystroke savings is estimated at 300 keystrokes based on an average of 10 keystrokes saved per fix over manual correction. Although this is a direct saving in typing work, the actual saving in developer work is probably greater. Manual detection of bugs relies heavily on time and mental processing, such as code examination, test execution, and identification of root causes. For multi-line bugs, it could be much more because of added cognitive complexity, which may lead to loss of productivity. With automation of bug detection using static analysis and accurate fix suggestions, CodeSense lowers such an overhead

tremendously, enabling developers to work at a higher level, which is augmenting productivity. The 300-stroke saving, then, is a cautious assessment of the productivity benefits, the real effect on development efficiency being far higher potentially, particularly in instances of more difficult, multi-line matters.

VI. CONCLUSION

In this paper, we presented **CodeSense**, a Visual Studio Code extension that combines conventional static analysis with large language model (LLM) reasoning to offer smart code improvement recommendations. The tool utilizes a static analysis linter to identify code problems rapidly and reliably without the need for program running, and employs a locally hosted CodeLlama instance via Ollama to produce context-aware fix suggestions. CodeSense is crafted to improve code quality and developer productivity through real-time diagnostics and actionable suggestions for fixes within an uninterrupted IDE experience.

Our test on the QuixBugs dataset of 40 small-scale Python programs, each containing a single-line bug, proved CodeSense's capability to identify and fix issues in code. With accuracy of 0.75, recall of 1.00, and F1 score of 0.86, the tool had complete coverage of all bugs while having 75% of the identified issues as real, balancing strongly between accuracy and completeness. For fix generation, CodeSense obtained a fix accuracy of 0.75, i.e., 75% of the LLM fixes were correct and fixed the detected issues without creating new defects. In addition, the tool yielded an estimated 300 keystroke savings for the 30 successfully repaired programs, with actual developer effort savings higher as a result of decreased time and cognitive burden to manually detect and fix bugs. These findings confirm the hybrid method, whereby the static analysis linter achieves fast and accurate issue detection, and the LLM provides context-sensitive and flexible fix suggestions. Interestingly, whereas the QuixBugs dataset consisted of single-line bugs, CodeSense is intended to address more complicated, multi-line bugs in more diverse situations based on the linter's capacity to quickly diagnose syntactic bugs.

In spite of these encouraging results, there are certain limitations that exist. The present LLM backend, although effective, sometimes produces fixes that are syntactically correct but semantically wrong, especially for complicated problems that might extend across several lines. The static analysis linter helps to alleviate some of these problems by guaranteeing syntactic correctness, but more can be done to augment semantic resilience, particularly for larger codebases. Second, CodeSense has been validated mainly for Python and single-file modules. Support must be extended to multi-file projects or other languages as an entirely new development.

Future directions for enhancing CodeSense include refining CodeLlama model on a set of human-authored real-world code bugs and their respective human fixes to enhance semantic precision of fix proposal, especially for multi-line bugs. We also plan to extend CodeSense to cover more programming languages, e.g., JavaScript and TypeScript, by adding suitable linters and language servers. We will explore adding automated test case generation to ensure the functional correctness of fixes beyond improvements to syntax. We also aim to implement a feedback loop that leverages developer interactions—like accepting or rejecting suggestions—as implicit signals to continue to improve prompt engineering and enhance the ranking of suggested fixes. This would allow CodeSense to evolve to understand individual coding styles and project-specific conventions over time. Lastly, we intend to extract the underlying CodeSense engine into an IDE-independent language server, which can be made available to editors other than VS Code, like Neovim, Atom, and JetBrains IDEs.

In short, CodeSense shows what is possible with hybrid systems that have the accuracy of static analysis and the adaptability of LLMs. By basing intelligent code suggestions on strong diagnostics and providing meaningful improvements in the developer's workflow, CodeSense is a major step forward in intelligent programming help. As software programming becomes more complex, tools such as CodeSense will be essential for assisting developers to write cleaner, more maintainable, and more efficient code.

REFERENCES

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

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

[3] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "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, "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. Accessed: Apr. 06, 2025. [Online]. Available: https://doi.org/10.1145/3292500.3330699

[5] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, Mar. 2015, pp. 161-170. Accessed: Apr. 06, 2025. [Online]. Available: https://doi.org/10.1109/saner.2015.7081826

[6] "Pylint 3.3.4 documentation." Accessed: Mar. 22, 2025. [Online]. Available: https://pylint.readthedocs.io/en/stable/

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

[8] "Ollama." Accessed: Mar. 23, 2025. [Online]. Available: https://ollama.com/

[9] "GitHub Copilot," GitHub. Accessed: Mar. 25, 2025. [Online]. Available: https://github.com/copilot

[10] P. Kruchten, R. L. Nord, and I. Ozkaya, "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, "Visual Studio Code," Microsoft. Accessed: Mar. 22, 2025. [Online]. Available: https://code.visualstudio.com/

[12] Microsoft, "Visual Studio Code," Microsoft. Accessed: Mar. 25, 2025. [Online]. Available: https://code.visualstudio.com/api/references/vscode-api

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

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

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

CodeSense: Combining Static Analysis and LLMs for Code Review

based CUDA extension for Visual Studio Code", Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, 2023

Publication