JADE: A VSCode Plugin for Static Analysis-Guided AI-Assisted Refactoring

Ruan Pablo de Assis Neres
Instituto Federal do Triângulo Mineiro
Uberlândia, Brazil
ruan.neres@estudante.iftm.edu.br

Marcelo de Almeida Maia
FACOM, Universidade Federal de Uberlândia
Uberlândia, Brazil
marcelo.maia@ufu.br

Carlos Eduardo de Carvalho Dantas
Instituto Federal do Triângulo Mineiro
Uberlândia, Brazil
carloseduardodantas@iftm.edu.br

Abstract

Software developers often use static analysis tools to identify code warnings related to code quality in general. However, despite the availability of automated warnings, developers still need to manually interpret and apply the suggested fixes. Recent advances in Large Language Models (LLMs) have created new opportunities to support automated refactoring suggestions directly within development environments. This paper presents JADE (Java Static Analysis Repair), a Visual Studio Code plugin that combines static analysis knowledge with LLM-based refactoring suggestions. JADE adopts a Retrieval-Augmented Generation (RAG) strategy based on SonarQube rules, retrieving semantically relevant static analysis heuristics to enrich structured prompts submitted to local LLMs. The plugin supports AI-assisted code diagnostics and refactoring generation integrated into the VSCode workflow. In addition, JADE incorporates a developer feedback mechanism that allows users to evaluate the usefulness and relevance of the generated recommendations. An exploratory study involving 50 Java code snippets suggests that JADE complements traditional static analysis by identifying additional semantic refactoring opportunities while preserving the strengths of rule-based analysis for detecting explicit code quality issues.

Demo Video: https://doi.org/10.6084/m9.figshare.32399817

Keywords

Static Analysis tools, Large Language models, SonarQube, Retrieval-Augmented Generation, Java

1 Introduction

Developers often use static analysis tools to analyze the source code without running the program in order to identify potential warnings such as bugs, coding standards violations, maintainability problems, and poor design practices [2, 7, 24]. During development, these tools raise warnings and recommendations, enabling developers to detect and correct problems early, contributing to improved software quality and maintainability [14, 17]. Additionally, recent previous studies have used warnings produced by static analysis tools as indicators to evaluate code quality attributes [3, 5, 25].

However, identifying code fragments that could potentially be improved does not guarantee that developers will actually improve the code. Developers may choose to follow these recommendations, ignore them, or even lack sufficient guidance on how to properly address the reported warnings. Previous studies have shown that only a few warnings reported by static analysis tools are actually resolved in practice [6, 17, 20]. Additionally, these tools may also generate numerous false-positive warnings [1, 15].

As Large Language Models (LLMs) have gained increasing attention from the software engineering community, several studies have evaluated how to combine them with static analysis tools to automate the refactoring of such suggestions [9, 13, 18]. LLMs can understand the context of the code to perform refactoring operations and provide feedback in natural language [10]. However, they are not always accurate. They may hallucinate [30], miss simple errors, or ignore coding standards if not properly guided [13, 27].

Based on this scenario, this article proposes JADE (Java Static Analysis Repair) [11], a plugin for the Visual Studio Code (VSCode) IDE [29] designed to support developers in identifying and refactoring potential code quality issues using LLMs guided by static analysis rules. JADE integrates the SonarQube static analysis rules [26] with locally executed LLMs through a Retrieval-Augmented Generation (RAG) strategy, retrieving semantically relevant rules and including them as contextual guidance in the prompts. More specifically, the plugin combines the source code written by the developer with semantically relevant static analysis rules retrieved from the SonarQube catalog, allowing locally executed LLMs such as DeepSeek-Coder (6.7B) and Qwen2.5-Coder (7B) to provide AI-assisted code review and AI-assisted refactoring. Unlike cloud-based assistants, JADE executes LLMs locally through the Ollama infrastructure [21], preserving source-code confidentiality and eliminating API costs. The tool is available publicly as open-source software [11].

JADE also incorporates a feedback mechanism designed to capture developers' perceptions about the usefulness and relevance of AI-generated recommendations. This mechanism was motivated by the fact that LLM-generated recommendations, such as traditional static analysis warnings, can include false-positive or low-relevance findings in practice [1, 15]. By collecting developer feedback for each recommendation, JADE aims to support future investigations on warning acceptance, perceived usefulness, and the practical relevance of static analysis recommendations during software development.

The paper is organized as follows. Section 2 discusses background concepts and related work on static analysis tools and static analysis-guided refactoring approaches. Section 3 presents the architecture and main features of JADE. Section 4 describes the experimental setup adopted in this work. Section 5 presents and discusses the results obtained. Section 6 discusses the limitations and threats to the validity of the proposed approach. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Background and Related Work

Static Analysis Tools - Static analysis tools analyze source code without running the program [19], detecting potential warnings such as logic errors, coding standard violations, and poor design practices [2]. By raising warnings during development, these tools support early issue detection and contribute to improved software quality [14]. One of the most widely adopted tools in this context is SonarQube [28], which supports more than 30 programming languages and provides over 700 static analysis rules for Java. SonarQube classifies warnings into maintainability problems, bugs, and vulnerabilities. Its rules and metrics have also been used in recent studies, for example, to analyze code smells in code snippets written by humans and LLM [5], detect readability changes in source code generated by LLM [25], and measure technical debt pull requests from GitHub [3]. In JADE, these static analysis warnings are incorporated into LLM prompts, allowing the generation of refactoring suggestions, i.e., modifications intended to improve code quality without changing the program's external behavior [8], guided by SonarQube static analysis rules.

Static Analysis-Guided Code Refactoring - Even before the emergent use of LLMs, previous works have already proposed to refactor source code based on static analysis tools recommendations. The Styler tool [16] is an approach that addresses readability-related improvements such as code styling. Their method collects warnings from the Checkstyle tool [4] for each class and trains a neural model to apply the linter recommendations. The JADE tool has a similar objective of automatically addressing static analysis warnings but focuses on SonarQube rules due to their widespread use in industry, extensive Java rule catalog, and well-documented quality model. Although the current implementation is limited to SonarQube, the proposed RAG-based architecture can be extended to incorporate rules from other static analysis tools (such as Checkstyle [4] and PMD [22]) in future work.

Recent work proposed a hybrid approach that integrates static analysis knowledge with LLMs at different stages of the generation pipeline to improve the relevance and completeness of automated code review comments [13]. In contrast, JADE focuses on assisting developers during code improvement by generating refactoring suggestions from static analysis warnings directly within the IDE.

Another work [9] combines SonarQube with LLMs to automatically improve Java source code by repeatedly analyzing the code with SonarQube, transforming the reported warnings into prompts, sending them to LLMs such as GPT-4-mini and Gemini, and reanalyzing the generated code over multiple iterations. The results showed that the approach was capable of significantly reducing static analysis warnings, achieving average reductions greater than 58% in several scenarios while preserving the original functionality of the code according to manual evaluation.

Similarly to this approach, JADE also combines LLMs with static analysis tools. However, instead of automatically applying iterative refactoring suggestions, JADE keeps developers in the decision loop by allowing them to decide whether the generated refactoring suggestions should be applied. This design decision is motivated by the fact that some static analysis warnings and generated recommendations may not be perceived as useful or relevant in practice.

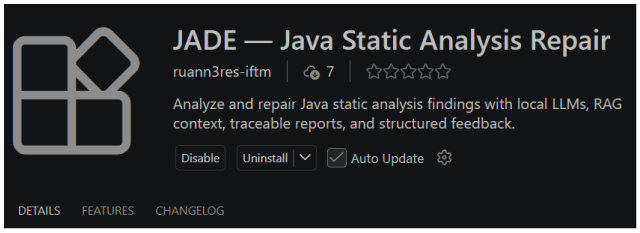


Figure 1: JADE VSCode plugin

3 JADE: Architecture and AI-Assisted Refactoring Workflow

This section presents the architecture of JADE, including its integration with static analysis tools and LLMs. Furthermore, the plugin's usage workflow within the VSCode IDE and its main features are described.

3.1 Tool Usage and Features

JADE was designed from the beginning as a plugin for the VSCode IDE [29], as illustrated in Figure 1. The plugin is also publicly available through the Microsoft Marketplace [11], allowing developers to install it directly from the VSCode extension menu. The decision to implement JADE as an IDE plugin was motivated by the goal of providing developers with direct access to refactoring suggestions without requiring them to leave the development environment. This integration also facilitates the collection of developer feedback on the recommendations generated during programming activities.

3.1.1 External Tools. JADE integrates three external tools to support static analysis retrieval, vector search, and AI-assisted refactoring generation:

- Ollama [21]: LLMs are executed locally through Ollama. By default, JADE is configured to use the *deepseek-coder:6.7b* model for AI-assisted code review and AI-assisted refactoring recommendation tasks.
- SonarQube [26]: JADE uses the static analysis rules provided by SonarQube as a knowledge base to support AI-assisted code review and AI-assisted refactoring recommendation tasks.
- Qdrant [23]: Qdrant is used as a local vector database to store embeddings generated from SonarQube static analysis rules.

3.1.2 Commands. Figure 2 presents the main commands currently supported by JADE. The commands are available through the VSCode IDE Command Palette, allowing developers to access the plugin functionalities directly from the development environment. The current version of JADE provides commands for the execution of static analysis, the generation of LLM-assisted refactoring, model comparison, report visualization, and developer feedback collection. The main commands include:

- **JADE: Export Feedback** opens the external feedback form used to collect developers' perceptions regarding the generated recommendations.

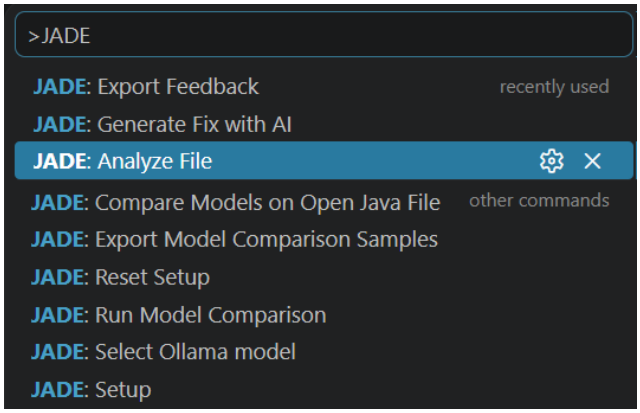


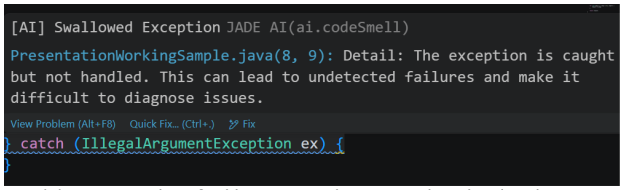
Figure 2: JADE Commands

- **JADE: Generate Fix with AI** generates refactoring suggestions based on SonarQube warnings using LLMs integrated into the plugin.
- **JADE: Analyze File** executes the static analysis process on the currently opened Java file and identifies potential code quality warnings.
- **JADE: Compare Models on Open Java File** Compares supported models on the currently open Java file.
- **JADE: Export Model Comparison Samples** Export official benchmark samples to the workspace.
- **JADE: Reset Setup** Reset RAG setup state and return to the embedded heuristics, such as SonarQube access token and organization input.
- **JADE: Run Model Comparison** and **JADE: Compare Models on Open Java File** support comparative analyses among different LLMs.
- **JADE: Select Ollama Model** allows developers to select the local LLM used during refactoring generation. In this work, the selected models are JADE *deepseek-coder:6.7b* and *qwen2.5-coder:7b*.
- **JADE: Setup** Configure the optional Docker, Qdrant, SonarCloud, and RAG setup.

3.1.3 AI-Assisted Refactoring Workflow. When a developer has a Java class open in VSCode, JADE can be executed through the `JADE: Analyze File` command. The plugin produces a report summarizing the detected warnings and provides contextual explanations describing the identified warning, as illustrated in Figure 3 (a). In this example, JADE identifies a code smell related to a swallowed exception, where an exception is caught but not properly handled.

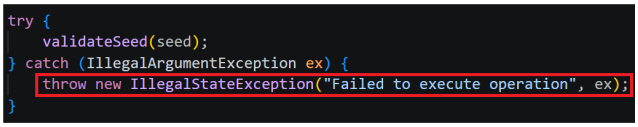
By selecting the underlined code fragment highlighted in blue color, the developer is presented with the `JADE: Generate Fix with AI` option, as shown in Figure 3 (b). Once selected, JADE sends the source code and the corresponding static analysis warning to the configured LLM, which generates a refactoring suggestion to address the reported warning.

Figure 3 (c) presents the resulting refactored code generated by the LLM through JADE. In this example, the generated solution throws an *IllegalStateException* with the message *"Failed to execute operation"* while preserving the original exception instance.



(a) Warning identified by JADE and presented to the developer inside VSCode

(b) JADE offering an AI-assisted refactoring option through the VSCode Quick Fix interface



(c) Refactored code generated by JADE using an LLM to address a static analysis warning

Figure 3: JADE Workflow

3.1.4 Report and Feedback. Figure 4 illustrates the feedback interface provided by JADE to collect developers' perceptions about the usefulness and validity of the recommendations generated from the SonarQube warnings. Through this mechanism, developers can report whether a recommendation is useful, irrelevant, or potentially false positive.

Previous studies have shown that developers' perceptions about code quality recommendations can be highly individual and context-dependent [6, 20]. However, some categories of recommendations tend to appear more frequently in practice, such as renaming-related suggestions. By collecting developers' feedback, JADE aims to support future investigations on recommendation acceptance and continuously refine its recommendation framework based on developer perceptions and usage patterns.

3.2 Tool Architecture

Figure 5 presents the architecture of JADE, which is divided into three main stages: RAG setup and preparation, AI-assisted code review, and AI-assisted refactoring. RAG is a technique that combines information retrieval with LLMs, allowing the model to access external knowledge sources during code analysis and refactoring generation. In JADE, the RAG stage is responsible for storing and retrieving static analysis rules and examples from the SonarQube ecosystem (SonarQube/SonarCloud), providing them as contextual information to the LLM to guide the analysis and refactoring process.

3.2.1 RAG Setup and Preparation. The first step represented in Figure 5 prepares the contextual knowledge base used by JADE. In this step, the developer executes the `JADE: Setup` command and provides the Sonar server URL, which defaults to https://sonarcloud.io. The plugin then redirects the developer to the platform web interface to authenticate and generate an access token and organizational key.

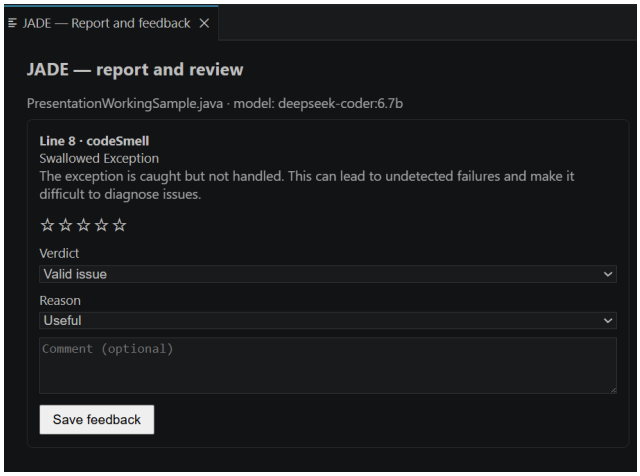


Figure 4: Feedback interface provided by JADE to collect developers' perceptions of the usefulness and validity of the recommendations.

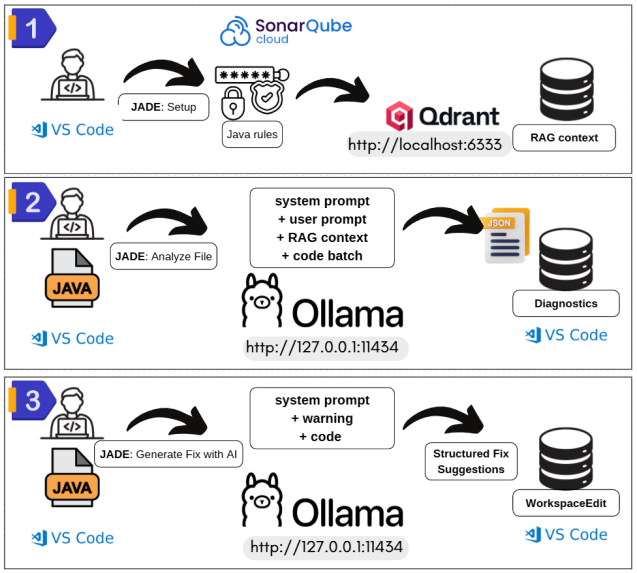


Figure 5: JADE VSCode plugin Architecture

Once authenticated, JADE dynamically retrieves Java static analysis rules directly from the Sonar web APIs. To enable efficient semantic retrieval, the tool generates vector embeddings of these rules and stores them in a local Qdrant vector database. This local database allows JADE to perform semantic similarity searches over the Sonar rules later in the pipeline, retrieving the most relevant coding standards and violation heuristics based on the source code fragment currently under analysis.

The adoption of a RAG-based approach is motivated by the extensive catalog of rules available in SonarQube. Including the complete rule set directly into the LLMs prompt would drastically increase token consumption and degrade prompt efficiency, a constraint that is particularly critical for locally executed LLMs operating with limited input token windows. Instead, JADE queries the local Qdrant database to fetch only the rules and heuristics directly relevant to the target code snippet, reducing the prompt size while preserving contextual relevance.

Furthermore, because JADE dynamically retrieves SonarQube rules during setup, changes to quality profiles or analysis rules are automatically propagated by re-running the setup command, without requiring modifications to the plugin. Consequently, the LLM is based on dynamically retrieved domain-specific knowledge rather than relying solely on its parametric knowledge.

3.2.2 AI-Assisted Code Review. During the review stage (Figure 5, step 2), the developer executes the `JADE: Analyze File` command over a Java source file opened in VSCode. JADE splits the source file into batches of 50 lines and constructs a structured prompt for each batch. This prompt is enriched with the specific SonarQube rules retrieved via RAG that map semantic similarities to the code batch. Each batch is submitted as an independent request to a local LLM through the Ollama infrastructure [21]. Finally, JADE aggregates the results, parses the model output, and converts the suggestions into native VSCode diagnostics, allowing warnings and recommendations to be visualized directly inside the IDE editor.

The system prompt follows a structured prompt engineering strategy composed of five main sections: *role* (defining the agent's expertise), *context* (the input code and retrieved RAG data), *requirements* (operational constraints), *critical* (strict output restrictions, such as valid JSON formatting), and *response* (the expected JSON schema).

Due to space constraints, a simplified excerpt of the code review prompt is presented in the following box. The complete prompt is available in the replication package [12]. For this stage, the LLM execution parameters were set to *NUM_PREDICT* = 4,096, *STREAM* = false, *FORMAT* = json, *TIMEOUT* = 15 minutes, and *TEMPERATURE* = 0.15, as lower temperatures favor stability in deterministic software engineering tasks [10].

Prompt Template Used by JADE for Code Analysis

<role> "You are an agent specialized in Java static code review, focusing on quality, maintainability, reliability.
<context> "In the following user message, data arrives inside ‹input›; with subtags. Variable parts may be wrapped in CDATA to preserve special characters in the code...
<requirements>Mandatory rules: 1..14 Applicable editor
<critical>Reply ONLY with valid JSON: the first character of your message MUST be "" and the last MUST be ""...
<response>Mandatory final format; all human-readable fields ("summary", "detail") must be in English

3.2.3 AI-Assisted Refactoring. The third stage (Figure 5 step 3) corresponds to the automated refactoring workflow. After selecting a warning generated by JADE, the developer can execute the `JADE: Generate Fix with AI` command. In this step, JADE submits a structured prompt containing the warning information and the corresponding source code to the LLM. The model then generates structured fix suggestions, such as line replacements or code range

modifications, which are converted into VSCode *WorkspaceEdit* operations and applied directly to the source code editor.

After a diagnostic is selected by the developer, JADE builds a fix-generation prompt containing the diagnostic information and the complete source code with absolute line numbers. The system prompt constrains the LLM to generate minimal and safe Java patches, requiring the response to be returned as a single JSON object. The model can either suggest replacing a single line, replacing a code range, or return no fix when a safe modification cannot be inferred. Before applying any generated fix, JADE validates the suggestion through multiple safety checks, including brace balance verification, placeholder text rejection, and indentation-aware range validation. Fixes that fail these checks are discarded and the developer is notified that no safe fix could be generated. This structured output allows JADE to convert the generated response into VSCode edit operations while avoiding free-form textual recommendations. All LLM requests were executed with *NUM_PREDICT* = 4,096, *STREAM* = false, *FORMAT* = json, *TIMEOUT* = 15 minutes, and *TEMPERATURE* = 0.05.

Prompt Template Used by JADE for Code Refactoring

You generate minimal safe Java patches for VS Code.
Allowed outputs: "fixKind":"replaceLine","line":1,"newLineText": "raw Java source line"
"fixKind":"replaceRange", "startLine":1, "startColumn":1, "endLine":1, "endColumn":1, "newText":"raw Java source"
"fixKind":"none"
Rules: - Replacement text must be raw Java source only. - Preserve indentation. - Prefer replacing the smallest statement or block.

4 Comparative Study

To provide an initial assessment of JADE, we conducted an exploratory comparative study aimed at analyzing the code review recommendations generated by the tool and comparing them with the warnings previously reported by static analysis in a previous study [5].

In a previous study, 351 Java classes were collected from Stack Overflow, a widely known question-and-answer platform for software developers [5]. The SonarQube static analysis tool was then used to extract code warnings from these classes.

JADE provides a feature called `JADE: Compare Models in Open Java File`, shown in Figure 6, which enables the comparison of recommendations generated by two LLMs: deepseek-coder:6.7b and qwen2.5-coder:7b. Using this feature, we manually analyzed the recommendations produced by both models for a sample of 50 Java classes and contrasted them with the warnings previously reported by SonarQube in the previous study.

Because the goal of this study is exploratory, we intentionally analyzed a smaller sample of 50 Java classes to allow for a detailed qualitative comparison between traditional static analysis and LLM-based recommendations. Although this sample does not support broad statistical generalizations, it provides useful information on the complementarity, strengths, and limitations of the proposed approach.

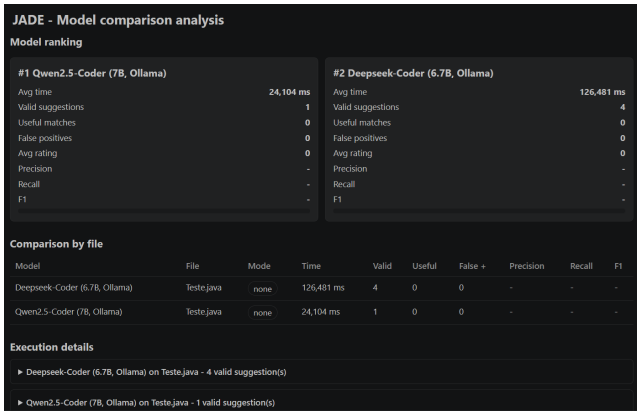


Figure 6: JADE comparing *deepseek-coder:6.7b* and *qwen2.5-coder:7b* models for a single Java file.

5 Results

Table 1 summarizes the distribution of warnings identified by the three approaches evaluated. Overall, the results indicate that the LLM-based approaches and the traditional static analysis baseline identify both overlapping and complementary code quality issues, suggesting that the approaches provide different perspectives during code review.

The baseline *Previous Study* refers to the warnings previously obtained through the execution of SonarQube on these code snippets in a replication package of a prior study [5]. These results are compared with the code review recommendations generated by the *deepseek-coder:6.7b* and *qwen2.5-coder:7b* models. Although LLMs are guided by SonarQube rules through the JADE architecture, models may still produce different recommendations and identify distinct code quality issues.

Table 1: Distribution of warnings across the previous study and LLM-based approaches.

Warning source	# Warnings	%
Only Previous Study	41	28.5
Only DeepSeek-Coder	32	22.2
Only Qwen-Coder	19	13.2
Previous Study + DeepSeek-Coder	18	12.5
Previous Study + Qwen-Coder	7	4.9
DeepSeek-Coder + Qwen-Coder	15	10.4
Reported by all approaches	12	8.3
Total warnings	144	100.0

Most of the warnings (92 out of 144, 63.9%) were reported exclusively by a single approach, while only 12 warnings (8.3%) were shared by all three approaches. This reinforces that the evaluated approaches are largely complementary rather than redundant.

Considering the warnings reported exclusively by each approach, the traditional static analysis approach in the previous study produced 41 instances of unique warnings, that is, code snippets for which warnings were reported only by the static analyzer and not by the evaluated LLMs. Examples include recommendations related

to transforming loops into lambda expressions and Java naming and coding conventions.

DeepSeek-Coder produced 32 instances of unique warnings, frequently identifying higher-level semantic refactoring opportunities, such as long method extraction and code structure refactorings, that were not reported by the static analysis baseline. The smallest number of unique warnings was observed for Qwen-Coder.

5.1 Discussion

By qualitatively analyzing the results, we observed that SonarQube alone tends to focus primarily on predefined static rules, such as naming conventions, variable declarations, and syntactic patterns. In contrast, even when guided by static analysis rules through the RAG context, the studied LLMs tend to perform code reviews considering more semantic and structural aspects of the source code. For example, the models frequently identified opportunities for extracting long methods, simplify nested loops or potential bug prevention.

An interesting case was observed in a code snippet containing the statement `boolean success = f.delete();`. While DeepSeek-Coder identified a potential bug related to the possibility that the file referenced by variable `f` might not exist, SonarQube focused on the fact that the variable `success` was declared but never used. In contrast, Qwen did not report any issue for this code fragment.

We also observed instances of false negatives produced by the LLMs. For example, in one code snippet, two occurrences of `catch (BadLocationException e){}` were present. While both SonarQube and *deepseek-coder* detected the two occurrences, Qwen identified only one of them. In another code snippet, both *deepseek-coder* and SonarQube detected unnecessary statements, whereas Qwen did not report any anomalies.

In general, the exploratory study indicates that the LLM-based recommendations complement, rather than replace, traditional static analysis. This interpretation is reinforced by the observation that 63.9% of the reported warnings were unique to a single approach, while only 8.3% were identified by all three approaches. Although SonarQube consistently detects predefined rule violations, the evaluated LLMs frequently identify higher-level semantic refactoring opportunities that fall outside the scope of rule-based analyzers. At the same time, LLMs may overlook explicit or repetitive issues that are reliably detected by static analysis.

6 Limitations

During the development of JADE, some limitations and opportunities for improvement were identified.

- Currently, JADE validates only the structural consistency of the generated patches. Future versions will automatically compile the project and execute the available test suites before presenting refactoring suggestions.
- Currently, the static analysis process must be explicitly triggered by the developer through the plugin commands. Although this approach allows JADE to generate a complete report regarding the analyzed class, some developers may prefer a fully automatic analysis process that is executed continuously during programming activities.
- JADE currently relies on locally executed LLMs through the Ollama framework [21]. Although this design avoids API usage costs and external rate limitations commonly found in cloud-based LLM services, it also introduces an additional installation and configuration step for developers before using the plugin.
- JADE currently relies exclusively on SonarQube [26] as its static analysis backend. Although SonarQube provides a large set of rules, future versions of the tool could benefit from integrating additional static analysis tools, such as Checkstyle [4], in order to expand the variety of supported recommendations and code quality analyses.
- The current version of JADE is limited to the Java programming language. Although Java is widely adopted in both industry and academia, static analysis tools for other programming languages could also be integrated into the approach in future work.

7 Conclusion and Future Work

This paper presented JADE, a VSCode plugin that combines static analysis warnings with LLM-based refactoring suggestions to support software quality improvement activities. The tool integrates SonarQube warnings into LLM prompts, allowing developers to request AI-assisted refactorings directly from the development environment. In addition, JADE incorporates a feedback mechanism designed to collect developers' perceptions regarding the usefulness and relevance of the generated recommendations.

In future work, one of the main goals is to promote a wider adoption of JADE in order to collect and analyze a larger volume of developer feedback. This feedback may support future investigations involving recommendation acceptance, frequently accepted improvements, perceived false positives, and recommendations considered irrelevant in practice. Future work also includes extending JADE to support additional programming languages, more static analysis tools, and optional integration with online LLM services in addition to locally executed models.

Artifact Availability

The artifacts associated with JADE, including the source code and installation instructions, are archived as a versioned Zenodo record (DOI: https://doi.org/10.5281/zenodo.22151529)

Acknowledgments

The authors appreciate the support provided by the Instituto Federal do Triângulo Mineiro (IFTM) and its Pró-Reitoria de Pesquisa, Pós-Graduação e Inovação (PROPI).

References

[1] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, and Danfeng Yao. 2023. Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks. *IEEE Transactions on Software Engineering* 49, 2 (2023), 485–497. doi:10.1109/TSE.2022.3154717

[2] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 470–481. https://doi.org/10.1109/SANER.2016.105

[3] Felipe Calixto, Eliane Araújo, and Everton Alves. 2024. How does Technical Debt Evolve within Pull Requests? An Empirical Study with Apache Projects. In *Anais*

do XXXVIII Simpósio Brasileiro de Engenharia de Software (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 212–223. https://doi.org/10.5753/sbes.2024.3368

[4] CHECKSTYLE. 2026. CheckStyle Web Page. https://checkstyle.sourceforge.io/. Accessed: 2026-02-20.

[5] Carlos E. Dantas, Adriano M. Rocha, and Marcelo A. Maia. 2023. Assessing the Readability of ChatGPT Code Snippet Recommendations: A Comparative Study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES '23)*. Association for Computing Machinery, New York, NY, USA, 283–292. doi:10.1145/3613372.3613413

[6] Carlos E. Dantas, Adriano M. Rocha, and Marcelo A. Maia. 2023. How do Developers Improve Code Readability? An Empirical Study of Pull Requests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 110–122. doi:10.1109/ICSME58846.2023.00022

[7] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2022), 835–847. doi:10.1109/TSE.2020.3004525

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[9] João Gonçalves and Marcelo Maia. 2025. An Empirical Study on the Effectiveness of Iterative LLM-Based Improvements for Static Analysis Issues. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 382–392. doi:10.5753/sbes.2025.9964

[10] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 34, 13 pages. doi:10.1145/3597503.3623306

[11] JADE. 2026. JADE Visual Studio Code Plugin. https://marketplace.visualstudio.com/items?itemName=ruann3res-iftm.jade-static-analysis-repair&ssr=false. Accessed: 2026-05-23.

[12] JADE_PROMPT. 2026. Prompt template used by JADE for AI-Assisted Refactoring. https://github.com/ruann3res/JADE/blob/main/src/services/ai/prompts/javaAnalysisPrompt.ts. Accessed: 2026-04-23.

[13] Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. 2025. Combining Large Language Models with Static Analyzers for Code Review Generation . In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 174–186. doi:10.1109/MSR66628.2025.00038

[14] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *2013 35th International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[15] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575. doi:10.1016/j.jss.2022.111575

[16] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. 2022. Styler: learning formatting conventions to repair Checkstyle violations. *Empirical Software Engineering* 27 (08 2022). doi:10.1007/s10664-021-10107-0

[17] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219. doi:10.1109/ICPC.2019.00040

[18] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of ChatGPT for Static Analysis: How Far Are We?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software* (Porto de Galinhas, Brazil) *(AIware 2024)*. Association for Computing Machinery, New York, NY, USA, 151–160. doi:10.1145/3664646.3664777

[19] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated. https://dl.acm.org/doi/book/10.5555/1965094

[20] Delano Oliveira, Reydne Santos, Benedito de Oliveira, Martin Monperrus, Fernando Castor, and Fernanda Madeiral. 2025. Understanding Code Understandability Improvements in Code Reviews. *IEEE Transactions on Software Engineering* 51, 1 (2025), 14–37. doi:10.1109/TSE.2024.3453783

[21] OLLAMA. 2026. Ollama Web Page. https://ollama.com/. Accessed: 2026-02-20.

[22] PMD. 2026. Pmd. https://pmd.github.io/. Accessed: 2026-02-20.

[23] QDRANT. 2026. Qdrant. https://qdrant.tech/. Accessed: 2026-04-23.

[24] Simone Romano, Fiorella Zampetti, Maria Teresa Baldassarre, Massimiliano Di Penta, and Giuseppe Scanniello. 2022. Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?. In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (Helsinki, Finland) *(ESEM '22)*. Association for Computing Machinery, New York, NY, USA, 80–91. doi:10.1145/3544902.3546233

[25] Igor Regis da Silva Simões and Elaine Venson. 2025. Measuring how changes in code readability attributes affect code quality evaluation by Large Language Models. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES 2025) (SBES 2025)*. Sociedade Brasileira de Computação, 13–24. doi:10.5753/sbes.2025.9603

[26] SONARQUBE. 2026. SonarQube Web Page. https://www.sonarsource.com/products/sonarqube/ide/. Accessed: 2026-04-23.

[27] Salma Begum Tamanna, Gias Uddin, Song Wang, Lan Xia, and Longyu Zhang. 2025. Chatgpt Inaccuracy Mitigation During Technical Report Understanding: Are we There Yet? . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2290–2302. doi:10.1109/ICSE55347.2025.00145

[28] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 38–49. https://doi.org/10.1109/SANER.2018.8330195

[29] VSCODE. 2026. Visual Studio Code Web Page. https://code.visualstudio.com/. Accessed: 2026-04-23.

[30] Guido Zuccon, Bevan Koopman, and Razia Shaik. 2023. ChatGPT Hallucinates when Attributing Answers. In *Proceedings of the Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region* (Beijing, China) *(SIGIR-AP '23)*. Association for Computing Machinery, New York, NY, USA, 46–51. doi:10.1145/3624918.3625329