Humans in Control: A Methodological Framework for Quality Assurance in Agentic Software Engineering

Matheus Silveira
University of São Paulo (IME-USP)
matheus.feitosa@usp.br

Paulo Meirelles
University of São Paulo (IME-USP)
paulormm@ime.usp.br

Igor Steinmacher
School of Informatics, Computing & Cyber Systems,
Northern Arizona University
igor.steinmacher@nau.edu

Fabio Kon
University of São Paulo (IME-USP)
kon@ime.usp.br

Abstract

As Large Language Models (LLMs) continue to advance, their application in software engineering has expanded considerably, particularly in automated code and test generation, which significantly accelerates the development process. However, the resulting increase in code output often leaves developers overwhelmed by the volume of code requiring review, often leading to a progressive degradation in code quality and the accumulation of cognitive debt among development teams as they struggle to maintain a comprehensive understanding of an increasingly complex codebase. To address the high volume of AI-generated code, this paper introduces a multi-agent methodology for quality assurance in agentic software engineering, specifically aimed at resolving code smells and other maintainability issues reported by static analysis tools, rather than targeting feature generation or software evolution in general. The framework establishes a collaborative workflow that preserves software integrity while keeping humans firmly in control of the engineering process, and it is being supported by an under development dedicated software tool.

Keywords

Code Refactoring, LLM, Agentic Software Engineering, Multi-agent

1 Introduction

With the continued adoption of AI agents in software engineering and the industry's demand for faster development workflows, generating a high volume of code that meets diverse requirements has become straightforward. As a consequence, much debate has arisen regarding the future maintenance of this AI-generated code. Several recent studies show that while agent-produced code is functional, it is often accompanied by a decline in code quality metrics such as Cyclomatic Complexity and Halstead metrics, as observed in AI agent-authored commits on GitHub [3]. Research also indicates that code duplication and structural degradation increase as agents are repeatedly asked to extend or modify the same codebase [5].

This phenomenon, combined with the growing volume of seemingly functional AI-generated code, places increasing pressure on developers responsible for code review. As AI-generated contributions become more prevalent in repositories, thoroughly reviewing every change becomes increasingly challenging, motivating the adoption of AI-assisted code review techniques to relieve reviewer workload while preserving human oversight [2, 9]. Consequently, implementations with unnecessarily high complexity may be integrated into project repositories because they preserve the system's functionality. Although such changes often preserve correctness, this increased complexity can undermine software maintainability and increase future maintenance costs.

Code refactoring is a key strategy for mitigating the long-term maintenance burden posed by increasingly complex software systems. Previous work has shown that higher code and architectural complexity are significantly associated with greater maintenance effort, with developers spending a larger proportion of their time on maintaining codebases rather than developing new features [1]. Consequently, reducing overall complexity through refactoring is important to preserve software maintainability over time. However, the cost of change and the continuous pressure to deliver new features frequently limit opportunities to refactor the codebase [4].

To address this limitation, the deployment of artificial intelligence (AI) agents has been increasingly explored [7, 8, 11]. This shift is justified by the AI agent's ability to complete tasks faster than independent developers and its ability to handle large volumes of data. This trend is evidenced by the increasing amount of code being produced by AI agents in both small and large-scale software projects [2]. Nonetheless, this also raises concerns about the actual impact of these architectural changes and the increase in cognitive and organizational debt, which may result in humans losing control over the codebase they maintain.

In this paper, we propose a multi-agent methodology, with the developer in control, to address code smells and other issues identified by static analysis tools. It aims to preserve the speed of AI code generation while shifting the developer's role from a passive code reviewer to an active participant in the most impactful refactoring decisions and their consequences.

Unlike existing software development agents that operate with limited human intervention, our approach explicitly incorporates the developer as the central component of the decision-making process for assuring code quality during development. This strategy follows a human-in-the-loop paradigm that has recently shown promising results in AI-assisted software engineering [8].

The multi-agent approach is adopted rather than a single-agent architecture because specialized agents can focus on distinct responsibilities throughout the development process. This separation enables more targeted reasoning and better coordination when solving specific tasks during refactoring [7, 11], thereby allowing

agents to provide more precise and contextualized information while ensuring the developer retains control of the overall process.

2 Related Work

Ottenhof *et al.* [6] analyzed how commercially available coding agents, such as GitHub Copilot and Claude Code, perform refactoring tasks in comparison with human developers. Their findings indicate that agent refactorings focus on simple annotation-level changes, in contrast to the more diverse structural refactorings performed by human developers, who prioritize higher-level architectural improvements.

To improve the reliability of AI-assisted code modifications, Wadhwa *et al.* [11] proposed CORE, a multi-stage framework for resolving code quality issues identified by static analysis tools using LLMs. The framework defines a proposer LLM to generate candidate code revisions, validates them with the original static analysis tool, and then uses a ranker LLM to assess whether the generated changes introduce unintended behavioral modifications before presenting them to the developer. This approach reduces false positives while preserving the flexibility of LLM-based code refactoring, demonstrating that a separate AI-review stage can improve the reliability of LLM-generated changes.

In real-world projects, different approaches are being developed by both open-source initiatives and proprietary platforms. In the open-source domain, tools such as Aider-AI[1] and Cline[2] provide conversational agents for interactive code editing and refactoring across multiple files. Similarly, proprietary solutions have implemented workflows for code modification, planning, and refactoring. For example, Claude Code can modify repository code, refactor existing implementations, run tests, and iteratively refine the generated changes based on the developer's feedback. These tools and their widespread adoption indicate the potential of LLMs to perform highly complex software refactoring tasks, even in practical, large-scale development environments.

More recently, Oueslati *et al.* [7] proposed RefAgent, a fully automated multi-agent framework for large-scale Java refactoring. The framework divides the refactoring process into specialized planning, refactoring, compilation, and testing agents, enabling iterative refinement through automated validation. Their results demonstrate that the multi-agent approach significantly outperforms single-agent models, capturing a significant majority of the refactoring opportunities that human developers typically address.

Building on RefAgent's idea, the framework proposed in this paper extends the fully autonomous workflow by explicitly incorporating the human developer into the decision-making process. CORE [11] targets the same class of static analysis issues, but the developer only intervenes at the end of its pipeline, and RefAgent [7] validates its own output automatically. In both cases, the developer remains outside the loop. Rather than treating developers as passive reviewers of the final generated code, in our paper, we focus on the interaction model itself, incentivizing developers to participate throughout the planning and execution stages, inspecting and validating architectural decisions, refactoring strategies, and code patterns before modifications are applied.

[1]https://github.com/aider-ai/aider
[2]https://github.com/Cline/Cline

3 Proposed Methodology

Our framework keeps the developer in control of the entire process, allowing them to make critical architectural decisions throughout the refactoring workflow while leveraging AI's speed and capabilities to manage large codebases. It is structured into two distinct phases, as illustrated in Figure 1.

3.1 Planning Phase

The primary objective of the initial phase is to allow the human developer to inspect the system's current issues while simultaneously gaining visibility into the specific points of attention identified by the AI. This phase utilizes three specialized agents:

Analyzer Agent: Extracts and interprets data from automated code metrics and code smell detection tools. This component enables the human developer to comprehend the system's current state and inspect the exact list of issues detected by the AI through a rich, interactive interface.

Explorer Agent: Since multiple strategies can resolve the same set of coding issues, this agent is responsible for investigating alternative refactoring approaches and presenting their respective expected architectural impacts.

Planner Agent: Following the selection of a refactoring technique, this agent creates an initial step-by-step execution plan, generating a human-verifiable sequence of operations.

After each agent step in this phase, the AI output is presented to the developer through an interactive, structured interface. Using natural language, the developer can then provide feedback, specifying whether the workflow should re-execute a step with new instructions or proceed to the next stage of the refactoring process.

3.2 Refactoring Phase

In the refactoring phase, the framework enables the AI agent to systematically apply code refactoring while ensuring that the modifications preserve the codebase's original behavioral semantics. Human developers validate this process at each step to guarantee that the core objectives of the refactoring are being continuously achieved. The refactoring phase utilizes two AI agents and two external tools, detailed as follows:

Scheduler: Orchestrates the underlying workflow by tracking the current execution step and determining the necessary reviews or modifications. It dynamically decides whether to re-execute a step to apply a refactoring operation or to proceed to the next phase of the plan, routing instructions to the Refactor Agent accordingly.

Refactor Agent: Receives a single, straightforward instruction to generate a set of code modifications corresponding strictly to a well-defined step in the execution plan.

Testing Tool: Verifies that the codebase has not suffered any functional regressions by executing the existing test suite and forwarding a comprehensive test report to the Reviewer Agent to provide additional execution context.

Reviewer Agent: Validates code against the original plan by interpreting user change requests or identifying flaws, then routing instructions back into the refactoring cycle.

In the final review stage, to facilitate the developer's understanding of the architectural changes, the framework presents the final

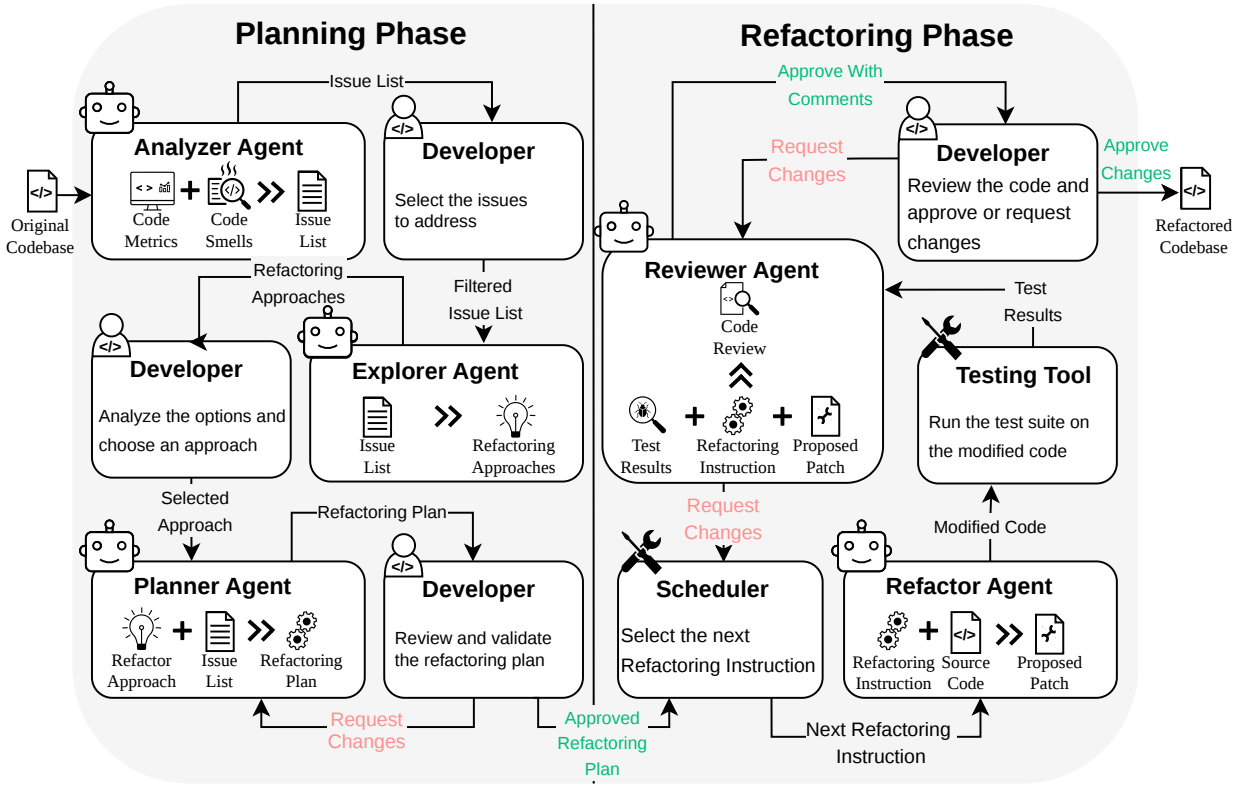

Figure 1: Framework pipeline overview

output within an interactive interface. This interface uses the developer's IDE to display the code diff between the original and modified codebases, overlaid with contextual comments generated by the Reviewer Agent for the approved sections of code.

4 Implementation Approaches

The proposed methodology is designed to be implemented as a VS Code extension, enabling developers to execute the entire refactoring workflow without leaving their development environment. This design is expected to reduce the cognitive overhead of repeatedly switching between the IDE and external tools while leveraging existing IDE utilities such as version control, source code navigation, and debugging. User interaction occurs through a conversational interface integrated within the extension. Instead of interacting directly with each specialized agent, developers can communicate in natural language through a single conversational interface that coordinates the underlying multi-agent workflow and routes requests to the appropriate framework components.

The initial implementation currently targets Python due to the availability of mature static analysis tools and benchmark repositories that facilitate experimental evaluation. However, the methodology is designed to remain language-independent. Unlike approaches that rely on language-specific representations, the framework primarily operates on high-level artifacts, including refactoring plans and test results, exploring LLMs' ability to reason about code without extensive language-specific definitions. Extending the framework to additional programming languages will require only adapting the language-specific analysis and testing components.

The framework is being structured around a modular architecture that separates AI agents from external analysis and testing tools via defined interfaces, allowing replacement with minimal architectural changes. Likewise, the LLM integration layer is abstracted using two independent interfaces that support both locally hosted models and cloud-based services. By isolating model providers from the rest of the infrastructure, the framework can support different models without requiring modifications to its core methodology, thereby facilitating future transitions between LLMs.

5 Framework Evaluation

Given the framework's strong dependence on developer interaction, we propose evaluating the methodology through a user study with software developers of varying experience levels from three groups:

Students. Undergraduate Computer Science students with no industrial experience.

Novice Developers. Software developers with less than two years of industrial experience.

Experienced Developers. Software Developers with at least two years of industrial experience.

Since the implementation targets Python, each participant will receive a repository selected from RefactorBench[3]. Participants will then be divided into a control group, who will use a "raw" agentic environment, and a treatment group, who will follow the proposed methodology step by step. Throughout the experiment, we will record the state of the repository after each accepted modification, enabling analysis of the codebase's evolution over the course of the refactoring process.

The effectiveness of the proposed methodology will be evaluated from three complementary perspectives:

Code Quality Metrics. To assess architectural improvement, we will measure software quality metrics throughout the refactoring process. For the initial Python implementation, these will include Cyclomatic Complexity, Maintainability Index, and raw metrics calculated by Radon[4].

Refactoring Effectiveness. As RefactorBench[3] provides a set of refactoring tasks with their corresponding target locations within each repository, we will measure how many tasks were successfully completed, which remained unresolved, and the overall code coverage achieved during each refactoring session.

Developer Experience. The framework will be evaluated in terms of perceived usability, cognitive workload, as well as user confidence in the performed refactoring [10], collected through surveys using five-point Likert scales, complemented by interaction logs informing the acceptance and editing rates of AI-generated suggestions.

By combining code metrics with developer usability indicators, we aim to verify that the framework supports correct refactoring while remaining intuitive and practical, keeping developers in control of the AI-assisted software engineering process. We also expect some challenges and limitations on this methodology implementation, such as its dependence on static analysis tools and on existing test suites, the possibility of incorrect LLM-generated changes and its effects on the quality of the modifications inside the codebase depending on the developer who is applying the methodology. These aspects will be investigated during the planned evaluation to measure their impact and to verify the viability of the methodology in real-world scenarios.

6 Preliminary Remark

As AI-driven code generation accelerates the pace of software development, it inadvertently risks overwhelming developers with cognitive debt and degrading system quality. In response, this vision paper proposes a paradigm shift in agentic software engineering: moving the developer from a passive reviewer to an active, central decision-maker. Through our methodological framework, structured into collaborative Planning and Refactoring phases, we harness the analytical speed of LLMs while enforcing strict human oversight at architectural junctures. Supported by a prospective IDE extension and an evaluation plan targeting both code integrity and user experience, this work lays the foundation for a more sustainable development life-cycle for the agentic AI era.

[3]https://github.com/microsoft/RefactorBench

[4]https://pypi.org/project/radon/

Artifact Availability

The initial implementation of the proposed framework as a VS Code extension is publicly available at github.com/MatheusSilver/Rick-Agentic-AI-refactoring-assistant-extension under the GNU General Public License v3.0.

Acknowledgments

This study is supported by the São Paulo Research Foundation (FAPESP 2023/00811-0 and 2025/22433-3) and the São Paulo State Data Analysis System Foundation (SEADE/FAPESP 2023/18026-8).

References

[1] Yuanfang Cai, Lanting He, Jun Qian, Yony Kochinski, Nan Zhang, Ciera Jaspan, and Antonio Bianco. 2025. Understanding Architectural Complexity, Maintenance Burden, and Developer Sentiment – A Large-Scale Study. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*. IEEE, Ottawa, ON, Canada, 2176–2187. doi:10.1109/ICSE55347.2025.00168

[2] GitLab and The Harris Poll. 2026. *Global DevSecOps Report: The Intelligent Software Development Era: How AI will redefine DevSecOps in 2026 and beyond.* Report. GitLab. https://about.gitlab.com/resources/developer-survey/ Survey conducted from July 31, 2025, to August 15, 2025, among 3,266 DevSecOps professionals..

[3] Kyogo Horikawa, Kosei Horikawa, Yutaro Kashiwa, Hidetake Uwano, and Hajimu Iida. 2026. Do AI Agents Really Improve Code Readability?. In *Proceedings of the 2026 IEEE/ACM 23rd International Conference on Mining Software Repositories (MSR '26)*. ACM, Rio de Janeiro, Brazil. MSR 2026 Mining Challenge Track.

[4] James Ivers, Robert L. Nord, Ipek Ozkaya, Chris Seifried, Christopher S. Timperley, and Marouane Kessentini. 2022. Industry experiences with large-scale refactoring. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, New York, NY, USA, 1544–1554. doi:10.1145/3540250.3558954

[5] Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. 2026. SlopCodeBench: Benchmarking How Coding Agents Degrade Over Long-Horizon Iterative Tasks. *arXiv preprint arXiv:2603.24755* (2026).

[6] Lukas Ottenhof, Daniel Penner, Abram Hindle, and Thibaud Lutellier. 2026. How Do Agents Refactor: An Empirical Study. In *Proceedings of the 2026 IEEE/ACM 23rd International Conference on Mining Software Repositories (MSR '26)*. ACM, Rio de Janeiro, Brazil. MSR 2026 Mining Challenge Track.

[7] Khouloud Oueslati, Maxime Lamothe, and Foutse Khomh. 2026. RefAgent: A Multi-Agent LLM-Based Framework for Automatic Software Refactoring. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*. ACM, Rio de Janeiro, Brazil, 92–104. doi:10.1145/3744916.3773153

[8] Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2025. Human-In-The-Loop Software Development Agents. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '25)*. IEEE, Ottawa, ON, Canada, 342–352. doi:10.1109/ICSE-SEIP66354.2025.00036

[9] Kla Tantithamthavorn, Yaotian Zou, Andy Wong, Michael Gupta, Zhe Wang, Mike Buller, Ryan Jiang, Matthew Watson, Minwoo Jeong, Kun Chen, and Ming Wu. 2026. RovoDev Code Reviewer: A Large-Scale Online Evaluation of LLM-Based Code Review Automation at Atlassian. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '26)*. ACM, Rio de Janeiro, Brazil, 34–45. doi:10.1145/3786583.3786851

[10] Priyan Vaithilingam, Tianyi Zhang, and Elena Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI EA '22)*. ACM, New Orleans, LA, USA, 1–7. doi:10.1145/3491101.3519665

[11] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 36 (2024), 23 pages. doi:10.1145/3643762