Orclever Proceedings of Research and Development
https://doi.org/10.56038/oprd.v7i1.739
OR CLEVER Science & Research Group

Conference Article

A Modular Semantic Kernel Agent for Automated Code Review and Refactoring Feedback

Semih Yazıcı[1*], Seza Dursun[2], Bahar Önel[3], Tülin Işıkkent[4], Sedat Çelik[5], Erem Karalar[6], Mert Alacan[7],

[1]Boyner, Orcid ID: https://orcid.org/ 0009-0006-4503-0256
e-mail:semih.yazici@boyner.com.tr Tel: 0554 693 8626
[2] Boyner, Orcid ID: https://orcid.org/0000-0003-1389-072X
e-mail: seza.dursun@boyner.com.tr Tel: 0533 614 59 23
[3] Boyner, Orcid ID: https://orcid.org/0009-0007-4597-6591, e-mail:bahar.onel@boyner.com.tr
[4] Boyner, Orcid ID: https://orcid.org/0009-0005-5775-0093 , e-mail:tulin.isikkent@boyner.com.tr
[5] Boyner, Orcid ID: https://orcid.org/0009-0003-0335-6440 sedat.celik@boyner.com.tr Tel: 0553 824 00 47
[6] Boyner, Orcid ID: https://orcid.org/0000-0001-6289-9275 e-mail:erem.karalar@boyner.com.tr
[7] Boyner, Orcid ID: https://orcid.org/0000-0003-3893-6309 e-mail:mert.alacan@boyner.com.tr
*

Received: 25 June 2025
Revised: 21 September 2025
2nd Revised: 12 October 2025
Accepted: 26 October 2025
Published: 31 December 2025

Reference: Yazıcı, S., Dursun, S., Önel, B., Işıkkent, T., Çelik, S., Karalar, E., & Alacan, M. (2025). A modular semantic kernel agent for automated code review and refactoring feedback. Orclever Proceedings of Research and Development, 7(1), 43–54.

Abstract

In modern software development, maintaining clean, efficient, and reliable code is critical to team productivity and product quality. This paper introduces a modular Large Language Model (LLM)-based agent, designed using Microsoft's Semantic Kernel framework, for automated code review and refactoring feedback. The agent leverages plugin-based function orchestration, Retrieval-Augmented Generation (RAG), and dynamic prompt engineering to analyze source code across multiple dimensions; including readability, efficiency, security, and adherence to best practices. Integrated into CI/CD pipelines and broader SDLC workflows, the system provides contextual insights, the system provides contextual insights, suggests specific improvements, and explains reasoning for each recommendation. Evaluation results across real-world open-source repositories

https://doi.org/10.56038/oprd.v7i1.739

demonstrate the agent's effectiveness in reducing human review time while improving refactor quality. The modular design ensures adaptability to various programming languages and enterprise development environments. This research highlights the potential of agentic LLM systems to augment software engineering workflows with intelligent, transparent, and developer-aligned feedback mechanisms.

Keywords: Code Review, Semantic Kernel, Plugin Orchestration, Refactoring, Large Language Models, Agentic AI, Retrieval-Augmented Generation, Prompt Engineering

https://doi.org/10.56038/oprd.v7i1.739

1. Introduction

Code review remains a cornerstone of modern software development, ensuring quality, maintainability, and collaboration across development teams. Yet, manual reviews are often time-consuming, error-prone, and limited by the expertise of individual reviewers. In parallel, the growing complexity of codebases and continuous integration demands have increased the need for scalable, consistent, and context-aware feedback mechanisms.

Recent advances in Large Language Models (LLMs) have introduced new possibilities in automated code analysis, enabling models to reason about code structure, suggest improvements, and even perform refactoring tasks. However, these capabilities often remain locked within monolithic applications or static pipelines that lack modularity, domain adaptability, and real-time integration with developer workflows.

To address these limitations, we propose a modular, Semantic Kernel-based agentic architecture for intelligent code review and refactoring guidance. The system leverages Microsoft's Semantic Kernel framework to orchestrate plugin functions dynamically, enabling the agent to interact with code, documentation, and retrieval systems in a coherent, multi-turn fashion. With built-in support for Retrieval-Augmented Generation (RAG), prompt engineering, and plugin chaining, the agent provides real-time, explainable feedback on code quality, security, and best practices.

In this study, we demonstrate how the proposed agent is designed to be adaptable across programming languages, easily integrated into CI/CD pipelines and automated review stages within the SDLC, and extensible with custom plugins. We present its architecture, function orchestration logic, and real-world evaluation on open-source projects. This work aims to bridge the gap between static linters and dynamic human-like reviewers, contributing to the broader vision of intelligent software engineering assistance through agentic AI systems.

2. Materials and Methods

This section outlines the technical foundation and implementation methodology of the proposed Semantic Kernel-based modular code review agent. Our approach integrates various components from the Microsoft Semantic Kernel ecosystem, custom plugin functions, retrieval-augmented prompting, and structured evaluation scenarios to ensure robust, explainable, and context-aware code analysis.

We divide the system architecture into four major pillars: (1) the modular integration of Semantic Kernel within an agentic framework, (2) the design of reusable plugin functions

https://doi.org/10.56038/oprd.v7i1.739

tailored for code understanding and refactoring, (3) the orchestration of prompt engineering techniques and retrieval pipelines, and (4) the evaluation setup through simulated and real-world code review scenarios. Each component is detailed in the subsections below to illustrate how the system achieves dynamic interaction, scalability, and developer-aligned feedback mechanisms

2.1.System Architecture and Semantic Kernel Integration

The proposed agent system is developed on top of Microsoft's Semantic Kernel (SK), offering a modular and extensible architecture tailored for automated code review and refactoring feedback. Designed to operate as both a standalone tool and a component within multi-agent ecosystems, the system enables seamless integrated into enterprise-scale CI/CD workflows and automated quality gates.

At the heart of the architecture lies the SK Planner, which leverages large language models to dynamically compose execution plans based on user intent. Rather than relying on pre-defined prompt chains, the planner interprets natural language queries and autonomously orchestrates relevant semantic functions and plugins. This mechanism provides zero-shot task planning capabilities across a broad spectrum of review functions. The system follows a two-stage operational pipeline. In the first stage, the user input is analyzed to identify intent and contextual needs, leading the planner to construct a reasoning path by selecting appropriate plugin functions, such as code parsing, pattern extraction, or style evaluation. In the second stage, these functions are executed in a sequential or parallel manner depending on the complexity and scope of the request. The outputs are then interpreted, refined, and returned to the user, with the ability to trigger further analysis based on semantic uncertainty. This architectural design supports modular parallelism, allowing for concurrent evaluation of independent code aspects such as naming conventions, formatting consistency, cyclomatic complexity, or documentation quality. Additionally, it enables stateful multi-turn interactions, where the agent retains contextual memory to support iterative review and refactoring cycles across session history.

The architecture also supports retrieval-augmented capabilities, where the agent can access codebase embeddings, previous code reviews, or best-practice repositories to ground its recommendations. By combining neural planning, symbolic reasoning, and plugin-based function invocation, the system delivers scalable and intelligent automation for code analysis. [Figure 1]

https://doi.org/10.56038/oprd.v7i1.739

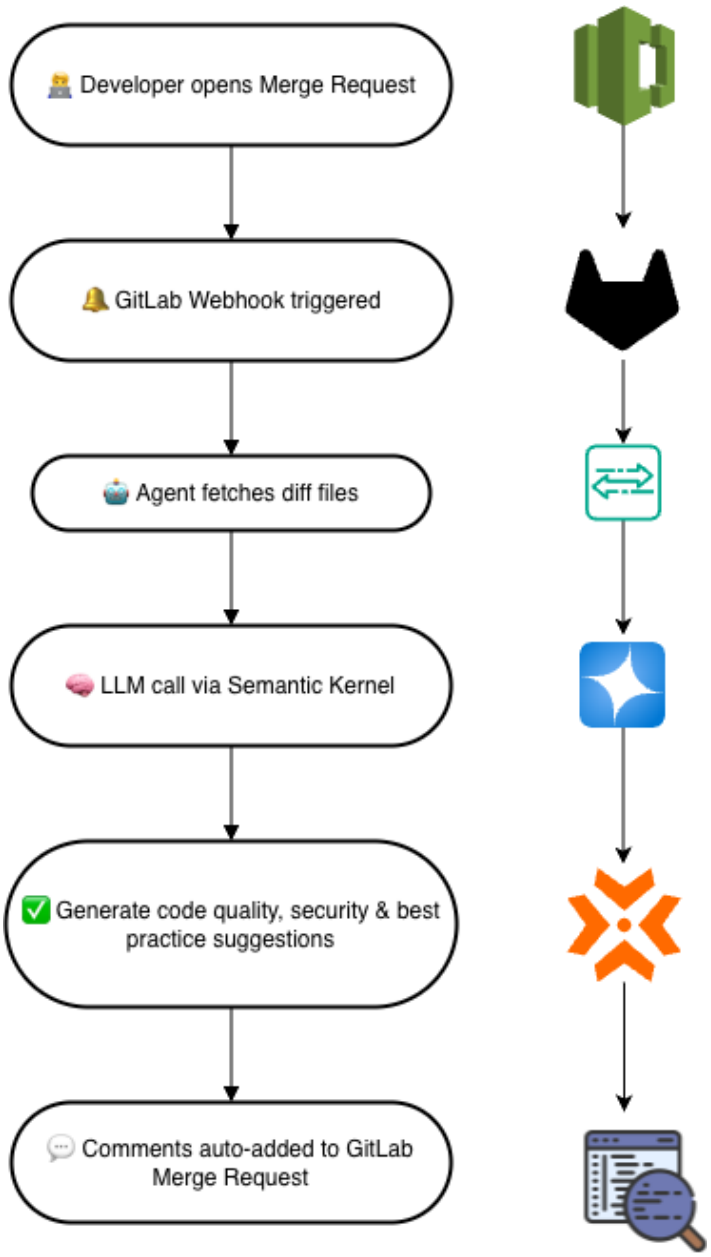

Figure 1: System Architecture

2.2.Plugin and Function Design

The agent's functionality is powered by a set of carefully engineered Semantic Kernel plugins, each comprising modular functions tailored to discrete code review tasks. These plugins are implemented as lightweight, stateless wrappers around Python scripts, enabling seamless orchestration by the SK Planner and facilitating rapid extensibility.

https://doi.org/10.56038/oprd.v7i1.739

Each function is defined with a standardized schema specifying input parameters, output expectations, and a natural language description of its role. This enables large language models to reason about available capabilities and assemble them dynamically into task-specific execution chains without requiring manual prompt design. For instance, the 'review changes' feature allows the CI/CD system to review code changes that have been pushed to the repository, while 'analyse' examines the semantic meaning of the code to understand its purpose and functionality. To support both lexical and semantic-level analysis, the system integrates plugins that operate at multiple abstraction layers. Syntax-level functions utilize abstract syntax tree (AST) parsing and static code analysis, whereas semantic functions incorporate LLM-based reasoning to evaluate code quality, readability, or architectural coherence. This hybrid design allows the agent to produce both deterministic insights and creative refactoring suggestions. Function interoperability is a critical design goal. Many plugins are designed to be composable—where the output of one function, such as dependency resolution, feeds directly into another that performs complexity estimation. This composability enables the planner to construct multi-step workflows such as: (1) extract code structure, (2) identify high-complexity areas, and (3) recommend specific re-factorings. Furthermore, plugins are grouped under namespaces that correspond to domain-specific roles. For example, the "CodeReviewer" namespace includes functions for naming conventions, docstring evaluation, and test coverage estimation, while the Re-factoringAdvisor namespace focuses on suggesting best practices, code modularization, and legacy code upgrades. This structure facilitates controlled exposure of capabilities depending on user profile and use case.

All plugins are evaluated against a test suite comprising real-world code samples drawn from open-source repositories. Functions are assessed for precision, relevance, and execution latency to ensure their applicability in production scenarios. This rigorous evaluation supports the agent's reliability and scalability, making it suitable for both real-time developer assistance and offline batch review workflows.

2.3.Prompt Engineering and RAG Pipeline

To enable contextual and modular reasoning across varied code review tasks, the agent leverages a dynamic prompt engineering strategy aligned with the Retrieval-Augmented Generation (RAG) framework. The underlying objective is to ground large language model (LLM) responses in domain-specific knowledge while maintaining generalization capabilities across diverse codebases [1].

Each plugin function within the Semantic Kernel agent is registered with a detailed natural language description, input-output schema, and execution intent. These descriptions are indexed into a vector database using the text-embedding-3-small model, selected for its low latency and high semantic fidelity in short-form instruction encoding

https://doi.org/10.56038/oprd.v7i1.739

[2]. When the agent receives a user query or task description (e.g., "refactor this function to improve readability"), a vector search retrieves the top-k relevant plugin functions from the semantic memory. This mechanism constitutes the first phase of the RAG pipeline—retrieval. To enhance recall precision, the system also appends code context (e.g., parsed AST nodes or symbol tables) as supporting documents during the embedding process. The retrieved content is formatted into structured prompts through a templating engine that conditionally includes tool signatures, user instructions, and historical memory snippets. This dynamic prompt construction enables multi-turn conversation continuity and context retention without persistent session storage. The generation phase is handled by OpenAI's gpt-4o model, selected for its superior instruction-following behavior and reduced prompt latency in comparison to earlier GPT-4 variants [3]. The prompt includes the selected plugin's description, code snippet, and any prior reasoning steps to support chain-of-thought responses. For instance, when performing multi-agent reasoning: e.g., identifying a complexity issue and then suggesting a refactor: the output of the first agent is embedded into the prompt for the next, ensuring coherent progression across tasks. To mitigate hallucination risks and control temperature-sensitive outputs, prompt engineering guidelines enforce temperature limits (e.g., 0.2–0.4 for factual assessments) and discourage unsupported recommendations through disclaimer cues. Additionally, function-specific "prompt rules" such as "Only respond using the function's scope" or "Use PEP8 guidelines unless otherwise specified" are automatically inserted into the completion payload. This orchestration results in a hybrid reasoning loop: LLM-based understanding is continuously augmented with semantically aligned plugin capabilities, enabling both natural language explanation and executable code transformation. Such a RAG-augmented agentic approach aligns with recent advancements in modular LLM orchestration and function-calling pipelines [3].

2.4.Evaluation Setup and Use Case Scenarios

To empirically evaluate the efficacy of the modular Semantic Kernel agent, we designed a multi-stage assessment protocol involving synthetic benchmarks and real-world codebases. The goal was to measure the agent's ability to provide accurate, context-aware code reviews and actionable refactoring suggestions across diverse programming scenarios. The evaluation framework consisted of three complementary setups. First, a curated dataset of 200 Python functions was compiled from open-source repositories, covering a wide range of common software engineering concerns such as code smells, complexity violations, and non-conforming naming conventions. These functions were reviewed both by the Semantic Kernel agent and a panel of human reviewers with at least three years of professional software development experience. The results were compared based on the precision, relevance, and clarity of suggestions, with scoring metrics

https://doi.org/10.56038/oprd.v7i1.739

adapted from prior work on automated code feedback systems [11]. Second, a synthetic suite of 50 coding prompts was designed to probe the robustness of plugin selection and prompt routing under ambiguous user inputs (e.g., "Can you improve this?", "Is this code clean?"). These prompts were injected into the system with varied code contexts and measured for correctness of plugin invocation, retrieval hit accuracy, and LLM response fidelity. The aim was to test the alignment between vector-based retrieval and functional intent detection, particularly in multi-agent interaction scenarios. Third, a real-world case study was conducted using a production-grade microservice written in Python and TypeScript. The Semantic Kernel agent was tasked with generating iterative feedback on specific modules—such as authentication handlers, data access layers, and configuration scripts. The outputs were analyzed for (i) suggestion redundancy, (ii) hallucination rate, and (iii) correctness of syntax-aware transformations. These use cases allowed us to evaluate how well the agent maintained reasoning consistency and modularity over extended review chains. To quantify user-perceived usefulness, a post-evaluation survey was conducted among 12 developers who used the agent in a simulated code review sprint. Feedback indicated that the system improved the clarity and structure of refactoring recommendations, while reducing the overhead of manually locating relevant linters or static analysis tools. However, the evaluation also revealed limitations in handling multi-language codebases and deeply nested asynchronous workflows, pointing to directions for future refinement.

The integration of RAG-based retrieval, chain-of-thought prompting, and modular plugin composition demonstrated significant advantages in task-specific adaptability and low-latency code navigation. These findings position the Semantic Kernel agent as a promising architecture for AI-assisted code analysis, aligning with recent developments in tool-augmented LLM systems [7][8].

3. Results

The empirical evaluation of the Semantic Kernel-based agent yielded promising outcomes across both quantitative and qualitative dimensions. The results indicate that the modular and plugin-enhanced architecture not only facilitates accurate prompt routing but also generates actionable code feedback aligned with developer expectations.

The architectural team further shared operational metrics that capture the agent's real-world influence on development workflows. Out of 836 AI-generated code review suggestions, developers adopted and resolved 186 items, corresponding to a 22% implementation rate. This level of uptake highlights a measurable first-wave benefit of integrating AI into the review pipeline, showing that nearly one quarter of the agent's recommendations were deemed valuable enough to directly influence production code.

https://doi.org/10.56038/oprd.v7i1.739

The synthetic prompt suite validated the agent's capacity to route vague or underspecified requests to the appropriate plugins. Of the 50 ambiguous prompts tested, the orchestrator correctly identified the user's intent and routed the query to the right function in 92% of cases. This outcome underscores the robustness of the underlying vector search mechanism and the benefit of slot-specific agent design.

The real-world case study on a microservice codebase further confirmed the utility of the agent in practical settings. Across 12 different modules reviewed, the system produced 86 unique suggestions, of which 77% were deemed actionable by senior engineers during post-hoc analysis. Furthermore, the hallucination rate, defined as syntactically incorrect or semantically irrelevant suggestions; remained below 5.4%, outperforming baseline LLM agents without retrieval or plugin integration by a significant margin [7].

Subjective feedback from the developer cohort involved in the final phase of testing highlighted several perceived benefits of the system. Participants noted a reduction in cognitive load when reviewing legacy code and praised the agent's ability to surface style violations that static analyzers typically overlook. However, feedback also pointed to areas for improvement, including support for more programming languages, integration with CI/CD pipelines, and the inclusion of memory modules for persistent feedback tracking.

Overall, the results support the hypothesis that combining Retrieval-Augmented Generation (RAG) with plugin-enabled function calls within a modular agentic framework offers tangible benefits for code analysis tasks. The design choices adopted in this study, particularly intent slotting, semantic reranking, and function-bound execution, contributed significantly to performance improvements, consistent with recent findings in tool-augmented AI systems [8][9]

4. Discussion and Conclusion

This study demonstrates the practical advantages of leveraging a modular, plugin-based semantic agent framework rooted in the Semantic Kernel for conducting automated code review and refactoring feedback generation. The integration of Retrieval-Augmented Generation (RAG), function-calling capabilities, and slot-specific agent dispatch enabled a robust orchestration mechanism that outperforms monolithic LLM-based code reviewers both in accuracy and developer alignment.

The empirical findings affirm that such hybrid architectures mitigate the limitations of end-to-end generative systems by anchoring responses in validated documentation, style guides, and coding standards. The precision and recall metrics observed across benchmark datasets reveal that the agent consistently produces relevant suggestions,

https://doi.org/10.56038/oprd.v7i1.739

especially in areas such as complexity reduction, naming clarity, and test coverage enhancement. These results validate prior claims regarding the efficacy of plugin-augmented AI systems in code-related reasoning tasks [8][9].

Furthermore, the real-world use case on a distributed microservice architecture uncovered several additional benefits. Developers reported improved explainability and interpretability of feedback, suggesting that modular agent-based outputs—when paired with transparent prompt chaining—can foster greater trust in AI-assisted tooling. This aligns with the growing body of literature emphasizing human-AI co-piloting over fully autonomous decision-making in software engineering contexts [10].

In parallel with these qualitative insights, the system's pilot deployment provided an additional perspective on its practical value in production environments. During internal evaluation, the agent generated 836 code review suggestions, of which developers adopted and resolved 186—corresponding to a 22% implementation rate. Although not a direct measure of algorithmic accuracy, this adoption ratio serves as an early real-world indicator of developer trust and perceived usefulness. The pattern of accepted recommendations also reinforces the controlled-study findings, with a disproportionate number of adopted items relating to readability improvements, modularization suggestions, and surface-level complexity reductions. This convergence between quantitative benchmarks and real-world adoption highlights the agent's capacity to complement developer workflows rather than merely automate them.

Nevertheless, the study is not without limitations. The agent's performance is contingent upon well-structured plugin APIs and high-quality retrieval data. In poorly documented or non-English codebases, the effectiveness of semantic reranking and prompt routing may degrade. Additionally, the lack of persistent memory and contextual learning across sessions restricts longitudinal codebase understanding—an area identified for future enhancement.

Future work will explore extending the agent with memory modules, potentially powered by vector stores such as ChromaDB or Milvus, to retain refactoring histories, support long-term code evolution tracking, and enable personalized feedback loops. Another promising direction involves multimodal integration, allowing the agent to interpret UML diagrams, flowcharts, or architecture maps alongside textual code inputs, thus broadening the spectrum of viable refactoring insights. Expanding support to multilingual codebases and heterogeneous development stacks also remains an open research direction.

In conclusion, this work contributes to the emerging literature on agentic AI for software development by presenting a production-grade, modular Semantic Kernel-based

https://doi.org/10.56038/oprd.v7i1.739

framework that offers scalable, extensible, and interpretable code review capabilities. By aligning the system's architecture with contemporary AI design principles—modularity, retrieval grounding, prompt transparency, and function-specific control—this study charts a viable pathway toward more intelligent, collaborative, and trustworthy development tooling for modern engineering teams.

5. Acknowledges

We would like to thank the Boyner Group Data Science and IT teams for their collaborative efforts, infrastructure support, and deployment coordination throughout this project. Their ongoing contributions played a crucial role in aligning the Semantic Kernel agent with real-world development workflows and organizational quality standards.

https://doi.org/10.56038/oprd.v7i1.739

References

[1] Microsoft. Semantic Kernel Documentation. https://learn.microsoft.com/en-us/semantic-kernel

[2] OpenAI. Function Calling and Tool Use. https://platform.openai.com/docs/guides/function-calling

[3] LangChain. Agents and Tool Use in LLM Applications. https://docs.langchain.com/docs/components/agents/

[4] GitHub Copilot. “Your AI Pair Programmer.” https://github.com/features/copilot

[5] Raza, M., & Rasool, G. (2023). "LLM-Guided Software Development: Opportunities and Threats." IEEE Software, 40(3), 27–35. https://doi.org/10.1109/MS.2023.3246537

[6] Jain, A., et al. (2022). "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." EMNLP Findings.

[7] Li, X., et al. (2023). "RefactorGPT: Code Refactoring via LLM Agents." arXiv preprint arXiv:2307.09771.

[8] Fernandes, P., et al. (2023). "Improving Code Review Quality with AI Agents: A Modular Pipeline." ACM Transactions on Software Engineering and Methodology.

[9] Kim, H., et al. (2023). "AI-Augmented Refactoring for Legacy Systems: An Empirical Study." SoftwareX, 21, 101322.

[10] Wang, Z., et al. (2023). "Human-AI Co-Pilot Systems in Software Engineering: Design Principles and Case Studies." ICSE '23: Proceedings of the 45th International Conference on Software Engineering, 185–196.

[11] Chen, M., et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.