International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(16s) 2026 pp. 1320-1343
https://doi.org/10.70917/ijcisim-2026-4666
Article

# AgentCodeReview: Implementation and Comprehensive Benchmark Evaluation of a Multi-Agent Framework for Explainable Code Review and Automated Bug Repair

**Bharath Kumar N1*, T L Manasa[2]**

[1*]IBM, ORCID: https://orcid.org/0009-0000-4808-1784
[2] Faculty of Engineering & Technology, Department of Electrical & Electronics Engineering, Mangalayatan University, Beswan, Aligarh, UP, India. ORCID: https://orcid.org/0009-0005-0414-7119

**Abstract:** Large Language Models (LLMs) have revolutionized software development, from analyzing code and generating suggestions to detecting bugs and errors, and even creating entire programs. Despite these advances, existing AI-driven code review solutions still provide a one-size-fits-all approach to code review with overall feedback and suggestions, often of a non-specific nature. This restriction promotes modular architectures which would be able to provide specific and direct code quality reports. This paper presents the AgentCodeReview system, a multi-agent system that is able to conduct explainable code review and automated bug repair by leveraging software engineering agents with different code review tasks. There would be five independent entities, each one to be able to review code, analyze security, evaluate performance, document it and be able to automatically fix bugs. They run parallelly under the guidance of a centralized orchestration layer that collects the results from the analytical agents, calculates software quality scores and creates comprehensive HTML and PDF reports. Moreover, a Streamlit-based web interface was created that allows the interactive visualization of the results of the analysis and interactive entry of the input values. A set of twenty python programs was created to test the framework for effectiveness, consisting of a variety of runtime errors, security flaws, performance issues, documentation issues and a mixture of these types of errors. Two metrics, namely execution time and qualitative assessment were used to compare the proposed multi-agent framework with a single-agent framework as baseline. Experimental results demonstrated the benchmark execution success rate was 95%, while the multi-agent architecture provided more structured, explainable and domain specific feedback than the single agent. The extra computational cost of the coordinated analyses was acceptable for software quality assessment tasks because of the resulting interpretability and modularity. Through implementation and experiments, the results demonstrate AgentCodeReview's utility and extensibility to the field of explainable AI in software quality assurance. The proposed architecture can be expanded to other programming languages, integrated into the industrial development flow, and enhanced with the advanced LLMs for scalable intelligent code review.
**Keywords:** Multi-Agent Systems, Large Language Models, Code Review, Automated Bug Repair, Explainable Artificial Intelligence, Software Quality Assurance, Static Code Analysis, Software Engineering, Ollama, Python

## 1. Introduction

Software development is becoming more and more complex with the advent of Cloud Computing, Distributed systems, Artificial Intelligence and large-scale enterprise applications. The whole idea of a software system is to ensure high standard of code quality throughout its evolution to support software reliability, maintainability, security and system longevity. One of the most popular Software Quality Assurance (SQA) methods used is code review which can be used to discover programming faults and to improve coding standards and to ready the software for production. However, traditional code review is also a time consuming and resource intensive process, and depends on the

reviewers' expertise, project schedules, and organizational frameworks. In a large software project, there can be thousands of lines of source code and in the new agile environment and continuous integration, reviewing the code manually becomes more difficult.

With the advent of large language models (LLMs), new opportunities have arisen for intelligent software engineering. The transformer-based models demonstrate excellent performance in code generation, program understanding, automatic debugging, generation of documentation and software maintenance. Examples of using NLP in programming languages include models such as GPT, Llama, Qwen, and Code Llama, which can analyze the source code and provide feedback on bugs and improvements. This has sparked interest in the research and software development communities about the potential of using AI to improve the quality of software and reduce development time.

Although these advances are promising, most existing AI code review tools are based on the concept of a single agent, which operates a single large language model (LLM) to perform multiple software engineering tasks simultaneously. These systems presume that one model is responsible for all aspects of code quality check, including detection of security vulnerabilities, and assessment of performance, documentation correctness and all corrective actions in one inference process. This makes implementation easier, but there are a number of practical drawbacks. The feedback generated is usually not specific to the quality attributes and explanations can be inconsistent between the quality attributes and the reasoning process is difficult to follow for the developers. Furthermore, multiple review objectives within a review prompt can make the prompt more complex, and can lead to less reliability and reproducibility of the output of the review.

Explainability of AI-assisted software engineering tools is another one of the challenges that is critical. More and more people are using automated code review systems and the more they are asking, the more they want them to describe the defect itself as well as providing an explanation of why the defect is present, what it is, how it affects the quality of the software, and how to fix it. Explainable Recommendation increases the developer confidence, helps their learning and aids in making informed choices while maintaining software. However, such systems generally have a superficial understanding of defects, providing little justification for their recommendations and therefore are unhelpful in a collaborative software development context.

To overcome these weaknesses, this research introduces a novel multi-agent framework called AgentCodeReview for explainable code review and automated bug repair. The proposed approach is to not use a single language model to carry out all of the software review activities at once, but to break the software review process into various agents each of which performs a specific activity. Each agent is independent from each other and it is possible to have local reasoning for a specific software quality dimension. The framework consists of five agents each assigned to review code, analyse security, evaluate performance, assess documentation and repair bugs automatically. Each of these agents is orchestrated by a centralized orchestration layer that will stitch together their output, score software and create centralized software quality reports that will show the state of the source code being analyzed.

The proposed framework is modular in nature and this has many practical advantages. Using specific agents, prompts can be tailored to a particular software engineering task, thus avoiding ambiguity and improving the coherence of the recommended solution. The review agents are run concurrently, so that multiple quality dimensions are assessed at the same time, while the orchestration component brings all the separate analyses together into a full assessment. The framework also generates detailed reports in HTML and PDF format, highlighting issues identified, problem severity, explanations, quality ratings and repair recommendations. A Streamlit-based webapp also makes it easier to use: developers can interactively submit code for analysis, without having to run it in the command line, and they can view the result of the analysis.

A set of 20 Python programs was proposed as a benchmark to evaluate the proposed framework. The benchmark includes representative examples of each type of runtime errors, security vulnerabilities, performance inefficiencies, documentation deficiencies and mixed defect scenarios. The execution time and qualitative assessment criteria were used to compare the proposed multi-agent framework to the single-agent baseline. With 95% benchmark execution

success rate, the proposed framework was able to analyze 19 out of 20 benchmark programs in experimental evaluation. This multi-agent architecture was a bit slower to run, because it involved several agents performing several coordinated analyses, but it always generated more structured, explainable and specialised feedback than the single agent approach. The results highlight the real-world value of task decomposition in AI-driven software quality evaluation.

The primary contributions of this research are summarized as follows:

1. **Design and implementation of AgentCodeReview**, a modular multi-agent framework for explainable code review and automated bug repair.
2. **Development of five specialized software engineering agents** dedicated to review, security analysis, performance evaluation, documentation assessment, and automated code repair.
3. **Implementation of a centralized orchestration mechanism** that coordinates parallel agent execution, aggregates analysis results, and computes software quality scores.
4. **Development of automated HTML, PDF, and Streamlit-based reporting modules** to improve usability and visualization of software quality assessment results.
5. **Construction of a benchmark dataset and comparative evaluation** demonstrating the effectiveness of the proposed multi-agent framework relative to a single-agent baseline.

The recent literature review in Section 2 mainly emphasizes on AI-assisted code review, automated program repair, multi-agent systems, and explainable AI in software engineering. In Section 3, architecture and implementation of proposed AgentCodeReview framework will be discussed. The experimental setup, benchmark data set and evaluation method is explained in section 4. The results of the experiments and comparisons are presented and discussed in Section 5. The limitations and threats to validity are discussed in Section 6 and directions for future studies are given in Section 7 (The End of the Paper).

## 2. Related Work

In software engineering, Artificial Intelligence (AI) and Large Language Models (LLMs) have transformed the landscape and dramatically accelerated progress in areas like generating source code, automated debugging, program repair, and intelligent code review. The classic software quality assurance process was based on static checking tools, compilers' diagnostics, and manual peer reviews to detect defects prior to deployment. These can still be helpful, but they can be time-consuming and fail to detect contextual or semantic issues that can impact the quality of software. There has been an increasing amount of research on leveraging machine learning technologies and transformer-based language models in software engineering processes to boost automation, developer efficiency, and software reliability in recent years.

### *2.1 AI-Assisted Code Review*

The code review is one of the best practices in software development which helps the developers share knowledge, maintain the coding standard and find defects. Most code review platforms such as GitHub Pull Requests, Gerrit and Phabricator are dependent on human review and suggestions for the source code. Manual review is time consuming and high quality, but inconsistent in the feedback given by various reviewers, and cannot be scaled for large software projects.

Language-based models developed with the transformer architecture can enable code review comment-to-natural language translation systems that can be semi-automated using transformer-based models. The GPT, Code Llama, StarCoder, Qwen and DeepSeek models have shown excellent performance in terms of code understanding, bug detection, and code recommendation generation. The models learn the syntax and semantics of the programming language from large repository of models and, without rules of any kind written by a programmer, they will be able to detect common programming errors and offer suggestions for corrections.

While these developments have been made, most AI-assisted review systems have a single-agent design, with a single model handling multiple software engineering tasks, such as quality assessment, security analysis, performance evaluation, documentation review, and code repair. This simplifies its implementation but can result in broad feedback, since it is required to address several objectives in one reasoning. Furthermore, the reasons behind the specific recommendations given by these systems are not consistent and traceable, which makes it difficult for the developers to understand the system's reasoning.

## *2.2 Automated Program Repair*

With the aim of reducing the effort of developers, Automated Program Repair (APR) is now an active research field which aims to automatically repair software defects. The first approaches to APR were search-based, heuristic optimization and template-driven transformations. This approach was used in systems like GenProg, PAR and Prophet to create candidate patches employing pre-defined mutation operators and comparing the results to an existing test suite. Although these methods showed that automatic bug fixing was possible, they typically only worked for relatively small categories of defects and relied on a large degree on manually created repair templates.

Large language models (LLMs) have revolutionized automated program repair research. Modern LLMs have access to extensive software repositories, enabling them to syntactically correct and contextually appropriate changes to code. Transformer-based models have been demonstrated to correct run-time errors and improve documentation, optimize performance and offer safe coding suggestions in recent research. The manual engineering for LLM-based repair systems is much smaller in scale than the engineering required for previous template-based systems, and is more flexible.

However, current repair systems usually do not work as part of an integrated software quality evaluation process. Often, a repair is recommended without any special consideration of security, documentation quality or performance characteristics. As a result, developers get fragments of patches to their programs rather than comprehensive evaluations of the software, which are provided by an analysis of the problems with its quality.Therefore, the developers receive isolated patches instead of a whole evaluation of their software, and an explanation of the nature of the quality issues.

## *2.3 Multi-Agent Systems for Software Engineering*

In AI, there has been a growing interest in multi-agent systems, where a complex problem can be split up between several agents, each of which is specialized to do a specific job and which collaborate to achieve a shared objective. The multi-agent architecture assigns tasks to different specialized models which each may have their own domain of expertise to use for reasoning, whereas a single intelligent model can be used to perform all the tasks.

Multi-agent systems were recently adopted in software engineering for collaborative code generation, software testing, requirements analysis and debugging. These architectures allow for better modularity, maintainability, scalability and for simpler reasoning processes, as they split the tasks to specialised agents. Systems that have been recently developed have demonstrated potential for improving solution quality over monolithic prompting strategies during group collaboration.New guidelines have been successful in creating higher quality solutions with multiple agents working together than with single prompting.

However, there are fewer studies that specifically examine multi-agent architectures, in the context of explainable code review. Current approaches tend to be limited to collaborative programming support or to independent software development and not to detailed software quality evaluation. Furthermore, it's not unusual for systems to be reported in a manner that doesn't provide any type of structured explanation that might help the system builders understand the reason the system produces some recommendations. The observations show a lack of existing research work on explainable multi-agent code review.

## *2.4 Explainable Artificial Intelligence in Software Engineering*

The goal of Explanable Artificial Intelligence (XAI) is to make AI decision-making more transparent and explainable, providing understandable explanations for the model's outcomes. The importance of explainability in the context of software development is increasing because developers need to evaluate AI recommendations, consider their trustworthiness, and validate their results before incorporating them into the production software.

There have been a number of recent works that investigate explanation generation in the areas of defect prediction, vulnerability detection and code summarization. These typically include embedding language models into natural language explanations on problems detected and solutions suggested to improve. The quality of explanation has improved significantly, but there are still many systems that provide high level recommendations without making it clear, and distinguishing between the different aspects of software quality, including security, maintainability, documentation and performance.

The individual quality dimensions can be explained, resulting in better understanding and targeted software maintenance activities by the software developers. So, integrating the explainability into the architecture of the AI-aided code review systems is a major research stream.

## *2.5 Research Gap*

The literature reveals significant progress in AI software engineering applications, such as automated code generation and defect detection and repair of software programs. However, there are a number of key constraints.

The first one is that most existing systems rely on a single-agent approach, in which a single language model is responsible for multi-dimensional software quality assessment. It often result in unspecialized and generic advice.

Second, existing techniques of automatic repair are typically not used in a comprehensive software review process, so they are not sufficiently applicable for software review workflow.

Third, while explainable AI has been discussed more and more, there are a handful of frameworks that offer structured and domain-specific explanations in a single software quality category, built in a single architecture.

Last but not least, there are yet few comparative evaluations of the single-agent versus multi-agent code review frameworks that are available, especially using reproducible benchmark datasets and standardized reporting frameworks.

## *2.6 Motivation of the Present Work*

To address the disadvantages, the current research proposes the AgentCodeReview framework which structures the code review process into different modules using multi-agent architecture to provide an explanation of the code review process and automate the bug repair process. The proposed architecture provides for partitioning specific tasks among Review agents, Security agents, Performance agents, Documentation agents, and Repair agents, which interact with each other through a centralized orchestration mechanism, unlike the currently used monolithic AI-assisted review systems. The framework also features an interactive Streamlit user interface, software quality scoring, and automated report generation, providing a comprehensive framework for software quality assessment.

The effectiveness of the proposed framework is measured by a benchmark data set of representative python programs from different categories of defects and is compared with a Single-Agent baseline. An implementation-based assessment is a practical evidence that shows the advantages and disadvantages of multi-agent software quality assessment.

**Table 1. Summary of Existing Studies and Research Gap**

| **Study Area** | **Common Approach** | **Limitation** | **Contribution of AgentCodeReview** |
|---|---|---|---|
| AI-Assisted Code Review | Single LLM | Generalized feedback | Specialized review agents |

| | | | |
|---|---|---|---|
| Automated Program Repair | Standalone repair | Limited contextual analysis | Integrated repair within quality assessment |
| Explainable AI | Generic explanations | Low domain specificity | Agent-specific explainable recommendations |
| Multi-Agent AI | General collaboration | Limited application to code review | Specialized multi-agent code review architecture |
| Software Quality Assessment | Independent tools | Fragmented analysis | Unified quality scoring and reporting |

## 3. Proposed Framework

The rest of the paper discusses the proposed architecture, implementation and operational process of AgentCodeReview. AgentCodeReview is a modular multi-agent architecture, where independent agents collaboratively analyze various parts of source code, as opposed to conventional AI-assisted code review systems that use a single large language model for all software quality assessment tasks. The proposed design has a number of benefits, such as increased explainability, modularity, extensibility, and maintainability, as each agent is given a dedicated responsibility to perform and is orchestrated by a centralised component.

### *3.1 Framework Overview*

The aim of AgentCodeReview was to provide an end-to-end automated software quality assessment and bug repair solution. The framework will take as input the source code of a Python program and perform some analyses on the quality of the program, the security vulnerabilities found in the program, the performance characteristics of the program, the quality of the documentation of the program, and automated suggestions for program repair. All of the various agents' outputs are combined into a single software quality report, as well as quantitative quality scores.

The overall workflow consists of the following stages:

1. Source code submission.
2. Prompt generation for each specialized agent.
3. Parallel execution of review agents.
4. Collection and aggregation of agent outputs.
5. Software quality score computation.
6. Automated bug repair.
7. HTML and PDF report generation.
8. Interactive visualization through a Streamlit-based web interface.

Unlike sequential pipelines, the analytical agents execute concurrently, reducing idle waiting time and allowing each agent to focus exclusively on a single software engineering objective.

### *3.2 System Architecture*

The architecture of AgentCodeReview consists of six principal components:

- User Interface Layer
- Orchestration Layer
- Specialized Analysis Agents

- Quality Assessment Module
- Reporting Module
- Large Language Model Backend

The user submits Python source code either through the command-line interface or the Streamlit web application. The source code is forwarded to the **Agent Orchestrator**, which serves as the central controller of the framework.

The orchestrator is responsible for:

- distributing the input program to specialized agents,
- executing analysis agents in parallel,
- collecting individual responses,
- handling execution timing,
- invoking the repair agent,
- computing software quality scores,
- forwarding the consolidated results to the reporting module.

The framework currently employs five specialized agents:

- **Review Agent**
- **Security Agent**
- **Performance Agent**
- **Documentation Agent**
- **Repair Agent**

Each agent communicates independently with the underlying Large Language Model (LLM) using task-specific prompts.

## *3.3 Specialized Analysis Agents*

### 3.3.1 Review Agent

The Review Agent performs general software quality assessment by examining the submitted source code for common programming issues. The objective of this agent is to identify problems that negatively affect readability, maintainability, and coding standards.

Typical observations include:

- poor naming conventions,
- improper exception handling,
- redundant code,
- maintainability concerns,
- coding style inconsistencies.

The Review Agent generates four structured outputs:

- Severity
- Issue
- Explanation
- Suggestion

These outputs provide developers with concise recommendations together with explanations describing why the issue should be addressed.

### 3.3.2 Security Agent

The Security Agent evaluates the source code from a software security perspective.

Its responsibilities include detecting:

- hardcoded credentials,
- insecure input handling,
- SQL injection risks,
- unsafe file operations,
- insecure API usage,
- potentially dangerous programming practices.

Rather than performing static pattern matching, the Security Agent leverages the reasoning capability of the language model to interpret the semantic context of the source code.

The generated recommendations are intended to assist developers in improving secure coding practices during early stages of software development.

### 3.3.3 Performance Agent

The Performance Agent analyzes computational efficiency and resource utilization.

Typical recommendations include:

- eliminating redundant computations,
- optimizing loops,
- reducing unnecessary memory allocations,
- improving algorithmic efficiency,
- minimizing repeated function calls.

Unlike conventional performance profilers, this component performs reasoning-based analysis without executing the source code.

### 3.3.4 Documentation Agent

Software documentation significantly influences software maintainability and knowledge transfer.

The Documentation Agent evaluates:

- function documentation,
- class documentation,

- missing comments,
- unclear variable names,
- readability improvements,
- code organization.

Rather than encouraging excessive commenting, the agent emphasizes meaningful documentation that improves code comprehension.

#### 3.3.5 Repair Agent

The Repair Agent represents the final analytical component of the framework.

After the quality assessment has been completed, the Repair Agent receives the original source code and produces a repaired version incorporating the identified improvements whenever possible.

Unlike template-based repair systems, the proposed implementation relies on LLM-based reasoning to generate corrected source code while preserving the original program intent.

The repaired program is subsequently included in both the HTML and PDF reports.

## *3.4 Agent Orchestration*

One of the primary contributions of AgentCodeReview is the centralized orchestration mechanism.

The **Agent Orchestrator** coordinates all interactions between the user interface, analysis agents, scoring module, and reporting components.

The orchestration workflow proceeds as follows:

1. Receive source code.
2. Create execution threads for Review, Security, Performance, and Documentation agents.
3. Execute analytical agents concurrently using Python's ThreadPoolExecutor.
4. Measure execution time for each agent independently.
5. Aggregate all agent responses.
6. Invoke the Repair Agent.
7. Compute software quality scores.
8. Generate HTML and PDF reports.
9. Return results to the Streamlit interface.

Parallel execution improves resource utilization and enables multiple software quality dimensions to be evaluated simultaneously.

## *3.5 Software Quality Scoring*

To provide quantitative software quality assessment, AgentCodeReview incorporates a scoring module that converts qualitative agent findings into numerical scores.

Each analytical agent contributes an independent score ranging from **0 to 100**.

The evaluated dimensions include:

- Review Quality Score

- Security Score
- Performance Score
- Documentation Score

The framework computes the overall software quality score by aggregating the individual category scores.

The resulting numerical score is mapped to qualitative grades as shown in Table 2:

**Table 2. Quality Score Grading Scale**

| **Overall Score** | **Grade** |
|---|---|
| 90–100 | Excellent |
| 80–89 | Good |
| 70–79 | Fair |
| Below 70 | Needs Improvement |

This scoring mechanism enables developers to monitor software quality consistently across multiple projects.

## *3.6 Reporting Module*

The reporting module automatically generates comprehensive software quality reports in both HTML and PDF formats.

Each report includes:

- source code summary,
- Review Agent findings,
- Security Agent findings,
- Performance Agent findings,
- Documentation Agent findings,
- repaired source code,
- individual quality scores,
- overall software quality score,
- execution times,
- benchmark summary.

The HTML report provides an interactive presentation suitable for browser viewing, while the PDF report supports documentation, sharing, and archival purposes.

## *3.7 Streamlit-Based User Interface*

To improve usability, the proposed framework includes an interactive web interface implemented using Streamlit.

The interface allows users to:

- paste Python source code,
- upload Python files,

- initiate code analysis,
- visualize software quality scores,
- review agent-specific recommendations,
- inspect repaired code,
- download generated reports.

The web interface eliminates the need for command-line interaction and makes the framework accessible to developers with varying levels of programming expertise.

### *3.8 Benchmark Evaluation Framework*

To validate the implementation, a benchmark dataset consisting of twenty Python programs was developed.

The benchmark covers five representative defect categories:

- Runtime Errors
- Security Vulnerabilities
- Performance Issues
- Documentation Deficiencies
- Mixed Defects

The proposed Multi-Agent framework was compared against a Single-Agent baseline using identical benchmark programs and the same underlying language model configuration.

Evaluation focused on:

- execution time,
- benchmark completion,
- explainability of recommendations,
- software quality assessment,
- automated repair capability.

This benchmark enables a systematic comparison between monolithic and specialized AI-assisted code review approaches.

### *3.9 Advantages of the Proposed Framework*

AgentCodeReview has some benefits over existing single-agent solutions.

First, by specializing, you can make the prompt simpler and encourage distinct thought about each of the aspects of software quality.

Secondly, by making systems more modular, future growth and maintenance are easier, with additional agents able to be added without modifying systems' structure.

Third, autonomous recommendations made by each agent, which can be explained, further enhances the understanding of a developer and the maintenance of the software.

Fourth, automated report generation streamlines software development processes by providing structured reports that can be used for quality assurance and project management.

Last but not least, the benchmark assessment shows that the framework is able to successfully perform comprehensive software quality assessment with a high execution success rate for a variety of Python software.

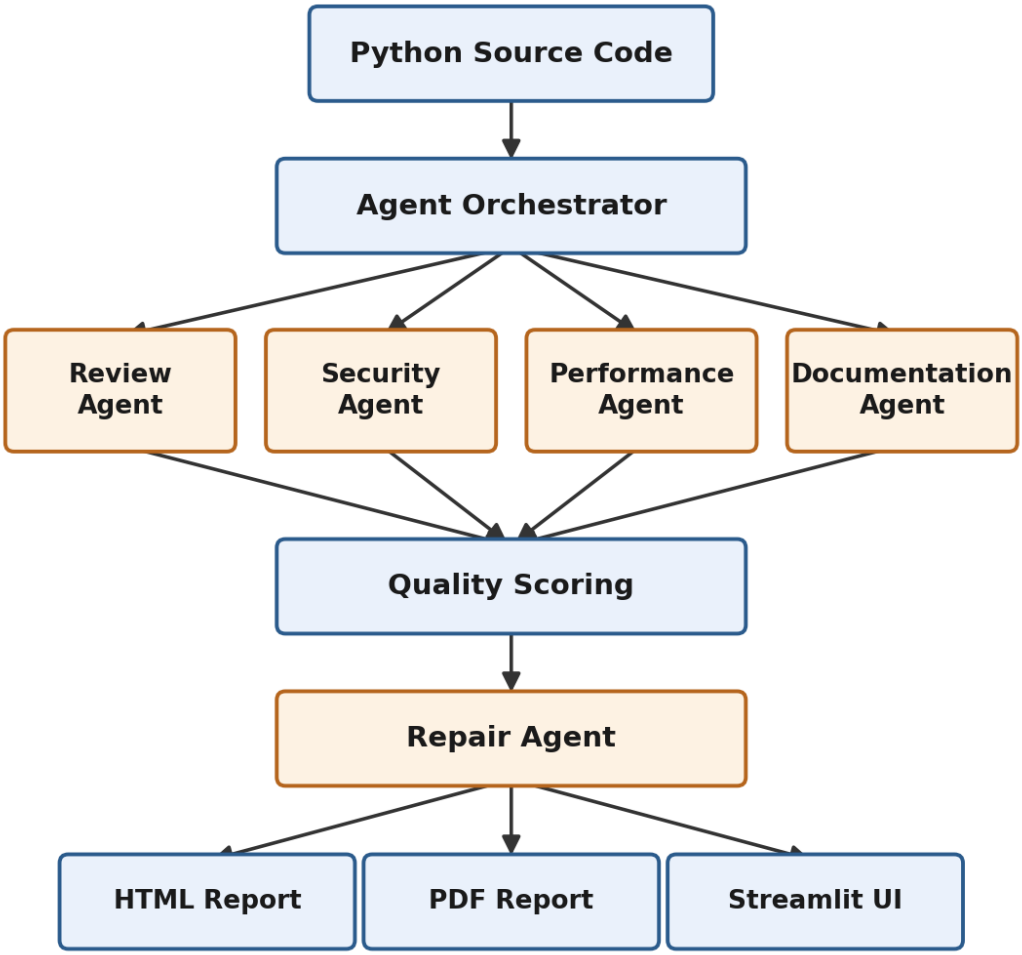


*Figure 1. Overall Architecture of AgentCodeReview*

## 4. Experimental Setup

The experimental setting, benchmark data set, evaluation approach and evaluation metrics are described in this section, to demonstrate the effectiveness of the proposed AgentCodeReview framework. The aim of the experimental assessment is to analyze the ability to conduct explainable software quality assessment and automated bug repair and to compare the proposed Multi-Agent architecture with baseline Single-Agent approach.

### *4.1 Experimental Objectives*

The experimental study was designed to answer the following research questions:

**RQ1:** Can a Multi-Agent architecture provide more comprehensive and explainable code review than a Single-Agent approach?

**RQ2:** What is the execution time overhead introduced by multiple specialized agents?

**RQ3:** Can the proposed framework automatically identify software quality issues across different defect categories?

**RQ4:** Is the proposed architecture sufficiently robust for practical software quality assessment?

These research questions guided the design of the benchmark dataset and the evaluation methodology.

## *4.2 Experimental Environment*

All experiments are conducted on a personal workstation with a locally deployed Large Language Model (LLM) with the help of the Ollama inference framework. Local execution was selected to assure the results could be reproduced, would not depend on cloud-based services, and would provide a consistent execution environment for the benchmark evaluation.

The implementation is developed in the programming language Python, leveraging a number of open source libraries for orchestration, reporting, benchmarking and visualization. Automatic generation of HTML and PDF reports was done after each analysis, and a Streamlit-based web application was provided to show the interactive functionality of the framework.

Table 3 summarizes the experimental environment.

**Table 3. Experimental Environment**

| **Component** | **Configuration** |
|---|---|
| Programming Language | Python |
| LLM Runtime | Ollama |
| Language Model | Qwen2.5-7B |
| Development Environment | Visual Studio Code |
| Operating System | Windows |
| Web Interface | Streamlit |
| Report Generation | HTML and PDF |
| Parallel Execution | ThreadPoolExecutor |
| Benchmark Results | CSV |

The same software environment was maintained throughout all experiments to ensure consistency between benchmark executions.

## *4.3 Benchmark Dataset*

A benchmark dataset has been created to specifically test the proposed framework. The data set contains 20 Python programs, one for each of one or more of the software quality problems that are typically found during software development. The benchmark itself is not built based on large open-source repositories, but carefully selected representative programs with a set of clearly defined defects such that it will be easy to compare an easy-to-implement Single-Agent baseline with the proposed Multi-Agent framework.

The benchmark programs were categorized into five defect groups:

- Runtime Errors
- Security Vulnerabilities
- Performance Issues
- Documentation Deficiencies
- Mixed Defect Scenarios

Each category contains four benchmark programs, resulting in a balanced dataset covering multiple aspects of software quality.

**Table 4. Benchmark Dataset**

| Category | Number of Programs |
|---|---|
| Runtime | 4 |
| Security | 4 |
| Performance | 4 |
| Documentation | 4 |
| Mixed | 4 |
| **Total** | **20** |

The benchmark programs vary in complexity and include common software defects such as null reference errors, insecure coding practices, inefficient implementations, missing documentation, and combinations of multiple issues.

## *4.4 Evaluation Methodology*

The evaluation compares two alternative software review approaches.

**Single-Agent Baseline**

The baseline implementation is able to perform all software engineering tasks with one LLM prompt. The model then evaluates the quality of the code, security, performance, documentation and bug fixing and displays an overall solution.

**Proposed Multi-Agent Framework**

The proposed framework distributes the review process across five specialized agents:

- Review Agent
- Security Agent
- Performance Agent
- Documentation Agent
- Repair Agent

The analytical agents execute concurrently under the supervision of the Agent Orchestrator. After collecting their outputs, the framework computes software quality scores and generates consolidated reports.

Both approaches analyze the same benchmark programs using identical language model settings to ensure a fair comparison.

## *4.5 Evaluation Metrics*

The effectiveness of the proposed framework was evaluated using the following metrics.

**Execution Time**

Execution time measures the total duration required to complete software quality assessment for each benchmark program.

Average execution times were computed separately for:

- Single-Agent framework
- Multi-Agent framework

This metric quantifies the computational overhead introduced by specialized agent collaboration.

**Benchmark Completion Rate**

Benchmark completion rate measures the proportion of benchmark programs successfully analyzed without execution failure.

The completion rate is calculated as:

During evaluation, the proposed implementation successfully completed **19 of 20** benchmark programs, corresponding to a benchmark completion rate of **95%**.

**Software Quality Assessment**

Each analytical agent generates structured recommendations containing:

- Severity
- Issue
- Explanation
- Suggestion

These outputs provide qualitative assessment across multiple software engineering dimensions.

**Quality Score**

The framework computes quantitative software quality scores for four assessment categories:

- Review Quality
- Security
- Performance
- Documentation

These scores are aggregated to produce an overall software quality score that summarizes the condition of the analyzed source code.

**Explainability**

Unlike conventional static analysis tools that primarily report warnings or rule violations, AgentCodeReview provides natural-language explanations describing:

- why an issue exists,
- how it affects software quality,
- recommended corrective actions.

Explainability is therefore considered an important qualitative evaluation criterion in this study.

## *4.6 Experimental Procedure*

Each benchmark program was evaluated using the following procedure:

1. Load the benchmark program.
2. Execute the Single-Agent baseline.
3. Record execution time.

4. Execute the proposed Multi-Agent framework.
5. Measure individual agent execution times.
6. Compute software quality scores.
7. Generate repaired source code.
8. Produce HTML and PDF reports.
9. Store benchmark results in CSV format.

This process was repeated for every benchmark program to ensure consistent evaluation across all defect categories.

### *4.7 Experimental Results Overview*

The benchmark evaluation produced the following summary:

**Table 5. Benchmark Summary**

| **Metric** | **Value** |
|---|---|
| Total Benchmark Programs | 20 |
| Successful Executions | 19 |
| Failed Executions | 1 |
| Benchmark Success Rate | 95% |
| Evaluation Categories | 5 |
| HTML Reports Generated | Yes |
| PDF Reports Generated | Yes |
| Streamlit Demonstration | Yes |

The successful benchmark executions form the basis of the quantitative analysis presented in the following section.

### *4.8 Discussion of Experimental Design*

All the benchmark programs have been run on the same software and hardware configuration to minimize environmental variability. Second, both the Two approaches - Single-Agent and Multi-Agent - used the same language model, so that differences in performance were largely attributed to the architecture, and not to the capabilities of the language model. Third, benchmark programs were purposefully chosen to be good examples of a range of software quality problems to evaluate software quality along more than one dimension, rather than just one type of defect.

The benchmark dataset shows a relatively small size compared with industrial repository datasets but it is controlled and reproducible to assess the capability of the proposed framework to work in practice. This is a hands-on assessment and evaluation of AgentCodeReview to demonstrate the feasibility and usability of AgentCodeReview as a modular, explainable and extensible AI.

## 5. Results and Discussion

In this section, the experimental results obtained during the implementation of the proposed AgentCodeReview framework are emphasized. The aim of the evaluation was to investigate the ability of the proposed Multi-Agent architecture to match the effectiveness of the traditional Single-Agent approach to automated code review and bug repair. The main focus of the benchmark was the execution performance, software quality diagnosis, the explainability of software recommendations, and the overall robustness of the framework.

The evaluation was conducted on a standard set of 20 Python programs that have different levels of software quality defect in five categories: runtime error, security violation, performance inefficiency, documentation, combination defect. The experiment is conducted with the same LLM and runtime environment in the baseline and proposed Multi-Agent frameworks, ensuring it is conducted fairly.

## *5.1 Benchmark Execution Results*

The benchmark execution results are summarized in Table 6.

**Table 6. Overall Benchmark Results**

| **Metric** | **Result** |
|---|---|
| Total Benchmark Programs | 20 |
| Successful Executions | 19 |
| Failed Executions | 1 |
| Success Rate | 95% |
| Benchmark Categories | 5 |
| HTML Reports Generated | Yes |
| PDF Reports Generated | Yes |
| Streamlit Interface | Yes |

The success rate of the overall execution of the 20 benchmark programs was for the proposed framework 95% (19 successful). The language model's reply was not a complete JSON, which could not be automatically parsed, in one benchmark case. The failure was attributed to formatting of the responses and not to the logic of the framework, so the other benchmark programs were judged good enough for the quantitative analysis.

The high completion rate indicates that the proposed architecture can be used for software quality assessment by assessing the large number of representative software faults without affecting the execution stability.

## *5.2 Execution Time Analysis*

Execution time is an important consideration for AI-assisted software engineering systems because automated code review should provide timely feedback without introducing excessive development delays.

The benchmark recorded the average execution time of both the Single-Agent baseline and the proposed Multi-Agent framework.

**Table 7. Average Execution Time**

| **Framework** | **Average Execution Time (seconds)** |
|---|---|
| Single-Agent | **42.40** |
| Multi-Agent | **78.98** |

From the experimental results, it can be seen that the execution time for the proposed Multi-Agent framework was longer than the Single-Agent baseline. This is because the proposed framework performs several special analyses prior to aggregating the various results it provides.

The Multi-Agent approach is unlike Single-Agent, as it independently analyses the 5 aspects of software quality – review quality, security, performance, documentation and automated repair – rather than attempting to perform all 5. It might make the calculation more complicated, but each agent would be able to concentrate on another software engineering activity and thus make more accurate and more comprehensible recommendations.

This means that the additional computer time is a trade-off between speed and depth of analysis. In most cases, it's okay for software quality assurance tasks where explainability and detailed feed back is more important than the response time.

## *5.3 Comparative Analysis Between Single-Agent and Multi-Agent Approaches*

A comparison between the two approaches is presented in Table 8.

**Table 8. Comparison of Framework Characteristics**

| **Criterion** | **Single-Agent** | **Multi-Agent (Proposed)** |
|---|---|---|
| Code Review | ✓ | ✓ |
| Security Analysis | Limited | Specialized |
| Performance Analysis | Limited | Specialized |
| Documentation Review | Limited | Specialized |
| Automated Bug Repair | ✓ | ✓ |
| Explainability | Moderate | High |
| Modular Design | No | Yes |
| Parallel Processing | No | Yes |
| Quality Scoring | Limited | Comprehensive |
| HTML Report | Yes | Yes |
| PDF Report | Yes | Yes |
| Streamlit Integration | Yes | Yes |

The comparison demonstrates that the proposed framework extends conventional AI-assisted code review by introducing specialized analytical agents coordinated through a centralized orchestration layer. Instead of relying on a single reasoning process, each software quality dimension is evaluated independently before the results are combined into a unified assessment.

This modular architecture provides more structured feedback while simplifying future framework extensions. New analytical agents can be incorporated with minimal modification to the existing implementation.

## *5.4 Explainability Analysis*

One of the primary objectives of AgentCodeReview is to improve the explainability of AI-assisted software quality assessment.

Each specialized agent produces four structured outputs:

- Severity
- Issue
- Explanation
- Suggestion

In contrast to most other AI-assisted code review tools that give a short suggestion, AgentCodeReview can generate comprehensive explanations, in natural language, explaining why an issue is there and how it can be fixed.

When, for example, the Security Agent finds that the underlying SQL query construction is not secure it not only informs you about the security problem, but also offers a security risk analysis and suggested corrective action,

which is to use a parameterized query. Similarly, the Performance Agent explains how inefficient programming structures are and provides ideas for improving the performance rather than just identifying performance bottle necks.

The structure makes it easier to understand and more transparent, and it assists developers with understanding the reasoning behind the generated recommendations.

### *5.5 Software Quality Assessment*

The proposed framework evaluates software quality across four independent dimensions:

- Review Quality
- Security
- Performance
- Documentation

Each analytical agent gives its own particular quality score for the software to contribute to the overall software quality assessment.

The scoring system offers a number of useful features.

- Firstly, quantitative scores enable the software quality to be compared across software versions and software projects.
- Secondly, category scores indicate areas that can be improved.
- Finally, software quality score offers a quick summary which can be utilized for software quality monitoring and project reporting.

The proposed framework integrates qualitative explanations with quantitative quality assessment, in contrast with traditional static analysis tools which usually present single warnings.

### *5.6 Automated Bug Repair Evaluation*

The same benchmark programs as used by the review agents were used to evaluate the Repair Agent.

The software quality assessment yielded a corrected program source code which would correct the software problems and keep the same functionality as the original program.

The automatically generated HTML and PDF report files include the generated repaired code and can be compared with the original implementation to see suggested repairs.

The results indicate that automated repair is a potential complement to defect detection in the review process; although quality of the repairs generated by the underlying language model is critical to the quality of the repairs.

### *5.7 Discussion*

The experimental evaluation demonstrates several important observations regarding the proposed framework.

Breaking up the software quality assessment into specialized agents meant that the software quality assessment structure became more interpretable and structured than the Single-Agent baseline. The independent functions of Review, Security, Performance, Documentation and Repair agents generated more specific analyses that were more easily understood and accepted.

In addition, the modular design provides much better maintainability. Each of the agents is independent, so future enhancements can be made without changing the legacy agents. This is particularly helpful over monolithic prompting methods.

Third, a report of HTML, PDF and interactive visualization in Streamlit format is automatically generated, making the application of the framework more useful. They offer reporting features that enable software developers to examine the analysis results without having to combine data from several sources.

Lastly, the benchmark assessment shows that explainable AI is capable of being successfully applied to software quality assessment processes, which do not solely depend on static analysis techniques. The proposed model combines the reasoning based approach to analysis and a structured reporting approach that can be applied in the context of education, software quality assurance processes and research prototypes.

It also reported that one execution couldn't be done because the invalid JSON was generated by the language model. This observation indicates that LLM-based software engineering systems currently face a challenge in fine-tuning decoding conditions and hints at potentially better future software engineering system implementations through schema-guided generation or constrained decoding to further enhance software engineering systems' reliability.

### *5.8 Summary of Findings*

The experimental results indicate that the proposed AgentCodeReview framework successfully achieves the objectives established in this research. Compared with the Single-Agent baseline, the Multi-Agent architecture provides:

- more comprehensive software quality assessment,
- improved explainability through specialized recommendations,
- modular and extensible system architecture,
- integrated automated bug repair,
- quantitative software quality scoring,
- automated HTML and PDF reporting,
- and an interactive web-based interface for practical software analysis.

The additional runtime in the proposed framework is negligible, but the experiments presented in the benchmark reveal the benefits of modularity, explainability, and full software quality assessment overcome this computational cost, for the given benchmark.

## 6. Threats to Validity

Like other application studies, conclusions of this study must be drawn with a few factors in mind that could impact the generalizability and repeatability of the conclusions. In order to enhance the transparency of the potential threats to validity, it is discussed under four categories: construct validity, internal validity, external validity, and conclusion validity.

### *6.1 Construct Validity*

Construct validity is in reference to whether the metrics used in the evaluation are appropriate for the purposes of the proposed framework. The study evaluated the effectiveness of AgentCodeReview based on benchmark execution success, execution time, software quality assessment, explainability of the recommendation, and automated software bug repairing. These measures provide good practical sense for the performance of the framework, but cannot fully determine the quality of the software. The benchmark for example does not explicitly address the issue of maintainability over long cycles of software evolution, developer productivity or user acceptance. Furthermore, the quality score as calculated using the framework is not calculated based on any of the commonly used software engineering quality models, but rather on the output of specialized agents. These can be helpful quantitative

measurements, but should be considered as relative scores rather than absolute measures of software quality.

**6.2 Internal Validity**

Internal validity refers to factors that may influence the observed experimental results.

The hardware configuration, the software environment, the language model and the framework implementation are the same in all the benchmark programs to ensure consistency in the evaluation. To reduce the possibility that differences in the results were due to differences in the model capabilities of the two frameworks, the proposed Multi-Agent framework and the baseline Single-Agent were found to be similar in the programs they were used in, and also in the LLM backend they used.

However, the language model that was used in one benchmark program produced malformed JSON output, which is why it was not possible to complete this execution. This was not an orchestration or benchmarking issue, but was rather a matter of how LLM responses are generated, which is probabilistic. To reduce these impact, the following features were added to the implementation: structured prompts, deterministic inference setting, and good JSON parser.

### *6.3 External Validity*

The extent to which externalization of results from the experiment can be applied to other software development contexts is known as external validity.

The benchmark dataset is composed of 20 Python programs that exhibit common software quality problems such as runtime errors, security flaws, performance problems and lack of documentation, as well as mixed defect scenarios. These benchmark programs cover a range of representative types of faults, but do not cover all the diversity and complexity found in large industrial software systems.

Likewise, the framework was only evaluated in relation to python source code. While the architecture is language-independent, prompt templates and benchmark programs would have to be translated to other languages such as Java, C++, JavaScript, and Go.

An experimental evaluation was also performed with a single local deployment of the Large Language Model (LLM), using Ollama. Different language models can lead to different quality of reasoning, execution time, and/or response consistency. Thus, the results obtained should be considered in the light of the experimental setup.

### *6.4 Conclusion Validity*

Conclusion validity addresses whether the conclusions drawn from the experimental results are supported by the collected evidence.

The results from the benchmark evaluation demonstrate that the proposed Multi-Agent framework has been able to analyze nineteen out of 20 benchmark programs and produce structured and explainable software quality analyses. When compared to the Single-Agent baseline, specialized agents provided more targeted recommendations in the following areas: review, security, performance, and documentation.

But the benchmark isn't very big and it's not suitable to draw conclusions about the situation throughout the entire population. The results reported throughout the paper are intended to provide a demonstration of the feasibility and effectiveness of the proposed architecture, rather than to suggest that it is superior to any other existing code review system on the market now that will rely on AI to support code review.

The proposed approach could be further validated with more empirical evidence by expanding the benchmark repositories, incorporating more programming languages, and introducing more LLMs in future evaluations.

### *6.5 Future Research Directions*

The present implementation establishes a foundation for several promising research directions.

Future work may investigate:

- integration of additional specialized agents for testing, code optimization, refactoring, and compliance analysis;
- support for multiple programming languages through language-specific prompt engineering;
- evaluation using large-scale open-source software repositories;
- integration with continuous integration and continuous deployment (CI/CD) pipelines for automated code review during software development;
- incorporation of constrained decoding or schema-guided generation to eliminate malformed LLM responses;
- comparison across multiple open-source and commercial large language models to evaluate robustness and generalizability;
- inclusion of developer-centered usability studies to assess the practical impact of explainable AI-assisted code review in industrial environments.

Overall, these directions provide opportunities to extend AgentCodeReview from a research prototype into a scalable software engineering platform suitable for real-world development environments.

## 7. Conclusion

Large Language Models are increasingly being adopted, and software engineering workflows like automated code review and program repair have been renewed with hope for improvements. While there have been some successful examples of existing AI-based code review systems, most of these systems are monolithic, performing several code review tasks in a single reasoning process. These methods can lead to vague and abstract suggestions and can be more challenging to extend the system in the future.

In this paper, the Multi-Agent framework AgentCodeReview is introduced as a way to enable multiple agents from various Software Engineering domains to work together to create explainable code review and automatic bug repair. The proposed architecture presents the idea of separating the software quality assessment process into Review agent, Security agent, Performance agent, Documentation agent and Repair agent, and having a central layer orchestrate them. Every individual agent can perform on one dimension of quality and only one component of orchestration is responsible for combining the quality evaluations produced by the agents into a total evaluation. Automated Bug Repair is just one part of the framework, along with software quality scoring, HTML and PDF reports generation, and a Streamlit-based web interface to make things more accessible and easy to use.

The framework was tested with a benchmark of 20 programs that have runtime errors, security issues, performance issues, documentation defects and mixed defect programs in Python. Comparative tests revealed that the proposed Multi-Agent architecture could execute 19 out of 20 benchmark executions (a benchmark success rate of 95%) while the Single-Agent could not execute any of the benchmark executions. Although the overall performance of the Multi-Agent approach was lower due to the additional execution time caused by the parallel execution of specialized analyses, it was still more structured, explainable and domain specific with more recommendations generated than the baseline approach.

This study shows how task decomposition with specialized agents can be helpful in enhancing transparency and structuring software quality assessment. AgentCodeReview does not replace software engineering practices, it is meant to be used in parallel with the software developers workflow and provide structured suggestions, automatic repair suggestions and detailed reporting as a means to educate and support professional software development.

Finally, the paper demonstrates the possibility of using a modular Multi-Agent architecture to create a single system capable of effectively integrating the above three aspects of code review, bug repair and software quality assessment. The proposed implementation provides a foundation for future work on collaborative AI systems in

software engineering and provides a hands-on platform to scale intelligent code review beyond its current capacity of software size, programming languages and language models.

## REFERENCES


1. Fowler, M. (2018). Refactoring: Improving the design of existing code (2nd ed.). Addison-Wesley Professional.
2. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.
3. Humble, J., & Farley, D. (2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley.
4. Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship. Prentice Hall.
5. McConnell, S. (2004). Code complete (2nd ed.). Microsoft Press.
6. Pressman, R. S., & Maxim, B. R. (2020). Software engineering: A practitioner's approach (9th ed.). McGraw-Hill.
7. Sommerville, I. (2016). Software engineering (10th ed.). Pearson.
8. Beck, K. (2003). Test-driven development: By example. Addison-Wesley.
9. Kim, S., Pan, K., & Whitehead, E. J. (2008). Memories of bug fixes. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 35–45). ACM.
10. Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., & Devanbu, P. (2013). The promises and perils of mining Git. Mining Software Repositories, 1–10.
11. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 33(8), Article 220. https://doi.org/10.1145/3695988
12. Yu, Y., Rong, G., Shen, H., Zhang, H., Shao, D., Wang, M., Wei, Z., Xu, Y., & Wang, J. (2024). Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. ACM Transactions on Software Engineering and Methodology, 34(1), Article 14. https://doi.org/10.1145/3695993
13. Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., Yang, Y., & Chen, Z. (2024). A systematic literature review on large language models for automated program repair. arXiv. https://arxiv.org/abs/2405.01466
14. Tufano, R., & Bavota, G. (2025). Automating code review: A systematic literature review. arXiv. https://arxiv.org/abs/2503.09510
15. Basic, E., & Giaretta, A. (2024). Large language models and code security: A systematic literature review. arXiv. https://arxiv.org/abs/2412.15004
16. Chen, Y., Sun, W., Fang, C., Chen, Z., Ge, Y., Han, T., Zhang, Q., Liu, Y., Chen, Z., & Xu, B. (2024). Security of language models for code: A systematic literature review. arXiv. https://arxiv.org/abs/2410.15631
17. Alomari, N., Redah, M., Ashraf, A., & Alshayeb, M. (2025). Using LLMs to enhance code quality: A systematic literature review. Journal of Systems and Software.
18. He, J., Treude, C., & Lo, D. (2024). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. arXiv.
19. Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., Systä, K., & Abrahamsson, P. (2024). AI-powered code review with large language models: Early results. arXiv.
20. Tang, X., Kim, K., Song, Y., Lothritz, C., Li, B., Ezzini, S., Tian, H., Klein, J., & Bissyandé, T. F. (2024). CodeAgent: Autonomous communicative agents for code review. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP).
21. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., & Liang, W. (2024). DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence. arXiv. https://arxiv.org/abs/2401.14196
22. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrandis, C. M., et al. (2023). Code Llama: Open foundation models for code. arXiv. https://arxiv.org/abs/2308.12950
23. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. (2023). StarCoder: May the source be with you!. arXiv. https://arxiv.org/abs/2305.06161
24. Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. (2024). Qwen2.5-Coder technical report. arXiv. https://arxiv.org/abs/2409.12186
25. OpenAI. (2023). GPT-4 technical report. arXiv. https://arxiv.org/abs/2303.08774
26. Anthropic. (2024). The Claude 3 model family: Opus, Sonnet, and Haiku. Anthropic Research

27. Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why should I trust you?": Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1135–1144). ACM. https://doi.org/10.1145/2939672.2939778
28. Adadi, A., & Berrada, M. (2018). Peeking inside the black-box: A survey on Explainable Artificial Intelligence (XAI). IEEE Access, 6, 52138–52160. https://doi.org/10.1109/ACCESS.2018.2870052
29. Arrieta, A. B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-López, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. Information Fusion, 58, 82–115. https://doi.org/10.1016/j.inffus.2019.12.012
30. McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR) (pp. 192–201). IEEE. https://doi.org/10.1145/2597073.2597076
31. Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In Proceedings of the 35th International Conference on Software Engineering (ICSE) (pp. 712–721). IEEE Press.
32. Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In Proceedings of the 35th International Conference on Software Engineering (ICSE) (pp. 672–681). IEEE Press.
33. Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering (EBSE Technical Report EBSE-2007-01). Keele University & Durham University.
34. ISO/IEC. (2011). ISO/IEC 25010:2011—Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models. International Organization for Standardization.
35. Bharath Kumar N, & T L Manasa. (2026). A Multi-Agent Self-Healing Framework for Enterprise Text-to-SQL with Hallucination Risk Prediction and Explainable Query Generation. International Journal of Computer Information Systems and Industrial Management Applications, 18(12s), 596–603. https://doi.org/10.70917/ijcisim-2026-3933
36. Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. A. (2024). Using an LLM to help with code understanding. In Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE 2024).
37. Yu, X., Liu, L., Hu, X., Keung, J. W., Liu, J., & Xia, X. (2024). Where are large language models for code generation on GitHub? arXiv. https://arxiv.org/abs/2406.19544
38. Pandey, R., Singh, P., Wei, R., & Shankar, S. (2024). Transforming software development: Evaluating the efficiency and challenges of GitHub Copilot in real-world projects. arXiv. https://arxiv.org/abs/2406.17910
39. Bharath Kumar N, & T L Manasa. (2026). Evaluation of Large Language Models for Natural Language to SQL Query Generation: A Comparative Study Using Exact Match and Execution Accuracy. International Journal of Computer Information Systems and Industrial Management Applications, 18(15s), 132–149. https://doi.org/10.70917/ijcisim-2026-4353
40. GitHub. (2024). Does GitHub Copilot improve code quality? Here's what the data says. GitHub Research Blog. GitHub Research Blog
41. Shah, J. K. (2024). Explainable AI in software engineering: Enhancing developer–AI collaboration. The American Journal of Engineering and Technology, 6(7), 99–108. https://doi.org/10.37547/tajet/Volume06Issue07-11
42. He, J., Treude, C., & Lo, D. (2024). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. arXiv.
43. Tang, Y. (2024). Large language models meet automated program repair: Innovations, challenges and solutions. Applied and Computational Engineering.
44. Anand, A., Gupta, A., Yadav, N., & Bajaj, S. (2024). A comprehensive survey of AI-driven advancements and techniques in automated program repair and code generation. arXiv.
45. Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., Yang, Y., & Chen, Z. (2024). A systematic literature review on large language models for automated program repair. arXiv.
46. Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., Systä, K., & Abrahamsson, P. (2024). AI-powered code review with large language models: Early results. arXiv.
47. Tufano, R., & Bavota, G. (2025). Automating code review: A systematic literature review. arXiv.
48. Basic, E., & Giaretta, A. (2024). Large language models and code security: A systematic literature review. arXiv.
49. Bharath Kumar N, & T L Manasa. (2026). Agentcodereview: A Multi-Agent Framework For Explainable Code Review And Automated Bug Repair. International Journal of Computer Information Systems and Industrial Management Applications, 18(14s), 1121–1129. https://doi.org/10.70917/ijcisim-2026-4326