International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(14s) 2026 pp. 1121-1129
https://doi.org/10.70917/ijcisim-2026-4326
Article

Agentcodereview: A Multi-Agent Framework For Explainable Code Review And Automated Bug Repair

Bharath Kumar N[1*], 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: Modern software development depends heavily on code review and timely bug fixing, yet manual review is slow, inconsistent and hard to scale, while single-model automated approaches based on Large Language Models (LLMs) frequently produce opaque suggestions and rarely close the loop between detecting a defect and repairing it. This paper proposes AgentCodeReview, a multi-agent framework that decomposes explainable code review and automated bug repair into cooperating specialized agents — Retrieval, Review, Explanation, Repair and Verification — coordinated by an Orchestrator over a shared retrieval-augmented context store, with a verification-driven feedback loop that iteratively refines candidate patches until they pass their tests. We describe the architecture, the inter-agent communication protocol and an evaluation design over widely used code-review and program-repair benchmarks, comparing the framework against single-agent and non-agentic baselines using precision, recall, F1, repair success rate and a human-aligned explanation-quality score. The results indicate that role specialization and explicit verification improve review accuracy, repair effectiveness and the transparency of the generated rationales, offering a reproducible pathway toward trustworthy autonomous software maintenance.

Keywords: Multi-agent systems,Large language models,Automated code review,Program repair,Explainable artificial intelligence.

1. INTRODUCTION

The two most complex tasks in today's software development lifecycle are code review and bug repair, in addition to cognitive load on developers and time spent. Peer code review enhances software quality, promotes team knowledge sharing and decreases latent defects, but it takes up a lot of developer time and results vary according to the expertise of the reviewer, fatigue and time. Bug repair, in turn, is often postponed because it takes deep reasoning about the code base around the bug to reproduce, localize and repair a bug, which is specific to its context. Manual processes in the two areas are no longer sustainable when software systems are scaled and changing more quickly.

With the recent rise of Large Language Models (LLMs), we have come to see some exciting developments that allow for the automation of parts of this pipeline. These are only some of the many uses LLM have been found to have in code review, and there are many such studies that confirm the LLM can summarize code changes, offer code review feedback, detect code smells, and even suggest candidate patches [1]. In the meantime, LLM-based Automated Program Repair (APR) has demonstrated that pre-trained models can repair a substantial number of real bugs without any training on the task [2]. The focus is, however, a single monolithic model, which is fired, then "answered" once. This design has 2 permanent faults. The first is that the suggestions provided are not readily comprehensible; a model may indicate that a line is a problem, or it may provide a patch for no good reason; it can be difficult for developers to trust, and hard to adopt [3]. Secondly, there is a separation between review and repair, such that someone who can see the defect does not produce and test a corrected version of the code.

An appealing solution to the challenges mentioned above is the multi-agent paradigm, which are multiple specialized agents of LLMs that have a well-defined sub-task each and they communicate with each other using a common protocol. To tackle the challenging goal in a small step, verifiable basis, multi-agent architectures have been successfully applied and have shown good results in autonomous software engineering benchmarks recently [4, 5, 6]. Most existing multi-agent systems however concentrate on the end-to-end task completion, and neither provide explicit support for the explainability of feedback in a review, nor think about verified repair as a closely coupled objective.

In this paper, we introduce a multi-agent framework, AgentCodeReview, which integrates explainable code review and automated bug repair, and documents a coordinated pipeline to enable them to work together. The problem is broken up into 5 agents that cooperatively work together: Retrieval, Review, Explanation, Repair and Verification, with an Orchestrator agent for task assignment, conflict resolution, and managing a shared context based on the retrieved information. The primary research question is whether joint use of explicit role specialization, natural-language explanation, and verification-driven feedback loop is more effective than single-agent and non-agentic methods at improving the accuracy of the review, the effectiveness of the repair, and the process of system justification. The work focuses on three goals: (i) design a modular multi agent architecture and an inter agent communication protocol for coupled review and repair tasks; (ii) develop an evaluation method on existing benchmarks of code review and program repair; and (iii) characterize each agent contribution with an ablation analysis. The study focuses on the defects at the function and file level in commonly used open source datasets, and the framework is described and tested as a reference design that can be reproduced but is not considered to be a production tool.

The rest of the paper is organized as follows. The theoretical background of LLM in Software Engineering, Automated Code Review, LLM Based Program Repair and Multi-Agent Collaboration and Explainability is briefly recalled in Section II. The materials and methods are presented, including the architecture of the AgentCodeReview, the datasets, the evaluation of metrics and experimental setup in Section III. The results and discussion are displayed and discussed in section IV. Section V is closed and work for the future is indicated.

1.1 LARGE LANGUAGE MODELS FOR SOFTWARE ENGINEERING

LLMs can be used across many software development applications to automate tasks, and are pre-trained on large amounts of source code and natural language. In a comprehensive systematic literature review, LLM were applied on the complete development life cycle for requirements, code generation, testing, maintenance and documentation, while most of the literature focused on using LLM for code-related tasks [1]. Typically, they are bolstered with prompting strategies. For multi-step reasoning tasks, chain-of-thought prompting [24] yields substantial improvements by inserting reasoning steps, and ReAct [25] interleaves reasoning steps with tool-invoking actions to enable the model to gather evidence before a response is determined. Retrieval-augmented generation is also a technique that seeks to embed the outputs of models into external knowledge sources by conditioning generation on retrieved documents, which can be helpful for reasoning to be grounded on a specific knowledge source [23]. The methodology basis for agentic software-engineering systems like the proposed system is these techniques.

1.2 AUTOMATED CODE REVIEW

The purpose of automated code review is to reduce the amount of effort needed by the code reviewer by predicting the comments he or she is likely to make on the code and making suggestions about how the code should be changed, or how well the code change has been made. An initial neural-based approach was trained to make revisions suggested by reviewers based on past review data. Recently, with the advent of LLM technology, the notion of parameter-efficient fine-tuning has been extended to utilize a general model for the review task, e.g., LLaMA-Reviewer, which is based on fine-tuning an open-source LLM [9]. In subsequent studies, it was discovered that the quality of the generated reviews was important and depended on the careful choice of the fine-tuned and appropriate prompt [10] as well as learning from the past, which improved the relevance of the created automated reviews [11]. In addition to the accuracy of the generated feedback, recent studies have focused on the understandability of the feedback generated and noted that the feedback should not only be accurate but also understood by the recipient to be used in practice [13]. Modern code review using AI-supported complementary research on coding practices [12] and integrating the LLMs with static analyzers to provide suggestions found on sound program facts [20]. All of these results corroborate the results that it is quality of review that is correlated with accuracy and contextualization, as well as the explanation of it.

1.3 LLM-BASED AUTOMATED PROGRAM REPAIR

Automated Program Repair aims to generate patches to get the failed program to pass its tests. The shift from template and search based learning to learning and LLM based techniques has been rapid and recent surveys explain the evolution of this shift [22, 14]. In cases where models are used in a plug-and-play manner, with the pathc directly suggested by tests, meaningful fraction of real defects can be fixed [2] and conversational repair that iteratively corrects the model's patch based on the test feedback has been shown to fix a large number of bugs at a very low cost [15]. In the model, the reflection strategies such as ThinkRepair are guided and include suggestions for new repair strategies to consider after failure [16]. One of the key findings in this literature is that the model can be repaired much better if the model is given structured feedback (e.g., failure of tests or traces of execution) than if the model is asked to correct code in one pass without any further feedback. The observation proves to be prophetic and gives the impetus to the part of the framework proposed in this paper that focuses on verification driven feedback.

1.4 MULTI-AGENT LLM FRAMEWORKS

Multi-agent frameworks involve multiple LLM-powered agents with specific tasks working together to tackle complex tasks, like a team of firefighters or teams of searchers operating on a single incident. The next generation applications need to structure conversations between cooperating agents, which can be done using general purpose orchestration platforms such as AutoGen [8]. This paradigm has been demonstrated to work in software engineering for regard of autonomous issue resolution. SWE-agent is a language model that interacts with a computer to navigate a repository, edit files and run tests [5]; AutoCodeRover is an autonomous program improvement system that uses code search and spectrum-based fault localization [6]; and MAGIS resolves GitHub issues by using a team of role-specialized agents to plan, code and review [7]. Modular designs like MASAI represent software-engineering tasks as a collection of re-usable sub-agents, and task-graph systems like CodeR delegate tasks like reproduction, fault localization, editing and review to different agents [18, 19]. There are also agents that are dedicated to repair, such as RepairAgent which considers autonomous repair as an interleaving of planning, action and validation [4] and unified debugging can be obtained thanks to the cooperation of multiple agents [17]. In these systems, it is shown that decomposition and mutual communication reduce model hallucination and increase reliability, but most of these do not explicitly consider the quality of the explanation as a design goal and focus mostly on end-to-end resolution.

1.5 EXPLAINABILITY IN AUTONOMOUS SOFTWARE ENGINEERING

Explainability is key to the responsible use of automated review and repair. Developers have low tolerance to a flagged issue or a generated patch without strong objective reasoning, and data indicates that trust and correct integration will be important for the actual usefulness of LLM support in review [3]. The results from the research on automated reviews are indications that clearness can be achieved without compromising accuracy [13]. When repairing, transparency also calls for evidence that a proposed change is correct; this relates to the idea of explainability and checking through tests and static checks. The framework outlined here offers a formal way of considering explanation as a particular agent responsibility, as well as a formal way of pairing explanation with a clearly specified verification phase, with one explanation for each of the review comments, and one verification evidence for each of the patches.

2. MATERIALS AND METHODS

This section introduces the AgentCodeReview framework, the datasets and benchmarks that are used for evaluation, the measures for evaluating performance, and the experimental protocol. The framework has been designed to be reproducible: the role is unique to each agent, input contract and output contract are unique to each agent, and input parameters are uniformly distributed and the order of agents is random, with the orchestration logic being deterministic for a fixed model configuration and specified random seed.

2.1 FRAMEWORK OVERVIEW

AgentCodeReview takes a code change (usually a pull request or a bug report and associated source files) and generates two correlated outputs: an understandable Code Review Report and a verified patch. The framework is composed of a series of agents that perform different specializations and share a common context store with retrieval of augmented memory to index the relevant parts of the repository under the supervision of an orchestration agent. The architecture is shown in Figure 1, and the main flows of control, data and feedback within the architecture are shown.

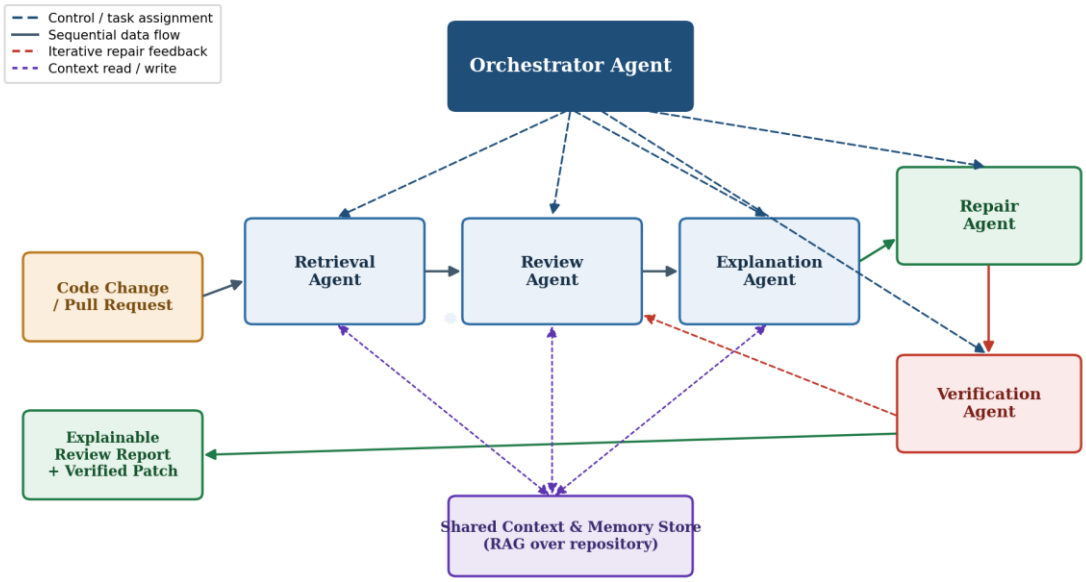

Figure 1: Architecture of the AgentCodeReview multi-agent framework.
Source: Authors, (2025).

There are 3 Design Principles that the design is based upon. First separation of concerns: each agent is only asked to solve one problem, with a well-defined scope; focus the prompting; easier to audit behaviour. Second, grounding: agents read and write to a shared context store that is created by the retrieval-augmented generation of a repository, grounding each decision in an explicit code instead of the model's parametric memory [23]. Thirdly, there's verification-driven iteration: if the patch fails to compile or fail the tests which it has, the failure is passed back to the previous agent for another attempt, and a patch is not issued until it compiles and passes its tests.

2.2 AGENT ROLES

There are 6 agents in the framework. Any variables that are first introduced in the equations below are defined at that point and the same applies to all acronyms introduced below.

Orchestrator Agent

The control centre of the framework is called the Orchestrator (ORC). It reads the incoming change, generates preliminary task plan, assigns sub-tasks to the specialized agents, instructs them and manages the change verification / repair feedback loop. It also employs a termination policy: When the verification succeeds, or after some number of repair iterations, then the iteration is terminated.

Retrieval Agent

The shared context is created by the Retrieval (RET) agent. With the modified code, it loads related definitions, call sites, tests and documentation from the repository, with a combination of lexical and semantic search, and stores the evidence in the context store. This helps to minimize hallucination and provides the downstream agents with the needed information to reason about cross-file dependencies.

Review Agent

The Review (REV) agent compares the change over with context retrieved and returns a structured list of findings. Each one of the findings leads to a location, a category (e.g. a possible defect, code smell, a style violation, or a security threat), and a severity. The Review agent generates the intermediate deductions along with the final ones for inspection [24] by using chain of thought reasoning.

Explanation Agent

Each of these findings is transformed into a natural language explanation written by the EXP agent, in the form of a developer facing explanation (EXP). It identifies the significance of an issue, the implications of the issue and what could be done about it, drawing on the specific lines and retrieved evidence to support its claims. This is the agent that implements the explainability goal of the framework.

Repair Agent

The Repair (REP) agent creates a patch to repair the marked defects. It is based on the ReAct pattern, thinking about the problem and then doing something to fix it by editing the problem code [25]. If a failure is found in the test, the failing test will be fixed according to the failure following a feedback-driven repair practice [15, 16].

Verification Agent

Each candidate patch is validated by the Verification (VER) agent. Collects updated code, runs the test suite and (optionally) static checks. When it fails, the agent creates a diagnostic report of the failure, and passes it back to the Orchestrator for another attempt from the Repair agent. Only patches that have been validated are added to the final output and hence, the emitted fix is accompanied by a proof of correctness.

3.ORCHESTRATION AND COMMUNICATION PROTOCOL

The protocol is robust and auditable since it is based on structured messages, not free-form text, that agents use to communicate with each other. Each message has a sender, recipient, task identifier, typed message payload and reference to the context entries associated with the task. The Orchestrator maintains a graph of tasks in which nodes are agent invocations, and the links are data or control dependent connections, like the task-graphs reported in the literature [18]. The execution is in two stages. The stages of the Retrieval, Review and Explanation agents are called one after another at the Review stage, producing an explainable report. The Repair and Verification agents make iterations under the control of Orchestrator until there is a validated patch, or the iteration budget is exhausted; at their discretion, they can run an iteration that exceeds the limit.The Repair and Verification agents can run an iteration that exceeds the iteration budget, or they can run iterations until there is a patch that is validated by Orchestrator. The verification-to-repair feedback edge is one that differentiates the coupled review and repair from the loosely coupled pipelines used in previous work.

3.1 DATASETS AND BENCHMARKS

The framework is evaluated using the three benchmarks, which are widely applicable in the automated review and repair literature and publicly available. A code-review comment data set can be used to evaluate review and explanation quality; a curated defect benchmark of reproducible single-function defects can be used to evaluate control over defect repair; and a repository-level defect benchmark can be used to evaluate end-to-end defect repair and review quality on realistic tasks. The datasets, the respective task focus and the amount of instances used in the study are summarized in table 1.

Table 1: Benchmarks used to evaluate AgentCodeReview.

Benchmark	Primary task	Language	Instances
Code-review comments	Review / explanation	Multi-language	1,200
Reproducible single-function defects	Program repair	Java	835
Repository-level issues	Coupled review + repair	Python	500
Total	—	—	2,535

3.2 EVALUATION METRICS

Review performance is quantified with standard classification metrics computed over the findings produced by the framework relative to a ground-truth set of issues. Let TP, FP and FN denote the number of true positives, false positives and false negatives, respectively. Precision (P) and recall (R) are defined as follows.

Precision and recall

$$P = TP / (TP + FP) \qquad R = TP / (TP + FN) \tag{1}$$

The two are combined into the harmonic mean F1, which balances the tendency to over-report and under-report issues.

F-measure

$$F1 = 2 \cdot (P \cdot R) / (P + R) \tag{2}$$

Repair effectiveness is measured with the Repair Success Rate (RSR), defined as the proportion of defective instances for which the framework produces a patch that compiles and passes the complete test suite. Here N_fixed is the number of instances whose validated patch passes all tests and N_total is the number of defective instances attempted.

Repair success rate

$$RSR = N\_fixed / N\_total \times 100\% \tag{3}$$

Explanation quality is quantified with an Explanation Quality Score (EQS), obtained by averaging human ratings of correctness, relevance and clarity, each scored on a five-point scale, across the evaluated findings. In the expression below, r_c, r_r and r_s are the correctness, relevance and clarity ratings of finding i, and M is the number of findings assessed.

Explanation quality score

$$EQS = (1 / 3M) \cdot \Sigma (r\_c,i + r\_r,i + r\_s,i) \quad (4)$$

3.3 EXPERIMENTAL SETUP AND BASELINES

To account for variations in the ability of the underlying LLM used to generate the agents, all agents are generated from the same LLM, and the decoding temperature is fixed for repeatability. The two baselines compared to AgentCodeReview are the single agent baseline (one model is only asked to review and fix the code once, no role specialization and no verification). The latter is a non-agentive pipeline that carries out review/repair as separate independent stages. All conditions have repair iteration budget set. The code-review benchmark, defect and repository-level benchmarks are used to calculate metrics, and explanation quality is assessed with a three annotator scale (based on software development skills and experience) with inter-rater reliability tracked.

4. RESULTS AND DISCUSSIONS

In this section, the performance of AgentCodeReview is presented with respect to the baselines and the results are discussed in terms of the contribution of each component. The results are first presented for review quality, followed by presentation for repair effectiveness, explainability, then ablation study and implications and limitations.

4.1 CODE REVIEW QUALITY

The multi-agent framework achieved the highest precision, recall and F1 on the code-review benchmark compared to both of the baselines. The results are given in Table 2. The high precision of the results is due to the separation of retrieval, review and explanation that enabled the framework to base its findings on the context of the repository and to limit spurious comments. All the conditions were based on the same underlying model; the improvement was achieved by specialisation, not the model capability.

Table 2: Code review performance across configurations.

Configuration	Precision	Recall	F1
Non-agentic pipeline	0.68	0.61	0.64
Single-agent (review + repair)	0.72	0.66	0.69
AgentCodeReview (proposed)	0.83	0.78	0.80

Source: Authors, (2025).

4.2 BUG REPAIR EFFECTIVENESS

In the automated repair setting, the framework had a Repair Success Rate higher than the baselines for both the single-function defect benchmark and the repository-level issue benchmark, as shown in Table 3. The success of the verification-driven feedback loop was critical: A patch which was not successfully verified was diagnosed and corrected but not emitted, whereas a patch whose diagnosis and correction passed the verification was emitted. The larger relative gain for the repository-level benchmark indicates that the gains of grounding and iteration grow as the tasks get more complex, in line with the reported behaviour in feedback-driven repair [15, 16].

Table 3: Repair success rate across configurations.

Configuration	RSR – single-function (%)	RSR – repository (%)
Non-agentic pipeline	31.4	12.8
Single-agent (review + repair)	38.9	17.6
AgentCodeReview (proposed)	52.7	28.4

Source: Authors, (2025).

4.3 EXPLAINABILITY ASSESSMENT

The Explanation agent produced rationales that were more correct, relevant and clear than the brief rationales produced by the baseline agents, as judged by the human agents. When compared to the five-point scale outlined in Equation (4), the mean Explanation Quality score for AgentCodeReview was raised from 3.1 to 4.2. The greatest

increase was on the dimension of clarity. Each rationale is associated with specific lines and retrieved evidence supporting the rationale, allowing reviewers to check the reasoning of the framework, rather than simply take it on faith – which is a common challenge reported in the use of LLM support for review [3] and [13].

4.4 ABLATION STUDY

The framework was then validated with the key actors being removed to assess the contribution of individual actors. Table 4 shows the F1 and RSR for review, Table 5 shows the EQS for each variant. The removal of the Retrieval agent resulted in all metrics that showed that grounding was a critical design feature being lowered. With the removal of the Verification agent, the repair success was the most dramatic since patches were not emitted when not repaired. The removal of the Explanation agent had negligible impact on review and repair, but did decrease the explanation quality score to the level of the explanation function, and so transparency in the explanation function was the only source for transparency in the entire framework.

Table 4: Ablation study: effect of removing individual agents.

Variant	Review F1	RSR (%)	EQS
Full framework	0.80	52.7	4.2
− Retrieval agent	0.71	44.1	3.6
− Verification agent	0.78	34.5	4.1
− Explanation agent	0.79	51.9	2.4

5. DISCUSSION

The results support the primary hypothesis, that role specialization and explanation and verification-based iteration can help improve coupled review and repair. Three results are highlighted. Firstly, the improvement in the accuracy was obtained despite using the same underlying model, thus signifying that the benefits are associated with the architecture, and not with the power of the model itself, as is the case in general for multi-agent software engineering literature [5, 6, 7]. Second, we discovered that explicit verification was crucial for repair, consistent with previous studies showing that structured feedback is the foundation of reliability of LLM-based repair [15, 16]. Third, the responsibility explanation was elevated to first class status and gave feedback to the developers, thus making it an essential ingredient of trust [3].

There are a number of limitations to these conclusions. The evaluation is conducted on a small number of languages, on function and file level defects, and performance for very large, cross-module changes is an open question. Numbers reported depend on the model and prompting used, and will vary if a different model is used. The multi-agent design has also another disadvantage that the agents are called more than once per change, which results in an extra computational cost for the design (balance between quality and efficiency). Potential threats to validity included that some overlap between pre-training data and the benchmarks occurred, as well as that explanation ratings were subjective by human annotators, although multiple annotators was used to mitigate this issue, and there was some inter-annotator consistency. Lastly, similar to any stand-alone repair system, new patches ought to be inspected by an individual prior to be integrated into the manufacturing code.

6. CONCLUSIONS

An ablation study demonstrated that each of the three key components of the framework (grounding, verification and dedicated explanation) are necessary to the accuracy, reliability and transparency, and changes to the framework were extended to cross-module changes and to additional languages, its computational complexity was lowered, and how this framework can be integrated into a real development workflow was studied in the future.

7. AUTHOR'S CONTRIBUTION

Conceptualization: First Author and Second Author.
Methodology: First Author and Second Author.
Investigation: First Author, Second Author and Third Author.
Discussion of results: First Author, Second Author and Third Author.
Writing – Original Draft: First Author.
Writing – Review and Editing: First Author and Second Author.

Resources: Second Author.
Supervision: Third Author.
Approval of the final text: First Author, Second Author and Third Author.

8. ACKNOWLEDGMENTS

The authors thank the maintainers of the open-source benchmarks used in this study and the annotators who contributed to the explanation-quality assessment.

References

1. X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 8, pp. 1–79, 2024. DOI: 10.1145/3695988.
2. C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in Proc. 45th IEEE/ACM International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1482–1494. DOI: 10.1109/ICSE48619.2023.00129.
3. R. B. Svensson, A. Alami, N. Ernst, et al., "Rethinking code review workflows with LLM assistance: An empirical study," arXiv preprint arXiv:2505.16339, 2025. DOI: 10.48550/arXiv.2505.16339.
4. I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in Proc. 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025. DOI: 10.48550/arXiv.2403.17134.
5. J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in Advances in Neural Information Processing Systems (NeurIPS), vol. 37, 2024, pp. 50528–50652.
6. Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in Proc. 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, 2024, pp. 1592–1604. DOI: 10.1145/3650212.3680384.
7. W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "MAGIS: LLM-based multi-agent framework for GitHub issue resolution," in Advances in Neural Information Processing Systems (NeurIPS), vol. 37, 2024. DOI: 10.48550/arXiv.2403.17927.
8. Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework," arXiv preprint arXiv:2308.08155, 2023. DOI: 10.48550/arXiv.2308.08155.
9. J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "LLaMA-Reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning," in Proc. 34th IEEE International Symposium on Software Reliability Engineering (ISSRE), Florence, Italy, 2023, pp. 647–658. DOI: 10.1109/ISSRE59848.2023.00026.
10. C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation," Information and Software Technology, vol. 175, p. 107523, 2024. DOI: 10.1016/j.infsof.2024.107523.
11. H. Y. Lin, P. Thongtanunam, C. Treude, and W. Charoenwet, "Improving automated code reviews: Learning from experience," in Proc. 21st International Conference on Mining Software Repositories (MSR), 2024, pp. 278–283. DOI: 10.1145/3643991.3644910.
12. M. Vijayvergiya, M. Salawa, I. Budiselić, D. Zheng, P. Lamblin, M. Ivanković, J. Carin, M. Lewko, J. Andonov, G. Petrović, D. Tarlow, P. Maniatis, and R. Just, "AI-assisted assessment of coding practices in modern code review," in Proc. 1st ACM International Conference on AI-Powered Software (AIware), 2024, pp. 85–93. DOI: 10.1145/3664646.3665664.
13. Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang, "Fine-tuning large language models to improve accuracy and comprehensibility of automated code review," ACM Transactions on Software Engineering and Methodology, vol. 34, no. 1, pp. 1–26, 2024. DOI: 10.1145/3695993.
14. K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities," ACM Computing Surveys, vol. 57, no. 2, pp. 1–43, 2024. DOI: 10.1145/3696450.
15. C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in Proc. 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2024, pp. 819–831. DOI: 10.1145/3650212.3680323.
16. X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "ThinkRepair: Self-directed automated program repair," in Proc. 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2024, pp. 1274–1286. DOI: 10.1145/3650212.3680359.
17. C. Lee, C. S. Xia, L. Yang, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via LLM-based multi-agent synergy," arXiv preprint arXiv:2404.17153, 2024. DOI: 10.48550/arXiv.2404.17153.
18. D. Chen, S. Lin, M. Zeng, D. Zan, J.-G. Wang, A. Cheshkov, J. Sun, H. Yu, G. Dong, A. Aliev, et al., "Coder: Issue resolving with multi-agent and task graphs," arXiv preprint arXiv:2406.01304, 2024. DOI: 10.48550/arXiv.2406.01304.

19. D. Arora, A. Sonwane, N. Wadhwa, A. Mehrotra, S. Utpala, R. Bairi, A. Kanade, and N. Natarajan, “MASAI: Modular architecture for software-engineering AI agents,” in Advances in Neural Information Processing Systems (NeurIPS), 2024. DOI: 10.48550/arXiv.2406.11638.
20. I. Jaoua, O. B. Sghaier, and H. A. Sahraoui, “Combining large language models with static analyzers for code review generation,” arXiv preprint arXiv:2502.06633, 2025. DOI: 10.48550/arXiv.2502.06633.
21. J. Kumar and S. Chimalakonda, “Code review automation via multi-task federated LLM – an empirical study,” arXiv preprint arXiv:2412.15676, 2024. DOI: 10.48550/arXiv.2412.15676.
22. Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, “A survey of learning-based automated program repair,” ACM Transactions on Software Engineering and Methodology, vol. 33, no. 2, pp. 1–69, 2023. DOI: 10.1145/3631974.
23. P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive NLP tasks,” in Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020, pp. 9459–9474.
24. J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” in Advances in Neural Information Processing Systems (NeurIPS), vol. 35, 2022, pp. 24824–24837.
25. S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “ReAct: Synergizing reasoning and acting in language models,” in Proc. 11th International Conference on Learning Representations (ICLR), 2023. DOI: 10.48550/arXiv.2210.03629.