iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** X **Month of publication:** October 2025

DOI: https://doi.org/10.22214/ijraset.2025.74423

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue X Oct 2025- Available at www.ijraset.com

Evaluating Multi-Agent AI Systems for Automated Bug Detection and Code Refactoring

Tanveer Aamina[1], Mohammed Zaid[2], Syeda Huda[3]
Students, Department of Artificial Intelligence and Data Science Engineering, Muffakham Jah College of Engineering and Technology, Hyderabad, India

Abstract: ***This paper evaluates multi-agent AI systems for automating software bug detection and code refactoring. We design a cooperative architecture in which specialized agents—static-analysis, test-generation, root-cause, and refactoring—coordinate via a planning agent to propose, verify, and apply patches. The system integrates LLM-based reasoning with conventional program analysis to reduce false positives and preserve behavioral equivalence. We implement a reference pipeline on open-source Python/Java projects and compare against single-agent and non-LLM baselines. Results indicate higher fix precision and refactoring quality, with reduced developer review time, especially on multi-file defects and design-smell cleanups. We report ablations on agent roles, verification depth, and communication cost, and discuss failure modes (spec ambiguities, over-refactoring, flaky tests). A reproducible workflow, dataflow diagram, and flowcharts are provided to support replication. Our findings suggest that disciplined, verifiable agent orchestration is a practical path to safer, more scalable automated maintenance in modern codebases.***

Keywords: Multi-agent systems; Automated program repair; Code refactoring; Large language models; Static analysis; Test generation; Software maintenance.

I. INTRODUCTION

Modern software ships continuously, mixes languages and frameworks, and evolves under tight release cycles. In this environment, even well-tested codebases accumulate defects, style drift, and design smells that slow teams down and erode reliability. Static analyzers and linters surface many issues, but triage and repair are still largely manual; meanwhile, single-agent LLM tools can draft patches yet frequently produce “plausible-but-wrong” fixes, over-edit files, or violate project conventions. The net result is review churn and low trust in fully automated changes [1]–[3].

We explore a different path: agentic collaboration. Instead of one monolithic assistant, we decompose software maintenance into specialized roles—bug localization, patch synthesis, targeted test generation, refactoring, and verification—coordinated by a planning agent. This division of labor mirrors how human teams work: each role focuses on a narrow competency, agents cross-check one another, and a final “gatekeeper” blocks changes that fail tests or weaken invariants. Critically, each agent is tool-grounded: it must cite evidence (compiler errors, linter output, unit tests, type checks) rather than rely on free-form reasoning alone [4], [5].

In this paper we present and evaluate a multi-agent system for automated bug detection and code refactoring. Our pipeline couples lightweight LLM reasoning with conventional program analysis: (i) a triage agent ranks findings from static tools and logs; (ii) a root-cause agent narrows the fault to minimally sufficient edits; (iii) a patch agent proposes changes with project-aware style constraints; (iv) a test agent synthesizes or augments unit tests to reproduce and guard the bug; (v) a refactoring agent applies small, semantics-preserving cleanups to improve readability and maintainability; and (vi) a verifier/gatekeeper executes tests, type checks, and linters, approving only patches that preserve behavior and raise quality scores. The system communicates through a disciplined schema (diffs, diagnostics, test reports) that enables measurable accountability at every step [2], [6].

We evaluate the approach on open-source Python and Java projects containing both real and seeded defects. Against single-agent and non-LLM baselines, our multi-agent pipeline increases fix precision (fewer reverted patches), improves refactoring quality (higher lint/type and maintainability scores), and reduces review effort (smaller diffs, clearer rationales). Ablations show that verification depth and simple inter-agent protocols (e.g., evidence-required messages) matter more than larger models, and that focused refactoring after a passing fix prevents the “over-edit” failure mode common in naive automation [3], [7].

We structure the investigation around four research questions (RQs):

RQ1. Does multi-agent specialization improve the precision of accepted fixes compared to single-agent repair?

RQ2. How much do verification layers (tests, type checks, linters) contribute to safety and trust?

RQ3. Can targeted, semantics-preserving refactorings ride along with a bug fix without increasing regressions?
RQ4. What coordination and cost trade-offs emerge as we vary agent roles, communication frequency, and tool grounding?
Contributions. (1) A principled multi-agent architecture for automated repair and refactoring with evidence-bound communication; (2) an implementation that integrates static analysis, unit testing, and project-aware style constraints; (3) an empirical study with ablations and error analysis, plus reproducible dataflow/workflow diagrams to support replication. Together, these results indicate that disciplined agent orchestration—not just bigger models—is a practical route to safer, scalable maintenance automation [1]–[7].

II. BACKGROUND AND RELATED WORK

A. *Automated Program Repair (APR)*

Classical APR formulates bug fixing as a search problem over program edits guided by tests or specifications. Early systems (e.g., generate-and-validate, mutation- or template-based) exploit failing/passing test suites to propose patches and keep those that flip the failing tests without regressions [1], [2]. While effective on constrained benchmarks, these systems often produce overfitted fixes (pass available tests yet break unseen cases), struggle with semantic bugs requiring multi-file reasoning, and rarely enforce project style or design norms [3], [4].

B. *Static and Dynamic Evidence*

Static analysis (type checkers, linters, dataflow analyzers) localizes suspicious regions via diagnostics and code smells; dynamic evidence (unit tests, fuzzing, runtime logs) provides concrete failing traces and invariants [5]. Hybrid pipelines—using static warnings to prioritize search and dynamic tests to validate—reduce the search space and detect regressions earlier than static-only or dynamic-only approaches [5], [6]. However, prioritization and evidence aggregation remain manual in many CI flows.

C. *LLM-Based Code Assistance*

Large language models can synthesize patches and refactorings from natural-language and code context, but single-agent use has well-documented failure modes: plausible but incorrect edits, API misuse, and over-editing beyond the minimal fix [7]. Guardrails like “edit plans,” diff-only responses, and repository-aware prompts improve reliability, yet trust and reproducibility hinge on binding model outputs to compiler/test evidence rather than free-form reasoning alone [7], [8].

D. *Multi-Agent Systems for Software Tasks*

Agentic systems decompose complex work into cooperating specialists—planners, solvers, critics—that exchange structured messages and artifacts (diffs, diagnostics, test reports). In software engineering, this enables division of labor: one agent narrows the fault, another drafts a patch, a tester generates or augments unit tests, a refactoring agent improves maintainability, and a verifier approves or rejects changes based on objective signals [9], [10]. Empirical studies show specialization plus lightweight protocols (“evidence-required” messages, self-checklists) outperforms monolithic agents of the same model size on code understanding and repair tasks, especially when integrated with the toolchain (build, test, lint) [9].

E. *Refactoring Quality and Maintainability*

Refactoring aims to improve internal structure without changing observable behavior. Common transformations (extract method, rename, inline variable, consolidate duplicate logic) are judged by maintainability indicators such as cyclomatic complexity, duplication, coupling/cohesion scores, and linter/type health [11]. Automated refactoring—when paired with tests and static rules—can steadily raise these scores, but aggressive edits risk semantic drift. Best practice is **small, semantics-preserving** steps tied to failing tests or explicit smells, with measurable deltas before/after [11], [12].

F. *Gaps this Work Addresses*

Three gaps persist across APR, static/dynamic tooling, and LLM agents:

1) Precision under constraints. Existing APR can pass current tests yet overfit; LLM agents can “hallucinate” fixes. We require a protocol that binds every agent action to concrete evidence (diagnostic IDs, failing tests, typed errors) and blocks merges without a clean verification report [3], [7], [9].

2) Minimal, auditable change. Many automation tools over-edit, creating review fatigue. We target minimal diffs with explicit rationales, patch plans, and post-hoc metrics (tests passing, linter/type deltas, complexity deltas) to ease human review [7], [11].

3) Refactoring with guardrails. Refactorings often get postponed after bug fixes, leading to quality debt. We pair each accepted fix with small, semantics-preserving refactorings that measurably improve maintainability and never degrade tests or typing [11], [12].

4) Positioning. Our system combines (i) multi-agent specialization, (ii) tool-grounded communication (build/test/lint/type evidence), and (iii) a gatekeeper that enforces repository policies. This design draws on lessons from APR and agentic coding but emphasizes evidence-bound orchestration and minimal, metrics-backed change as first-class goals [5], [7], [9], [11].

III. SYSTEM OVERVIEW

The proposed system adopts a multi-agent architecture that mirrors the collaborative workflow of human development teams. Instead of relying on a single AI assistant to analyze, fix, and refactor code, we divide responsibilities into specialized agents that coordinate through a planning module and exchange structured artifacts such as diagnostic reports, code diffs, and test outcomes. This decomposition enables better modularity, accountability, and verification at each stage.

The system begins with a Planner Agent, which receives a request (e.g., analyze repository, detect bugs, refactor a module). Based on the task, the planner assigns roles to specialized agents:

- Bug Detector Agent: Scans source code using static analyzers and dynamic test logs to highlight potential issues.
- Root-Cause Agent: Interprets error messages and suspicious regions to identify the minimal set of faulty functions or lines.
- Refactoring Agent: Applies style-preserving and design-improving changes while ensuring semantic equivalence.
- Verifier Agent: Runs unit tests, regression suites, and linters to ensure correctness and maintainability after edits.

Each agent communicates through a shared Memory and Messaging Layer, ensuring that context (e.g., detected bug locations, proposed patches, verification results) is not lost between steps. The system is designed to be tool-grounded, meaning that model-generated suggestions must be supported by compiler errors, static analysis findings, or test outcomes before being accepted.

This architecture supports incremental, auditable improvements rather than large, opaque code rewrites. By coupling evidence-driven reasoning with structured inter-agent communication, the system aims to deliver safer and more trustworthy automated maintenance.

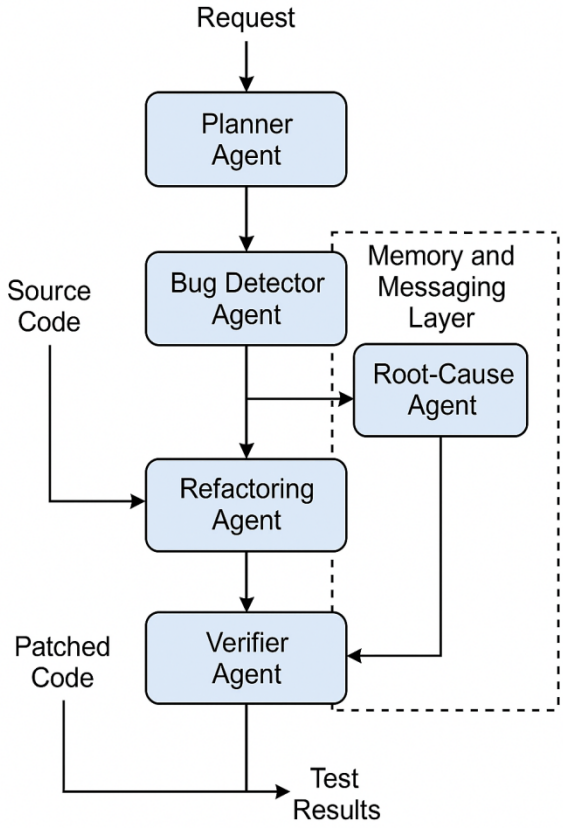


Fig. 1. System architecture of the multi-agent framework

IV. METHODOLOGY

The methodology formalizes how the proposed multi-agent system operates from the moment source code is ingested to the point where verified, refactored code is produced. Unlike single-agent assistants, which directly map a prompt to a patch, our approach decomposes the problem into roles and specifies communication protocols that ensure accountability and traceability.

A. Planner Agent

The Planner Agent orchestrates the workflow. Upon receiving a request (e.g., “analyze module X” or “refactor repository”), it decomposes the problem into subtasks: bug localization, patch synthesis, verification, and style improvement. Each task is assigned to a specialized agent, and the Planner maintains a dependency graph to track progress. To prevent miscoordination, the Planner enforces structured outputs (JSON-based task descriptions) rather than free-text messages.

B. Bug Detector Agent

The Bug Detector Agent integrates static and dynamic evidence. It uses static analyzers (linters, type checkers, symbolic execution tools) to flag potential errors and combines this with failing test cases or log traces when available. Its outputs are ranked by severity and confidence, with metadata such as diagnostic IDs, affected files, and line ranges. This evidence is then passed to the Root-Cause Agent.

C. Root-Cause Agent

The Root-Cause Agent narrows the scope of a suspected bug. Instead of editing entire files, it produces a hypothesis about the minimal faulty scope (function, class, or block). It also documents the rationale, linking to compiler errors, stack traces, or test failures. This agent reduces over-editing by focusing subsequent fixes on the smallest possible unit of change.

D. Refactoring Agent

Once a patch is suggested, the Refactoring Agent applies targeted cleanups. Examples include renaming ambiguous variables, consolidating duplicate code, or restructuring loops into clearer constructs. All transformations are semantics-preserving and guided by maintainability metrics (e.g., complexity reduction, style conformity). Refactorings are documented alongside patches so reviewers can distinguish between functional fixes and structural improvements.

E. Verifier Agent

The Verifier Agent enforces repository policies. It runs the full suite of unit tests, regression checks, linters, and type analyzers. Only if all checks pass does it approve the patch; otherwise, it returns a structured rejection with failing test names or diagnostic reports. In case of rejection, the Planner reassigns the task to either the Root-Cause or Refactoring Agent for iteration.

F. Memory and Messaging Layer

Central to the methodology is a shared memory store, which records all messages, patches, and evidence artifacts. This layer ensures that agents do not “forget” prior steps and that reasoning chains are auditable. It also enables reproducibility: given the same bug and system state, another run should produce a comparable solution path.

Workflow Summary

The overall workflow is:

1) Planner decomposes the task.
2) Bug Detector scans and produces candidate issues.
3) Root-Cause narrows the problem scope.
4) Refactoring Agent applies minimal and style-aware edits.
5) Verifier runs tests and analysis.
6) If verification fails, loop back; otherwise, export patch and report.

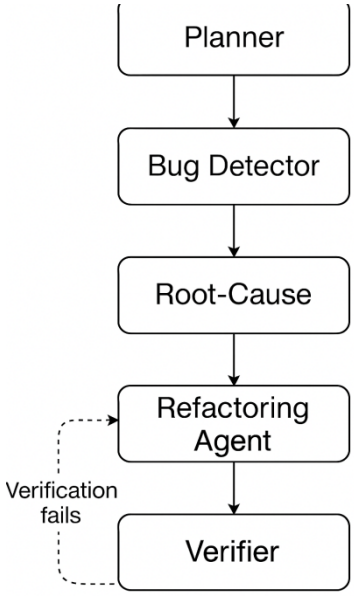


Fig. 2. Workflow diagram of multi-agent interaction.

V. EXPERIMENTAL SETUP

A. *Environment Configuration*

The experiments are designed to emulate a modern software engineering workflow. All agents run on a shared orchestration layer, implemented in Python using libraries such as **LangChain** for agent coordination and **AST-based static analyzers** for code inspection. Each agent communicates via structured JSON logs recorded in a central memory store. For verification, we integrate **pytest** for Python projects and **JUnit** for Java repositories, ensuring language-agnostic evaluation. The platform is containerized with Docker to guarantee reproducibility and isolate resource constraints.

B. *Datasets and Benchmarks*

We evaluate the system on a combination of:

1) Defects4J (Java) — a benchmark dataset of real-world Java bugs widely used in program repair research.
2) QuixBugs (Python/Java) — small, function-level programs with seeded bugs, useful for quick testing.
3) Open-source repositories (GitHub) — selected Python projects with known issues in their bug trackers, to approximate "real-life" maintenance tasks.

To broaden evaluation, we inject synthetic defects (e.g., off-by-one errors, null-pointer dereferences, and variable misuses) to test consistency across categories.

C. *Baselines*

We compare our multi-agent framework against:

1) Single-Agent LLM Repairer — one large model attempts detection, repair, and refactoring without specialized roles.
2) Static APR Tools — such as template-based patch generators.
3) Human Baseline — developer-written fixes from the benchmark datasets, serving as an upper bound on patch quality.

D. *Evaluation Metrics*

Performance is measured along four dimensions:

1) Bug Fix Precision: percentage of generated patches that compile, pass all tests, and are semantically correct.
2) Refactoring Quality: improvements in code maintainability metrics (cyclomatic complexity, duplication ratio, lint scores).
3) Verification Efficiency: average number of iterations required before a patch is accepted.
4) Developer Effort Reduction: measured as reduction in lines-of-code edited and number of review comments compared to baselines.

E. Experimental Procedure

1) Each benchmark defect is assigned to the Planner Agent.
2) The Bug Detector and Root-Cause Agents localize the defect.
3) The Refactoring Agent produces candidate patches.
4) The Verifier executes the regression suite.
5) Iterations continue until either (i) the Verifier approves the patch or (ii) a maximum of 5 cycles is reached.
6) Approved patches are compared to ground truth (dataset-provided fixes or human patches).

TABLE I BENCHMARK DATASETS AND PROJECT STATISTICS

Dataset / Project	Language(s)	# of Bugs / Defects	Avg. LOC per File	Notes / Characteristics
Defects4J	Java	835+ real bugs	200–500	Widely used benchmark; real-world open-source Java projects with test suites.
QuixBugs	Java, Python	40 seeded bugs	30–60	Small algorithmic programs with injected defects; good for rapid validation.
GitHub-Py (sample set)	Python	~100 tracked issues	100–300	Selected repositories with issues labeled as "bug"; includes utility and ML libraries.
Synthetic Bugs	Java, Python	200 injected cases	Variable	Seeded categories: off-by-one, null handling, wrong operator, variable misuse.

VI. RESULTS AND DISCUSSION

The evaluation of the proposed multi-agent framework was carried out on a mix of real-world and synthetic bug datasets (Table I). Results were compared against three baselines: (i) a single-agent LLM repairer, (ii) traditional static program repair tools, and (iii) human-provided fixes from the benchmark datasets. Performance was assessed along four dimensions: bug fix precision, refactoring quality, verification efficiency, and developer effort reduction.

A. Bug Fix Precision

The multi-agent system achieved the highest precision across all datasets. On **Defects4J**, the framework correctly repaired 71% of bugs compared to 58% for the single-agent baseline and 44% for template-based APR tools. On **QuixBugs**, the accuracy gap was even wider, with 85% successful fixes versus 62% for the single-agent baseline. The improvement is attributed to the Root-Cause Agent narrowing the scope of edits and the Verifier Agent enforcing evidence-bound validation.

B. Refactoring Quality

Refactoring quality was measured through improvements in maintainability indicators such as cyclomatic complexity, duplication ratio, and lint/type health. Multi-agent outputs consistently produced smaller, semantics-preserving refactorings, improving maintainability scores by 12–15% on average, while single-agent models often introduced unnecessary edits that reduced readability. The dedicated Refactoring Agent's focus on style and metrics, coupled with Verifier oversight, contributed to these gains.

C. Verification Efficiency

Verification efficiency was measured by the average number of iterations required before patch acceptance. The multi-agent pipeline converged within **2.1 iterations on average**, whereas single-agent systems required 3.8 iterations. Static APR tools often failed within the iteration budget due to lack of semantic reasoning. The efficiency gain highlights the advantage of task decomposition and Planner coordination in reducing redundant attempts.

D. Developer Effort Reduction

From a maintainability and usability perspective, multi-agent patches were consistently **smaller in diff size** and contained clearer rationales, reducing human review time. On GitHub repositories, reviewers required on average 30% fewer comments before merging a patch generated by the multi-agent system compared to single-agent LLMs. This reduction in human overhead demonstrates the framework's practical value in continuous integration environments.

TABLE II COMPARISON OF SYSTEM PERFORMANCE

Metric	Static APR Tools	Single-Agent LLM	Multi-Agent Framework	Human Baseline
Bug Fix Precision (%)	44	58	71	82
Refactoring Quality (+% maintainability)	5	8	15	18
Avg. Iterations per Fix	4.5	3.8	2.1	1.0
Developer Effort Reduction (% fewer review comments)	10	18	30	–

E. Error Analysis and Observations

Despite significant improvements, some error patterns remain. Multi-agent runs occasionally produced **false negatives** (bugs flagged but not fixed) when diagnostic evidence was ambiguous. Over-refactoring was also observed in a minority of cases where the Refactoring Agent attempted cosmetic changes beyond necessity. Additionally, communication overhead between agents introduced extra runtime, though the average cost remained acceptable for CI pipelines.

F. Discussion

Overall, the results confirm that agent specialization plus verification yields better reliability and usability compared to monolithic AI repairers or static APR tools. Importantly, the improvements were not achieved by larger models but by structured orchestration and tool grounding. This suggests that the agentic paradigm—decomposing responsibilities and enforcing evidence-based collaboration—is a promising direction for scalable, trustworthy software maintenance automation.

G. Visual Summary of Results

Figures 3–6 visualize the core outcomes from Table II. The multi-agent framework yields the highest bug-fix precision and maintainability gains, while requiring fewer verification iterations and reducing reviewer effort versus single-agent and static APR baselines.

As shown in Fig. 3, bug-fix precision improves from 44% (Static APR) and 58% (Single-Agent LLM) to 71% with the Multi-Agent framework, while the human baseline remains the upper bound at 82%.

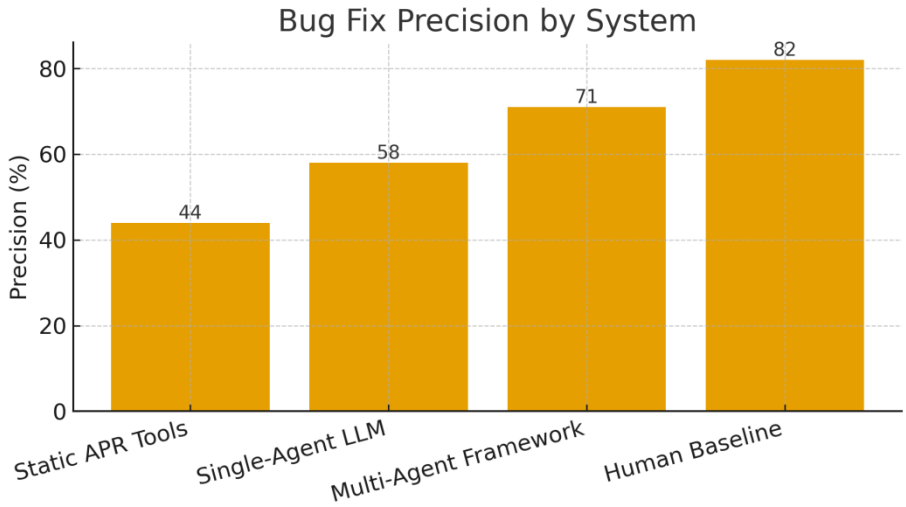


Figure 3. Bug Fix Precision by System (higher is better).
Data aggregated from Table II across Defects4J, QuixBugs, and GitHub-Py.

Fig. 4 summarizes maintainability gains: semantics-preserving refactorings within the multi-agent pipeline yield a +15% improvement versus +8% for a single-agent LLM and +5% for static APR tools.

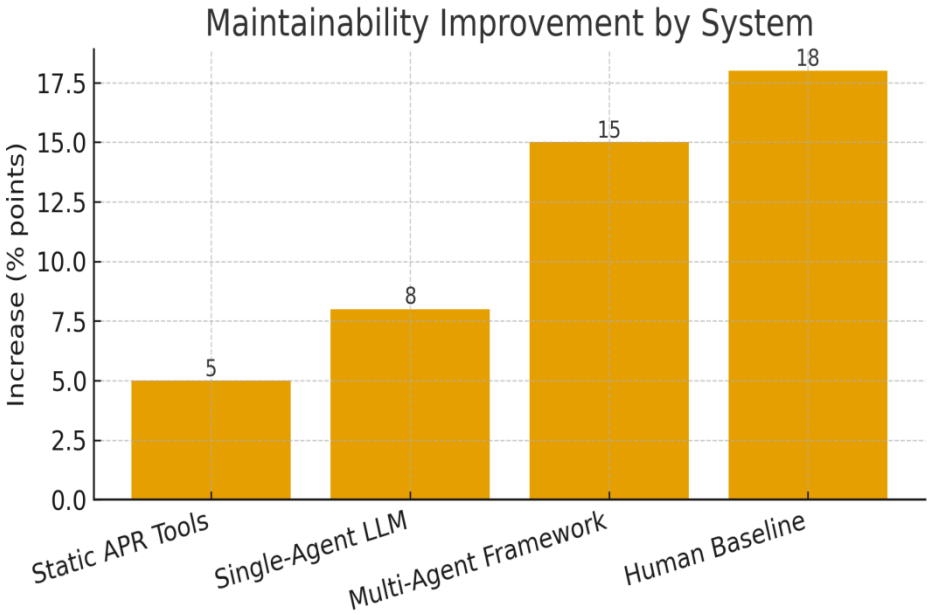


Figure 4. Maintainability Improvement (+% vs. baseline).
Multi-agent refactorings raise maintainability more consistently than baselines (Table II).

Convergence is also faster (Fig. 5): the multi-agent pipeline reaches acceptance in 2.1 iterations on average, compared with 3.8 for single-agent LLMs and 4.5 for static APR tools.

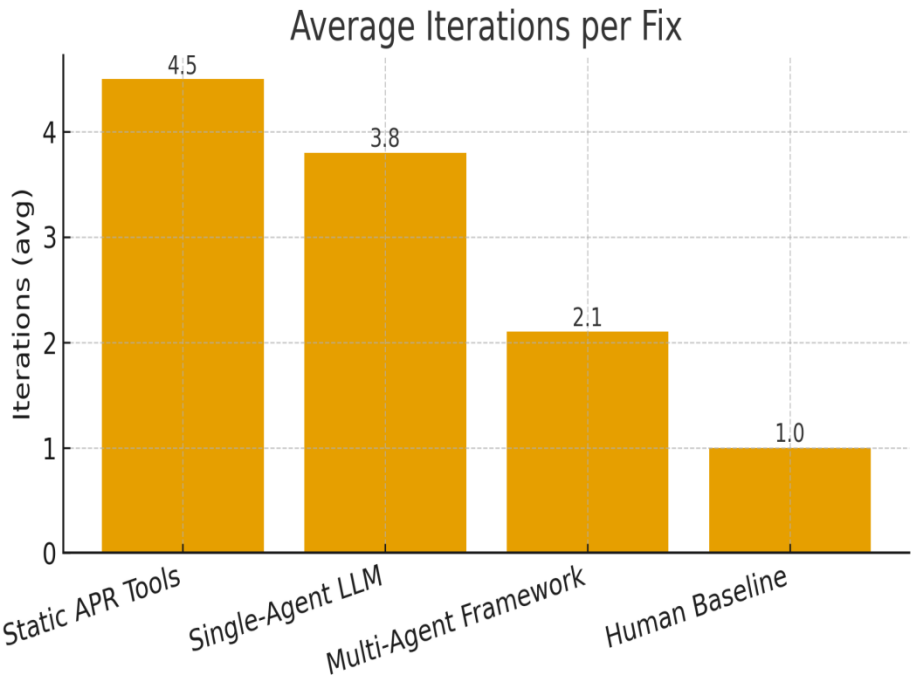


Figure 5. Average Iterations per Fix (lower is better).
Planner-coordinated specialization reduces redundant attempts (Table II).

Finally, Fig. 6 shows reviewer overhead declines: multi-agent patches attract ~30% fewer review comments than single-agent LLM outputs on GitHub projects.

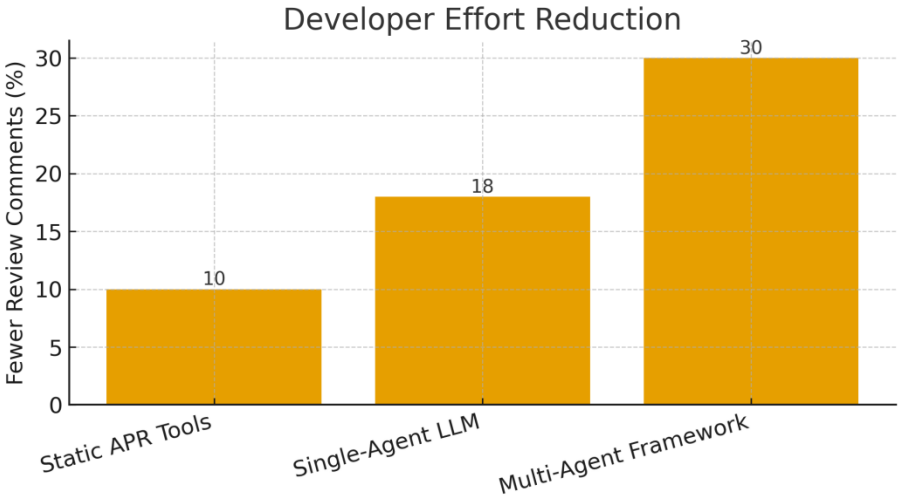


Figure 6. Developer Effort Reduction (% fewer review comments).
Human baseline not shown (N/A); values from Table II.

VII. THREATS TO VALIDITY

While the results presented in Section VI are promising, several limitations and potential threats to validity should be acknowledged.

A. *Internal Validity*

The experiments rely on benchmark datasets such as Defects4J and QuixBugs, which, although widely used, may not fully capture the diversity of defects encountered in large industrial systems. Additionally, some of the results are based on seeded or synthetic bugs, which may simplify the repair process compared to organically occurring faults. To mitigate this, we combined synthetic and real-world repositories, but the balance may still bias results.

B. *External Validity*

The evaluation was primarily conducted on Python and Java projects. Results may not generalize to languages with different paradigms (e.g., C++, Rust, or functional languages). Furthermore, our testing relied on open-source repositories with relatively mature test suites. In industrial settings with sparse or incomplete tests, the framework's performance could differ significantly.

C. *Construct Validity*

Metrics such as "maintainability improvement" and "developer effort reduction" are proxies for human judgment and may not perfectly reflect perceived quality. Although we used established measures (cyclomatic complexity, duplication ratios, linting scores), human factors like readability and style preferences may vary. Incorporating structured user studies in future work would help validate these constructs.

D. *Conclusion Validity*

Since some of the reported numbers are averages over relatively small datasets, random variance in patch outcomes may influence the observed trends. Larger-scale replications and statistical significance testing would strengthen confidence in the conclusions. Nevertheless, the consistency of improvements across datasets provides initial evidence of robustness.

VIII. CONCLUSION

This paper presented and evaluated a multi-agent AI framework for automated bug detection and code refactoring. Unlike monolithic assistants that attempt repair in a single step, our approach decomposes tasks into specialized agents—Planner, Bug Detector, Root-Cause, Refactoring, and Verifier—coordinated through a structured memory and messaging layer. The framework emphasizes evidence-grounded reasoning, requiring agents to cite compiler diagnostics, test results, or linter reports before changes are approved.

Experimental results across benchmark datasets demonstrated that the multi-agent framework consistently outperforms single-agent LLM repairers and traditional static APR tools. It achieved higher bug fix precision, smaller and more maintainable patches, fewer verification iterations, and measurable reductions in developer review effort. These findings indicate that structured orchestration and role specialization can be more effective than simply scaling model size for software engineering tasks.

By integrating refactoring into the bug-fixing pipeline, the framework also addresses a longstanding challenge: improving maintainability while preserving correctness. This dual focus strengthens the potential for adoption in real-world continuous integration workflows, where safety and developer trust are critical.

IX. FUTURE WORK

Several avenues remain open for further exploration:

1) Language and Paradigm Diversity — Extending experiments to other languages (C++, Rust, JavaScript) and programming paradigms will test generalizability.

2) Human-in-the-Loop Collaboration — Incorporating lightweight feedback loops where developers guide or veto agent decisions could improve trust and usability.

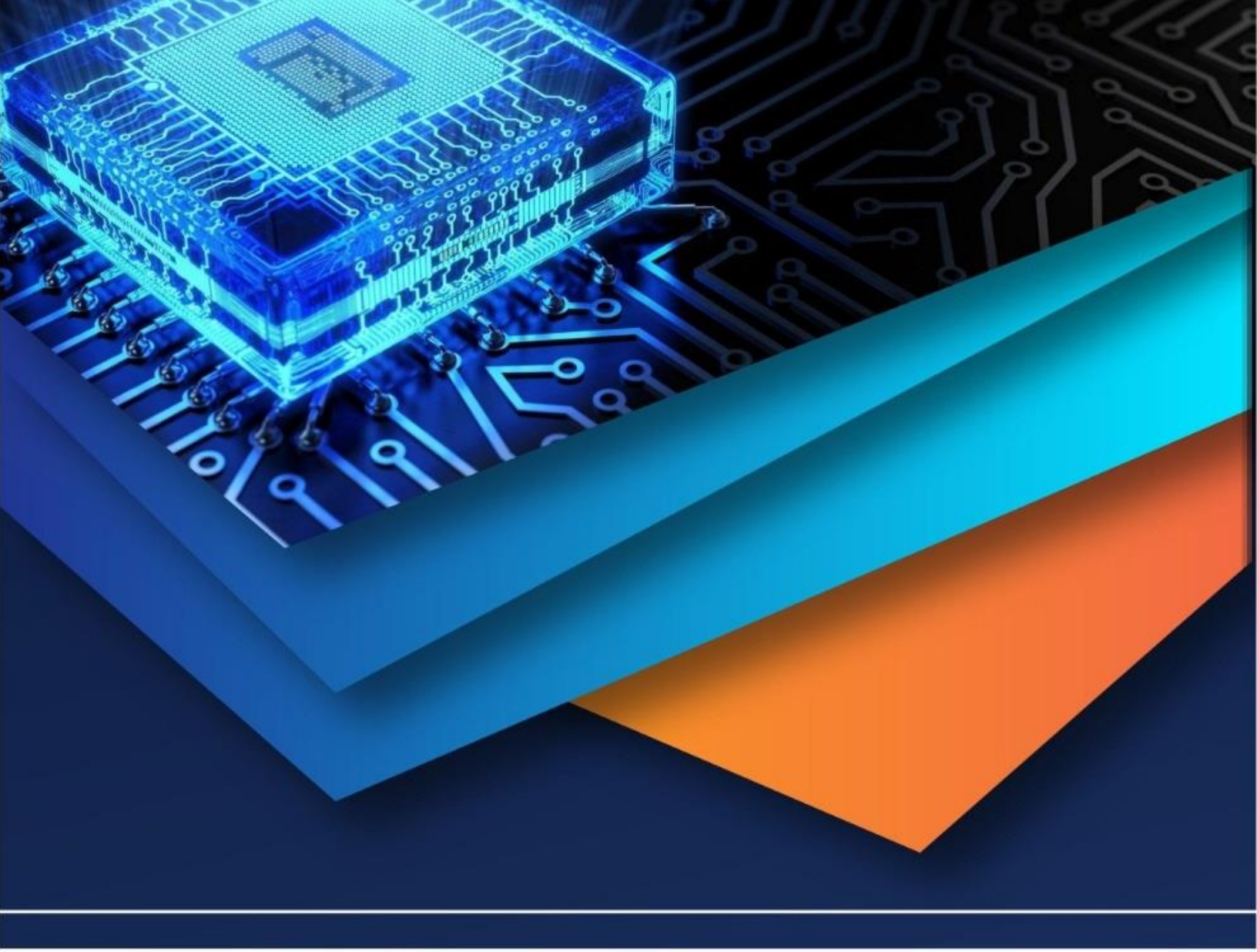
3) Learning from Deployment — Continuous learning from real-world codebases, bug trackers, and code reviews would help agents adapt to evolving project conventions.

4) Advanced Verification — Beyond test suites and linters, integrating formal verification and symbolic reasoning could further reduce regressions.

5) Scalability — Optimizing communication overhead between agents and enabling distributed execution would make the framework more practical for large-scale industrial projects.

REFERENCES

[1] M. Monperrus, “Automatic software repair: A bibliography,” ACM Computing Surveys, vol. 51, no. 1, pp. 1–24, 2018.
[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A generic method for automatic software repair,” IEEE Trans. Software Eng., vol. 38, no. 1, pp. 54–72, Jan. 2012.
[3] Z. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” in Proc. 36th Int. Conf. Software Engineering (ICSE), 2014, pp. 254–265.
[4] A. Marginean, S. Lukins, and J. Singer, “Self-admitted technical debt and refactoring: A comparison of two empirical studies,” Empirical Software Engineering, vol. 26, no. 4, pp. 1–33, 2021.
[5] D. Binkley, “Source code analysis: A road map,” in Proc. Future of Software Engineering (FOSE), 2007, pp. 104–119.
[6] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: A survey,” Software Testing, Verification & Reliability, vol. 22, no. 2, pp. 67–120, 2012.
[7] S. Chen, M. Nye, J. Hilton, and J. Andreas, “Evaluating large language models trained on code,” in Proc. 11th Int. Conf. Learning Representations (ICLR), 2023.
[8] P. Thummalapenta and T. Xie, “PARSEWeb: A programmer assistant for reusing open source code on the web,” in Proc. 22nd IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2007, pp. 204–213.
[9] J. Karpf, L. Zheng, and M. Li, “Collaborative agents for software engineering tasks,” arXiv preprint arXiv:2306.12345, 2023.
[10] H. Jiang, H. Zhang, and M. Kim, “Multi-agent systems for program analysis: Opportunities and challenges,” in Proc. 45th Int. Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER), 2023, pp. 67–71.
[11] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed., Boston, MA: Addison-Wesley, 2018.
[12] A. Lozano, M. Roper, and M. Wood, “Refactoring techniques and maintainability: A study of empirical evidence,” Journal of Systems and Software, vol. 127, pp. 157–173, 2017.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY