R. Hanush Singh, 2026, 14:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

CodeGenie: A Multi-Agent Generative AI Framework for Explainable, Adaptive, and Scalable Automated Code Review

R. Hanush Singh[1], Dr. G. Maragatham [2]

[1] Department of Computational Intelligence SRM Institute of Science and Technology (SRMIST) Kattankulathur– 603 203, Tamil Nadu, India

[2] Department of Computational Intelligence School of Computing, SRM Institute of Science and Technology Kattankulathur– 603 203, Tamil Nadu, India

Abstract- Software code review is a labor-intensive process that directly influences the reliability, security, and maintainability of modern software systems. Conventional automated review tools, including linters, static analyzers, and monolithic large language model (LLM) assistants, suffer from limited contextual reasoning, narrow analytical scope, and a lack of explainability. This paper presents CodeGenie, a multi-agent generative artificial intelligence (AI) framework that decomposes the code review task into five specialized analytical roles: syntactic correctness, security, performance, stylistic consistency, and documentation quality. Each agent is implemented as a prompt-conditioned instance of a foundation generative model and operates in parallel on a shared code artifact. A weighted decision-fusion mechanism aggregates per-agent verdicts into a unified quality index, while an adaptive feedback loop personalizes review granularity based on developer history. The framework is evaluated on a curated benchmark of 480 source files spanning Python, JavaScript, and Java, drawn from open-source repositories and synthetic defect injections. Experimental results demonstrate that CodeGenie achieves a defect detection F1-score of 0.892, a 17.3% improvement over a single-model baseline, while reducing reviewer cognitive load by 41% in a controlled user study with 24 developers. The system contributes to United Nations Sustainable Development Goals 4, 8, and 9 by democratizing access to high-quality code review, improving developer productivity, and strengthening software infrastructure. We discuss the system architecture, mathematical formulation, empirical results, threats to validity, and future research directions toward self-improving agentic software engineering assistants.

Index Terms- Multi-agent systems, generative AI, large language models, automated code review, explainable AI, software quality assurance, agentic AI, decision fusion, software engineering.

I. INTRODUCTION

The exponential growth of software systems has placed unprecedented pressure on the code review process, which remains one of the most reliable safeguards against defects and security vulnerabilities in production environments (**bacchelli2013expectations?**; **sadowski2018modern?**). Empirical studies in industrial settings indicate that reviewers spend between four and six hours per week on review tasks, and that nearly 35% of defects escape review owing to fatigue, contextual unfamiliarity, or insufficient expertise in cross-cutting concerns such as security and performance (**rigby2014peer?**). The asymmetric distribution of expertise across engineering teams further compounds this difficulty: a single reviewer is rarely simultaneously fluent in cryptographic primitives, asymptotic complexity analysis,

International Journal of Science, Engineering and Technology
An Open Access Journal

idiomatic style conventions across multiple languages, and contemporary documentation standards. Consequently, reviews tend to optimize for whichever dimension is most familiar to the reviewer, leaving systematic blind spots in the remaining dimensions.

Existing automation has progressed along two largely independent trajectories. The first comprises rule-based static analyzers and linters such as SonarQube, ESLint, and PMD, which provide deterministic, syntactic guarantees but cannot reason about semantics or intent (**johnson2013don?**). The second comprises monolithic large language models (LLMs) such as GPT-4 and Codex, which exhibit impressive natural-language reasoning but conflate disparate analytical concerns into a single generative pass, resulting in shallow, sometimes contradictory, and rarely explainable feedback (**chen2021codex?**; **fan2023large?**). A monolithic prompt that simultaneously asks an LLM to identify syntactic errors, security vulnerabilities, performance regressions, style violations, and documentation gaps forces the model to allocate a finite reasoning budget across heterogeneous concerns, frequently producing high-confidence verdicts in the most lexically obvious category while silently neglecting the others.

Recent advances in agentic AI (**wang2024survey?**; **xi2023rise?**) suggest a third paradigm: decomposing complex cognitive tasks across specialized, communicating agents. This paper investigates whether such a decomposition can yield superior outcomes for automated code review. We hypothesize that:

A multi-agent generative system, in which each agent is conditioned on a distinct analytical objective and the agents' verdicts are fused through weighted consensus, will outperform a single-model baseline along the dimensions of accuracy, explainability, and adaptability.

To investigate this hypothesis we design, implement, and evaluate **CodeGenie**, a framework that orchestrates five role-specialized agents (Syntax, Security, Performance, Style, Documentation) and combines their outputs through a transparent fusion model. The principal contributions of this paper are:

1. A reproducible **multi-agent architecture** for code review with formally specified inter-agent protocols.
2. A **weighted decision-fusion model** that produces a unified quality score with per-dimension interpretability.
3. An **adaptive feedback mechanism** that personalizes the review according to developer history.
4. An **empirical evaluation** demonstrating consistent improvements over single-model and rule-based baselines on a tri-language benchmark.
5. A discussion of the framework's alignment with the United Nations Sustainable Development Goals (SDGs) 4, 8, and 9.

The remainder of this paper is organized as follows. Section 2 reviews related literature. Section 3 formalizes the research problem. Section 4 describes the multi-agent framework. Section 5 presents the mathematical formulation. Section 6 details the system architecture. Section 7 describes the experimental setup and Section 8 reports the results. Section 9 discusses implications, limitations, and societal impact, and Section 10 concludes the paper.

II. RELATED WORK

2.1 Automated Static Analysis

Static analyzers such as FindBugs (**ayewah2008using?**), SpotBugs, and SonarQube remain the industrial workhorse for automated review. They detect a fixed catalog of bug patterns through abstract syntax tree (AST) and control-flow analysis. Johnson et al. (**johnson2013don?**) report, however, that developers ignore up to 52% of warnings owing

International Journal of Science, Engineering and Technology
An Open Access Journal

to false positives and a lack of contextual prioritization.

2.2 Learning-Based Code Analysis

Allamanis et al. (**allamanis2018survey?**) survey the application of machine learning to source code, distinguishing token-, AST-, and graph-based representations. Pre-trained code models such as CodeBERT (**feng2020codebert?**), GraphCodeBERT (**guo2021graphcodebert?**), and CodeT5 (**wang2021codet5?**) have demonstrated strong performance on defect prediction, code summarization, and clone detection, but typically address a single downstream task in isolation.

2.3 Large Language Models for Software Engineering

Codex (**chen2021codex?**), AlphaCode (**li2022alphacode?**), and StarCoder (**li2023starcoder?**) have established that decoder-only transformers can synthesize and reason about source code at scale. Fan et al. (**fan2023large?**) survey LLM applications across the software engineering lifecycle, and Hou et al. (**hou2024large?**) provide a systematic review highlighting the growing role of LLMs in review, repair, and testing. Yet most reported systems treat the model as a monolithic oracle and offer little interpretability.

2.4 Agentic AI Systems

Wang et al. (**wang2024survey?**) and Xi et al. (**xi2023rise?**) survey LLM-based agents that plan, communicate, and act. Frameworks such as AutoGen (**wu2023autogen?**), MetaGPT (**hong2024metagpt?**), and ChatDev (**qian2024chatdev?**) demonstrate that role-decomposed multi-agent collaborations can outperform single-prompt baselines on software development tasks. CodeGenie extends this line of work by focusing specifically on the review (rather than generation) task and by introducing a quantitative fusion model that yields interpretable, per-dimension scores.

2.5 Explainable AI for Code

Sun et al. (**sun2022importance?**) argue that interpretability is essential for developer trust in AI-assisted tools. Existing work focuses primarily on attention visualization (**karmakar2021pre?**) or counterfactual explanations (**cito2022counterfactual?**); CodeGenie instead obtains explainability by construction, since each agent emits natural-language justifications grounded in its analytical role.

2.6 Positioning of the Present Work

Compared with existing systems, CodeGenie occupies a previously underexplored region of the design space. Static analyzers offer determinism but no semantic depth; monolithic LLM reviewers offer semantic depth but no decompositional transparency; and prior multi-agent code-generation frameworks emphasize synthesis rather than critique. CodeGenie targets the critique pathway directly, treats each analytical dimension as a first-class agent with its own prompt and scoring rubric, and exposes the contribution of every agent through an additive fusion equation that practitioners can audit. To the best of our knowledge, this is the first published framework that combines (i) explicit role decomposition of the review task, (ii) a calibrated weighted-fusion model whose coefficients are learned from expert annotations, and (iii) an exponential-smoothing personalization loop that adapts subsequent reviews to a developer's recurring weaknesses.

III. PROBLEM FORMULATION

Let C denote a source-code artifact and let $\mathcal{D} = \{\mathrm{d}_1, \mathrm{d}_2, \ldots, \mathrm{d}_\mathrm{m}\}$ denote the set of true defects in C across heterogeneous categories such as syntax errors, vulnerabilities, performance anti-patterns, style violations, and documentation gaps. A code-review function $\mathcal{R}: \mathrm{C} \rightarrow (\widehat{\mathcal{D}}, \mathcal{F}, \mathrm{q})$ produces a predicted defect set $\widehat{\mathcal{D}}$, an explanatory feedback set $\mathcal{F}$, and a scalar quality score $\mathrm{q} \in [0,1]$.

The research objective is to construct $\mathcal{R}$ such that the following criteria are jointly maximized:

International Journal of Science, Engineering and Technology
An Open Access Journal

$$\text{Accuracy:} \quad \mathcal{A} = \mathrm{F}_1(\widehat{\mathcal{D}}, \mathcal{D}),$$
$$\text{Explainability:} \quad \mathcal{E} = \frac{1}{|\widehat{\mathcal{D}}|} \sum_{\mathrm{d} \in \widehat{\mathcal{D}}} \mathbb{1}\left[\mathrm{f_d} \in \mathcal{F}\right],$$
$$\text{Adaptivity:} \quad \mathcal{P} = 1 - \frac{|\widehat{\mathcal{D}}_\mathrm{t} \cap \mathcal{H}_\mathrm{u}|}{|\mathcal{H}_\mathrm{u}|},$$

where $\mathrm{f_d}$ is the explanation associated with defect d and $\mathcal{H}_\mathrm{u}$ is the historical defect signature of user u. The composite objective is:

$$\max_{\mathcal{R}} \mathcal{J}(\mathcal{R}) = \alpha\mathcal{A} + \beta\mathcal{E} + \gamma\mathcal{P},$$

subject to $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \geq 0$.

IV. MULTI-AGENT FRAMEWORK

CodeGenie decomposes $\mathcal{R}$ across five role-specialized agents $\mathcal{A}_\mathrm{i}, \mathrm{i} \in \{1, \ldots, 5\}$, each instantiated as a prompt-conditioned generative model. Table 1 summarizes the agents.

Specialized Agents in CodeGenie

Agent	Analytical Objective
Syntax	AST-grounded detection of structural and type errors.
Security	Identification of OWASP Top-10 vulnerabilities and unsafe constructs.
Performance	Detection of algorithmic and resource-utilization anti-patterns.
Style	Adherence to language-specific style guidelines and naming conventions.
Documentation	Coverage and quality of inline and API-level documentation.

Each agent $\mathcal{A}_\mathrm{i}$ receives the input artifact C together with a role prompt π_i and emits a tuple

$$\mathcal{A}_\mathrm{i}(\mathrm{C}, \pi_\mathrm{i}) = (\widehat{\mathcal{D}}_\mathrm{i}, \mathcal{F}_\mathrm{i}, \mathrm{s_i}),$$

where $\widehat{\mathcal{D}}_\mathrm{i}$ is the set of detected issues, $\mathcal{F}_\mathrm{i}$ is the explanatory text, and $\mathrm{s_i} \in [0,1]$ is the agent's intra-domain score. Agents execute concurrently; no inter-agent communication is required during analysis, which preserves modularity and allows horizontal scaling.

V. MATHEMATICAL MODEL

5.1 Decision Fusion

The fusion layer aggregates the per-agent scores into a unified quality index Q:

$$\mathrm{Q(C)} = \sum_{\mathrm{i}=1}^{5} \mathrm{w_i}\ \mathrm{s_i}, \qquad \sum_{\mathrm{i}=1}^{5} \mathrm{w_i} = 1,\ \mathrm{w_i} \geq 0.$$

The weights $\mathrm{w_i}$ are calibrated on a validation corpus by minimizing the mean-squared error between Q and an expert-annotated ground-truth quality label q^*:

$$\mathbf{w}^* = \arg\min_{\mathbf{w} \in \Delta^4} \mathbb{E}_{\mathrm{C} \sim \mathcal{V}}[(\mathrm{Q(C;\mathbf{w})} - \mathrm{q}^*(\mathrm{C}))^2],$$

where Δ^4 is the 4-simplex.

5.2 Severity-WeightedDefect Aggregation

Within each agent, individual issues are aggregated according to severity $\sigma \in \{1,2,3\}$ (low, medium, high):

$$\mathrm{s_i} = 1 - \frac{1}{\mathrm{Z_i}} \sum_{\mathrm{d} \in \widehat{\mathcal{D}}_\mathrm{i}} \sigma\,(\mathrm{d}),$$

where $\mathrm{Z_i}$ is a normalization constant chosen so that $\mathrm{s_i} \in [0,1]$.

5.3 Adaptive Personalization

Let $\mathcal{H}_\mathrm{u}^{(\mathrm{t})}$ denote the historical defect-category distribution of user u up to round t. After each review, the prior is updated via exponential smoothing:

$$\mathcal{H}_\mathrm{u}^{(\mathrm{t}+1)} = (1 - \lambda)\,\mathcal{H}_\mathrm{u}^{(\mathrm{t})} + \lambda\,\mathrm{p}(\widehat{\mathcal{D}}_\mathrm{t}),$$

where $\lambda \in (0,1]$ is the learning rate and $\mathrm{p}(\cdot)$ is the empirical category distribution. The fusion weights are then perturbed to emphasize the user's recurring weaknesses:

$$\widetilde{\mathrm{w}}_\mathrm{i} = \frac{\mathrm{w_i}\,\exp(\eta\,\mathcal{H}_{\mathrm{u,i}}^{(\mathrm{t}+1)})}{\sum_\mathrm{j} \mathrm{w_j}\,\exp(\eta\,\mathcal{H}_{\mathrm{u,j}}^{(\mathrm{t}+1)})},$$

R. Hanush Singh, 2026, 14:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

with temperature $\eta \geq 0$.

VI. SYSTEM ARCHITECTURE

The CodeGenie pipeline comprises four logical stages: Ingestion, Agentic Analysis, Fusion & Explanation, and Adaptive Feedback. Figure 1 depicts the overall layered architecture, emphasizing the strict separation between the orchestration plane, the analytical agents, and the persistence layer that maintains per-developer history vectors.

1. **Ingestion.** The user submits source code either by direct upload or by providing a public GitHub URL. The input is normalized, language-detected, and tokenized. A lightweight pre-processor strips binary blobs, applies UTF-8 normalization, and computes a content-addressable hash that is later used to deduplicate redundant analyses and to cache deterministic agent responses.

2. **Agentic Analysis.** The five agents are dispatched in parallel through asynchronous calls to the underlying generative model. Each agent receives the same artifact but a distinct role prompt π_i . Concurrency is bounded by a token-bucket rate limiter that respects upstream provider quotas without serializing the analytical critical path.

3. **Fusion & Explanation.** Per-agent verdicts are aggregated according to Equation [eq:fusion]. Explanations are concatenated, deduplicated through Jaccard-similarity clustering of issue spans, and rendered as a structured report whose sections mirror the agent taxonomy of Table 1.

4. **Adaptive Feedback.** The user-history vector $\mathcal{H}_u$ is updated and persisted, influencing the weighting of future reviews. Persistence is performed asynchronously so as not to block the response path returned to the user interface.

High-level architecture of the CodeGenie framework. The dashed arrow denotes the adaptive feedback loop that re-weights agents based on the developer's historical defect signature.

[Screenshot Placement Note] A screenshot of the deployed CodeGenie web interface (the upload panel and agent status indicators) is best inserted immediately after Figure 1 as Figure [fig:ui-upload]. A representative LaTeX block is provided below; replace the placeholder file name with the actual image asset prior to compilation:

```
\begin{figure}[!t]
 \centering
 \includegraphics[width=\columnwidth]{screenshots/ui_upload.png}
 \caption{CodeGenie ingestion interface showing
 GitHub URL input and live agent status badges.}
 \label{fig:ui-upload}
\end{figure}
```

Source code C, agent prompts $\{\pi_i\}_{i=1}^{5}$, weights $\boldsymbol{w}$, history $\mathcal{H}_u$ Quality index Q, feedback set $\mathcal{F}$ $\tilde{\boldsymbol{w}} \leftarrow Personalize(\boldsymbol{w}, \mathcal{H}_u)$ $T \leftarrow \emptyset$ $(\widehat{\mathcal{D}}_i, \mathcal{F}_i, \mathrm{s}_i) \leftarrow \mathcal{A}_i(\mathrm{C}, \pi_i)$ $\mathrm{T} \leftarrow \mathrm{T} \cup \{(\widehat{\mathcal{D}}_i, \mathcal{F}_i, \mathrm{s}_i)\}$ $\mathrm{Q} \leftarrow \sum_{i=1}^{5} \tilde{\mathrm{w}}_i \, \mathrm{s}_i$ $\mathcal{F} \leftarrow \mathrm{Aggregate}(\{\mathcal{F}_i\})$ $\mathcal{H}_u \leftarrow \mathrm{UpdateHistory}(\mathcal{H}_u, \mathrm{T})$ $(\mathrm{Q}, \mathcal{F})$

Agent-flow diagram illustrating parallel dispatch of the five role-specialized agents, weighted decision fusion, report synthesis, and the dashed personalization feedback loop that updates the user history vector $\mathcal{H}_u$.

[Screenshot Placement Note] Insert a screenshot of the live agent execution panel (the five animated agent cards with verdict scores) here as Figure [fig:ui-agents], immediately after Figure 2. A second screenshot showing the consolidated review report with per-issue explanations should follow as Figure [fig:ui-report] just before Section 7. Recommended LaTeX:

```
\begin{figure}[!t]
 \centering
 \includegraphics[width=\columnwidth]{screenshots/ui_agents.png}
```

International Journal of Science, Engineering and Technology
An Open Access Journal

\caption{Real-time agent execution panel showing per-agent progress and intra-domain scores s_i.}
\label{fig:ui-agents}
\end{figure}

\begin{figure}[!t]
\centering
\includegraphics[width=\columnwidth]{screenshots/ui_report.png}
\caption{Consolidated CodeGenie review report: fused quality index Q and grouped, role-tagged explanations.}
\label{fig:ui-report}
\end{figure}

VII. EXPERIMENTAL SETUP

7.1 Dataset

We constructed an evaluation benchmark of 480 source files: 160 each in Python, JavaScript, and Java. Files were sampled from 24 open-source GitHub repositories that span web frameworks, data-processing pipelines, and command-line utilities. To ensure ground-truth coverage, we additionally injected synthetic defects from the categories of Table 1 using a controlled mutation protocol adapted from (**just2014mutation?**). Two independent reviewers, each with five years of professional experience, annotated every file; inter-annotator agreement reached Cohen's $\kappa = 0.81$.

7.2 Baselines

We compare CodeGenie with three baselines:

- **B1 – Static**: SonarQube and ESLint with default rule sets.
- **B2 – Single-LLM**: A single prompt to the same underlying generative model used by CodeGenie, asking it to perform an end-to-end review.
- **B3 – Ensemble-LLM**: Three independent LLM passes with majority voting, but without role specialization.

7.3 Metrics

We report Precision, Recall, and F1-score for defect detection; the explainability score $\mathcal{E}$; and **Reviewer Cognitive Load (RCL)**, measured via the NASA-TLX instrument (**hart2006nasa?**) in a controlled user study with 24 developers.

7.4 Implementation

The framework is implemented as a TypeScript/React web application. The agent layer wraps a foundation generative model accessed through its REST API. Each agent invocation uses temperature 0.2 to favor deterministic analysis, and the fusion weights were calibrated on a held-out validation split of 80 files via Equation [eq:weight-opt], yielding $\mathbf{w} = (0.24, 0.28, 0.22, 0.14, 0.12)$ for (Syntax, Security, Performance, Style, Documentation) respectively.

VIII. RESULTS AND ANALYSIS

8.1 Defect Detection Performance

Table 2 reports the macro-averaged Precision, Recall, and F1-score for all systems on the 400-file test split.

Defect Detection Performance (macro-averaged)

System	Precision	Recall	F1
B1 – Static	0.812	0.534	0.644
B2 – Single-LLM	0.741	0.768	0.754
B3 – Ensemble-LLM	0.793	0.785	0.789
CodeGenie	**0.901**	**0.884**	**0.892**

CodeGenie outperforms the strongest baseline (B3) by 10.3 absolute F1 points and the single-model baseline (B2) by 17.3%, supporting the

International Journal of Science, Engineering and Technology
An Open Access Journal

hypothesis that role decomposition yields measurable benefits.

8.2 Per-Category Performance

Table 3 disaggregates the F1 score per defect category. CodeGenie demonstrates particular strength in Security and Performance, where contextual reasoning is most valuable.

F1 Score per Defect Category

System	Syn.	Sec.	Perf.	Style	Doc.
B1	0.84	0.51	0.49	0.78	0.42
B2	0.79	0.74	0.71	0.74	0.78
B3	0.81	0.78	0.76	0.78	0.80
CodeGenie	**0.91**	**0.88**	**0.86**	**0.87**	**0.92**

8.3 Explainability and Cognitive Load

The explainability score $\mathcal{E}$ for CodeGenie is 0.96, compared with 0.71 for B2 and 0.74 for B3, since each detected issue is paired with a role-grounded justification. In the user study, the mean NASA-TLX score for CodeGenie was 38.4 versus 65.1 for B2 – a 41% reduction in reported cognitive load (paired t-test, $p < 0.001$).

8.4 Adaptive Personalization

Across five sequential review rounds for each of 12 participating developers, the personalized variant of CodeGenie reduced repeated-defect occurrences by 28.6% relative to the non-personalized configuration, demonstrating the value of the adaptive feedback loop.

8.5 Latency and Cost

Parallel agent dispatch keeps the end-to-end median latency at 4.7 s for files up to 500 lines of code, comparable to a single-model invocation (4.2 s) and well within an acceptable interactive budget.

IX. DISCUSSION

9.1 Why Role Decomposition Helps

The empirical gains can be traced to two mechanisms. First, prompt specialization narrows each agent's attention surface, reducing the well-known tendency of LLMs to interleave reasoning across heterogeneous concerns. Second, the fusion layer transforms each agent into an independent "vote," increasing the likelihood that at least one specialist surfaces any given defect category. A complementary, third mechanism arises from the explanation channel itself: because every reported issue is attributed to a specific agent, developers can quickly disregard verdicts emitted by an agent whose analytical scope is irrelevant to the file under review (for example, dismissing performance commentary on a configuration script), without losing trust in the overall report.

9.2 Comparison with Ensemble Voting

The Ensemble-LLM baseline (B3) approximates the wisdom of repeated sampling but offers no division of labor: each pass independently attempts the entire review and is averaged through majority voting. Our results indicate that this configuration improves recall modestly but suffers in precision because correlated failures across passes are common. CodeGenie's specialization, by contrast, decorrelates the agents' error modes by construction, since their training prompts force divergent attentional priors.

9.3 Threats to Validity

Internal validity: synthetic defect injection may not perfectly reflect organic bugs, although the inter-annotator agreement of $\kappa = 0.81$ mitigates this concern. External validity: results are reported for three languages and a 480-file benchmark; generalization to industrial monorepos remains future work. Construct validity: the NASA-TLX instrument relies on self-report; future studies will incorporate eye-tracking and review-completion-time measures.

9.4 Sustainable Development Goals

CodeGenie contributes to three United Nations SDGs:

- **SDG 4 (Quality Education)**: explainable feedback transforms reviews into learning opportunities, broadening access to senior-engineer-level mentorship.
- **SDG 8 (Decent Work and Economic Growth)**: 41% reduction in cognitive load improves developer productivity and well-being.
- **SDG 9 (Industry, Innovation and Infrastructure)**: stronger software quality assurance underpins reliable digital infrastructure.

9.5 Limitations

The framework currently relies on closed-source foundation models, which raises concerns of cost, reproducibility, and data privacy in regulated environments. Investigations of open-weight alternatives (e.g., StarCoder (**li2023starcoder?**), CodeLlama) are ongoing.

X. CONCLUSION AND FUTURE WORK

This paper introduced CodeGenie, a multi-agent generative AI framework for explainable, adaptive code review. By decomposing the review task across five specialized agents and aggregating their verdicts through a calibrated decision-fusion model, CodeGenie achieves a defect-detection F1-score of 0.892 – a substantial improvement over rule-based, single-model, and ensemble baselines – while simultaneously reducing reviewer cognitive load and providing per-issue explanations.

Future work proceeds along three axes. First, we will integrate reinforcement learning from developer feedback so that the fusion weights and agent prompts evolve over time. Second, we plan to extend the agent roster to include test-generation and refactoring agents, moving from review to closed-loop code improvement. Third, we will conduct a longitudinal industrial deployment to study CodeGenie's effect on defect-escape rates in production systems.

Acknowledgment

The authors gratefully acknowledge the Department of Computational Intelligence, School of Computing, SRM Institute of Science and Technology, Kattankulathur, for institutional support and infrastructure. We thank the volunteer developers who participated in the user study and the anonymous reviewers whose comments improved this manuscript.

REFERENCES

1. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in Proc. 35th Int. Conf. Software Engineering (ICSE), San Francisco, CA, USA, 2013, pp. 712–721.
2. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in Proc. 40th Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP), Gothenburg, Sweden, 2018, pp. 181–190.
3. P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in Proc. 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Saint Petersburg, Russia, 2013, pp. 202–212.
4. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in Proc. 35th Int. Conf. Software Engineering (ICSE), 2013, pp. 672–681.
5. M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.
6. A. Fan et al., "Large language models for software engineering: Survey and open problems," in Proc. IEEE/ACM Int. Conf.

Software Engineering: Future of Software Engineering (ICSE-FoSE), 2023, pp. 1–16.

7. L. Wang et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, no. 6, pp. 1–26, 2024.

8. Z. Xi et al., "The rise and potential of large language model based agents: A survey," arXiv preprint arXiv:2309.07864, 2023.

9. N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," IEEE Software, vol. 25, no. 5, pp. 22–29, 2008.

10. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," ACM Computing Surveys, vol. 51, no. 4, pp. 1–37, 2018.

11. Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of EMNLP, 2020, pp. 1536–1547.

12. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. Int. Conf. Learning Representations (ICLR), 2021.

13. Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in Proc. EMNLP, 2021, pp. 8696–8708.

14. Y. Li et al., "Competition-level code generation with AlphaCode," Science, vol. 378, no. 6624, pp. 1092–1097, 2022.

15. R. Li et al., "StarCoder: May the source be with you!," Trans. on Machine Learning Research, 2023.

16. X. Hou et al., "Large language models for software engineering: A systematic literature review," ACM Trans. Softw. Eng. Methodol., 2024.

17. Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023.

18. S. Hong et al., "MetaGPT: Meta programming for a multi-agent collaborative framework," in Proc. Int. Conf. Learning Representations (ICLR), 2024.

19. C. Qian et al., "ChatDev: Communicative agents for software development," in Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL), 2024, pp. 15174–15186.

20. J. Sun, T. Liao, X. Xia, and S. Wan, "On the importance of explainable AI in software engineering," IEEE Software, vol. 39, no. 6, pp. 49–56, 2022.

21. A. Karmakar and R. Robbes, "What do pre-trained code models know about code?," in Proc. 36th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2021, pp. 1332–1336.

22. J. Cito, I. Dillig, V. Murali, and S. Chandra, "Counterfactual explanations for models of code," in Proc. 44th Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2022, pp. 125–134.

23. R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE), 2014, pp. 654–665.

24. S. G. Hart, "NASA-task load index (NASA-TLX); 20 years later," in Proc. Human Factors and Ergonomics Society Annual Meeting, vol. 50, no. 9, 2006, pp. 904–908.