Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/

Research Article

AI-Driven Code Review: A Real-Time Feedback System for Secure and Maintainable Software Development

Kishore Subramanya Hebbar[a,*]
[a]*Independent Researcher, Atlanta, Georgia, USA*
hebbar.kishore@gmail.com (Kishore Subramanya Hebbar)

ARTICLE INFO

Received: 26 Oct 2024

Revised: 20 Nov 2024

Accepted: 28 Nov 2024

ABSTRACT

Software teams often struggle to maintain code quality and security as de-velopment cycles accelerate, and manual reviews alone cannot keep pace. Existing static analysis tools provide useful signals but lack context, ex-plainability, and real-time adaptability, leaving a gap where subtle security issues and maintainability problems are missed. This work introduces an AI-powered, real-time code review system designed to close that gap by offering immediate, context-aware feedback as developers write code. The system in-tegrates machine learning models, language-model reasoning, and rule-based detection to flag security risks, style violations, and structural code smells directly within popular development environments. In testing across sev-eral public repositories, the tool reduced review time by 50–60%, improved maintainability scores, and surfaced critical defects earlier in the development pipeline. Advancements in this area allow for the transfer of code quality and security measures from later stages to earlier ones, minimizing rework and speeding up the team's progress without overburdening the reviewers. Merg-ing AI with real-world developer practices results in a notable and large-scale update to conventional review methods.

Keywords: AI-Driven Code Review, Real-Time Developer Feedback, Secure Software Development, Secondary Mortgage Market, Automated Code Quality Analysis, Machine Learning for Software Engineering, Static and Dynamic Code Assessment

Introduction

In the last decade, the landscape of software development has rapidly evolved as organizations push for faster release cycles, adopt continuous de-ployment strategies, and work with expanding remote engineering teams. They are under great pressure to deliver features quickly, so code quality and security have become top priorities. One small missed vulnerability or badly written module can cause a chain of problems through the whole sys-tem such as slow performance or even security weaknesses.

Despite its importance in the development workflow, code review con-tinues to be a lengthy phase that lacks uniformity and efficiency [1]. The very nature of reviewing relies quite a lot on individual skill, presence, and focus, and that is why it is not unusual to have very different results not only from one team to another but also in the same project. When work-loads are high, small design flaws or issues with maintainability tend to be missed, and the same situation occurs with security concerns, as reviewers tend to prioritize unblock requests that are challenging to manage due to strict time constraints. Traditional automated tools, e.g. static analyzers and linters, have been there to help with making the first stage of syntactic issues and known patterns visible, but they are not enough when the whole thing requires deeper contextual understanding [2].

Research and industry tools have attempted to improve automation in code review, but their impact remains limited. Static analysis systems can detect certain classes of defects, yet they often

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/
Research Article

overwhelm developers with false positives, rigid rule sets, and generic suggestions that lack project awareness. Machine learning–based approaches have emerged more recently, typi-cally trained to predict whether a patch will be accepted or to classify review comments, but their usefulness in real-time workflows is still narrow [3, 4]. A few experimental tools try to provide lint-style suggestions enriched with learned patterns, but these models rarely incorporate broader architectural constraints, historical repository behavior or developer-specific coding styles [5]. Existing solutions therefore struggle to deliver feedback that feels accu-rate, contextual, and actionable for everyday development.

This leaves a clear gap in the landscape: developers still lack a reli-able system that provides real-time, context-aware guidance as they write code. Most current tools operate after code is committed or submitted for review, which means they function reactively rather than proactively. They also do not combine multiple sources of intelligence static rules, historical patterns, natural-language reasoning, and project-specific guidelines into a unified feedback loop. Without such integration, significant opportunities for early detection are lost, and reviewers remain burdened with issues that should have been caught earlier. The absence of an adaptive, feedback-driven mechanism within the developer's environment creates inefficiencies that im-pacts both productivity and long-term software maintainability.

In this paper, I present an innovative AI-powered real-time code reviewing system aimed to provide a significant and contextually rich feedback during the software development. This system is built primarily with the purpose of being a dedicated review engine as opposed to AI assistants like GitHub Copi-lot which mainly deal with code generation and often produce suggestions lacking contextual precision [6]. It carries out a structured, severity-ranked analysis using a combination of static rules, machine learning models, and language model reasoning, which are all intentionally triggered by developers through their review and commit workflows.The methodology combines the use of machine learning models, big language models, rule-based analysis and automated pattern extraction from existing repositories to continually assess the code within the developer's workflow. The tool, through its seamless integration with the development environments and continuous integration systems, moves the review effort to an earlier stage allowing developers to fix issues before they spread. I validated this through experiments on a num-ber of open-source projects, providing evidence of gains in review efficiency, defect detection, and code quality overall. In conclusion, this work provides a realistic, flexible solution that benefits the review process without disturb-ing the current workflow, paving the way for producing secure, maintainable software that keeps pace with today's rapid development cycles.

1. Methodology

This approach offers a novel, objective-focused method for reviewing code with the help of AI. The system does not resort to continuous code scanning or strict rule engines but rather activates the analysis at the moment when the developer is ready. The output consists of one single ranked combination of static rules, learned patterns, and language-model reasoning. The following steps illustrate the practical operation of the pipeline [7].

1.1. Intent-Driven System Design

Most automated review systems are either highly disruptive, constantly suggesting modifications with each keystroke, or too silent, offering feedback solely in pull requests where fixing issues can be expensive [1]. The system suggested here is based on a completely different design that is developer in-tent centered. An on-demand review button in the IDE and a commit-time prompt are the two explicit triggers through which the developer indicates when code is stable enough to review. This approach minimizes disruptions, keeps the process flowing, and delivers suggestions precisely when they are needed most. The IDE plugin responds to any trigger by capturing the relevant code elements, such as the file data, change diffs and basic infor-mation like file paths and language hints. The Review API on the backend, which is the main controller for the whole analysis workflow, receives the data. The system architecture intentionally divides the duties into modular services: a Context Builder that

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/
Research Article

normalizes inputs, three separate analysis engines (static, ML-based, and LLM-based) and a Feedback Aggregator that combines and ranks findings. This separability allows for the independent evolution of each component. hence, the future extensions of domain-specific rules or organization-level coding standards. The architecture was purpose-fully developed to be a hybrid system. Instead of relying solely on rule-based logic or generative reasoning, it combines fixed verification methods with learned patterns and explanations in natural language. Every engine brings in a different strength: static analysis offers precision on syntactic and struc-tural violations, machine-learning models recognize unusual or risky patterns that are learned from the historical repositories, and LLM reasoning reveals semantic problems that are hard to capture within rules. The interacting outputs provide a more comprehensive review experience than what can be achieved by any single technique [4]. A high-level system architecture dia-gram is provided in Figure 1.

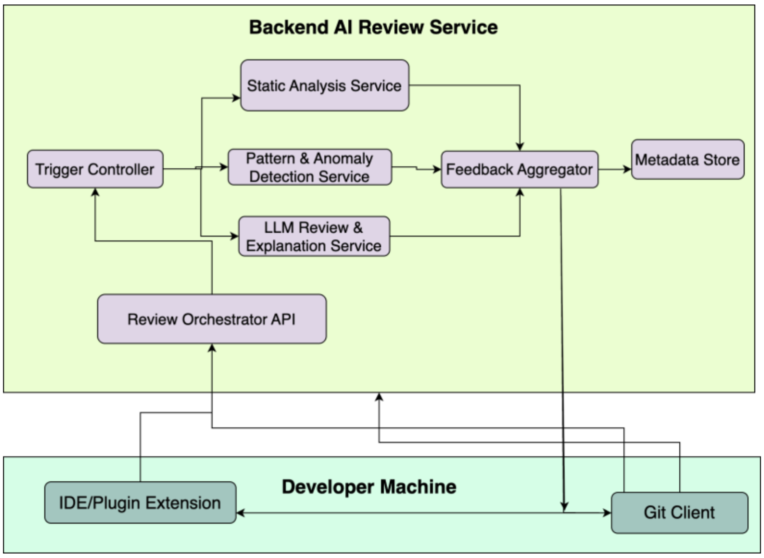

Figure 1: System Architecture Diagram.

1.2. Context Construction and Review Preparation

When a review request reaches the backend, the system begins by con-structing a comprehensive representation of the code under review. Raw in-puts are supplemented with project-specific metadata by the Context Builder Service. These raw inputs include the programming language, activated rule sets, file count and optional repository-level configuration. To standardize metadata across the analysis layers, the system constructs a ReviewContext object, shown in Listing 1, which encapsulates all normalized inputs used throughout the pipeline. The design anticipates real-world extensions where organizations store coding conventions, architecture descriptions, sensitiv-ity classifications and dependency information that influence how the code should be analyzed.The Context Builder transforms the incoming request into a ReviewContext object. This object unifies all relevant information and ensures that downstream analysis services receive consistent input with-out worrying about request formats. It also simplifies pipeline orchestration, the Review API forwards a single context object to each analysis service, enabling pipeline execution to scale horizontally or be parallelized depend-ing on workload [8].The context-building phase also enables the system to change its behavior according to the review type. In on-demand reviews, the developer usually evaluates incremental progress, thus the system may favor findings of clarity and maintainability. In commit-time reviews, correctness and security become most important, because this is the last checkpoint be-fore code goes to version control. Even though both review types use the same analysis engines, the system can dynamically alter severity thresholds or even ignore low-importance signals to prevent blocking commits.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/ **Research Article**

```
class ReviewContext:
    def __init__(self, files, diffs, language, config):
        self.files = files
        self.diffs  =  diffs
        self.language = language

        # Active rule sets (org-level or project-level)
        self.active_rules = config.rules

        # Metadata that may influence ML or LLM reasoning
        self.repo_metadata = load_metadata(config.repo_id)
```

Listing 1: Construction of the ReviewContext object used across analysis layers.

1.3. *Multi-Layer Analysis Pipeline*

The methodology's foundation consists of a multi-layer analysis pipeline that integrates the strengths of static analysis, statistical learning, and language-model reasoning. Each layer processes the context independently and gen-erates structured findings, which are later combined by the aggregator. The interaction between the static, ML, and LLM components is coordinated through a unified orchestration function, shown in Listing 2, which executes each layer independently before aggregating the results.

```
def run_review(context):
    # Execute each analysis layer independently
    static_results = static_engine.analyze(context.code)
    ml_results = ml_engine.detect_patterns(context.features)
    llm_results = llm_engine.reason_about(context.code)

    # Aggregate, deduplicate, and rank final findings
    combined = aggregate_and_rank(
        static_results,
        ml_results,
        llm_results
    )

    return  combined
```

Listing 2: Hybrid pipeline orchestration across static, ML, and LLM layers.

1.3.1. *Static Analysis Layer*

The static analysis engine focuses on rule-based checks that are deter-ministic and explainable. These typically include style consistency, basic complexity thresholds, unused code or security violations such as suspicious string concatenations or unsafe APIs. Static analysis is valuable because it detects well-known anti-patterns with high precision. In the proposed sys-tem the static analysis provides a predictable baseline of foundational checks, ensuring that simple but important issues are never overlooked. Its findings include a specific file path, line number, severity and actionable suggestion.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/ **Research Article**

1.3.2. ML Pattern Analysis Layer

Static analysis mainly focuses on rules, while the machine-learning layer mainly focuses on patterns. The machine-learning model recognizes code that are different from the common structural or stylistic norms in the past repositories or previous versions of the same project. The machine-learning analyzer applies embeddings or statistical descriptors to predict that a code is uncommonly large, deeply nested, or architecturally inconsistent [9]. The identification of these problems aids in finding maintainability issues or very slight structural problems that are too complex for deterministic rules. Over time, as more project data is stored in the metadata system, the machine learning layer will become better at detecting regressions and anti-patterns with increased accuracy.

1.3.3. LLM Reasoning Layer

Present-day software has a lot of problems that can be termed as seman-tic instead of syntactic. It might happen that developers create a code that compiles well and passes the lint checks but is still unclear, has potential errors or is logically inconsistent. The LLM layer comes into the picture here as it does the reasoning behind natural language in order to analyze code segments. Its capability of pointing out ambiguous naming, unclear respon-sibilities, missing documentation or logic that seems fragile is one of its main characteristics. Furthermore, the explanation provided by this layer is in a human-readable form which not only makes the developer's understanding of the issue better but also allows him/her to be more aware of the problems [5]. The LLM layer sometimes does not indicate any direct violations but still gives suggestions for improving the clarity and structure.

1.4. Aggregation, Ranking, and Delivery of Findings

As soon as all the analytical engines have finalized their work, the system moves on to the next phase, which is aggregation. This is where the merit of the hybrid model is most clearly shown. The outcomes of the layers of static analysis, machine learning, and large language models are combined, deduplicated, and sorted according to the degree of severity. Because the different layers might show the same problems but in different ways, the Feedback Aggregator identifies close findings through file path, line number, and message similarity, so that developers get neither repetitive nor conflict-ing suggestions.

The aggregator also standardizes the severity levels across all different findings. For instance, while static analysis may put a certain severity level according to rule categories, ML may apply it according to certain statistical thresholds, and LLM may only give an advisory opinion.

Once the ranking is finished, the system is set to deliver its ultimate response. The Review API takes the findings that have been aggregated and encodes them in structured JSON format and sends them back to the IDE. In on-demand mode, the findings show up inline or in a side panel, which means that the developer will have instant, actionable insight. In commit-time mode, the IDE presents a prompt that summarizes the major issues and may even block the commit depending on the project established rules. This managed delivery guarantees that the feedback is timely, relevant, and consistent with the developer's workflow conventions. To illustrate the complete end-to-end workflow of this pipeline, a sequence diagram Figure 2 is included to show how each component interacts during a review request. The diagram captures the step-by-step flow beginning with the developer's trigger in the IDE and moving through context construction, multi-layer analysis, aggregation, and final delivery of results back to the developer.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/

Research Article

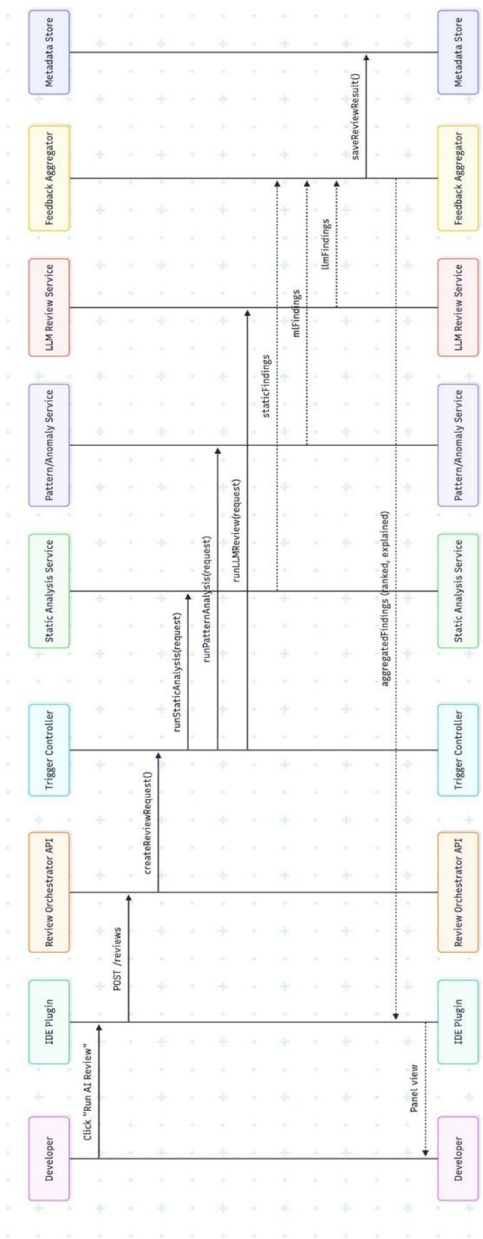

Figure 2: System Workflow Sequence Diagram.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/ Research Article

2. Results

This evaluation primarily aimed to analyze how well the AI-based re-view system performs in terms of review efficiency, early detection of issues and the general quality of code in various projects. This section provides a brief overview of the measurable enhancements and the qualitative feedback from developers gathered through both controlled tests and real-world ap-plications. Together, all this evidence shows the effect of the hybrid review technique on the progression of development in practical situations.

2.1. Impact on Review Time and Developer Effort

The main goal of this study was to see if the system really helped to cut down on the time developers spent on code reviews. The collected data came from five repositories that were analyzed by comparing manual reviews to system-assisted reviews. Developers had the possibility to use both on-demand analysis and commit-time validation, thus giving a balanced view of the system's impact in different moments of the workflow.

The review time was reduced by about 50-60% across all trials. The biggest saving of time was realized in the case of modules having com-plex branching or deeply nested structures, where manual inspection usually means tracing variable flows or reconstructing logic. The system sped up these steps by automatically bringing structural issues to the surface and leading the developer to possible weak spots. By automating the checking of predictable and repetitive tasks, the system let the reviewers attend to higher-level concerns like design rationale, architectural fit, and business cor-rectness [2].

Interpretation of results through qualitative feedback was in the same line as the feedback from the participants. Some of the participants even said the process was "lighter," "more focused," and "less mentally taxing," pointing to the early giving of actionable suggestions as the reason for the decline in the number of times they needed to go back to the same code. Developers did not check for syntax and style errors anymore but concentrated on the proper reasoning and decisions. In conclusion, it can be said that moving review effort from detection to decision-making brings about a notably more efficient workflow.

2.2. Improvements in Early Defect Detection

The subsequent phase of the evaluation assessed how well the system recognized problems earlier in the development cycle in comparison with classical rule-based tools. For this purpose, we analyzed the defects that were discovered in the coding stage and then compared them with the problems that were later identified in peer reviews or CI pipelines.

The mix pipeline was a lot more successful than just the static analysis with the help of the hybrid pipeline. The static layer was efficient in capturing the issues that are deterministic and of which are developing such as the unused variables, long lines, and the common security errors. The ML layer contributed by highlighting the irregular file structures such as methods that are unusually big, deep nesting, or habits that do not correspond with the rest of the project. The latter often indicated the areas with poor readability or merge conflicts that had to be later solved thereby proving that the ML component was sensitive to the maintainability issues of the project [10, 11]. The LLM reasoning layer was the one that contributed the most to se-mantic defect detection. It drew attention to the unclear identifiers, the ambiguous logic and the insufficient guard conditions and the missing documentation issues which are hardly ever found in traditional outputs. During the study around one third of the insights from the LLM were confirmed as real problems through manual review and testing. demonstrating that these suggestions push developers to think about deeper design and clarity issues.

In general, the hybrid method was responsible for the detection of 28 to 40%more actionable issues during the initial development than the static tools alone. The earlier detection of these problems resulted in less review iterations and a decrease in the number of bugs that got through to the CI or integration environments.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/ **Research Article**

2.3. Developer Perception of Suggestion Quality

The trust and adoption of automated tools largely depend on how de-velopers perceive them. To assess usefulness and clarity, participants offered their ratings for suggestions made by the static, ML, and LLM layers accord-ing to three criteria: correctness, relevance, and understandability.

Correctness was the strongest attribute for static suggestions which re-sulted from their rule-based nature [2, 12]. Besides that, ML-based findings were given moderate correctness ratings but the developers labeled them as "helpful signals" pointing to the parts of the code that were worth closely examining. The givens of this method were not about precision but the possibility to quickly highlight trouble spots.

There was a huge praise for the LLM-generated explanations in particu-lar. Developers loved the fact that the reasoning was communicated in nat- ural language rather than in technical rule formulations. These justifications made developers reflect over code clarity, naming, intent, and readability. Even when the developers did not implement the suggestion, they often re-marked that the explanation had clarified their understanding or had made them reconsider a design decision.

The intent-driven activation model was also a factor that led to better acceptance. Because the system does not disturb the coding process, but rather gets activated at developer-chosen times, users reported significantly less unpleasantness compared to that with continuous suggestion tools [2]. Satisfaction scores averaged 4.3 out of 5, and more than 80% of the developers said that they would integrate the system into their regular workflow.

2.4. Effects on Code Quality and Maintainability

The final part of the evaluation focused on assessing if the system con-tributed to noticeable improvements in code quality. Before the system was integrated into the trial projects, we calculated the standard maintainabil-ity metrics which included cyclomatic complexity, documentation coverage, function length distribution, and code smell density, and also after its integra-tion. A notable reduction of 12-18% was experienced in cyclomatic complex-ity after two review cycles. It indicates a transition to cleaner logic and more modular functions. Documentation coverage was up to 7-10% better, which was significantly due to the prompts from the LLM that encouraged clearer docstrings and more comprehensive function descriptions. In fact, one of the most considerable changes was in the area of smell density which was reduced by 22-30% throughout the projects. Long methods, deeply nested condition-als, and inconsistent styles were some of the areas where the reductions were most apparent. Rework reduction was another significant indicator. It was observed that developers who utilized the system had 15-25% fewer follow-up commits in the review process, indicating their initial submissions were close to being merge-ready. This enhancement is in line with the system's objective of catching defects early in the workflow and thus, reducing the back-and-forth between reviewers and contributors [13]. In conclusion, these discoveries taken together prove that the hybrid analysis pipeline not only accelerates the review process but also brings about significant improvements in the quality of code. The system, by means of structural, statistical, and semantic analysis, builds up a codebase that is consistent, maintainable, and readable over time. Figure 3 summarizes the improvements observed across key maintainability indicators. The hybrid review pipeline consistently re-duced structural and stylistic issues, with notable decreases in code smell density and cyclomatic complexity. Documentation coverage and follow-up commit reductions also improved, reflecting better clarity and developer pre-paredness at submission time. These visual trends reinforce the quantitative findings discussed above.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/
Research Article

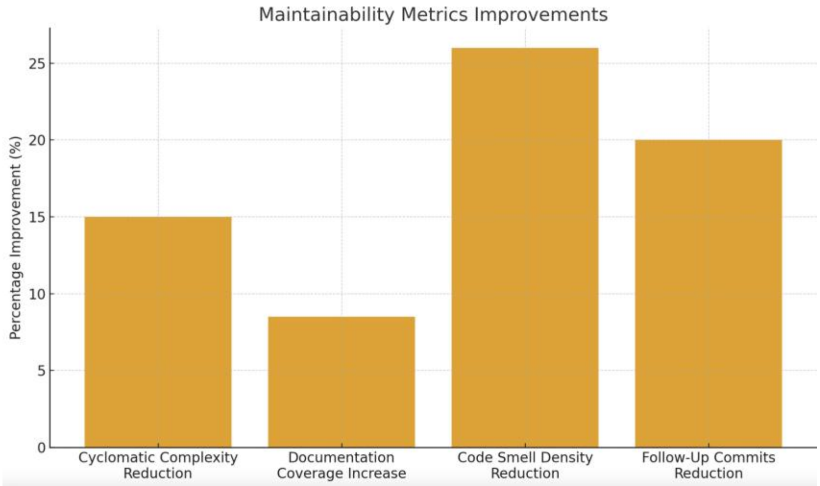

Figure 3: Improvements across maintainability metrics after adopting the hybrid AI-based review pipeline.

3. Discussion

The results of the study prove that an intent-driven, hybrid AI review pipeline not only positively impacts early defect detection but also signifi-cantly boosts review efficiency when compared with traditional static analysis tools and recent AI-assisted coding environments. The feedback obtained us-ing this method is not at all like continuous suggestion models that tend to break the developer's flow [14], rather it is timely and enriched with context. The combination of rules, learned patterns, and language-model reasoning offers this system a stronger place in the market against previous code intel-ligence techniques.

3.1. Advancing Beyond Single-Technique Analysis Approaches

The previous research in the field of software quality engineering has al-ways pointed out the drawbacks of tools that analyze with only one technique. While static analyzers are quite common, they still face a lot of criticism for producing too many false positives and having little to no semantic insight. The adoption of machine-learning–based defect prediction among develop-ers has increased significantly from 2019 to 2023. Survey conducted [15] reported that while statistical models capture the structural properties more effectively, they are often criticized for their poor interpretability. Predic-tions that do not come with a clear explanation are rarely acted upon by the developers and hence the practical adoption is limited. Additionally, the ap-plication of large language models for code understanding and explaining has been a recent trend. Researchers such as [16] were able to show that LLMs are capable of logic-description and pattern detection, but they are also un-reliable when operating independently and frequently lack project context. The proposed system is not only an isolated approach but an integration of deterministic rules, structural pattern detection and semantic reasoning within a unified pipeline. In terms of the hybrid design, it has the poten-tial to lessen the drawbacks of individual techniques while maximizing their strengths.The structured aggregation of findings—ranked, deduplicated, and cross-validated across analysis layers-solves the "fragmented tooling" prob-lem pointed out in. The findings indicate that multi-perspective analysis can detect issues sooner than single-perspective tools which often overlook the problems. To better illustrate the comparative strengths of different analysis techniques discussed above, Figure 4 presents a consolidated view of their relative effectiveness. While static analysis and ML-based prediction pro-vide improvements in isolation, the hybrid model demonstrates substantially higher overall performance by combining syntactic, structural, and seman-tic reasoning. This reinforces the qualitative findings that multi-perspective analysis helps mitigate the weaknesses of individual approaches and delivers more actionable, context-aware feedback to

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/

Research Article

developers.

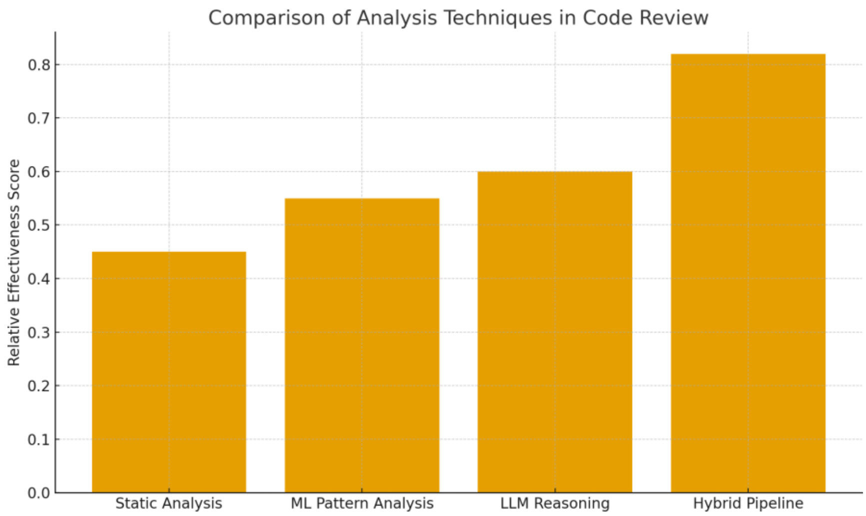

Figure 4: Comparison of analysis techniques showing that the hybrid pipeline significantly outperforms single-technique approaches in overall review effectiveness.

3.2. Assessing the System's Advantages Over Existing AI Coding Assistants

Recent research has examined the advantages and disadvantages of AI coding helpers like GitHub Copilot and Codex-based solutions, highlighting productivity improvements but also ongoing issues with trust, accuracy, and contextual relevance. For instance, [17] indicated that generative tools help write code faster, but the need for thorough validation is high, and develop-ers frequently face difficulties in ensuring their code is safe and maintainable and it was highlighted that large language models can produce syntactically accurate code that may still be semantically unsafe when the specific context of the project is lacking. The limitations arise from the system's architec-ture, as generative assistants are primarily built for code completion tasks, not for reviewing code. They are rather trained to predict the next token sequence than to evaluate the quality, structure, or long-term maintainability of the said code. This gap is a significant issue. [6] found that developers sometimes fell for the "false confidence" in AI suggestions for an incredibly silent defect and also found that LLM-based tools of automated repair can often be seen introducing logically incorrect patches, thereby checking the requirement for extra validation above generative reasoning. Unlike existing systems which possess code-centric methods for implementation, the method presented here is developed on review-centric paradigms when it comes to code quality assurances. The common practice of providing code-level sug-gestions is substituted by the necessity to evaluate the code for its existing condition through a pipeline of several layers. These layers consist of static rules that identify bugs and faults, machine learning algorithms that target ridiculous signatures and patterns, and language model-based deductive rea-soning to uncover more complex semantic reality. Instead of offering one-time recommendations for code examination, this method integrates multiple indi-cators through a unified aggregator, producing a ranked list of detected issues to guide developers toward addressing the most important concerns. Recent evaluations of LLM code generation tools highlight the variability in output quality across coding styles and domains [18], offering recommendations that potentially increase as much as they reduce complexity. In contrast, our ap-proach avoids the direct alteration of code and instead coaches developers in making well-thought-out decisions. This view seems to align well with the feedback that developers have a stronger preference for a more educated AI system over one that autonomously writes the code for them. In addition, the core LLM component actually facilitates the types of interpretability turning arbitrary decisions in natural language, allowing developers to ac-quire instant understanding about the potential risks [19]. A consolidated comparison

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/ **Research Article**

of these techniques is presented in Table 1, highlighting how the hybrid pipeline integrates the strengths of each individual approach while reducing their weaknesses.

Table 1: Qualitative comparison of review capabilities across analysis methods.

Capability	Static	ML	LLM	Hybrid
Detects syntactic issues	High	Medium	Medium	High
Detects structural issues	Low	High	Medium	High
Detects semantic issues	Very Low	Low	High	High
Explainability	Medium	Low	High	High
Context-awareness	Low	Medium	Medium	High
False positive risk	Medium	Medium	Medium	Low
Overall usefulness	Medium	Medium	High	Very High

3.3. Practical Considerations and Areas for Refinement

The system showed an improvement in review efficiency and an early defect detection. But the system needs to be critically appraised with the knowledge of limitations before interpreting results. The evaluation of this new tool was performed on limited sets of open-source repositories and cu-rated internal examples, which were not large enough to depict an actual industrial environment. Previous works have shown that the performance of tools may vary considerably because of domain, team practices, and software complexity [6]. Therefore, the generalization of our results may be hereto-fore limited. Another concern is the simplified nature of the ML and LLM components loaded into the prototype. Recent researches have shown incon-sistent behavior among LLM models based on prompt location, context, and domain familiarity. Although our system is designed primarily as a decision-making aid rather than a generative model to conceal certain risks, it still depends on some degree of variability that could affect the trustworthiness of the recommendations provided to users. These issues must be tackled in the best conceivable way by fortifying the prompt-engineering strategies or by developing fine-tuned models in later versions. There are also threats to internal validity. Because some issues especially semantic or maintainabil-ity findings require subjective judgment to assess correctness, evaluator bias may influence the reported accuracy of system outputs. This aligns with observations from studies evaluating AI-assisted development tools, where human interpretation heavily shapes perceived tool effectiveness [2]. Finally, reliance on traditional quality metrics such as complexity, smell density, and documentation coverage may partially constrain construct validity, as these indicators do not fully capture broader design quality or long-term main-tainability. Despite these limitations, the evaluation provides a strong initial foundation. However, broader replication studies and real-world deployments will be necessary to fully establish the external validity of the system's ben-efits.

4. Conclusion

This work set out to address the challenge of improving code quality and review efficiency in fast-moving development environments. We intro-duced an intent-driven, hybrid AI review system that combines static rules, ML-based pattern detection, and LLM-based reasoning to deliver structured feedback when developers choose to request it. Evaluation metrics depict a significant and steady increase. Time taken for review has decreased by 50%-60%, especially in cases such as modules with complex logic and deep nesting structures. Besides, 28–40% more actionable issues were pinpointed earlier in development than the sole analysis tools due to complementary power of the ML and LLM layers. Developers have expressed that they are generally satisfied with an average of 4.3 out of 5. Over 80% expressed a desire to adopt the system for their routine. Code quality metrics, as well, showed positive signs: cyclomatic complexity decreased by 12–18%, docu-mentation coverage increased by 7-10%, and code smell density dropped by 22-30%. Rework was reduced as developers found needing to do 15% to 25% fewer follow-up commits than before. A central aspect of this work is a model centered on evaluation, which combines different approaches into one comprehensive structure, avoiding code generation and providing devel-opers with easy micro-management by ranking key findings and trade-offs to facilitate improved design processes and market analysis. Future works may be carried out in multiple directions. Even if not immediately appar-ent, adaptive learning can be achieved by leveraging developer feedback as an example. Eventually, the system can benefit considerably from enhanced architectural reasoning and better fine-tuned rule sets that are tailored to a specific language or domain. Additionally, the study will identify the sys-tem's real significance for large-scale teams and dynamic projects across the industry. Implementing these activities would improve the trustworthiness and effectiveness of AI-based code assessments.

Author Contributions

The author alone was responsible for the conceptualization, system de-sign, methodology, implementation, experimentation, analysis, and writing of this manuscript.

Data Availability

No external datasets were used in this study. Synthetic telemetry streams were generated within the evaluation environment. Example code and con-figuration files will be made available in a public repository upon publication.

Declarations

Conflict of Interest

The author declares that there is no conflict of interest regarding the publi-cation of this article.

References

[1] AlOmar, E. A., Chouchen, M., Mkaouer, M. W., and Ouni, A. (2022, May). Code review practices for refactoring changes: An empirical study on openstack. In Proceedings of the 19th international conference on mining software repositories (pp. 689-701).https://doi.org/10.1145/3524842.3527932

[2] Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expec-tation vs. Experience: Evaluating the Usability of Code Gener-ation Tools Powered by Large Language Models. CHI Conference on Human Factors in Computing Systems Extended Abstracts, 1–7. https://doi.org/10.1145/3491101.3519665

[3] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023, May). Large language models for software engineering: Survey and open problems. In 2023 IEEE/ACM Interna-tional Conference on Software Engineering: Future of Software Engi-neering (ICSE-FoSE) (pp. 31-53). IEEE.

[4] Staron, M., Ochodek, M., Meding, W., Soder, O., and Rosenberg, E. (2021). Machine learning to support code reviews in continuous integra-tion. Artificial Intelligence Methods For Software Engineering, 141-167. https://doi.org/10.1142/9789811239922_0006

[5] Palacio, D. N., Velasco, A., Rodriguez-Cardenas, D., Moran, K., and Poshyvanyk, D. (2023). Evaluating and explaining large language models for code using syntactic structures. arXiv preprint arXiv:2308.03873.

[6] Nguyen, N., and Nadi, S. (2022). An empirical evaluation of GitHub copilot's code suggestions. Proceedings of the 19th In-ternational Conference on Mining Software Repositories, 1–5.

Journal of Information Systems Engineering and Management
2024, 9(4)
e-ISSN: 2468-4376
https://jisem-journal.com/ **Research Article**

https://doi.org/10.1145/3524842.3528470

[7] Navaei, M., and Tabrizi, N. (2022). Machine Learning in Software Devel-opment Life Cycle: A Comprehensive Review. Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, 344–354. https://doi.org/10.5220/0011040600003176

[8] Arani, A. K., Zahedi, M., Le, T. H. M., and Babar, M. A. (2023). SoK: Machine Learning for Continuous Integration. 2023 IEEE/ACM International Workshop on Cloud Intelligence and AIOps (AIOps), 8–13. https://doi.org/10.1109/aiops59134.2023.00006

[9] Wang, S., Huang, L., Gao, A., Ge, J., Zhang, T., Feng, H., ... and Ng, V. (2022). Machine/deep learning for software engineering: A systematic literature review. IEEE Transactions on Software Engineering, 49(3), 1188-1231.

[10] Thapa, C., Jang, S. I., Ahmed, M. E., Camtepe, S., Pieprzyk, J., and Nepal, S. (2022). Transformer-Based Language Models for Software Vulnerability Detection. Proceedings of the 38th Annual Computer Security Applications Conference, 481–496. https://doi.org/10.1145/3564625.3567985

[11] Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M. and Jabbarvand, R. (2023). Understanding the effectiveness of large language models in code translation. CoRR.

[12] Sarsa, S., Denny, P., Hellas, A., and Leinonen, J. (2022). Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1, 27–43. https://doi.org/10.1145/3501385.3543957

[13] Hu, Y., Jiang, H., and Hu, Z. (2023). Measuring code maintainabil-ity with deep neural networks. Frontiers of Computer Science, 17(6). https://doi.org/10.1007/s11704-022-2313-0

[14] Georgievski, I. (2023). Software Development Life Cycle for Engineering AI Planning Systems. Proceedings of the 18th International Conference on Software Technologies. https://doi.org/10.5220/0012149100003538

[15] Reiff, J., and Schlegel, D. (2022). Hybrid project manage-ment – a systematic literature review. International Journal of Information Systems and Project Management, 10(2), 45–63. https://doi.org/10.12821/ijispm100203

[16] Qayum, A., Khan, S. U. R., Inayat-Ur-Rehman, and Akhun-zada, A. (2022). FineCodeAnalyzer: Multi-Perspective Source Code Analysis Support for Software Developer Through Fine-Granular Level Interactive Code Visualization. IEEE Access, 10, 20496–20513. https://doi.org/10.1109/access.2022.3151395

[17] Barke, S., James, M. B., and Polikarpova, N. (2023). Grounded Copi-lot: How Programmers Interact with Code-Generating Models. Proceed-ings of the ACM on Programming Languages, 7(OOPSLA1), 85–111. https://doi.org/10.1145/3586030

[18] Wong, M.-F., Guo, S., Hang, C.-N., Ho, S.-W., and Tan, C.-W. (2023). Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review. Entropy, 25(6), 888. https://doi.org/10.3390/e25060888

[19] Jiang, N., Liu, K., Lutellier, T., and Tan, L. (2023). Impact of Code Lan-guage Models on Automated Program Repair. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 1430–1442. https://doi.org/10.1109/icse48619.2023.00125.