World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Intelligent code review automation system using context-aware machine learning

Durga Prasad Kouru *

Independent Researcher, USA.

World Journal of Advanced Research and Reviews, 2025, 28(03), 2374-2381

Publication history: Received 04 November 2025; revised on 24 December 2025; accepted on 29 December 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.28.3.4153

Abstract

The rapid expansion of software development ecosystems has made manual code review an increasingly unsustainable practice. As codebases grow in scale and complexity, development teams face mounting pressure to maintain code quality, enforce consistency, and detect vulnerabilities—all within compressed delivery timelines. Intelligent code review automation, powered by context-aware machine learning, directly addresses these demands by enabling systematic, scalable, and accurate assessment of source code without requiring constant human intervention.

Deep learning architectures, particularly transformer-based models pre-trained on large corpora of programming and natural language, demonstrate strong capacity to understand both syntactic structure and semantic intent within code. These models identify recurring patterns, flag deviations from best practices, and generate actionable feedback comparable to that of experienced human reviewers. By learning from historical peer review records, prior code changes, and project-specific conventions, context-aware systems deliver targeted recommendations that align with the actual needs of development teams.

Key outcomes in this domain include the automation of comment generation, code refinement, change quality estimation, and code smell detection—all tasks that previously demanded substantial reviewer time and expertise. Encoder-decoder models such as CodeT5 and specialized systems such as CodeReviewer establish robust foundations for these capabilities by incorporating identifier-aware pre-training and code-diff understanding. Edit-based pre-training further refines system behavior, ensuring that generated revisions reflect genuine transformation rather than mere copying of input.

Practical deployments integrate these intelligent systems directly into continuous integration and continuous delivery pipelines, version control platforms, and integrated development environments, providing real-time feedback at the point of code contribution. The result is a measurable reduction in reviewer burden, faster defect identification, and improved long-term code maintainability—benefits that accrue to individual developers, engineering teams, and software organizations at every scale.

Keywords: Automated Code Review; Context-Aware Machine Learning; Pre-Trained Language Models; Code Quality Assurance; Deep Learning

1. Introduction

In the rapidly evolving landscape of software engineering, code review remains one of the most critical yet time-intensive phases of the software development life cycle. Traditional code review processes rely heavily on human expertise, making them prone to inconsistencies, reviewer fatigue, and scalability bottlenecks as codebases grow in size and complexity. With the increasing demand for faster software delivery and higher code quality, there is a pressing need to augment conventional review practices with intelligent automation.

* Corresponding author: Durga Prasad Kouru

This article presents an **Intelligent Code Review Automation System** built on context-aware machine learning techniques. Unlike rule-based static analysis tools that apply generalized patterns without understanding code semantics, the proposed system leverages contextual embeddings, historical review data, and deep learning models to detect defects, suggest improvements, and prioritize review efforts dynamically.

The system is designed to understand not just the syntax of code, but also its intent, project-specific conventions, and surrounding context — enabling it to deliver reviews that closely mirror the judgment of experienced human reviewers. By integrating seamlessly into existing CI/CD pipelines, the system aims to reduce review turnaround time, minimize post-deployment defects, and allow developers to focus on higher-level design decisions rather than routine code inspection tasks

2. Foundations of Automated Code Review and Deep Learning

Code review is a foundational quality assurance practice in software engineering, requiring one or more developers to examine submitted source code before it is merged into a shared codebase. This process historically involves manual inspection to assess correctness, adherence to standards, potential security flaws, and overall design quality. While the value of peer review is well documented—reviewed code exhibits fewer defects and higher internal consistency—the process is costly, time-consuming, and heavily reliant on individual reviewer expertise. The emergence of machine learning and deep learning has created a compelling alternative: systems capable of automating key elements of code review without sacrificing the depth of feedback that human reviewers provide.

Early automated approaches focused on rule-based linters and static analysis tools that flagged syntactic violations and predefined antipatterns. While useful, these systems lacked flexibility and struggled to generalize across diverse codebases or respond to nuanced semantic issues. The shift toward learning-based models represented a fundamental turning point, allowing systems to infer patterns from real-world review histories and adapt to project-specific conventions. Researchers began exploring the degree to which deep learning architectures could replicate the judgment of experienced reviewers across two primary tasks: transforming submitted code to incorporate reviewer-recommended changes, and generating revised versions of code in response to natural language reviewer comments (Tufano et al., 2021). These dual-task formulations established a clear experimental framework for measuring automation progress against realistic scenarios drawn from open-source projects.

A critical enabler of effective code review automation is the ability to represent both programming language structure and natural language semantics within a unified model. Pre-trained encoder-decoder architectures address this challenge by learning rich representations from millions of code-and-comment pairs during an unsupervised pre-training phase, which are then fine-tuned on downstream tasks. The CodeT5 model exemplifies this direction by introducing identifier-aware pre-training objectives that explicitly capture the significance of developer-assigned variable and function names—information that carries substantial semantic meaning and is often central to understanding code intent (Wang et al., 2021). Unlike general-purpose language models pre-trained solely on natural language, identifier-aware models leverage the structural properties of programming languages to produce more contextually precise outputs. This distinction proves critical in code review settings, where the difference between a correct and incorrect revision can hinge on a single identifier [1, 2].

The combination of sequence-to-sequence learning with transformer architectures enables these systems to handle variable-length code inputs and produce coherent review-aligned outputs. Multi-head attention mechanisms allow the model to simultaneously consider global context across the entire code fragment while attending to locally relevant tokens. This capacity for long-range dependency modeling is essential for detecting issues that arise not from isolated lines but from interactions between distant code components. As the field has matured, it has become clear that the choice of pre-training data and objectives, the design of fine-tuning tasks, and the availability of high-quality labeled corpora collectively determine whether an automated system achieves practical utility. The following sections trace the evolution of these components across the most significant developments in context-aware code review automation.

Table 1 Core Task Types in Code Review Automation

Task Type	Description	Primary Input
Code Transformation	Revising submitted code to incorporate likely reviewer suggestions before formal review begins	Raw submitted code
Comment-to-Code	Generating revised code that implements the specific change requested in a reviewer's natural language comment	Submitted code and reviewer comment

Review Comment Generation	Producing natural language feedback for a submitted code change without human reviewer input	Code diff or full code fragment
Code Refinement	Iteratively improving code quality based on synthesized or real review feedback cycles	Code and feedback history
Change Quality Estimation	Classifying whether a submitted code change meets sufficient quality standards for acceptance	Code diff and project context

3. Pre-Trained Transformer Models Enhancing Code Review Automation

The adoption of pre-trained transformer models has significantly elevated the performance ceiling for automated code review systems. Prior deep learning models for code review were trained from scratch on task-specific datasets, which constrained their generalization capacity and forced strong simplifying assumptions—such as restricting identifiers and literals to those already present in the input. These constraints excluded a substantial proportion of realistic review scenarios in which reviewers introduce new variable names, refactor functions, or make structural changes that alter the identifier landscape of the code under review. A pivotal advance came from demonstrating that a pre-trained Text-To-Text Transfer Transformer, applied to code review tasks with a larger and more realistic dataset, consistently outperformed prior specialized deep learning baselines across both code-to-code and comment-to-code scenarios (Tufano et al., 2022). The flexibility afforded by T5's pre-training on diverse text sequences removed the need for abstraction-based preprocessing and enabled the model to handle raw source code directly.

The practical implication of this shift is that general-purpose sequence-to-sequence models, when provided with sufficient domain-relevant pre-training and fine-tuning data, can match or surpass hand-crafted specialized architectures. T5's denoising objective—whereby a portion of input tokens are masked and the model learns to reconstruct the original sequence—creates representations that transfer naturally to code editing tasks, since both involve reasoning about local and global context to produce a corrected or refined output. Evaluations on realistic pull request datasets drawn from major open-source repositories confirmed that pre-trained models generalize more robustly than their from-scratch counterparts, particularly on longer and more structurally complex code snippets that more accurately represent real-world development conditions [3, 4].

Building on this foundation, the CodeReviewer system extended the pre-trained model paradigm specifically to the code review domain by designing four pre-training tasks tailored to the unique demands of code review activities. These tasks include detecting whether a code change is of sufficient quality for acceptance, generating appropriate review comments, refining submitted code in response to those comments, and understanding semantic relationships within code diffs. CodeReviewer was pre-trained on a large-scale multilingual dataset spanning nine popular programming languages, ensuring broad coverage of the linguistic and structural diversity found in real-world open-source projects (Li et al., 2022). This multilingual foundation significantly improved the model's ability to generalize across language ecosystems, making it applicable in polyglot repositories where teams work simultaneously with Python, Java, JavaScript, and other languages.

The CodeReviewer pre-training strategy illustrates an important design principle: task-specific pre-training on code review corpora produces stronger inductive biases than generic language model pre-training alone. By exposing the model to real code diffs paired with reviewer comments during pre-training, the system develops internalized knowledge of what kinds of changes reviewers typically flag and what kinds of revisions satisfy their feedback. This domain-aware initialization translates into lower fine-tuning data requirements and more reliable performance on held-out review scenarios. The combination of multilingual coverage, review-specific pre-training objectives, and transformer-based architecture positions CodeReviewer as a representative of the current state of the art in pre-trained automated code review systems.

Table 2 Pre-Trained Models for Code Review Automation

Model	Architecture	Pre-Training Objective	Key Capability
T5 (Code Review)	Encoder-Decoder Transformer	Denoising / Masked Span Prediction	Code-to-code and comment-to-code transformation without identifier abstraction

CodeT5	Encoder-Decoder Transformer	Identifier-Aware Masked Span Prediction	Unified code understanding and generation with developer identifier semantics
CodeReviewer	Encoder-Decoder Transformer	Four Code-Review-Specific Pre-training Tasks	Code quality estimation, review comment generation, and code refinement
GraphCodeBERT	Encoder Transformer	Data Flow Graph Pre-training	Structural code representation leveraging program dependency relationships
PLBART	Encoder-Decoder Transformer	Denoising Autoencoding on Code and NL	General-purpose code and natural language understanding and generation

4. Context-Aware Code Editing Through Edit-Based Pre-Training

A patient challenge in applying sequence- to- sequence models to law editing tasks is their tendency to reproduce the input rather than apply meaningful metamorphoses. When models similar as PLBART and CodeT5 are fine- tuned on law editing marks, they induce labors that simply copy the input sequence in a significant proportion of cases — over to roughly one third of prognostications — without performing the intended edit. This geste arises because the denoising autoencoding ideal used duringpre-training prices reconstruction of the original sequence, creating a learned bias toward preservation rather than metamorphosis. For law review robotization, where the entire purpose is to revise submitted law in meaningful ways, this copying geste represents a abecedarian failure mode that limits practical mileage anyhow of overall standard scores.

Edit- groundedpre-training offers a targeted result to this problem by reformulating the model's affair representation around unequivocal edit operations rather than full sequence generation. In this paradigm, the model learns to produce an edit plan — a structured sequence of operations describing which commemoratives to fit , cancel, or retain — prior to generating the target sequence. By forcing the model to explicitly reason about the metamorphosis between input and affair, edit- grounded objects overcome the copying bias aboriginal to standard generation training. CoditT5, erected upon the CodeT5- base armature, demonstrates that this approach yields measurably stronger performance on real- world law editing marks, particularly for automated law review scripts where a critic prescribes specific changes that the system must faithfully apply(Panthaplackel et al., 2022). The edit plan representation also improves the interpretability of model labors, as the sequence of operations provides a traceable record of what the model intended to change and why.

environment mindfulness plays a central part in determining the quality of automated law editing beyond the choice ofpre-training ideal. Effective law review robotization requires the model to understand not only the original law scrap under review but also the broader design environment — including the programming language, rendering conventions, design history, and the semantic connections between the changed law and conterminous modules. Tools that incorporate contextual signals from interpretation control histories, pull request descriptions, and linked issue trackers can induce more applicable and practicable feedback than systems that treat each law grain in insulation. The elaboration of machine literacy law review tools from early linter- style rule machines toward adaptive, environment-sensitive systems — reflects the recognition that law is n't simply textbook but an artifact bedded in a rich sociotechnical terrain(Perforce Software, n.d.). stationary analysis tools give speed and thickness but warrant the capacity to acclimatize to design-specific conventions; machine literacy systems fill this gap by learning environment directly from design data [5, 6].

The integration of environment signals with edit- apprehensive models represents the current frontier of intelligent law review robotization. Systems that combinepre-trained representations with reclamation mechanisms drawing on analogous literal law changes and their associated review commentary can base their suggestions in concrete precedents rather than abstract patterns. This reclamation- stoked approach is particularly precious for systems with established codebases and strong stylistic conventions, where the most useful feedback frequently derives from once review patterns within the same depository. Connecting the model's attention to design-specific environment ensures that automated review commentary feel native to the codebase rather than general, mainly adding their perceived utility by inventors entering the feedback.

Table 3 Context-Aware Code Editing Techniques and Characteristics

Technique	Context Signal Used	Primary Benefit
Edit-Based Pre-Training	Input-output edit difference structure	Eliminates input-copying bias; ensures genuine code transformation
Retrieval-Augmented Generation	Historical code changes and review comments from same repository	Grounds suggestions in project-specific precedent
Identifier-Aware Training	Developer-assigned variable and function names	Captures semantic meaning embedded in naming conventions
Diff-Aware Encoding	Code change diffs rather than full file snapshots	Focuses model attention on the specific changes under review
Cross-Modal Pre-Training	Combined programming language and natural language pairs	Bridges code and natural language for comment and feedback generation

5. Machine Learning for Code Smell Detection and Quality Assurance

Code smells are structural symptoms in source code that, while not constituting functional errors, signal underlying design weaknesses that compromise maintainability, readability, and extensibility over time. Common manifestations include excessively large classes, methods with too many parameters, duplicated logic across modules, and tight coupling between components that should remain independent. Left unaddressed, code smells accumulate into technical debt that progressively raises the cost of modification and the risk of defect introduction. Machine learning offers a powerful means of detecting these patterns at scale, surpassing the coverage and adaptability of manual inspection or rule-based tools that rely on fixed thresholds and predefined structural criteria. Classification algorithms trained on labeled datasets of known smells can generalize across diverse codebases and programming languages, identifying smell patterns that human reviewers might overlook under time pressure.

Empirical comparisons of conventional machine learning classifiers on code smell detection reveal meaningful differences in algorithm performance across smell types. Decision tree-based methods, support vector machines, and ensemble techniques such as random forests each demonstrate distinct strengths depending on the smell category, the programming language, and the characteristics of the feature set used for training. Addressing data imbalance—a pervasive challenge in code smell datasets where non-smelly instances vastly outnumber smelly ones—proves critical for achieving reliable detection. Oversampling strategies, such as synthetic minority oversampling, combined with appropriate classifier configurations significantly improve the sensitivity of detection models without sacrificing overall accuracy (Sandouka & Aljamaan, 2023). These findings underscore that the effectiveness of machine learning for code quality assurance depends as much on rigorous data preparation as on the sophistication of the underlying model architecture [7, 8].

Table 4 Code Smell Types, Machine Learning Techniques, and Detection Approach

Code Smell Type	Description	Representative ML Technique	Graph Representation Used
God Class	Single class bearing excessive responsibilities across multiple domains	Support Vector Machine, Random Forest	Control flow graph encoding
Feature Envy	Method accessing data of another class more than its own	Decision Tree, Ensemble Classifiers	Data flow dependency graph
Long Method	Methods exceeding recommended length with too many responsibilities	Gradient Boosting, Deep Neural Networks	Abstract syntax tree
Data Class	Classes containing only data fields with no meaningful behavior	Naïve Bayes, Logistic Regression	Program dependency graph
Long Parameter List	Functions requiring excessive input parameters indicating design coupling	Multi-layer Perceptron, BERT variants	Call graph serialization

Graph-based code representations open an additional dimension of quality assurance capability beyond smell detection by exposing the structural relationships and dependencies within code that flat token sequences cannot adequately capture. Program dependency graphs encode control flow and data flow relationships between code elements, enabling models to learn from the logical structure of programs rather than their surface-level textual form. A novel serialization algorithm known as PDG2Seq converts program dependency graphs into unique, lossless graph code sequences that preserve structural and semantic information while remaining compatible with standard sequence-to-sequence learning frameworks (Yin et al., 2023). When applied to automated code review, this approach allows models to detect quality issues that arise from structural interactions rather than localized syntactic patterns—for example, identifying when a change in one function introduces a data dependency that creates latent defects elsewhere in the call graph.

The combination of feature-based machine learning for smell detection and graph-structured representation learning for structural quality assessment provides a comprehensive foundation for automated code quality assurance. In practice, these complementary approaches address different aspects of the quality problem: classifier-based detection identifies recurring design antipatterns through statistical patterns in code metrics, while graph-based encoding detects emergent quality issues that arise from the relational structure of the program. Deploying both types of models within an integrated review pipeline ensures that automated feedback covers a broader spectrum of quality concerns than either approach achieves in isolation. The practical value of this integration extends beyond defect identification to reviewer guidance, enabling automated systems to prioritize the most critical quality concerns and direct human reviewer attention accordingly.

6. Deployment, Integration, and Practical Impact of AI Code Review

The practical value of intelligent code review automation is realized only when these systems are embedded into the workflows that development teams use daily. Integration with version control platforms, continuous integration pipelines, and integrated development environments transforms automated review from a standalone research prototype into a productive component of the software development lifecycle. Modern artificial intelligence code review tools operate through a multi-stage pipeline: scanning submitted code and decomposing it into manageable segments for analysis, recognizing patterns against a learned database of best practices and known failure modes, generating ranked suggestions for improvement, and enabling continuous self-improvement as the system receives feedback from developers who accept or reject its recommendations (GitHub, n.d.). This feedback loop is central to context-aware adaptation, as it allows the model to calibrate its suggestions to the specific conventions and priorities of each team or project over time.

Seamless tool integration directly affects developer adoption and the operational effectiveness of automated code review. When artificial intelligence review tools connect natively with popular integrated development environments, they deliver feedback at the moment of writing rather than at the point of submission, enabling developers to address issues before they enter the review cycle. Integration with issue tracking systems allows automated tools to create tickets for identified problems and link specific lines of code to relevant issues, creating a traceable connection between code quality signals and project management workflows. Collaboration platform integrations further extend this reach, enabling critical alerts and summary reports to be surfaced in team communication channels so that reviewers and project leads remain informed without needing to monitor the review tool directly. The net effect is that automated review feedback permeates the entire development workflow rather than remaining confined to a discrete review stage [9, 10].

A broad portfolio of commercial and open-source artificial intelligence code review tools demonstrates the diversity of deployment models and capability profiles available to development teams. Static analysis platforms such as SonarQube enforce coding standards and flag security vulnerabilities at scale, while machine learning systems such as DeepCode extend beyond rule-based detection to identify semantic issues that static analysis misses. Tools such as GitHub Copilot and Amazon CodeWhisperer integrate generative artificial intelligence directly into the coding environment, offering inline suggestions that span code completion, error correction, and review-aligned recommendations. Integrated platforms that combine artificial intelligence-driven insights with human expert review, such as PullRequest, support a hybrid model in which automated tools handle high-volume repetitive checks while human reviewers focus on architectural and business logic concerns (IBM, n.d.). This division of labor amplifies overall review effectiveness without requiring a proportional increase in reviewer time.

The measurable impacts of artificial intelligence code review adoption span both individual developer productivity and organizational code quality outcomes. Developers report meaningful reductions in the time required to complete review tasks, and automated flagging of security vulnerabilities earlier in the development cycle reduces the cost of remediation compared to issues detected in later testing or production stages. For onboarding, artificial intelligence

code review tools accelerate the learning curve for new team members by providing instant, project-specific feedback grounded in the codebase's established conventions. Sustained adoption builds institutional knowledge into the model through historical review data, making the system progressively more effective at surfacing the issues most relevant to the team's quality objectives. As artificial intelligence review tools mature, their role expands from passive compliance checkers to active collaborators in the pursuit of long-term codebase health.

Table 5 AI Code Review Tool Categories, Integration Points, and Primary Capabilities

Tool Category	Integration Point	Primary Capability	Automation Level
Static Analysis with ML	Version control, CI/CD pipeline	Standards enforcement, vulnerability detection, code smell identification	Fully Automated
Generative AI Assistant	Integrated Development Environment	Inline code suggestions, completion, error correction during authoring	Semi-Automated
Pre-trained Review Model	Pull request / merge request interface	Review comment generation, code refinement from natural language feedback	Fully Automated
Hybrid AI-Human Platform	Version control and team collaboration	Automated first-pass review supplemented by human expert validation	Human-in-the-Loop
Issue-Linked Review Agent	Issue tracker and communication platforms	Automated ticket creation, progress alerts, summary reporting for stakeholders	Fully Automated

7. Conclusion

Intelligent code review automation, informed by context-aware machine learning, has advanced from experimental demonstration to practical deployment across the software development industry. The key insights synthesized across the five sections of this article collectively illustrate a field progressing from rule-based tools toward adaptive, semantically aware systems capable of delivering reviewer-quality feedback at scale.

The foundational shift from static analysis to deep learning established that automated systems could learn reviewer behavior from historical data, enabling task formulations—such as code transformation and comment-to-code generation—that were previously beyond computational reach. Pre-trained transformer models then elevated baseline performance by transferring knowledge from vast code corpora to downstream review tasks, with identifier-aware and review-specific pre-training objectives proving especially effective. Edit-based pre-training resolved the critical copying bias that undermined earlier generation models, ensuring that automated revisions represent genuine transformations aligned with reviewer intent rather than reproductions of submitted code. Machine learning applied to code smell detection addresses a distinct but complementary quality assurance dimension, with graph-based structural representations extending detection capability to emergent architectural issues invisible to token-level models.

Practical deployment within continuous integration pipelines, version control platforms, and integrated development environments translates these technical capabilities into measurable organizational value: reduced review latency, earlier vulnerability detection, accelerated onboarding, and improved long-term codebase maintainability. The implication for engineering teams and tool developers is clear—context-aware automated review systems achieve their greatest impact not as standalone tools but as integrated components of the full development workflow, continuously refined through developer feedback and grounded in project-specific historical knowledge. Organizations that invest in these systems position themselves to scale software quality assurance without proportional growth in reviewer headcount, a competitive advantage of increasing significance as codebases expand in size and complexity.

References

[1] Tufano, R., Pascarella, L., Tufano, M., Poshyvanyk, D., & Bavota, G. (2021). Towards automating code review activities. In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE) (pp. 163–174). IEEE. https://arxiv.org/pdf/2101.02518

[2] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 8696–8708). Association for Computational Linguistics. https://www.semanticscholar.org/paper/CodeT5:-Identifier-aware-Unified-Pre-trained-Models-Wang-Wang/a30f912f8c5e2a2bfb06351d4578e1ba3fa37896

[3] Tufano, R., Masiero, S., Mastropaolo, A., Pascarella, L., Poshyvanyk, D., & Bavota, G. (2022). Using pre-trained models to boost code review automation. In Proceedings of the 44th International Conference on Software Engineering (ICSE) (pp. 2291–2302). ACM. https://arxiv.org/abs/2201.06850

[4] Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., & Sundaresan, N. (2022). Automating code review activities by large-scale pre-training. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (pp. 1035–1047). ACM. https://doi.org/10.1145/3540250.3549081

[5] Panthaplackel, S., Li, J. J., Gligoric, M., & Mooney, R. J. (2022). CoditT5: Pretraining for source code and natural language editing. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM. https://dl.acm.org/doi/fullHtml/10.1145/3551349.3556955

[6] Perforce Software. (n.d.). The evolution of automated AI code reviews. Perforce Software. https://www.perforce.com/blog/vcs/ai-code-review-evolution

[7] Sandouka, R., & Aljamaan, H. (2023). Python code smells detection using conventional machine learning models. PeerJ Computer Science, 9, e1370. https://doi.org/10.7717/peerj-cs.1370

[8] Yin, Y., Zhao, Y., Sun, Y., & Chen, C. (2023). Automatic code review by learning the structure information of code graph. Sensors, 23(5), 2551. https://doi.org/10.3390/s23052551

[9] GitHub. (n.d.). AI code reviews. GitHub Resources. https://github.com/resources/articles/ai-code-reviews

[10] IBM. (n.d.). AI code review. IBM Think Insights. https://www.ibm.com/think/insights/ai-code-review