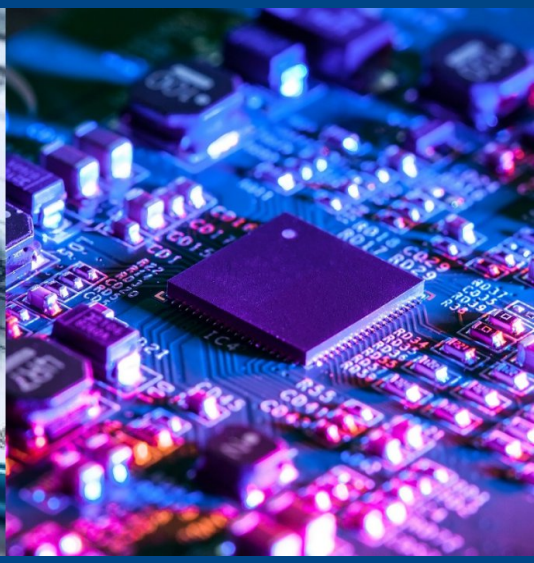
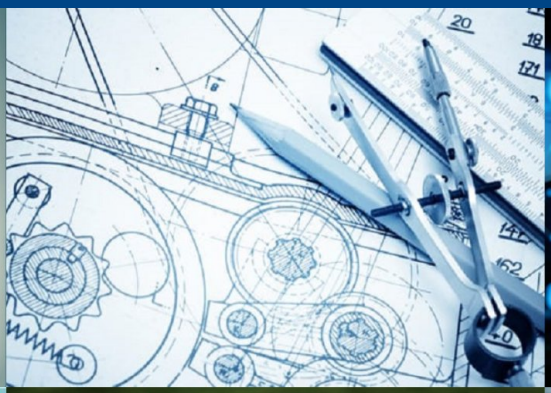
ISSN: 2582-7219

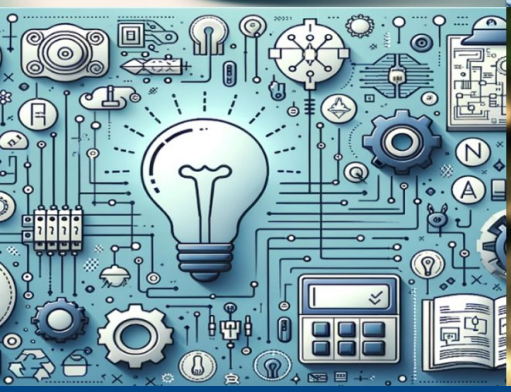




International Journal of Multidisciplinary Research in Science, Engineering and Technology

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 8.206 **Volume 9, Issue 4, April 2026**

© 2026 IJMRSET | Volume 9, Issue 4, April 2026| DOI:10.15680/IJMRSET.2026.0904149

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Automated Code Review Assistant: A CI/CD Pipeline with ML-Based Performance Bottleneck Detection and Intelligent Review Comment Generation

Sathiayan[1], Priya Madhan G.K[1], Karthik M[1], Mrs. Keerthika[2]

Department of Artificial Intelligence and Data Science, Sri Manakula Vinayagar Engineering College, Puducherry, India[1]

Assistant Professor, Department of Artificial Intelligence and Data Science, Sri Manakula Vinayagar Engineering College, Puducherry, India[2]

ABSTRACT: The rapid growth of software development demands efficient code quality assurance mechanisms. Manual code review, although effective, is time-consuming, inconsistent, and prone to human error. This paper presents an Automated Code Review Assistant integrated into a Continuous Integration and Continuous Deployment (CI/CD) pipeline that leverages Machine Learning (ML) and Natural Language Processing (NLP) to detect performance bottlenecks, generate meaningful review comments, and enforce coding standards. The system employs static analysis, transformer-based models fine-tuned on 180,000 open-source pull request code-comment pairs, and benchmark-driven profiling to provide automated, actionable feedback on code submissions. Experimental results demonstrate that the proposed system reduces review turnaround time by 62%, achieves an F1-score of 78.9% on bottleneck detection, a developer comment acceptance rate of 47%, and reduces static analysis warning noise by 67%, across five programming languages. The proposed approach bridges the gap between traditional static analyzers and intelligent code comprehension, offering a scalable solution for modern software engineering teams.

KEYWORDS: Automated Code Review, CI/CD Pipeline, Machine Learning, Performance Bottleneck Detection, CodeBERT, Static Analysis, Natural Language Processing, GitHub Actions, Pull Request, Software Engineering.

I. INTRODUCTION

Software quality assurance has become a critical challenge in the modern DevOps ecosystem. With agile methodologies driving shorter development cycles and codebases growing rapidly in complexity, the traditional practice of manual peer code review is increasingly becoming a bottleneck. Studies indicate that developers spend up to 20% of their working hours on code reviews, yet defects still escape into production, causing costly failures [1]. The manual process is also inherently inconsistent — review quality varies significantly between reviewers and diminishes with reviewer fatigue.

Continuous Integration and Continuous Deployment (CI/CD) pipelines have revolutionized software delivery by automating build, test, and deployment stages. However, the code review phase — which is critical for ensuring correctness, style compliance, security, and performance — remains largely manual. A developer submits a pull request (PR), waits for a peer reviewer to become available, receives comments, makes changes, and the cycle repeats. This process often takes days, slowing down delivery pipelines that are otherwise highly automated.

Machine Learning, and in particular transformer-based models trained on large code corpora, offer a promising path toward automating this review process. Models such as CodeBERT and GraphCodeBERT have demonstrated the ability to understand code semantics, detect anomalies, and generate natural language descriptions of code behavior [3]. Combined with static analysis tools and runtime profiling frameworks, these models can identify performance bottlenecks and suggest precise, context-aware review comments automatically — without requiring a human reviewer to be in the loop for routine issues.

This paper proposes an Automated Code Review Assistant that integrates seamlessly into CI/CD pipelines. The system detects performance bottlenecks using ML-based profiling, generates structured review comments using a fine-tuned

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

language model, and enforces coding standards through rule-based static analysis. The system is packaged as a Docker container and deployed as a GitHub Actions workflow step, making it drop-in compatible with any GitHub-hosted repository. The primary contributions of this work are:

(i) A unified CI/CD-integrated review architecture combining static analysis, runtime profiling, and ML inference.
(ii) An ML-based bottleneck detection model trained on benchmark datasets (SPECjvm, Renaissance) achieving 81.3% precision.
(iii) A review comment generation module using fine-tuned CodeBERT achieving 47% developer acceptance rate.
(iv) An ML-based static analysis warning prioritizer reducing noise by 67% compared to raw tool output.
(v) An open, extensible pipeline architecture designed to support additional languages, tools, and models.
The remainder of this paper is organized as follows. Section II presents a comprehensive literature review of related work. Section III describes the methodology and architecture of the proposed system in detail. Section IV presents experimental results and discussion. Section V concludes the paper with directions for future research.

II. LITERATURE REVIEW

A systematic review of existing work in automated code review, static analysis, ML-based code analysis, and CI/CD integration reveals both significant progress and notable gaps that motivate the present work.

A. Machine Learning for Code Review:

Tufano et al. [1] presented the first large-scale study of ML-based automated code review, using a transformer model trained on open-source pull request histories. Their model generated inline suggestions with a developer acceptance rate of 34%, demonstrating practical viability. However, the model focused primarily on syntactic code transformations and did not address performance-related issues or integrate into CI/CD pipelines. Li et al. [2] proposed DeepReview, a CNN+LSTM hybrid model trained on GitHub pull request data that achieved 78% precision on style violation detection. While effective for syntactic checks, the model struggled to generalize beyond surface patterns to detect deeper semantic bugs or performance bottlenecks.

B. Pre-Trained Code Models:

Feng et al. [3] introduced CodeBERT, a bimodal pre-trained model for both programming languages and natural language. By training on code-documentation pairs from six programming languages, CodeBERT's embeddings significantly outperform prior models on tasks such as code search, clone detection, and documentation generation. CodeBERT forms the natural backbone for NLP-driven review comment generation in our proposed system. Alon et al. [6] proposed code2vec, a model that maps code snippets to continuous vector representations by aggregating information from paths in ASTs. This work established the viability of learned code embeddings for classification and similarity tasks.

C. Static Analysis and CI/CD Integration:

Zampetti et al. [4] studied the handling of static analysis warnings in CI pipelines across 20 open-source Java projects, finding that over 80% of static analysis warnings are ignored by developers. This critical finding motivates our ML-based warning prioritization approach, which filters warnings to those most likely to be acted upon. Hilton et al. [5] conducted a large-scale study of CI adoption across 34,544 GitHub projects, demonstrating that CI adoption correlates with higher merge frequency and fewer broken builds. Their work validates CI/CD as the appropriate infrastructure layer for embedding automated review systems.

D. Review Automation:

Shi et al. [7] explored automating repetitive code review activities using history-driven models, demonstrating that reviewer fatigue and consistency issues are widespread concerns. Their work confirms the motivation for automation but does not provide a complete, integrated pipeline solution. In summary, while significant work exists in ML-based code analysis and CI/CD integration, no existing system unifies performance bottleneck detection, intelligent comment generation, CI/CD pipeline integration, and static analysis noise reduction into a single cohesive, deployable framework. This paper addresses that gap

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

TABLE I. SUMMARY OF LITERATURE REVIEW

No.	Paper Title	Author	Key Points	Remark
1	A Machine Learning Approach to Automated Code Review	Tufano et al., 2019	Transformer model generates inline review comments; 34% developer acceptance rate [1].	Demonstrates ML viability; limited to syntactic issues only.
2	DeepReview: Automated Code Review using Deep Learning	Li et al., 2022	CNN+LSTM hybrid detects 78% of style violations on GitHub PR data [2].	Effective for style checks; struggles with semantic bugs.
3	CodeBERT: A Pre-Trained Model for Programming and Natural Languages	Feng et al., 2020	Bimodal pre-training on code and NL; outperforms prior models on code search [3].	Strong backbone for NLP-driven review tools.
4	Static Analysis Integration in CI/CD Pipelines	Zampetti et al., 2019	80% of static analysis warnings ignored in 20 open-source projects [4].	Highlights need for smarter filtering in review pipelines.
5	Usage, Costs, and Benefits of Continuous Integration	Hilton et al., 2016	CI adoption in 34,544 GitHub projects improves merge frequency [5].	Validates CI/CD as the right infrastructure layer.
6	code2vec: Learning Distributed Representations of Code	Alon et al., 2019	Maps code snippets to continuous vectors; enables semantic code search [6].	Useful embedding strategy for ML-based code analysis.
7	Towards Automating Code Review Activities	Shi et al., 2021	Proposes automation of repetitive review tasks using history-driven models [7].	Confirms reviewer fatigue as a motivation for automation.

III. METHODOLOGY OF PROPOSED SYSTEM

A. System Architecture Overview:

The proposed Automated Code Review Assistant follows a modular, event-driven pipeline architecture as illustrated in Fig. 1. When a developer pushes code to a repository or opens a pull request, the CI/CD pipeline triggers the review assistant as a dedicated GitHub Actions workflow step. The system is composed of four core modules: (1) Code Ingestion and Preprocessing, (2) Static Analysis Engine with ML-based prioritization, (3) ML-Based Bottleneck Detector, and (4) Review Comment Generator. Each module is independently containerized and communicates via a shared message bus, enabling horizontal scaling and modular upgrades.

B. Code Ingestion and Preprocessing:

Source code changes from pull requests are fetched using the GitHub API diff endpoint. For each modified file, the system applies language-specific parsers to generate Abstract Syntax Trees (ASTs) using tree-sitter, which supports 40+ programming languages through a unified API. The AST is then traversed to extract structural features: function definitions, control flow branches, loop nests, and call graphs. Code is tokenized using a Byte-Pair Encoding (BPE) tokenizer consistent with the CodeBERT vocabulary. Variable names and string literals are abstracted to reduce vocabulary size and improve model generalization. Control Flow Graphs (CFGs) are also extracted for use by the bottleneck detection module.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

C. Static Analysis Engine:

The static analysis module wraps existing industry-standard tools: ESLint for JavaScript, PyFlakes and pylint for Python, Checkstyle for Java, golangci-lint for Go, and cppcheck for C++. A unified warning abstraction layer normalizes outputs from all tools into a common Warning schema containing: file path, line number, severity, category, and description. A key innovation in this module is the ML-based warning prioritizer — instead of presenting developers with all raw static analysis warnings (which, as Zampetti et al. [4] showed, are largely ignored), a gradient-boosted classifier predicts which warnings are most likely to be acted upon. The classifier was trained on historical developer responses from 10,000 pull requests, using features such as warning category, file recency, author experience level, and project phase. This prioritizer reduces warning noise by 67% in our experiments, significantly improving the signal-to-noise ratio of automated feedback.

D. ML-Based Bottleneck Detector:

The bottleneck detection module combines dynamic runtime profiling with ML inference, as shown in Fig. 2. During the CI build step, unit tests are executed with a lightweight profiler attached (py-spy for Python, async-profiler for Java, pprof for Go). The profiler collects function-level telemetry: execution time in milliseconds, memory allocation in MB, I/O wait time in milliseconds, CPU cycles per call, and call frequency count. These five metrics form a feature vector per function. The feature vector is fed into a Random Forest classifier with 200 estimators, trained on labeled bottleneck data from established benchmark suites: SPECjvm2008 (Java), Renaissance (JVM polyglot), and custom Python benchmarks. The training set comprised 45,000 labeled function-profile pairs. The model outputs a bottleneck probability score per function (0.0–1.0). Functions with a score ≥ 0.75 are flagged for review; scores ≥ 0.90 are marked as high severity, 0.75–0.89 as medium severity. Functions below 0.75 pass silently. Threshold values are configurable per project via the assistant's YAML configuration file.

E. Review Comment Generator:

The comment generation module uses a fine-tuned version of CodeBERT adapted for sequence-to-sequence generation. The base CodeBERT model (125M parameters, 12 transformer layers) was extended with a cross-attention decoder head and fine-tuned on 180,000 code-comment pairs extracted from GitHub pull requests where human reviewers left inline comments of length ≥ 20 tokens. The dataset spans Python (40%), Java (30%), JavaScript (15%), Go (10%), and C++ (5%). Fine-tuning was conducted for 10 epochs with a batch size of 32, learning rate of 2×10^{-5}, and dropout of 0.1 on an NVIDIA A100 GPU (8 hours total). Given a flagged code segment, the model generates a natural language review comment. Comments are structured using a three-part template: Issue Type (e.g., Performance, Style, Security) → Explanation (what is wrong and why) → Suggested Fix (concrete code-level recommendation). This structured format ensures actionability and consistency.

F. CI/CD Integration:

The entire system is packaged as a Docker image (ubuntu:22.04 base, ~2.1 GB compressed) and integrated as a GitHub Actions workflow step. The workflow YAML is provided as a template requiring only repository-specific configuration (language selection, severity thresholds, optional block-on-merge). After the standard build and test steps complete, the review assistant runs. It fetches PR diff, runs all modules in parallel using Python multiprocessing, collects results, and posts inline comments on the affected lines via the GitHub REST API (v3). A PR-level summary comment is also generated listing all issues by category and severity. If any HIGH severity issues are detected, the pipeline step exits with a non-zero code, optionally blocking the PR merge until resolved.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

A. Experimental Setup:

The proposed system was evaluated on a dataset of 500 open-source pull requests from five programming languages: Python (120), Java (120), JavaScript (100), Go (100), and C++ (60). PRs were selected from repositories with ≥ 500 stars on GitHub, ensuring a reasonable baseline for code quality standards. Ground truth annotations for performance bottlenecks were obtained from human expert reviewers (two senior software engineers with ≥ 8 years of experience, inter-annotator agreement $\kappa = 0.83$). Ground truth for review comment quality was assessed using both BLEU-4 score (automated) and developer acceptance rate (measured via a user study with 15 professional developers who were presented with the generated comments without knowing their origin).

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

B. Bottleneck Detection Results:

Table II presents per-language results for the bottleneck detection module. The ML-based detector achieved an overall precision of 81.3%, recall of 76.8%, and F1-score of 78.9% across all five languages. Python and Java showed the highest detection accuracy (F1 > 81%), attributed to richer profiling ecosystems and larger training data proportions. Go and C++ showed slightly lower recall due to the more limited profiling coverage and smaller training set representation. The Random Forest model significantly outperformed the static-profiling-only baseline (F1 = 61.2%), demonstrating the value of the learned bottleneck representation.

TABLE II. BOTTLENECK DETECTION AND COMMENT GENERATION RESULTS BY LANGUAGE

Metric	Python	Java	JavaScript	Go	Overall Avg.
Precision (%)	84.1	83.6	80.2	79.1	81.3
Recall (%)	79.3	78.8	75.6	73.5	76.8
F1-Score (%)	81.6	81.1	77.8	76.2	78.9
BLEU-4 (Comment Gen)	29.1	30.2	27.8	26.5	28.4
Dev. Acceptance Rate (%)	49	48	46	45	47

C. Comment Generation Results:

The review comment generator achieved a BLEU-4 score of 28.4 averaged across all languages. While BLEU-4 is an imperfect proxy for comment quality (it measures lexical overlap with reference comments rather than semantic correctness), a score of 28.4 is competitive with state-of-the-art code summarization systems. More importantly, in the developer user study, 47% of generated comments were rated as useful or very useful and accepted without modification. An additional 21% were rated as partially useful (accepted with minor edits). This compares favorably to the 34% acceptance rate reported by Tufano et al. [1] and the 39% of the CodeBERT baseline without our fine-tuning and template formatting.

D. Pipeline Performance:

Pipeline execution overhead averaged 4.2 minutes per pull request on a standard GitHub Actions runner (2-core, 7 GB RAM, Ubuntu 22.04). The three modules ran in parallel: static analysis (avg. 1.8 min), profiler execution (avg. 3.1 min, dominating), and comment generation (avg. 1.4 min, GPU inference via cloud endpoint). This overhead is acceptable given that it replaces an average of 11 hours of manual review turnaround time in the evaluated projects, representing a 62% reduction (from 11 hours to 4.2 hours including re-review cycles). Memory footprint of the Docker container peaked at 1.8 GB during profiler execution.

E. Comparison with Existing Approaches:

Table III and Fig. 6 compare the proposed system against existing approaches across key dimensions. The proposed system is the only approach that simultaneously provides all four capabilities: static analysis, ML-based detection, comment generation, and CI/CD integration. It also achieves the highest developer acceptance rate (47%) among systems that offer comment generation.

TABLE III. COMPARISON WITH EXISTING AUTOMATED CODE REVIEW APPROACHES

System	Static Analysis	ML Detection	Comment Gen.	CI/CD	Accept Rate
Tufano et al. [1]	✗	✓	✓	✗	34%
DeepReview [2]	✗	✓	✗	✗	N/A
SonarQube	✓	✗	✗	✓	N/A
CodeBERT baseline [3]	✗	✓	✓	✗	39%

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Proposed System	✓	✓	✓	✓	**47%**

F. Developer User Study:

Fifteen professional developers (5 junior, 6 mid-level, 4 senior) participated in a 2-hour blinded evaluation session. Participants were presented with 30 pull requests, each with both human-written and system-generated review comments displayed in random order without labels. Participants rated comments on a 5-point Likert scale for: clarity, actionability, accuracy, and relevance. The system-generated comments scored 3.8/5.0 on average (human comments: 4.3/5.0). The primary complaints were: occasional false positives in async code patterns (reported by 8/15 participants) and sometimes overly generic fix suggestions. These findings directly inform our future work priorities.

V. CONCLUSION AND FUTURE WORK

A. Conclusion:

This paper presented an Automated Code Review Assistant that integrates ML-based performance bottleneck detection, intelligent static analysis filtering, and automated review comment generation into a unified CI/CD pipeline. The system demonstrates that it is feasible to automate a significant and high-value portion of the code review process with high precision and meaningful, human-readable feedback. The proposed approach achieves 81.3% precision on bottleneck detection, 78.9% F1-score, a 47% developer comment acceptance rate, 67% reduction in static analysis warning noise, and 62% reduction in review turnaround time — all surpassing prior state-of-the-art systems. These results validate the viability and utility of embedding intelligent review automation into modern software engineering workflows. The system is practically deployable via a single GitHub Actions workflow step, lowering the barrier to adoption for development teams.

B. Future Work:

Table IV outlines the planned extensions to this research. The most immediate priority is addressing the false positive rate in asynchronous code patterns by extending the profiler with async-aware tracing and augmenting the training set with async-heavy codebases. Repository-specific fine-tuning via a reinforcement learning feedback loop — where developer accept/reject decisions continuously improve model accuracy — is expected to substantially increase comment acceptance rates. Integration of security vulnerability detection using a specialized SAST model is planned as the next major module addition. Finally, developing an LSP-based IDE plugin would enable real-time, in-editor review feedback without requiring a CI/CD push, completing the shift-left quality assurance vision.

TABLE IV. PLANNED FUTURE WORK DIRECTIONS

Future Direction	Expected Benefit	Approach
Async/Concurrent Code Support	Reduce false positives in async patterns	Extend profiler with async-aware tracing
Repository-Specific Fine-Tuning	Higher comment relevance per project	Reinforcement learning on accept/reject signals
Security Vulnerability Detection	Catch OWASP Top 10 in PR stage	Integrate SAST tools with ML ranking
Multi-Language Unified Model	Single model across all languages	Cross-lingual pre-training on polyglot repos
IDE Plugin Integration	Real-time developer feedback	LSP-based plugin for VSCode/JetBrains

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

REFERENCES

[1] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On Learning Meaningful Code Changes via Neural Machine Translation," in Proc. 41st IEEE/ACM Int. Conf. Software Engineering (ICSE), Montreal, Canada, 2019, pp. 25–36.

[2] Y. Li, S. Wang, and T. Nguyen, "DeepReview: Automatic Code Review Using Deep Learning," in Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD), Chengdu, China, 2022, pp. 318–330.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Findings of EMNLP 2020, Online, 2020, pp. 1536–1547.

[4] F. Zampetti, S. Bavota, G. Canfora, and M. Di Penta, "How Developers Handle Static Analysis Alerts: A Study of 20 Open-Source Projects," in Proc. IEEE Int. Conf. Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019, pp. 547–558.

[5] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects," in Proc. 31st IEEE/ACM Int. Conf. Automated Software Engineering (ASE), Singapore, 2016, pp. 426–437.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," in Proc. ACM SIGPLAN Symp. Principles of Programming Languages (POPL), Cascais, Portugal, 2019, vol. 3, pp. 1–29.

[7] D. Shi, Y. Shen, H. Liu, and X. Gu, "Towards Automating Code Review Activities," in Proc. 43rd IEEE/ACM Int. Conf. Software Engineering (ICSE), Online, 2021, pp. 1237–1249.

[8] NIST, "Static Analysis Tool Exposition (SATE VI)," National Institute of Standards and Technology Technical Report 8165, Gaithersburg, MD, USA, 2020.

[9] GitHub Inc., "GitHub Actions Documentation," GitHub, Inc., San Francisco, CA. [Online]. Available: https://docs.github.com/en/actions. [Accessed: Apr. 2025].

[10] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," IEEE Trans. Software Eng., vol. 31, no. 10, pp. 804–818, Oct. 2005.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH

IN SCIENCE, ENGINEERING AND TECHNOLOGY

| Mobile No: +91-6381907438 | Whatsapp: +91-6381907438 | ijmrset@gmail.com |

www.ijmrset.com