International Journal of Mathematics and Computer Research
ISSN: 2320-7167
Volume 14 Special Issue 03 March 2026, Page no. – 71-75
Impact Factor: 8.751
DOI: 10.47191/ijmcr/v14iSPC3.15

Applying Natural Language Processing (NLP) For Automated Code Review

Prajakta Patil, Vandana Nemane
Department of Computer Science, Dr. D. Y. Patil, Arts, Commerce & Science College, Pimpri, Pune, Maharashtra, India

ARTICLE INFO

Published Online:
14 March 2026

Corresponding Author:
Prajakta Patil

ABSTRACT

Automated code review aims to reduce manual effort, catch defects early, and improve code quality by using machine learning and Natural Language Processing (NLP) to analyze source changes, generate review comments, and prioritize reviewer attention. This paper surveys prior work and presents a practical methodology combining transformer-based code models (e.g., CodeBERT/CodeT5-style encoders), structural program representations (ASTs/graphs), and comment-generation components to build an automated code review assistant. We describe dataset collection from open-source pull requests, preprocessing, model design, evaluation metrics, and an implementation plan. Finally, we discuss expected benefits, limitations, and directions for future work. (arXiv)

KEYWORDS: Automated code review, Natural Language Processing, transformers, CodeBERT, comment generation, program analysis, pull requests.

1. INTRODUCTION

Code review is an essential software-engineering practice: besides defect detection, it supports knowledge sharing and maintainability. However, manual review is time-consuming — industry surveys show developers spend nontrivial hours weekly on reviews, motivating automation to reduce routine work and highlight high-value issues. Modern NLP and large pre-trained transformer models adapted to source code have made automating parts of code review feasible: models can classify review-relevant files, detect likely defects, and even generate reviewer-style comments or suggested patches. Prior work has demonstrated early progress in automating individual review tasks and explored transformer-based encoders and graph-based structures for source-code understanding. (arXiv)

This paper proposes a replicable methodology for building an NLP-driven automated code review system that (1) classifies whether a pull request (PR) likely needs manual attention, (2) recommends targeted reviewer comments, and (3) proposes small automated repairs for certain classes of issues.

2. RELATED WORK

Code scrutiny is a crucial stage in software development that guarantees code quality, safety, and maintainability. It involves closely reviewing source code before to delivery in order to identify errors, security flaws, and performance issues.

2.1. Existing automated tools for code review (e.g., linters, static analysis tools)

To help with code review and expedite the procedure, automated tools have been created. Among the most popular automated solutions are linters and static analysis tools. Linters check code for grammatical mistakes, style infractions, and possible defects to make sure coding standards are followed. By examining the code's flow and identifying potential weaknesses, performance bottlenecks, or places for improvement, static analysis tools can go farther. By automatically identifying problems, these tools drastically cut down on the amount of effort needed on human review; but, their capacity to comprehend code context and offer more detailed, high-level comments is constrained. They can enforce style requirements and identify basic faults, but they frequently overlook more complicated problems like logic errors or the code's overall meaning.

2.2. Role of NLP in software engineering, specifically in code review automation

By making it possible to comprehend both the syntax and semantics of code, natural language processing (NLP) significantly advances code review automation. NLP models can capture more complicated structures in code by comprehending purpose and context, whereas traditional static analysis techniques just concentrate on syntax. Text classification, sentiment analysis, and code summarization are examples of NLP techniques that can be used to automate issue identification, provide suggestions for enhancements,

and even produce natural language explanations. In order to make code review tools smarter and more context-aware, recent software engineering research has investigated the integration of natural language processing (NLP). For example, a more comprehensive approach to automated code review can be achieved by training NLP models to automatically classify code changes, detect code smells, and recommend best practices.

Table 1. NLP-Based Automation Capabilities in Code Review

NLP Technique	Use Case in Code Review	Strengths	Limitations
Text classification	Categorizing changes: bug, security, style, refactor	Improves feedback quality and learning outcomes.	Requires labeled data, limited generalization
Sentiment analysis	Evaluating tone of reviewer comments, constructive vs negative	Ensures constructive and respectful communication.	Not universally reliable in domain context
Code summarization	Generating summaries of code diffs or functions	Helps reviewers understand complex code segments quickly.	May miss semantics, language-specific nuances
Transformer models	Bug detection, refactoring suggestions, summarization (e.g., CodeBERT,GPT-3)	Learns patterns beyond simple rules, provides natural language feedback	High compute cost; may generate inaccurate suggestions
Graph-aware models	Using AST or PDG enriched data (e.g., GraphCodeBERT)	Better semantics understanding, structure-aware analysis	Complex training, less mature tools

2.3. Approaches to Automating Code Review with NLP

An overview of NLP methods (such as text categorization, sentiment analysis, and code summarization) that are pertinent to code review Code review and other software development chores can now be automated with the help of Natural Language Processing (NLP). When it comes to code review automation, a number of NLP approaches are especially helpful. Text classification is one such method in which the system classifies code modifications according to pre-established labels. This could assist in identifying whether a specific modification is only an enhancement in style or adds a bug or security problem. Sentiment analysis is another pertinent NLP approach that can be used to evaluate the "tone" or quality of a code review, even though it is often employed to examine human sentiment. For instance, it could assist in spotting excessively positive or negative comments, guaranteeing that reviewers offer helpful critiques. Another crucial NLP method is code summarizing, in which a model creates a concise synopsis of the code to help reviewers grasp the main points of modifications without having to read the full code. This enables quicker and more effective code reviews, which is especially helpful when working with complicated algorithms or big codebases.

3. PROPOSED METHODOLOGY

3.1 Objectives

1. **PR triage** — predicts which PRs are likely to require in-depth human review vs. those that can be auto-approved or require only minor fixes.
2. **Comment generation** — produce reviewer-style comments (e.g., pointing out bugs, style violations, or missing tests).
3. **Suggested fixes** — for a subset of comments (e.g., formatting, docstring consistency, small API misuse), generate patch suggestions.

3.2 Overall architecture

The system has three main modules:

A. **Preprocessing & Feature Extraction**

- Parse changed files and extract: diff hunks, file-level metadata (language, lines changed), code ASTs, and surrounding context (function/class).
- Extract natural-language artifacts: commit message, PR title/body, existing reviewer comments, and linked issue text.

B. Representation Learning

- **Code encoder**: a transformer-based encoder pre-trained or fine-tuned on code (e.g., CodeBERT/CodeT5). For structural signals, augment tokens with AST-derived positional embeddings or feed a graph neural network (GNN) over program dependency graph, combined with transformer outputs.
- **Text encoder**: a standard transformer encoder for PR text and issue descriptions.
- Fuse code and text representations via concatenation + cross-attention layers to capture correspondence between code changes and natural-language context.

C. Task Heads

- **Triage classifier**: binary/multi-class head (e.g., softmax) that predicts review urgency/severity buckets.
- **Comment generator**: sequence-to-sequence decoder (transformer/encoder–decoder) conditioned on fused representation to generate reviewer comments. Use constrained decoding to encourage concise, actionable outputs.

- **Patch proposer (optional)**: for certain comment classes, decode a suggested code edit (e.g., small AST rewrite) or produce a code snippet replacement, validated by lightweight static checks or unit tests.

3.3 Training objectives and multi-task learning

- **Classification loss**: cross-entropy on triage labels extracted from historical PR outcomes (e.g., number of requested changes, review time).
- **Generation loss**: teacher-forcing cross-entropy for comment generation against historical reviewer comments. Augment with sequence-level metrics (policy-gradient style fine-tuning using BLEU/ROUGE/CodeBLEU as reward).
- **Auxiliary tasks**: language modeling on code, comment classification (issue/bug/style), and binary "auto-fixable" label for patch proposer training.

3.4 Datasets

Source: large-scale GitHub pull request datasets (public PRs with diffs, comments, and merge metadata). Use curated datasets from prior research when available. Preprocessing filters: exclude extremely large PRs, remove private/repo-sensitive data, and anonymize identifiers. (Examples of such datasets and prior collection methodologies are described in the literature.) (arXiv)

3.5 Evaluation metrics

Triage: precision, recall, F1, ROC-AUC for urgency/severity prediction.

Comment generation: BLEU/ROUGE, METEOR, and **Code-aware metrics** (CodeBLEU) plus human evaluation for helpfulness, actionability, and false-positive rate.

Patch proposer: correctness measured by unit-test pass (when available), static-analysis checks, and human reviewer acceptance rate.

4. IMPLEMENTATION PLAN

4.1 Data collection & preprocessing

1. Clone datasets of public PRs (e.g., via GH Archive / GHTorrent or datasets used in earlier studies). Extract: diffs, file paths, languages, PR title/body, and reviewer comments.
2. Map reviewer comments to specific hunks when possible (align comment positions to diff hunks). Label triage target from historical review signals (e.g., PRs with >N comments or >T hours between opening and merge = "high attention required").
3. Tokenize code using language-aware tokenizers; construct ASTs with tree-sitter or language-specific parsers. Normalize identifiers for generalization.

4.2 Model & training details

- **Base models**: initialize with a pre-trained CodeBERT/CodeT5-style encoder for code tokens and a BERT-like encoder for PR text; use a transformer decoder for generation. Use multi-task fine-tuning.
- **Hyperparameters** (example starting point): learning rate 1e-5 (encoder), 3e-5 (task heads), batch size 16–64 depending on GPU memory, sequence length up to 1024 tokens (longer for diffs — consider Longformer/LED variants).
- **Infrastructure**: training on one or more GPUs (e.g., A100/RTX-class), experiment tracking (Weights & Biases), and CI to run evaluation. For production, serve a distilled/lightweight model for inference latency.

4.3 Safety and human-in-the-loop design

Always present generated comments and suggested patches as **recommendations**; require human approval for merging. Provide provenance (confidence score, supporting rationale/trace-back to code lines) to build reviewer trust. Include a feedback loop: when a reviewer accepts/rejects a suggestion, log for continual fine-tuning.

5. IMPLEMENTATION AND (ILLUSTRATIVE) RESULTS

The present document describes an implementation plan and evaluation framework. Running the experiments on real PR datasets is required to report actual quantitative results; below we provide an illustrative example of how results would be presented and interpreted.

5.1 Experimental setup (example)

- **Training set**: 50k PRs from mid-sized repos across Java, Python, and JavaScript.
- **Validation/test**: 10k PRs held out, ensuring no file overlap with training.
- **Baselines**: (1) heuristic rule-based linters, (2) a classifier using only metadata (lines changed, files touched), (3) CodeBERT fine-tuned only for the target task.

5.2 Example (hypothetical) results table

Task	Metric	Baseline (rule)	CodeBERT-only	Proposed (Code+AST+Fusion)
Triage	F1 (high-attention)	0.62	0.72	**0.78**
Comment gen	BLEU	12.5	18.2	**20.6**
Comment gen	Human helpfulness (%)	45%	61%	**69%**
Patch proposer	Acceptance rate (human)	N/A	18%	**31%**

Note: numbers above are illustrative, not empirical results. They demonstrate expected directions of improvement when adding structural code signals and multi-task learning.

5.3 Qualitative observations (expected)

- Transformer-based encoders reduce trivial false positives compared to naive linters by capturing semantics (e.g., flagging a potential API misuse rather than stylistic differences).
- Comment generation benefits from conditioning on PR text and historical context — e.g., models can reference linked issues or tests that changed.
- The patch proposer is valuable for small, deterministic fixes (formatting, off-by-one in loops where tests exist) but should be restricted for complex semantic changes.

5.4 Practical deployment notes

- Latency: full encoder–decoder inference over large diffs can be slow; practical deployments use early triage for small diffs and serverless inference with model caching for common code patterns.
- Trust & evaluation: continuous human evaluation is required to detect model drift and to tune false-positive thresholds.

6. DISCUSSION

Recent industry and academic studies indicate growing adoption of LLM/transformer-based automated code review tools and user studies measuring their impact on review speed and reviewer workload. However, challenges remain in ensuring comment correctness, avoiding hallucinations (i.e., confidently suggesting incorrect fixes), and integrating suggestion provenance so reviewers trust and efficiently use recommendations. Human-in-the-loop designs and conservative auto-fix thresholds are important safety measures. (arXiv)

7. CONCLUSION AND FUTURE SCOPE

We presented a complete methodology for applying NLP to automate parts of the code review process: from data collection and preprocessing to a multi-headed model combining code and text representations, and to an evaluation plan that balances automatic metrics with human judgement. While automated triage and comment generation can reduce reviewer workload, real-world adoption requires careful interface design, conservative automated edits, and continuous evaluation.

Future work includes:

- Extending to multilingual codebases and cross-repo transfer learning.
- Integrating stronger formal verification or symbolic checks for higher-assurance patch proposals.
- Exploring continual learning with reviewer feedback to reduce model drift.
- Better human-centered evaluation (longitudinal studies measuring team productivity and code quality). (PMC)

REFERENCES

1. R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards Automating Code Review Activities," Proc. 43rd Int. Conf. Software Engineering (ICSE), 2021, pp. 163–174.
2. Y. Yin, Y. Zhao, Y. Sun, and C. Chen, "Automatic Code Review by Learning the Structure Information of Code Graph," Sensors, vol. 23, no. 5, art. 2551, Feb. 2023.
3. G. Tucudean, C. Pop, and D. Petrescu, "Natural Language Processing with Transformers: A Review," MDPI Applied Sciences, 2024.
4. A. Frömmgen, M. Beller, and A. Zeller, "Resolving Code Review Comments with Machine Learning," Google Research Technical Report, 2024.
5. G. Zhao, D. A. da Costa, and Y. Zou, "Improving the Pull Requests Review Process Using Learning-to-Rank Algorithms," Empirical Software Engineering, vol. 24, no. 3, pp. 1759–1787, 2019.
6. X. Chen, Y. Liu, and M. Zhao, "Automatic Code Review by Learning Code Semantics and Reviewer Behaviors," IEEE Trans. Software Engineering, vol. 46, no. 8, pp. 850–862, 2020.
7. S. Nate, O. Patil, S. Medar, and J. Deshmukh, "A Survey on Transformer-based Models in Code Summarization," Int. Res. J. Adv. Eng. Hub (IRJAEH), vol. 3, no. 3, pp. 740–745, Mar. 2025.
8. "Promises and Perils of Using Transformer-based Models for Software Engineering Research," Empirical Software Engineering, Elsevier, 2024.
9. R. Tufano, M. Tufano, D. Poshyvanyk, and G. Bavota, "Impact Studies on LLM-Generated Review Comments and Reviewer Interactions," arXiv Preprint arXiv:2405.10234, 2024.
10. Y. Yin, Y. Zhao, Y. Sun, and C. Chen, "Automatic Code Review by Learning the Structure of Code Graph," MDPI Sensors, 2023.
11. U. Cihan, V. Haratian, A. İcöz, M. K. Gül, Ö. Devran, E. F. Bayendur, B. M. Uçar, and E. Tüzün, "Automated Code Review In Practice," Bilkent University Technical Report, 2024. ResearchGate
12. Y. Kartal, "Automating Modern Code Review Processes with Transformer-based Models," Computers & Security, vol. ?, no. ?, pp. ?, 2024. ScienceDirect
13. H. Y. Lin, P. Thongtanunam, C. Treude, M. W. Godfrey, C. Liu, and W. Charoenwet, "Leveraging Reviewer Experience in Code Review Comment Generation," arXiv Preprint arXiv:2409.10959, 2024. arXiv+1
14. Z. Rasheed, M. A. Sami, M. Waseem, K.-K. Kemell, X. Wang, A. Nguyen, K. Systä, and P.

Abrahamsson, "AI-Powered Code Review with Large Language Models: Early Results," arXiv Preprint arXiv:2404.18496, 2024. arXiv

15. T. Sun, J. Xu, Y. Li, Z. Yan, G. Zhang, L. Geng, Z. Wang, Y. Chen, Q. Lin, and W. Duan, "BitsAI-CR: Automated Code Review via Large Language Models in Practice," arXiv Preprint arXiv:2501.15134, 2025.