IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510105

An AI Powered Automated Code Review System Using Large Language Models and Static Analysis

Mrs. Nita Meshram[1], C. Balaji Naidu[2], Ganni Naveen Raj Anudeep[3], K. Vedavyas[4], K. Harsha Nandhan[5]

Associate Professor, Department of Computer Science and Engineering,

K. S. School of Engineering and Management Bengaluru, Karnataka, India[1]

Department of Computer Science and Engineering,

K. S. School of Engineering and Management Bengaluru, Karnataka, India[2-5]

Abstract: Reviewing pull requests by hand is slow and in-consistent, and the automated aids available today each solve only half of the problem: static analysers enforce fixed rules but cannot grasp the context of a change, while large language models (LLMs) can reason about code yet may return inaccu-rate or non-reproducible feedback. This paper surveys work that pairs LLMs with static or program analysis for automated code review, looking at system architectures, the way the two components are combined, the evidence reported, and the de-sign questions that remain open. Ten sources were examined, and five systems with enough accessible detail were set side by side. Combining an LLM with a rule-based or analysis-based component recurs across the set, yet the designs differ in how analysis supports the model, in deployment cost, and in the strength of their evidence. That evidence is uneven: one source gives a precision figure for generated comments, one records favourable reactions from developers, and the rest offer quali-tative or limited detail, so ranking the systems would be unjus-tified. Six open issues are identified: comparable quantitative evaluation, deployment cost and accessibility, merging of con-textual and rule-based findings, verification and reproducibility of model output, handling of failures, and language coverage. A related pull request review system is placed against these is-sues; its software modules are reported as implemented, while end-to-end validation and the final merged report are still in progress. The contribution is therefore a comparison limited to its sources, a gap analysis that stays within the evidence, and a plan for evaluation, not a claim that any system is better.

Index Terms: automated code review, large language mod-els, static analysis, Semgrep, pull request, hybrid review

I. INTRODUCTION

Code review is a gate that code changes pass before they are merged. In practice, a reviewer must read a pull re-quest, understand its purpose, and check it for defects, security risks, performance problems, and maintainability concerns. This takes time and concentration. Small repeti-tive points, such as unused variables, naming and clear-cut hazards, can consume attention that design and logic de-serve, and large or frequent pull requests make it easier for defects to slip through.

Two families of automated support exist, and they should not be treated as interchangeable. Static analysis tools match code against fixed rules and give consistent, reproducible reports, but they cannot judge the intent or context of a change. LLMs can reason about code and ex-plain their findings in natural language, but their output may be inaccurate, inconsistent, or impossible to repro-duce. A tool that applies both and delivers one structured report is therefore attractive, yet the term "hybrid review" covers systems with materially different designs, costs, and levels of evidence.

This review focuses on approaches that combine an LLM with static or program analysis for code review, with particular interest in lightweight designs that operate on GitHub pull requests. Out of scope are the fine-tuning of code models, automated program repair in general, and large-scale benchmarks of LLMs that lack an analysis com-ponent. A pull request review system built by the authors with Gemini AI and Semgrep prompted this study; Sec-tion VII describes it, and it is not used as validated evi-dence.

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026
DOI: 10.17148/IJARCCE.2026.1510105

Four questions guide the paper. RQ1: what architec-tures combining LLMs with static or program analysis are described in the available literature? RQ2: how does the analysis component support or check the model? RQ3: what evidence of evaluation is given, and can it be com-pared across sources? RQ4: which open issues matter for a lightweight design that works at pull request level? The pa-per offers a structured comparison of the accessible sources, a separation of measured results from design claims, a gap analysis kept within the evidence, and an open account of the status of a related project.

II. REVIEW METHOD AND SCOPE

The approach is a structured narrative review rather than a systematic review or a meta-analysis. Sources were taken from the reference list and literature survey of the re-lated project report, which cover LLM-based review, sym-bolic and static-analysis support for LLMs, and industrial deployments.

The retained set contains ten sources. Five systems form the main comparison, since the project literature survey gives enough information to discuss the approach, the role of analysis, and the reported evidence: Ericsson's experi-ence report [1], an LLM approach with symbolic reason-ing [2], FORGE [3], a work on LLMs with program analysis for code understanding [4], and BitsAI-CR [5]. Five fur-ther sources [6]–[10] appear in the project's reference list as related background but are not analysed in the project record; they are cited for context only and play no part in the detailed technical comparison.

The project report does not preserve how the sources were found: the search engines or databases, the query strings and dates, any removal of duplicates, the screen-ing counts, and the inclusion and exclusion criteria are all missing. Rebuilding them now would mean guessing, so no such account is given. The selection is therefore neither exhaustive nor fully repeatable, and the findings are a syn-thesis of a limited source set, not a survey of everything published on the topic.

Each source was read along five lines: approach and components, role of the analysis component, role of the LLM, type of evaluation with its reported outcomes, and limitations visible in the accessible text. Measurements, qualitative remarks, design descriptions and the authors' own inferences are each labelled as such. When the acces-sible text does not establish something, it is recorded as unreported in the accessible text and is not assumed to be missing from the full work.

III. TECHNICAL BACKGROUND AND SYSTEM ARCHITECTURES

A common conceptual architecture for hybrid review has four interacting stages. An input stage obtains the change under review, typically a diff and, where needed, the surrounding source. An analysis stage applies rule-based or program-analysis techniques that produce deter-ministic findings or structured context. A reasoning stage uses an LLM to evaluate the change, explain issues, and suggest fixes. A reporting stage combines the outputs into a form that developers can act on. A robust design must also define how outputs of the stages relate: a rule viola-tion and an LLM comment about the same line are not automatically the same finding, and a fluent explanation does not by itself establish that a defect exists.

The reviewed sources use the analysis component in dif-ferent ways. In one pattern, analysis supplies context to the model, for example by extracting the method that en-closes each changed line [1]. In another, symbolic reason-ing is added to the LLM to counter shallow or hallucinated output [2]. In a third, a static analysis framework detects issues and the LLM rewrites the affected code, with a com-parison tool catching erroneous edits [3]. A fourth pattern uses a rule-based checker first and a filter stage afterwards to verify the generated comments [5]. These patterns mea-sure different proxies for review quality, and none of the sources benchmarks these patterns against one another, so no pattern can be declared categorically superior on the evidence considered.

Input choice also matters. A unified diff keeps the model focused on modified lines and limits input size, but it omits code outside the change. Rule-based tools such as Semgrep operate on complete source files and cannot analyse a diff fragment, so a pull-request-level design must obtain both representations and tie them to one commit. Pinning the analysis to the head commit SHA ensures that the version analysed is the version under review even if the branch changes afterwards.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510105

Finally, the output layer is equally important. Struc-tured output with fixed fields such as category, severity, ex-planation, and suggested fix makes findings easier to store, merge, and display. The accessible descriptions say lit-tle about how findings from different sources are merged, deduplicated and mapped to one severity scale, or how such systems behave when APIs fail or rate limits are hit. Without this, nothing can be said about how reliable such systems are in operation.

IV. FINDINGS FROM THE REVIEWED SYSTEMS

The reviewed work ranges from industrial deployments to research prototypes. Ericsson's experience report de-scribes a light tool that pairs an LLM with static pro-gram analysis, in which the analysis extracts the enclosing method of each changed line so the model receives suitable context [1]. İçöz and Biricik start from the observation that manual review is subjective and slow and that review by an LLM alone is frequently shallow and liable to hal-lucinate; they propose adding symbolic reasoning to open-source models specialised for software engineering tasks [2]. Abtahi and Azim combine a static analysis framework with LLMs such as GPT-3.5 Turbo and GPT-4o, refine prompts iteratively, use retrieval-augmented generation, and em-ploy a comparison application to catch erroneous LLM changes [3]. Chen et al. use LLMs together with program analysis to strengthen a model's understanding of code [4]. BitsAI-CR, used in practice at ByteDance, operates in two stages: a RuleChecker flags issues using a taxonomy of re-view rules and a ReviewFilter checks comments to improve precision [5].

The evidence they offer is of unequal depth. The BitsAI-CR authors state 75% precision for the generated review comments [5]; this is a measurement for a deployed en-terprise framework and should not be read as transferable to other designs. The Ericsson study reports that expe-rienced developers reacted positively to the generated re-views [1], which shows feasibility in an industrial setting but is a qualitative result for a tool that is not publicly accessible. İçöz and Biricik report improved accuracy and efficiency [2] and Abtahi and Azim found markedly fewer code issues in follow-up scans [3], but the accessible record used for this review does not expose the underlying figures needed to compare them with the precision value in [5]. For [4], no evaluation outcome is reported in the available source.

Costs and constraints also differ. The Ericsson tool was built for an internal environment [1]. The symbolic-reasoning approach is costly to compute and a poor fit for real-time workflows [2]. The static-analysis-plus-LLM repair approach needs complex integration and additional processing pipelines [3]. The work on program analysis for code understanding is not designed for pull request review [4]. BitsAI-CR targets enterprise scale and needs complex infrastructure [5]. These are limitations visible in the accessible descriptions, not confirmed weaknesses of every implementation.

V. COMPARATIVE SYNTHESIS

Table I lays out the five systems in the comparison. No numerical ranking is given, since the systems were never tested under one protocol and their outcomes cannot be placed on one scale. An entry marked "not reported" in Table I only means that the accessible text did not estab-lish it; the full work may well contain it.

Three patterns are apparent. First, pairing an LLM with an analysis or rule component is common, but the compo-nent plays different roles: context supplier, reasoning aid, change detector, or comment filter. Second, the reported limitations cluster around deployment cost and integration complexity rather than around whether the idea works, which leaves a lightweight pull-request-level design under-explored in the accessible record. Third, the way results are reported varies widely. One source gives a precision figure, one gives developer reactions, and the others give qualitative statements or none. This variation limits cross-system conclusions about accuracy, reliability, and cost.

VI. RESEARCH GAPS AND DESIGN IMPLICATIONS

G1—Comparable quantitative evaluation. Within the reviewed set, only one accessible source provides a figure that can be stated directly [5]. Future evaluations should report precision and recall of findings against a labelled reference, false-positive and false-negative rates, review la-tency, and the test conditions. The number, size, and diversity of pull requests and the repositories used should be stated explicitly.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510105

G2—Deployment cost and accessibility. Several systems are tied to an internal environment, enterprise infrastruc-ture, or computationally expensive pipelines [1], [2], [5]. A lightweight design built on open tools and hosted APIs is a design opportunity inferred from these descriptions, not a claim that no such system exists.
G3—Merging contextual and rule-based findings. When an LLM and a static analyser both report on a change, the findings must be combined into one report. The ac-cessible descriptions do not consistently specify how du-plicates are detected, how severity levels are mapped to a common scale, or how null or missing results are handled. These remain open design questions that the review sur-faces; settling them needs implementation and testing, and until then no claim of a unified report is made.

G4—Verification and reproducibility of LLM output. Generated feedback may be inaccurate or inconsistent [2]. Verification stages such as comparison tools or comment fil-ters [3], [5] address this in different ways. A design should treat model output as a recommendation, define how it is checked, and record the model, prompt, and input so that a result can be traced. Rule-based findings are repro-ducible for a given source and ruleset, which gives a stable baseline against which model output can be assessed.

G5—Failure handling and operational recovery. The ac-cessible sources provide limited evidence about behaviour when external services fail. A pull-request-level system depends on a source-hosting API and a model API, and should specify handling of invalid input, HTTP errors, timeouts, and rate limits, and what the user is shown when a stage fails. A system that runs in a demo yet has no planned way to fail safely is not ready for everyday use.

G6—Language and rule coverage. Static analysis is lim-ited to enabled rules and supported languages. Coverage across languages, and the effort of choosing or adding rules to suit team standards, are rarely reported in the accessi-ble record. Language coverage should be stated explicitly and not implied.

VII. RELATED AI-POWERED PULL REQUEST REVIEW SYSTEM: STATUS AND WORKFLOW

The related project proposes an automated first-pass re-view workflow. A user submits a GitHub pull request URL through a React frontend. A Spring Boot backend parses the URL into owner, repository, and pull request num-ber, and obtains the diff, the changed Java files, and the head commit SHA through the GitHub REST API. The diff is sent to Gemini AI, which returns a structured re-sult containing an overall score, a summary, category-wise reviews, findings, severities, explanations, and suggested fixes. In parallel, the changed Java files are downloaded at the head SHA, written to temporary files, and analysed by Semgrep; the temporary paths are mapped back to repos-itory paths and the files are deleted. Findings from both sources are intended to be merged into one list, serialised as JSON, and stored through a repository layer so that earlier reviews can be retrieved.

According to the project records, the layered design sep-arates the controller, GitHub client, AI service, Semgrep services, and persistence. Data transfer objects (AIRe-viewResult and ReviewFinding) give both analysers a com-mon output shape. The frontend provides a review dash-board, review history, and analytics views. Maven tests report four tests run with no failures or errors for the Sem-grep analysis and the pull request Semgrep integration. In end-to-end testing, however, the frontend displayed an er-ror message, and the stack trace implicates the stage that fetches the pull request diff; the exact network or API cause has not yet been found. The final merge of Sem-grep and AI findings, with deduplication and consistent severity mapping, is also still being validated.

Table II separates what is reported as implemented from what is unverified or planned. The distinction matters: passing component tests does not show that the whole workflow works, and counts shown on a dashboard are not evaluation results. The project is best seen as working soft-ware components awaiting end-to-end validation, not as a validated system.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510105

TABLE I

Comparison of the Reviewed Systems and the Evidence Available for Each

Ref.	Approach/components	Role of analysis and LLM	Evidence available in reviewed record
[1]	LLM with static program analysis; lightweight tool	Analysis extracts the method enclosing each changed line to give the model context	Reviews tried with experienced developers; positive reactions reported; built for an internal environment and not publicly accessible
[2]	LLM with symbolic reasoning; open-source models for software engineering	Symbolic stage added to counter shallow or hallucinated LLM output	Improved accuracy and efficiency reported; figures not available in accessible record; costly to compute and a poor fit for real-time workflows
[3]	Static analysis framework with GPT-3.5 Turbo and GPT-4o	Analysis detects issues; LLM revises code; comparison tool catches erroneous edits; retrieval-augmented generation and iterative prompts	Follow-up scans found markedly fewer code issues; figures not available in accessible record; complex integration and extra processing pipelines
[4]	LLM with program analysis for code understanding	Program-analysis structure used to strengthen the model's understanding of code	Evaluation outcome not reported in the available source; not designed for pull request review
[5]	BitsAI-CR: RuleChecker and ReviewFilter	Rule taxonomy detects issues; filter verifies comments to raise precision	Conference industry paper; 75% precision stated for generated comments; enterprise-scale, needs complex infrastructure

VIII. EVALUATION PLAN FOR THE PROPOSED DESIGN

Evaluation should be staged. Unit and integration tests come first, covering URL validation and parsing, head-SHA pinning, Java-file filtering, temporary-file cleanup, path mapping, null-list handling, deduplication and sever-ity mapping. Failure paths should then be exercised on purpose: invalid URLs, HTTP errors, timeouts, rate lim-its and malformed model responses should each produce a clear message without a crash. Finally, end-to-end tri-als should time the path from submission to a stored and displayed review.

Dataset and protocol should be written down before any testing. Pull requests should come from a stated set of repositories and differ in size and content, with human-labelled reference findings prepared for a sample. Useful measures are precision and recall against that reference, the false-positive rate, agreement between AI and Semgrep findings, the share of duplicates removed in merging, run-to-run variation of model output on identical input, and median and 95th-percentile latency. Results from repeated runs should carry sample counts and a measure of spread, not just the best case.

No target figure should be quoted as achieved before it has been measured. The test setup, limitations, failure cases and anonymised results should be made public where suitable, and findings from a small sample of pull requests should be described as that, not as general validation.

IX. LIMITATIONS

The review rests on a small, bounded set of sources, there is no record of how they were found, and full texts were not equally accessible. Five sources are cited only for background and were not analysed. No systematic cov-erage, formal risk-of-bias assessment or meta-analysis is claimed. The systems differ in scope, components, deploy-ment setting and evaluation method, so direct performance comparisons would not be valid. Some technical details in the comparison come from the project literature sur-vey and should be checked against the full papers before a final journal submission. The related project has not fin-ished end-to-end validation or performance evaluation, so it would be premature to claim better accuracy, reliability or developer productivity.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510105

X. CONCLUSION

Ten sources on combining LLMs with static or program analysis for code review were examined here, and five sys-tems were compared where the accessible detail allowed. The reviewed work shows that several design patterns are workable: analysis as a context supplier, symbolic reason-ing added to a model, analysis-driven detection followed by LLM repair, and rule-based detection followed by comment filtering. The evidence is too mixed to rank the systems numerically. The main opportunities that emerge concern comparable quantitative evaluation, lightweight and acces-sible deployment, explicit merging of contextual and rule-based findings, verification of model output, failure han-dling, and language coverage.

The related AI-powered pull request review system ad-dresses some of these opportunities at the software-design level through a hybrid of Gemini AI and Semgrep, a fixed

TABLE II
Reported Implementation Status of the Related Project

Component/capability	Reported status	Evidence boundary
URL parsing and GitHub data retrieval (diff, changed Java files, head SHA)	Implemented (project-reported)	End-to-end fetch failure under investigation
Gemini AI structured review (AIReviewResult)	Implemented (project-reported)	Output may be inaccurate; no accuracy measure
Semgrep analysis of changed Java files	Implemented; 4 Maven tests passed	Component tests only; enabled rules and Java only
Review persistence and history	Implemented (project-reported)	Stored output not independently evaluated
React frontend: dashboard, history, analytics	Implemented (project-reported)	Display of stored data; not an evaluation
Merge of Semgrep and AI findings with deduplication and severity mapping	In progress	Final behaviour unverified
End-to-end validation	In progress	No system-level claims yet
Webhooks, PR comments, other languages, custom rules, user accounts	Planned	Not implemented

structured output, and a persistent review history. Its end-to-end validation and final merged report remain future work. The sensible next step is a documented evaluation under a repeatable protocol; stronger claims should wait for measurements.

DECLARATIONS

Funding: None was reported in the source materials. Conflict of interest: To be confirmed by the authors and stated before submission.
Data availability: No new experimental dataset was pro-duced by this review.
Author contributions: Not specified in the source materi-als.
Ethics statement: Not applicable to this review.



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026
DOI: 10.17148/IJARCCE.2026.1510105

REFERENCES

[1] S. Ramesh et al., "Automated code review using large lan-guage models at Ericsson: An experience report," arXiv preprint arXiv:2507.19115, 2025.
[2] B. İçöz and G. Biricik, "Automated code review using large language models with symbolic reasoning," arXiv preprint arXiv:2507.18476, 2025.
[3] S. M. Abtahi and A. Azim, "Augmenting large language models with static code analysis for automated code quality improve-ments," in Proc. FORGE 2025 (co-located with ICSE 2025), Ottawa, ON, Canada, Apr. 2025; arXiv:2506.10330.
[4] X. Chen et al., "LLMs with program analysis for code under- standing," 2025.
[5] T. Sun et al., "BitsAI-CR: Automated code review via LLM in practice," in Proc. FSE 2025 (Industry Papers), Jun. 2025; arXiv:2501.15134.
[6] Z. Rasheed et al., "AI-powered code review with LLMs: Early results," arXiv preprint arXiv:2404.18496, 2024.
[7] I. Jaoua, O. Ben Sghaier, and H. Sahraoui, "Combining large language models with static analyzers for code review genera-tion," arXiv preprint arXiv:2502.06633, 2025.
[8] R. Tufano, A. Martin-Lopez, A. Tayeb, O. Dabić, S. Haiduc, and G. Bavota, "Deep learning-based code reviews: A paradigm shift or a double-edged sword?," in Proc. 47th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), Ottawa, ON, Canada, 2025; arXiv:2411.11401.
[9] U. Cihan, A. İçöz, V. Haratian, and E. Tüzün, "Evaluat-ing large language models for code review," arXiv preprint arXiv:2505.20206, 2025.
[10] S. P. Kumar, S. Bararia, and K. Raj, "Bigger isn't always better: A comparative evaluation of LLMs for automated code review," arXiv preprint arXiv:2606.15689, 2026.

 This work is licensed under a Creative Commons Attribution 4.0 International License