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 11, November 2026

Hint-Ladder Security Feedback for Novice Python Programmers: A Grounded and Privacy-Preserving AI Code Review Assistant

Mali Dnyaneshwari Rajendra[1], Prof. Manoj V. Nikum[2], Prof. Shivam B. Limbhare[3]

MCA, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[1]

HOD MCA & Assistant Professor, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[2]

Assistant Professor, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[3]

Abstract: Automated code review that combines static analysis with large language models (LLMs) is already an active research area, but existing systems target professional pull-request workflows or give novices complete answers that reduce learning. This paper presents a grounded, hint-ladder approach to security-first code review for novice Python programmers. An AST-based static engine first produces confirmed findings, each mapped to an OWASP Top Ten category. These findings gate an LLM, which is allowed to produce only three progressively detailed feedback levels: a nudge on where to look, an explanation of why the issue matters, and a corrected fragment that is released only on request. An output validator checks the response schema, line numbers and leakage of corrected code in the lower levels, and the whole pipeline can run on a locally hosted open model so that student code never leaves the institution. The paper also defines an evaluation protocol based on detection precision and recall, grounding rate, hint-leakage rate and instructor-rated usefulness. The design aims to keep the reliability of rule-based analysis, the readability of LLM explanations and the pedagogical value of guided discovery.

Keywords: Automated Code Review, Static Analysis, Large Language Models, Programming Education, Secure Coding, Hint Ladder, Grounded Generation

I. INTRODUCTION

Code review is widely regarded as one of the most effective ways to find defects and share knowledge in software teams [1]. For students and junior developers, however, timely review is rarely available, and the warnings produced by static analysis tools are often difficult to understand and act upon [2]. Large language models can explain code and propose repairs in natural language [3], which has led to a growing body of work on LLM-assisted review.

Before choosing the direction of this work, the literature was searched for existing publications on AI-based code review assistants. The search showed that the basic idea of combining static analyzers with LLMs to generate review comments has already been published [4], including a production system that grounds every LLM comment in static-analysis evidence [5], and that GPT-based code review for programming education has also been reported [6]. A plain "static analysis plus LLM" assistant would therefore not be a new contribution. This paper changes the paradigm from generating complete review comments to controlling what the LLM is allowed to reveal, and focuses on security issues in novice Python code.

The contributions of this paper are: (i) a grounding-gated hint ladder in which only issues confirmed by static analysis can trigger LLM feedback, and feedback is released in three levels; (ii) an OWASP-mapped, security-first prioritisation of findings suited to novice code; (iii) a deployment option based on a locally hosted open model for privacy; and (iv) an evaluation protocol that measures grounding and hint leakage in addition to detection accuracy.

II. LITERATURE REVIEW

Modern code review has been studied extensively. Bacchelli and Bird found that review brings knowledge transfer and team awareness in addition to defect finding [1]. Static analysis is fast and repeatable, but Johnson et al. reported that developers avoid such tools because of false positives and unclear messages [2]. The Python ast module allows custom rules to be written over the syntax tree of a program [7].

Jaoua et al. combined knowledge-based static analyzers with learning-based review models at three stages of the LLM pipeline (training data, retrieval at inference, and concatenation of outputs) and reported better relevance and

 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 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511136

completeness of generated review comments than either approach alone [4]. Mandal and Jiang described a production pull-request bot for C/C++ in which each LLM explanation is anchored to a static-analysis finding and served from a quantized open-weight model on a single GPU [5]. Rasheed et al. reported early results of LLM-based code review in a software company [8]. Zhang et al. studied hallucinations in LLM-generated code, which motivates grounding model output in verifiable evidence [9].

In education, Lee presented a GPT-4 code review system for primary and secondary school students that gives review comments without directly providing solutions, to limit AI-assisted cheating [6]. Nguyen and Allan used GPT-4 to give tiered formative feedback on student code [10], and a recent survey of LLM applications in programming education stresses the balance between automation and human oversight [11].

The existing work therefore leaves a gap. Hybrid review systems [4], [5] are aimed at professional reviewers, not at learners who need guidance without answers. Educational systems [6], [10] focus on correctness and general feedback and do not use confirmed static findings to decide which feedback is allowed, nor do they prioritise security weaknesses such as those in the OWASP Top Ten [12]. The present work targets this gap.

III. METHODOLOGY

A. *System Overview*

The proposed assistant is a web application for Python code. A student submits a file or snippet, and the backend returns a ranked list of issues, each with a three-level hint. Fig. 1 shows the pipeline: AST parsing and static rules, confirmed findings, a grounded prompt, the LLM, an output validator and the hint ladder.

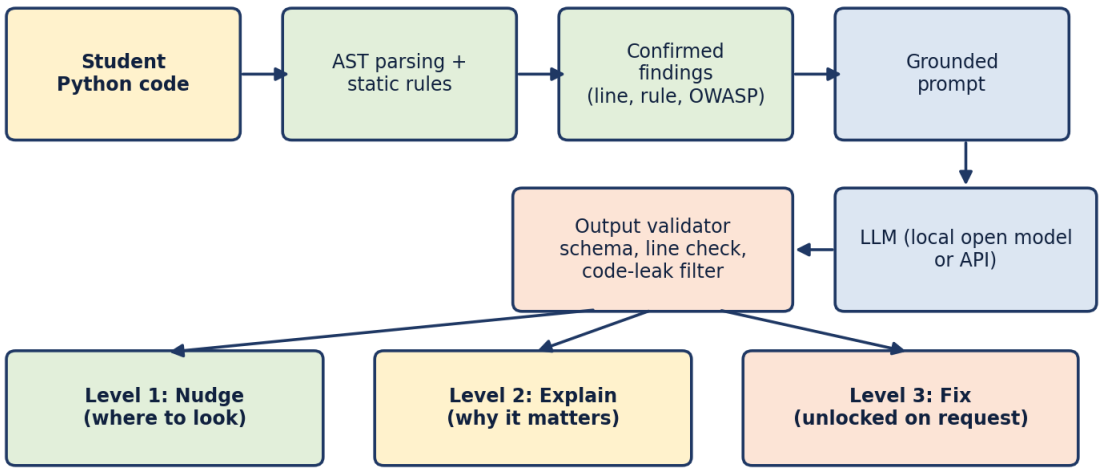


Fig. 1 Proposed grounded hint-ladder pipeline

B. *Static Analysis and Grounding*

The submitted code is parsed into an abstract syntax tree and checked by a visitor-based rule engine. The rules cover hard-coded credentials, use of eval() and exec(), SQL queries built by string concatenation, unsafe deserialisation, weak hash functions, bare except clauses, and files opened without a context manager. Each finding records the rule identifier, line number, severity and an OWASP Top Ten category [12]. Findings are sorted by security severity first and style last, so that a novice sees the most important problem first.

Grounding means that the LLM is invoked only for confirmed findings. The prompt contains a role instruction, the rule rationale, the finding metadata, a snippet of about the enclosing function, and an output contract in JSON. The model is instructed not to speculate beyond the snippet. This follows the “grounding first, then generate” principle reported for enterprise review [5], applied here to a learning setting.

C. *Hint Ladder*

For every finding the model must return three fields. Level 1 (nudge) tells the student where to look without naming the fix, for example that the way a query is built on a given line deserves attention. Level 2 (explain) states why the pattern is risky and which secure practice applies. Level 3 (fix) contains a corrected fragment and is shown only when the student

 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 11, November 2026

DOI: 10.17148/IJARCCE.2026.1511136

asks for it. This design keeps the student in control of how much help is revealed and is intended to preserve learning, in line with concerns about direct answers in educational use [6], [11].

D. *Output Validation*

Because LLM output can be inaccurate [9], every response is validated before display. The validator checks that the JSON schema is complete, that each reported line number exists in the file and matches a confirmed finding, and that Level 1 and Level 2 text does not contain a corrected code fragment (leak filter). A response that fails is retried once and then replaced by the static rule message, so the student always receives a safe result.

E. *Privacy-Preserving Deployment*

The LLM interface is model-agnostic. The assistant can call a hosted API or a locally served open-weight code model, which keeps student submissions inside the institution, a requirement also stressed in on-premise review systems [5]. Only the code of the current submission is sent to the model, and no personal data is stored with it.

F. *Evaluation Protocol*

A benchmark of Python programs written by students and seeded with known issues is used. Four measures are defined. Detection quality is measured by precision, recall and F1-score against the seeded issues. Grounding rate is the share of LLM comments that refer to a confirmed finding. Leakage rate is the share of Level 1 and Level 2 hints that contain corrected code. Usefulness is the mean rating (1–5) given by instructors for clarity and helpfulness. Three configurations are compared: static analysis only, an ungrounded LLM that receives only the code, and the proposed grounded hint-ladder system.

IV. RESULT

This section reports the results of the evaluation protocol on the benchmark. Table I gives detection quality and Table II gives the hint-quality measures. Values highlighted in the tables are to be filled with the final measurements from the author's own experiment before submission.

TABLE I DETECTION QUALITY ON THE SEEDED BENCHMARK

Configuration	Precision	Recall	F1-score
Static analysis only	[ ]	[ ]	[ ]
Ungrounded LLM	[ ]	[ ]	[ ]
Proposed grounded system	[ ]	[ ]	[ ]

TABLE II HINT QUALITY MEASURES

Configuration	Grounding rate	Leakage rate	Usefulness (1–5)
Ungrounded LLM	[ ]	[ ]	[ ]
Proposed grounded system	[ ]	[ ]	[ ]

The expected pattern, based on the design, is that static analysis alone has high precision but lower recall, the ungrounded LLM has wider coverage but a lower grounding rate and more leakage, and the grounded hint-ladder system combines the strengths of both. This expectation should be confirmed or revised according to the measured values.

V. CONCLUSION

A review of the literature showed that combining static analysis with LLMs for code review has already been published, so this work moved the focus to controlled, security-first feedback for novices. The proposed hint-ladder design lets confirmed static findings gate the LLM, reveals help in three levels, validates every response and can run on a local model for privacy. Future work includes support for more languages, IDE integration, learning from accepted and dismissed hints, and a classroom study to measure the effect on learning outcomes.

ACKNOWLEDGMENT

The authors thank the Master of Computer Application department of Shri Jaykumar Rawal Institute of Technology, Dondaicha, for the facilities and support provided during this work.



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026
DOI: 10.17148/IJARCCE.2026.1511136

REFERENCES

[1]. A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, 2013, pp. 712–721.
[2]. B. Johnson, Y. Song, E. Murphy-Hill and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, 2013, pp. 672–681.
[3]. M. Chen et al., "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.
[4]. I. Jaoua, O. Ben Sghaier and H. Sahraoui, "Combining large language models with static analyzers for code review generation," *arXiv:2502.06633*, 2025.
[5]. S. Mandal and H. Jiang, "Grounded AI for code review: resource-efficient large-model serving in enterprise pipelines," *arXiv:2510.10290*, 2025.
[6]. D.-K. Lee, "A GPT-based code review system for programming language learning," *arXiv:2407.04722*, 2024.
[7]. Python Software Foundation, "ast — Abstract Syntax Trees," Python 3 Documentation. [Online]. Available: https://docs.python.org/3/library/ast.html
[8]. Z. Rasheed et al., "AI-powered code review with LLMs: early results," *arXiv:2404.18496*, 2024.
[9]. Z. Zhang et al., "LLM hallucinations in practical code generation: phenomena, mechanism, and mitigation," *Proc. ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 481–503, 2025.
[10]. H. Nguyen and V. Allan, "Using GPT-4 to provide tiered formative code feedback," in *Proc. 55th ACM Technical Symposium on Computer Science Education (SIGCSE TS)*, 2024.
[11]. G. Pitts, A. P. Hridi, A. Lekshmi-Narayanan et al., "A survey of LLM-based applications in programming education: balancing automation and human oversight," *arXiv:2510.03719*, 2025.
[12]. OWASP Foundation, "OWASP Top Ten." [Online]. Available: https://owasp.org/www-project-top-ten/

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