BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

AI-Assisted Software Repository Understanding and Intelligent Code Analysis applying Machine Learning and Large Language Models

Kush Vishal Agarwal[1], Prof. Neelam Jain[2]

[1]Department of Artificial Intelligence and Data Science,
Ajeenkya DY Patil School of Engineering, Pune, India
[2]Department of Artificial Intelligence and Data Science,
Ajeenkya DY Patil School of Engineering, Pune, India (Guide)

Abstract

Growing codebases force developers to spend more and more effort working out how the store is organised, to a significant degree, and also, where its weak points are. Current tooling tackles this one slice at a time, like rule-based analysers count metrics and flag violations, learned models rank modules by likelihood of defects, in real-world scenarios; also, large Language Models (LLMs) are outlining what code does in plain language. In this broader context, few of all these slices are connected, and most operate on one file at one more time; this review surveys ten recent papers on LLMs in software engineering, neural defect prediction, LLM-based judgement of maintainability, retrieval over code for repository-scale context, and agent-based defect prediction; checking them uncovers that no open, reproducible pipeline has been gauged that links repository reading, metrics, learned risk ranking, LLM explanation, and recommendations in one single output. A conceptual architecture is, in fact, therefore sketched in which a dependency graph and code metrics are driving a risk model; also, only the riskiest modules are then explained by an LLM applying setting drawn from that graph; this architecture is a proposal on paper. Nothing has been realized. Still tested yet, and a prototype and evaluation sketch are given for follow-up work.

Keywords : Software repository mining, large language models, defect prediction, code quality assessment, retrieval-augmented generation, software maintenance, broadly speaking.

I. Introduction

Large software projects often include thousands of files that many contributors have written over several years, on the whole; in fact, going through the such a project from start to finish is impractical, which is why developers hinge on tooling, in actual application. SonarQube, Snyk and GitHub Copilot are standard examples, and still, what each covers is a narrow job. Within this framework, none hands a newcomer a single view of what the framework does, how its elements relate; in fact, also which domains are most likely to cause trouble.

Ways are being promised by Machine Learning (ML) and Large Language Models (LLMs) to close this gap, in many respects. In light of this dynamic, classifiers fed with code metrics can, in this setting,

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

single out modules that tend to comprise bugs [7], [10], and LLMs can outline code and judge how easy it is frequently to preserve [8], [9], broadly speaking. Nonetheless, research on each individual strand has mostly continued independently, and the outputs seldom arrange reliances into the account.

Automated software analysis can be viewed as three stacked layers, including rule and metric based inspection, learned prediction of defect-proneness, and LLM-driven comprehension of code, in real-world scenarios. Viewed through this lens, the concrete question tackled here is how the findings of all three layers could (in practical terms) be merged into one assessment spanning the entire repository, broadly speaking. Any single layer is occupied by most existing frameworks, and often the single file.

The paper followingly contributes, and groups ten representative studies by the analysis task they target, it spots the combination of capabilities that all these studies are leaving uncovered; it is providing a further conceptual architecture together with one testing outline for future empirical work, broadly speaking.

The rest of the paper is following this order, like the obstacle and motivation, review objectives, pertinent trends, the literature survey, the research gap, the proposed architecture, and the conclusion, broadly speaking.

II. Problem Statement and Motivation

Today's practice suffers from five shortcomings. First findings are, to be sure, being yielded on its own by each individual tool. Nothing is combining them. Second, many AI coding assistants reason over any file or snippet and have no picture of the wider architecture, to a significant degree; fifth, on account of all of the above, several disconnected tools must be juggled by maintainers, in real-world scenarios. Fourth, static analysers can, in practice, emit very long warning lists with no sense of which modules deserve attention first; third, LLMs are good at outlining semantics but are infrequently supplied with metrics or reliance information.

There are signs that progress is regularly possible. From this perspective, neural models can (in actual operation) learn advantageous code representations for defect prediction [7]; LLMs are revealing some intuition about code quality without being trained for it [8]; in actual application; also retrieval over code graphs can, broadly speaking, give an LLM repository-wide context [4]. The guiding question of this review is therefore, like how can (to some degree) store arrangement, software metrics, learned risk prediction and LLM comprehension be gathered into one pipeline whose output is the ranked, doable list of maintenance recommendations?

III. Targets of the Review

This review aims to:

1) Study existing ML and LLM schemes for repo assessment, code understanding and maintenance support.
2) Explain how repo architecture, spanning files, modules, classes and their links, can be showed.
3) Analyze how metrics and ML models spot high-risk, defect-prone modules.

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

4) Highlighting the value of LLMs for understanding, assessing quality, extracting summaries and providing recommendations.

5) Highlighting limitations and issues of existing tools for analysing repositories.

6) Sketch a conceptual architecture combining ML and LLMs for repository-level understanding.

IV. Relevance to Current Technology Trends

A. AI for Software Engineering and Repository Mining

Mining Software Repositories (MSR) is used to extract data from version control systems. AI techniques are now common place in this space, and there are large studies listing many different LLMs that act upon code for a wide variety of problems [9].

B. Language Models for Source Code

General-purpose and code-specific models, incorporating GPT-4, CodeBERT, CodeT5, StarCoder and LLaMA, are applied to write documentation, summarise functions and comment on quality or maintainability, broadly speaking.

C. Learned Defect Prediction

Convolutional, recurrent and graph neural networks are typically being evaluated to estimate which modules are likely to bug before a release [7], [10].

D. Repository-Scale and Retrieval-Based Intelligence

Retrieval-augmented generation (RAG) and agent-style frameworks are trying to give models access to the surrounding codebase, through code graphs and semantic search [4], to a significant degree. Some commercial products already are building a graph of functions, classes and imports for this purpose [11], to a significant degree.

E. Automated Review and Suggestion Systems

LLM-based reviewers have been put into production use and produce refactoring and quality suggestions backed by evidence from the code [5].

V. Literature Survey

A. How the Field Devised

The field has moved through the recognisable phases, such as rule-based static analysis, then ML defect prediction, then neural code models, then Transformer-based comprehension, and now LLM-driven investigation of whole projects, in many respects; early prediction work paired hand-designed metrics with classic classifiers such as Random Forests and SVMs, in many respects. Taken together, neural networks later removed the need for manual feature design by learning representations directly from code [7], [10]; in fact, Pre-trained Transformers such as CodeBERT and CodeT5 added the ability to summarise and document code. Current LLMs can (to a meaningful extent) comment on quality and maintainability; in turn, still, much of that work still treats one file in isolation [8], [9].

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

B. Summary of Existing Methods

The studies reviewed fall into the four groups: (i) LLM-based comprehension and quality judgement, (ii) ML and neural defect prediction, (iii) retrieval-based repository-scale intelligence, and (iv) automated review and recommendation, broadly speaking. Under these conditions, tables I and II list the ten studies, and Fig. 1 is revealing whether each works at file or repo scope, in many respects.

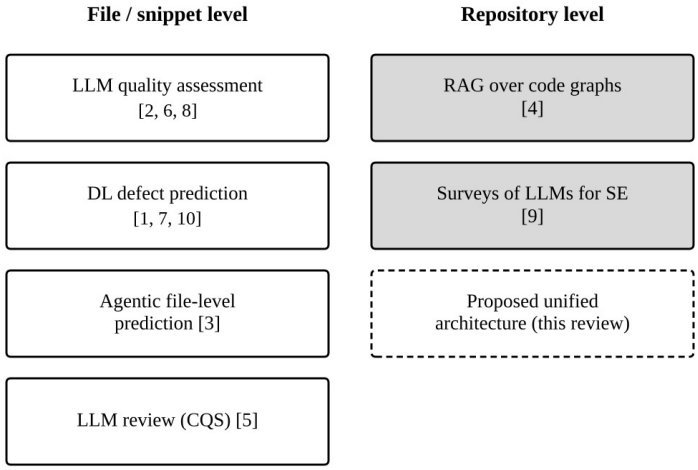


Fig. 1. Building on this observation, the proposed architecture, shown dashed, works at repository level and is conceptual; indeed, scope of each reviewed strategy.

A few patterns stand out. LLMs can judge maintainability without task-specific training, yet only per file or class [8], and their verdicts shift, across the model versions [6], broadly speaking; indeed, precision is sharpened by neural defect predictors, but obstacles with data quality, class imbalance and transfer between projects maintain, and many papers do not release code or data [7], [10], on the whole. Along these dimensions, retrieval work is revealing that repository context is broadly attainable but does not tie it to metrics or ML predictors [4]; production reviewers and agent-based predictors either stay at file scope or concentrate on quality, and none ranks risk, across this repository [3], [5].

More recent results point towards the combining signals, with catches. A 2026 examination into the defects that escape to production in Python projects found that models trained on metrics beat LLM-based predictors, and reported that code embeddings carry different information from metrics, although naive combination of the two degraded cross-project performance [1]. Against this backdrop, a further further 2026 paper reported that LLMs can rate code quality when revealed metrics alone. Also, that supplying code plus metrics is one workable hybrid [2]. Such findings motivate, but do not settle, the hybrid design explored in Section VII.

C. Comparative Analysis

Table III is setting the principal families side by side on five dimensions, with the proposed architecture in the final row, to a significant degree. Against this backdrop, rule and metric tools are quick and repeatable but explain nothing; LLM tools explain well but typically see one file and can, in practice, be

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

inconsistent; ML predictors rank risk but seldom come with an explanation, to a significant degree; indeed, a trade-off is revealed by the comparison.

D. Evaluation Measures in published research

Studies in this subfield are generally assessed on three benchmarks; and thus, the first is prediction quality, meaning how well risky modules are separated from safe ones, measured with precision, recall, F1 or domain under the ROC curve; effort-aware studies also report values such as Recall@Top20%LOC, on the whole. Building on this observation, the third is cost, which is covering delay, token consumption and indexing effort; the second is explanation quality, meaning whether yielded descriptions and advice are correct and advantageous, generally rated by people or in comparison to a reference tool, to a significant degree. An integrated strategy should be measured on all three. Also, in addition, on ranking usefulness, i.e. whether genuinely faulty modules are appearing near the top of the list.

TABLE I

LITERATURE SURVEY SUMMARY: PART 1

Ref.	Author / Year	Approach	Main Result	Weakness
[1]	De Rosa and Liguori, 2026	Supervised ML/DL on 83 metrics plus code embeddings to forecast residual (post-release) faults in Python, tested, across the projects	Metric-based models (RandomForest, XGBoost, CatBoost) beat LLM and unsupervised predictors; embeddings add different information, and still, despite this, cross-project findings were hurt by naive combination	Python only; no explanation layer or repository-level ranking
[2]	Cernău et al., 2026	Tests GPT-4o and Gemini 2.0 Flash on file-level quality assessment given code alone, code plus metrics, or metrics alone	LLMs can often spot hurdles from metrics alone; supplying code plus metrics is a workable hybrid	Single-file assessment; no risk ranking or repository-level grounding
[3]	Hesamolhokama et al., 2025	LLM-based multi-agent debate over code changes to anticipate file-level defects	Exposes inflated exactness of snapshot-based check; better sensitivity to newly introduced defects	File scope only; no repository-level ranking or combined report

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

Ref.	Author / Year	Approach	Main Result	Weakness
[4]	Tao et al., 2025	Survey of retrieval for code generation, including code graphs and semantic search	Retrieval gives repository-level understanding and better accuracy through grounding	Retrieval quality uneven; not joined to metrics or ML
[5]	Wong et al., 2025	Pair of fine-tuned LLaMA-3 models, one for spotting issues and one for writing feedback	About 60% week-over-week user helpfulness rate; hand-written rules filter invented comments; applied in industry	Concentrated on quality; no cross-repository risk ranking or reliance grounding

TABLE II

LITERATURE SURVEY SUMMARY: PART 2

Ref.	Author / Year	Approach	Main Result	Weakness
[6]	Simões and Venson, 2024	Compares GPT-3.5, GPT-4o and SonarQube on 1,641 Java classes	GPT-3.5 tracked SonarQube closely; GPT-4o was weaker on arranged metrics	Outputs differ by model version; connection and risk data unused
[7]	Chen et al., 2024	Task-oriented survey of deep learning across the twelve software engineering subareas, including defect prediction	Neural models are raising findings across the subareas, including fault-related tasks	Concerns over data quality; weak transfer between projects
[8]	Dillmann et al., 2024	Uses LLM cross-entropy as a "surprise" signal of maintainability over 10 models	Higher cross-entropy is suggesting lower maintainability once code size is widely controlled; no task-specific training	Class-level only; code size confounds the signal; nothing at repository level

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

Ref.	Author / Year	Approach	Main Result	Weakness
[9]	Zhang et al., 2023	Survey of 62 LLMs for code across 16 downstream tasks	Strong findings for generation, summarising and bug repair; increasing employ in practice	Mostly file-scoped; whole-repository investigation left out
[10]	Giray et al., 2022	Systematic review of deep learning for software defect prediction	Recommends handling class imbalance and publishing replication packages	Replication packages often unavailable; restricted cross-project proof

TABLE III

COMPARISON OF REPOSITORY ANALYSIS APPROACHES

Approach	Scope	Input signal	Strength	Weakness
Static analysis / metrics tools	File or module	Rules and metrics	Repeatable; thoroughly applied	Explains; long, unranked warning lists
ML / DL defect prediction [1], [7], [10]	Module or file	Metrics and learned code features	Ranks modules by risk before release	Poor transfer between projects; no explanation
LLM quality assessment [2], [6], [8]	File or class	Model probabilities or judgement	Describes code; no task training needed	File-level only; varies by version
RAG over code graphs [4]	Repository	Retrieved code and graph context	Brings in repository context	Not linked to metrics or risk models
LLM code review [5], [11]	Change or repo	LLM reading with codebase setting	Concrete comments inside developer workflow	Quality-centred; no open, risk-ranked, combined report
Proposed architecture (conceptual)	Repository	Graph, metrics, ML risk and LLM output	One ranked, explained result	Conceptual; untested

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

VI. Research Gap

There are five key problems identified from this survey:

1) Metrics do not give insight into what the code does. Tools such as SonarQube, CodeClimate and PMD provide numerical values and issues but do not provide insight into what the code actually does, or how it is structured, or how the maintenance will change based on the issue.

2) LLM analysis is not grounded in the repository. Most LLM studies focus on individual files [8] [9]. This means the results will vary depending on which version of the model is used [6].

3) Defect prediction is isolated. Predictors identify risky modules however few modules are passed onto graph-based explanations/recommendations [3] [10].

4) No measured end to end pipeline. There are existing partial pipelines such as store aware LLM analysis [11] and using metrics alongside LLMs [2]. However, no open pipeline has been found that fundamentally measures how well the pipeline performs across parsing, metrics, risk ranking, LLM analysis and recommendations.

5) Advice lacks context**.** From this perspective, existing systems flag problems but limitedly guide on maintainability and developer productivity [5].

Table IV highlights the studies that cover each pipeline module .

TABLE IV

COVERAGE OF PIPELINE MODULES BY REVIEWED STUDIES

Module	**Studies**
Store scanning / dependency graph	[4]
Metrics extraction	[1], [7], [10]
ML risk prediction	[1], [3], [7], [10]
LLM semantic understanding	[2], [8], [9]
Code quality assessment	[2], [5], [6], [8]
Recommendation engine	[5]
Joined pipeline and report	None (gap)

Primary gap, namely a transparent, repeatable repository-level method that is fusing architecture, metrics, ML risk and LLM explanation and is tested against single-signal baselines, in actual application.

VII. Proposed Conceptual Architecture

To close this gap we suggest the seven-stage pipeline shown in Fig. 2, and the store is read by the first stage and builds a dependency graph over files and modules, especially for classes, in actual application. The second is computing metrics for each module, such as size, intricacy, coupling, and change history, to a significant degree, and the third allocates each module a defect-risk score; in turn, the fifth is merging everything into the prioritised recommendations, and the last assembles the final report. Within this

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

framework, the fourth is passing the highest-risk modules to an LLM, together with context taken from the graph.

Let M represent the set of modules. Each module m has a feature vector $\mathbf{f}_m$ made of size and intricacy metrics, coupling values read from the graph, and history features, in many respects. In parallel with this, the risk model outputs

$$R_m = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{f}_m + b) \qquad (1)$$

with factors fitted on labelled modules applying binary cross-entropy; a tree-based classifier could be replaced for the linear form. Modules are generally sorted by score and only the top k go on to the LLM stage, in many respects. In parallel with this, for a chosen module, the LLM is given its source plus a context set $N(m)$ spanning its direct callers, callees and imports. To favour modules whose failure would spread generally, a priority score weights risk by importance in the graph:

$$P_m = R_m \cdot (1 + \lambda c_m) \qquad (2)$$

where cm is the normalised centrality of m and $\lambda \geq 0$ is a weight to tune. Recommendations are then produced in descending order based on Pm.

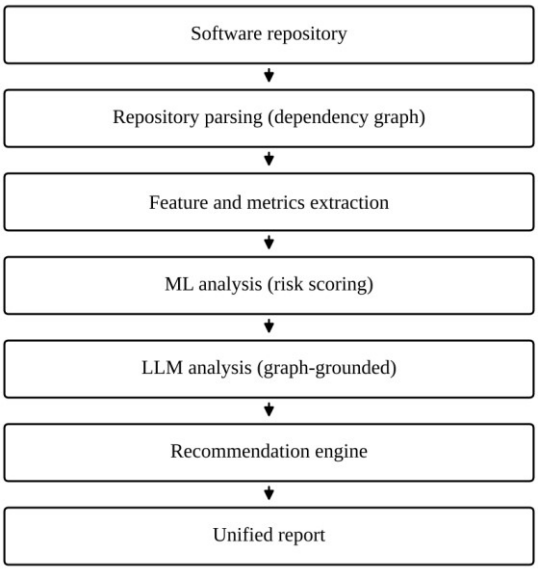


Figure 2: Seven step conceptual framework for risk based upon a journey from retail to consolidated reporting.

A. Signal Groups

The design uses 4 different signals. Structural signals are derived from the dependency graph to show how far the change may have an impact. Metric signals such as size, cyclomatic complexity and coupling provide insight into how difficult to maintain the module. Historical signals such as churn and previous defects suggest how often the module has failed previously. Semantic signals are derived from the LLM and provide insight into the intent of the module and confusing parts. None of these signals individually would be sufficient however it is assumed that combining them provides additional information. This is

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

consistent with research showing that metrics and learnt representations contain varied information [1]. However this research also shows that Naive Cross-project performance can be negatively affected by combining signals, this is what the following check is testing.

B. Managing Cost and Reliability of the LLM Stage

Many practical scenarios can benefit from the fact that only the top-k modules are sent to the LLM, reducing cost, and the settings are constrained to graph neighbours as opposed to the complete store.. Building on this note, to diminish change between model versions [6] and unsupported claims, outputs should, in essence, cite precise files and lines, be cross-checked with metric values, and pass through the elementary rule-based filters, a practice already used in an industrial reviewer [5], in many respects.

C. Summarize of an Evaluation

This can be further extended into a four part study. Part 2 is training the risk model on some of the data and comparing it to the baseline of metrics only models on projects outside of training. Part 3 compares the performance of the metrics, ML and LLM models on the same projects. The metrics and LLM are compared based on their performance in predicting risk, how useful they are ranked and cost. Part 1 identifies public stores with known defect status and collects the metrics and history. Part 4 looks at how well it transfers to other languages/fields.. Within this framework, also removing one signal group at a time would disclose which contributes most; and thus, also, a small user study could rate the generated recommendations.

D. Possible Uses

If laid out to work, the pipeline could, by definition, help new developers find their bearings in an unfamiliar codebase, support maintenance Technical debt is estimated by planning, prioritise code review, and create documentation. Along all these dimensions, its chief value would lie in directing scarce reviewer time to the modules that matter most.

E. Ethical and Practical Considerations

Source code can, strictly speaking, broadly be proprietary or hold credentials, so sending it to an external LLM service raises confidentiality questions; local models or redaction are, in practice, routinely options. The risk scores should be considered as guidance to help make decisions and not definitive. Weights and characteristics should be noted. Concrete hurdles are encompassing getting labelled defect data, the cost of LLM calls on large stores, and tuning the ranking weights. Overall this is a high level architecture to provide guidance for experimentation.

VIII. Conclusion

That review analysed ten studies covering LLM-based code comprehension, quality assessment, neural defect prediction, retrieval-based repo intelligence and agent-based prediction, in many respects; in fact, the typical shortcoming is that approaches stay at file level, tackle one task, or stop before combining metrics, ML risk and LLM explanation into one repository-level report, broadly speaking. Still, an gauged, reproducible end-to-end pipeline is still missing; in fact, we sketched one possible answer, including a dependency graph, metrics, an ML risk model, graph-grounded LLM analysis and a recommendation engine working together. In this broader context, whether it conducts well is not yet

BODHIVRUKSHA JOURNAL OF DIVERSE DISCIPLINE
E-ISSN : 3139-1486
An Open Access, Peer-Reviewed, Multidisciplinary Scholarly Journal
Bi-monthly | Multilingual (English, Hindi, Marathi) | Academic Research
@ Volume 2, Issue 5 (September-October 2026) (Special Issue 1) ISSN: 3139-1486 (Online)

known, and future work is encompassing a prototype on open-source repositories, comparison against metrics-only and LLM-only baselines, studying which signals carry over between projects and languages, lowering LLM cost, and fitting the tool into the developer workflows.

References

[1] G. De Rosa and P. Liguori, "Will it break in production? Metric-driven prediction of residual defects in Python systems," arXiv preprint arXiv:2604.26667, 2026.

[2] L. Cernău, A.-P. Dobrescu, E. Cărbune, and G. Asandei, "Comparative analysis of LLMs for software quality assessment via code and metrics," in *Proc. 18th Int. Conf. Agents and Artificial Intelligence (ICAART)*, 2026.

[3] M. Hesamolhokama, B. Rohani, A. Shafiee, M. Fazli, and J. Habibi, "From illusion to insight: Change-aware file-level software defect prediction using agentic AI," arXiv preprint arXiv:2512.23875, 2025.

[4] Y. Tao, Y. Qin, and Y. Liu, "Retrieval-augmented code generation: A survey with focus on repository-level approaches," arXiv preprint arXiv:2510.04905, 2025.

[5] S. Wong *et al.*, "A note on Code Quality Score: LLMs for maintainable large codebases," arXiv preprint arXiv:2508.02732, 2025.

[6] I. R. da S. Simões and E. Venson, "Evaluating source code quality with large language models: A comparative study," in *Proc. XXIII Brazilian Symp. Software Quality (SBQS)*, Salvador, Brazil, Nov. 2024, arXiv:2408.07082.

[7] X. Chen *et al.*, "Deep learning-based software engineering: Progress, challenges, and opportunities," arXiv preprint arXiv:2410.13110, 2024; also in *Sci. China Inf. Building on this observation, sci.*, vol. 68, art. 111102, 2025.

[8] M. Dillmann, J. Siebert, and A. Trendowicz, "Evaluation of large language models for assessing code maintainability," arXiv preprint arXiv:2401.12714, 2024.

[9] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, "A survey on large language models for software engineering," arXiv preprint arXiv:2312.15223, 2023.

[10] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," arXiv preprint arXiv:2210.02236, 2022, in many respects.

[11] Greptile, "Key features: Full codebase context," product documentation. [Online]. Available: https://greptile.com/docs/code-review/key-features