



EXPLAINABLE AI FOR DETECTING AND EXPLAINING SOFTWARE BUGS IN JAVA APPLICATIONS

Ms. Deshmukh V.H¹, Komal bhosale²

¹ Faculty Guide & Supervisor | ² Dattakala Group Of Institutions Faculty Of Engineering Swami - Chincholi tal. Daund dist. Pune, India

Corresponding author: bhosalekomal796@gmail.com

ARTICLE ID IJREK-2026-00153	ARTICLE TYPE Research Article	RESEARCH AREA Artificial Intelligence	PUBLICATION Volume 1, Issue Issue 1 (2026)
RECEIVED 23 October 2026	REVISED 27 October 2026	ACCEPTED 26 October 2026	PUBLISHED 28 October 2026

Abstract

Software bugs can cause incorrect results, crashes, security weaknesses, and expensive maintenance work. Traditional testing and static-analysis tools can identify many defects, but their warnings may be difficult to prioritise or understand, especially in large Java applications. Artificial intelligence (AI) can help rank suspicious code and recognise patterns associated with defects; however, a prediction is less useful when developers cannot understand why it was produced. This paper proposes an explainable AI (XAI) workflow for detecting and explaining potential bugs in Java applications. The proposed approach combines source-code features, static-analysis findings, test and version-history signals, and a prediction component with explanations tied to observable evidence such as a risky code path, a missing null check, a resource-handling pattern, or a related test failure. The design aims to present a confidence score, relevant code location, supporting signals, and suggested next steps while keeping the developer responsible for confirming the defect. The paper discusses the problem, related approaches, architecture, methodology, evaluation plan, applications, limitations, and future work. Since this is a proposed design, performance improvements must be established through experiments rather than assumed.

Keywords: Explainable AI; XAI; Java; software bug detection; machine learning; static analysis; defect prediction; software quality; code explanation

2. PROBLEM STATEMENT AND OBJECTIVES

A bug-detection system that only reports a risk score leaves developers uncertain about which code pattern triggered the warning, how serious the issue may be, and what should be inspected next. Conversely, an explanation that sounds plausible but is not grounded in code or test evidence can mislead users. The central problem is therefore to combine useful defect prediction with explanations that are understandable, traceable, and appropriately cautious. The main objectives are: • To identify potential defects in Java source code using static features and available project evidence. • To rank warnings so that developers can inspect higher-risk findings first. • To provide explanations linked to code locations, analysis rules, control-flow clues, or test outcomes. • To make uncertainty and limitations visible rather than presenting predictions as confirmed bugs. • To define an evaluation plan measuring detection quality, explanation usefulness, and developer effort.

3. LITERATURE REVIEW

Static analysis examines source code without executing it and can detect issues such as suspicious null usage, unreachable code, unsafe API calls, and resource-management problems. Rule-based tools are often interpretable because a warning can be linked to a named rule and a specific code location. Their limitations include configuration effort, false positives, and incomplete understanding of program context. Machine-learning-based defect prediction uses code metrics, change history, issue reports, or learned representations of source code to estimate whether a file, method, or statement is likely to contain a defect. These approaches can capture patterns that are difficult to express as individual rules, but results depend on the quality and representativeness of training data. A model trained on one project may not generalise to another project with different coding practices. Explainable AI methods are commonly grouped into interpretable-by-design models and post-hoc explanations. Feature-importance techniques can show which input signals influenced a prediction; local explanation methods can approximate a model's behaviour near one example; and example-based explanations can show similar past

cases. For source code, explanations should be checked carefully because a highlighted token or feature is not necessarily the true causal reason for a defect. The proposed system combines the strengths of rule-based analysis and learned risk ranking. Static-analysis findings provide direct evidence, while a prediction model can help prioritise findings or identify patterns not covered by a single rule. Explanations are constrained to evidence that can be inspected by a developer. This is an integration-oriented proposal, not a claim that one explanation algorithm is universally best.

4. SCOPE AND ASSUMPTIONS

The initial scope is Java applications with accessible source code, build configuration, and, where available, test results and version-control history. The system is intended for development-time use in an IDE, code-review process, or continuous-integration pipeline. It may analyse methods, files, or pull-request changes depending on project size and available labels. The proposal assumes that code can be parsed reliably and that any training or evaluation dataset has clearly defined defect labels. It does not assume that every warning is a real bug, nor that all bugs are detectable from source code alone. The system does not replace developers, security review, tests, formal verification, or production monitoring. Confidential source code and repository metadata must be processed according to the organisation's privacy and access policies.

5. PROPOSED SYSTEM ARCHITECTURE



Figure 1. High-level architecture of the proposed explainable bug-detection system.

5.1 SOURCE AND PROJECT INGESTION

The input stage loads Java files, build metadata, dependencies, and optional test or commit information. Parsing should preserve file names, line numbers, method

boundaries, and syntax-tree relationships so that every warning can be traced back to its source. Unsupported files or parse failures should be reported rather than silently ignored.

5.2 FEATURE AND EVIDENCE EXTRACTION

Potential signals include code complexity, nesting depth, method size, null-related patterns, exception handling, resource closing, dependency use, static-analysis warnings, test failures, and recent code changes. Feature selection should avoid leaking future information into training data. Evidence should retain the rule name, code range, and analysis version when available.

5.3 DEFECT-RISK PREDICTION

A baseline system may begin with transparent rules and a conventional classifier, then compare more complex models only if they provide measurable benefit. The prediction output can include a risk category or calibrated probability, but the interface must explain that the value is an estimate based on the model and dataset, not a certainty that a defect exists. The system should allow the model to abstain or mark a finding as low-confidence when evidence is insufficient.

5.4 EXPLANATION LAYER

For each warning, the explanation layer gathers the strongest available evidence: a relevant static-analysis rule, a suspicious code path, a nearby null-sensitive operation, a resource that may not be closed, a failing test, or influential model features. It should distinguish direct observations from model-derived indicators. Explanations should be concise, reproducible, and linked to the source code. A generated natural-language description must not invent program behaviour that has not been checked.

5.5 DEVELOPER INTERFACE

The interface presents the file and line range, defect category, risk level, confidence or uncertainty, evidence, and recommended inspection steps. Developers can confirm, reject, or defer a finding and optionally record a reason. Feedback is useful for evaluation and future model improvement, but should not be treated as a reliable label without review.

6. SYSTEM WORKFLOW AND METHODOLOGY

Load project Analyse code Rank bug risks Explain evidence Developer verifies



Figure 2. Proposed processing sequence from project loading to developer review.

6.1 DATA PREPARATION

The project should define what counts as a bug and how labels are obtained—for example, linked defect fixes, curated issue records, or manually reviewed examples. Duplicate code versions and repeated instances of the same defect should be controlled to avoid overly optimistic evaluation. Training, validation, and test partitions should be separated by project or time where possible so that near-duplicate code does not appear on both sides.

6.2 ANALYSIS AND FEATURE GENERATION

Each Java project is parsed and analysed using the same documented configuration. The pipeline extracts code metrics and tool findings, maps them to stable source locations, and records missing or unavailable inputs. For changed code, the system may focus on modified methods while retaining relevant context from callers and dependencies. Build and test failures are stored separately from predicted defects because they are not interchangeable labels.

6.3 PREDICTION AND EXPLANATION

The prediction component assigns a risk score or category to each candidate location. The explanation component then links the output to evidence. For example, a warning about a possible null dereference may point to a nullable return value and a

later dereference without a visible guard. The system should say 'possible null dereference' and describe the evidence, not claim that the failure will definitely occur.

6.4 HUMAN REVIEW AND FEEDBACK

A developer reviews the code, runs or adds tests, and decides whether the finding is a true defect, a false positive, or an unresolved concern. Confirmed results can be used to calculate evaluation metrics. If feedback is reused for model training, the process should version the dataset, document label quality, and avoid automatically learning from unreviewed clicks.

6.5 SUGGESTED TECHNOLOGY STACK

A prototype may use a Java parser or compiler APIs, one or more static-analysis tools, a Python-based model service or Java-compatible inference component, a REST interface, and an IDE or web dashboard. A relational store can retain findings and review status. The exact stack should be selected according to team skills, data availability, and deployment constraints.

7. EXPLANATION DESIGN AND EXAMPLE FINDINGS

Table 1. Example explanation record presented to a developer		
Field	Example content	Purpose
Location	OrderService.java, method placeOrder(), lines 42–45	Lets the developer inspect the exact code
Finding	Possible null dereference	States the suspected defect category
Evidence	Lookup result is used before a visible null guard	Connects the warning to observable code
Uncertainty	Potential issue; context and API contract need review	Avoids presenting a prediction as fact
Next step	Check return contract and add a test for missing result	Provides a practical verification action

7.4 EXPLANATION QUALITY REQUIREMENTS

- Faithfulness: explanations should reflect the signals actually used by the model or tool.
- Traceability: each claim should link to a code location, rule, test, or recorded feature.
- Actionability: suggest a verification step rather than only naming a risk.
- Uncertainty: distinguish likely, possible, and confirmed defects.
- Consistency: similar evidence should produce explanations with comparable structure.

8. IMPLEMENTATION PLAN

The prototype can be implemented in stages so that the prediction model is not introduced before the data and baseline are understood.

- Phase 1 — Requirements: define bug categories, users, privacy constraints, and supported Java versions.
- Phase 2 — Baseline analysis: integrate a parser and static-analysis rules; save reproducible findings.
- Phase 3 — Dataset preparation: collect and label examples, document provenance, and split data carefully.
- Phase 4 — Prediction: train and compare a simple baseline model with rule-only results.
- Phase 5 — Explanations: link predictions to source evidence and generate concise descriptions.
- Phase 6 — Interface: display warnings, evidence, uncertainty, and review status.
- Phase 7 — Testing: evaluate detection, explanation faithfulness, usability, and failure handling.

9. EVALUATION PLAN

Table 2. Proposed evaluation metrics

Metric	Method	What it measures
Precision	True positives divided by all predicted positives	How often reported bugs are confirmed
Recall	True positives divided by all known positives	How many labelled defects are found
F1-score	Harmonic mean of precision and recall	Balance between missed bugs and false alarms
Top-k usefulness	Check whether confirmed defects appear among top-ranked findings	Supports prioritisation in large projects
Explanation faithfulness	Perturb or remove cited evidence and compare output changes	Whether explanation reflects model behaviour
Developer usefulness	Task study, review time, and feedback quality	Whether explanations support practical debugging
Runtime overhead	Measure analysis duration and resource use	Suitability for IDE and CI workflows

9.1 EXPERIMENTAL DESIGN

The evaluation should compare at least three conditions: static analysis alone, AI ranking without explanations, and the proposed explanation-supported workflow. Use projects not seen during training when feasible, record tool and model versions, and report the number and source of labelled examples. If developer studies are conducted, participants should receive the same tasks and access to comparable project context. Any human-subject study should follow applicable institutional requirements.

9.2 ERROR ANALYSIS

False positives should be grouped by cause, such as missing context, conservative rules, misleading training labels, or code patterns that are safe under a project-specific contract. False negatives should be reviewed to identify defect types not represented in the features or dataset. Explanation errors should be tracked separately: an accurate prediction paired with a misleading explanation is still a serious usability problem. 9.3 Reproducibility The report should include the Java version, build tools, static-analysis configuration, dataset construction procedure, train/test split, model parameters, hardware, and measurement method. Numerical results must not be reported until tests have been run. Where source code cannot be shared, provide enough methodological detail for independent review without exposing confidential code. 10. Applications Potential uses include pull-request review, prioritisation of static-analysis findings, code-quality dashboards, educational programming environments, legacy-code maintenance, and continuous-integration checks. In an IDE, an explanation can help a developer move from a warning to the relevant statement and test. In code review, a ranked list can focus attention on high-risk changes. In education, explanations may support learning if they are accurate and do not simply provide unverified answers. The system may also assist security-oriented review by highlighting suspicious input validation, unsafe API use, or error-handling patterns. However, security findings require dedicated threat modelling and security tools; a general bug predictor should not be presented as a complete vulnerability scanner.

11. BENEFITS AND EXPECTED OUTCOMES

The expected outcome is a prototype that loads Java projects, runs analysis, ranks potential defects, attaches evidence-based explanations, and supports developer review. The main anticipated benefit is improved transparency: developers can inspect why a warning appeared and decide whether it is relevant. Other possible benefits—such as reduced review time or fewer missed defects—remain hypotheses until measured in a controlled evaluation.

12. LIMITATIONS AND RISK MANAGEMENT

AI predictions depend on training data, code representation, and the projects used for evaluation. If historical bug reports are incomplete or labels are noisy, a model may learn patterns of reporting rather than underlying defect causes. A model may also perform poorly on a new framework, coding style, or Java version. Class imbalance

can make accuracy misleading, so precision, recall, and class-specific error analysis are important. Explanations have limitations as well. Feature importance may describe correlation rather than causation, and a natural-language explanation may omit a relevant program path. A source snippet may not include the caller, configuration, or runtime contract needed to judge the warning. Therefore, explanations should cite evidence, expose uncertainty, and invite verification. The interface must not imply that an explanation proves a bug exists. Privacy and intellectual-property risks arise when source code or repository data is sent to external services. A practical implementation should prefer approved local processing where required, enforce access control, minimise stored code, protect logs, and avoid sending secrets or credentials to third-party services. Model artefacts and datasets should be versioned and access-controlled. Risk management should include secure handling of repositories, validation of model inputs, dependency management, audit logs for system actions, clear user feedback controls, and a process for disabling unreliable models. Human review remains essential for high-impact changes. The tool should be treated as decision support rather than an autonomous code approver.

13. FUTURE SCOPE

Future work may explore graph-based representations of Java programs, call-graph and data-flow context, sequence models for source code, and retrieval of similar confirmed defects from a project's history. Explanations could be improved by combining static-analysis traces with tests that reproduce a suspected issue. A developer study could compare explanation formats such as rule-based evidence, code highlights, feature contributions, and counterfactual examples. Additional work could support multiple languages, integration with IDEs and continuous-integration platforms, and explanations tailored to novice or experienced developers. Cross-project evaluation is particularly important to measure generalisation. Future versions should also explore confidence calibration, abstention when evidence is weak, and ways to prevent feedback loops from reinforcing incorrect labels. 14. Conclusion This paper proposes an explainable AI workflow for detecting and explaining potential bugs in Java applications. The design combines source-code analysis, static-analysis evidence, optional project history and test signals, a defect-risk predictor, and a developer-facing explanation layer. Instead of presenting a score alone, the system aims to show the code location, supporting evidence, uncertainty, and a practical verification step. The approach can complement code review and testing, but it cannot guarantee that every defect will be found or that every warning is correct. Its usefulness must be established through reproducible experiments that measure precision, recall, ranking quality, explanation faithfulness, developer usefulness, and runtime overhead. Responsible deployment requires careful dataset design, privacy protection, transparent limitations, and continued human review.

REFERENCES

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, 2018.

[2] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[3] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in *Advances in Neural Information Processing Systems*, 2017.

[4] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why Should I Trust You?': Explaining the Predictions of Any Classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144.

[5] N. Japkowicz and S. Stephen, "The Class Imbalance Problem: A Systematic Study," *Intelligent Data Analysis*, vol. 6, no. 5, pp. 429–449, 2002.

[6] OWASP Foundation, OWASP Code Review Guide and applicable secure coding guidance. Consult the current official resources.

[7] SpotBugs Project, SpotBugs Documentation. Official project documentation.

[8] PMD Project, PMD Documentation. Official project documentation.

[9] Oracle, Java Language and Tools Documentation. Consult the documentation for the target Java version.

[10] National Institute of Standards and Technology, Artificial Intelligence Risk Management Framework (AI RMF 1.0), NIST AI 100-1, 2023.

Publication archival note: Official scholarly contribution published under CC BY 4.0 International License. Permanent record on <https://www.ijrek.org> | Article ID: IJREK-2026-00153