International Journal of Research in Advanced Engineering and Technology
www.allengineeringjournal.in
ISSN: 2455-0876
Received: 24-05-2026, Accepted: 23-06-2026, Published: 08-07-2026
Volume 12, Issue 3, 2026, Page No. 18-22

Data-driven software quality evaluation using artificial intelligence

Tamanna
Assistant Professor, Department of Computer Science, Chhotu Ram Arya College, Sonipat, Haryana, India
DOI: https://doi.org/10.66856/ijraet.2026.12.3.12026

Abstract
Software quality evaluation has traditionally relied on manually defined metrics, expert judgment, and testing activities performed at selected stages of the software development life cycle. Although these techniques remain valuable, they often provide limited support for identifying complex quality patterns across large and heterogeneous software repositories. Artificial intelligence (AI), combined with data-driven analysis, enables organizations to evaluate software quality continuously by learning from source code, development history, testing records, issue reports, and operational data. This paper presents a research framework for data-driven software quality evaluation using AI. The framework integrates data collection, preprocessing, feature engineering, machine learning, quality prediction, explainability, and continuous feedback. It considers functional correctness, reliability, maintainability, security, performance, and usability as interconnected quality dimensions. The paper discusses suitable data sources, machine learning techniques, evaluation metrics, implementation challenges, ethical considerations, and threats to validity. It also proposes a reference architecture for integrating AI-based quality assessment into modern software engineering workflows. The analysis indicates that AI can improve early defect detection, prioritize testing activities, estimate maintainability risks, and support evidence-based quality management. However, successful adoption depends on data quality, model transparency, domain adaptation, human oversight, and continuous monitoring.

Keywords: Artificial intelligence, data-driven analysis, machine learning, software quality, software metrics, defect prediction, software testing, explainable artificial intelligence

Introduction

Software systems support critical activities in healthcare, finance, transportation, communication, manufacturing, and public administration. As software becomes more complex, organizations require reliable methods for evaluating and improving its quality. Software quality encompasses multiple characteristics, including functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. These characteristics are influenced by source-code structure, development practices, system architecture, testing effectiveness, operational conditions, and user feedback.

Conventional software quality evaluation commonly depends on predefined rules, static analysis tools, test coverage, code reviews, defect counts, and expert assessment. These methods provide useful information, but they may not adequately capture nonlinear relationships among quality factors. For example, a module with moderate complexity may still present a high defect risk if it is frequently modified, poorly tested, and connected to unstable components. Identifying such relationships manually is difficult when projects contain millions of lines of code and extensive development histories.

Artificial intelligence provides techniques for discovering patterns in large datasets and generating predictions or recommendations. Supervised learning can estimate the likelihood of defects, unsupervised learning can identify anomalous development behavior, and natural language processing can analyze issue reports and requirements. Deep learning and large language models can additionally support code representation, vulnerability detection, test generation, and documentation analysis.

This paper investigates how data-driven AI methods can be applied to software quality evaluation. The main contributions are as follows:

- A multidimensional framework for evaluating software quality using heterogeneous engineering data.
- A reference architecture connecting software repositories, AI models, quality indicators, and development workflows.
- A discussion of appropriate machine learning methods and evaluation metrics.
- An analysis of practical limitations, explainability requirements, and threats to validity.
- A research agenda for reliable and responsible AI-assisted software quality management.

Background

a. Software Quality Evaluation

Software quality evaluation is the process of measuring and judging whether a software product satisfies specified requirements and stakeholder expectations. Quality can be assessed through direct measurements, such as response time and code complexity, or indirect indicators, such as defect density and code-review latency.

The ISO/IEC 25010 quality model organizes software quality into product characteristics and subcharacteristics. These include functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. A data-driven evaluation system should not treat these dimensions as isolated variables. Instead, it should model their interactions and identify trade-offs. For example, increased security controls may affect performance, while rapid feature development may reduce maintainability if technical debt accumulates.

b. Data Sources

AI-based quality evaluation depends on the availability of relevant and reliable data. Common data sources include:

- Source-code repositories containing files, commits, branches, and pull requests.
- Issue-tracking systems containing defects, feature requests, priorities, and resolutions.
- Continuous integration systems containing build outcomes, test results, and execution times.
- Static-analysis tools producing complexity, coupling, duplication, and vulnerability measurements.
- Code-review systems containing review comments, approvals, and change histories.
- Runtime monitoring systems containing failures, latency, resource use, and availability information.
- Requirements and documentation containing natural-language descriptions of expected behavior.
- User feedback containing ratings, complaints, support requests, and usage observations.

These sources differ in structure, quality, frequency, and reliability. Consequently, data integration is a central challenge in any proposed evaluation framework.

c. AI Techniques for Software Engineering

Machine learning methods can be grouped according to the type of task being performed. Classification models predict discrete outcomes, such as whether a file is defective. Regression models predict continuous values, such as the expected number of defects or maintenance effort. Clustering methods group components with similar characteristics. Anomaly-detection methods identify unusual changes, test failures, or operational events.

Natural language processing is useful for analyzing issue descriptions, requirements, review comments, and user feedback. Deep learning can learn representations of source code and software-development sequences. Ensemble methods can combine multiple models to improve predictive robustness. Explainable AI methods provide information about the variables that influence model predictions, which is particularly important when quality decisions affect release approval or developer workload.

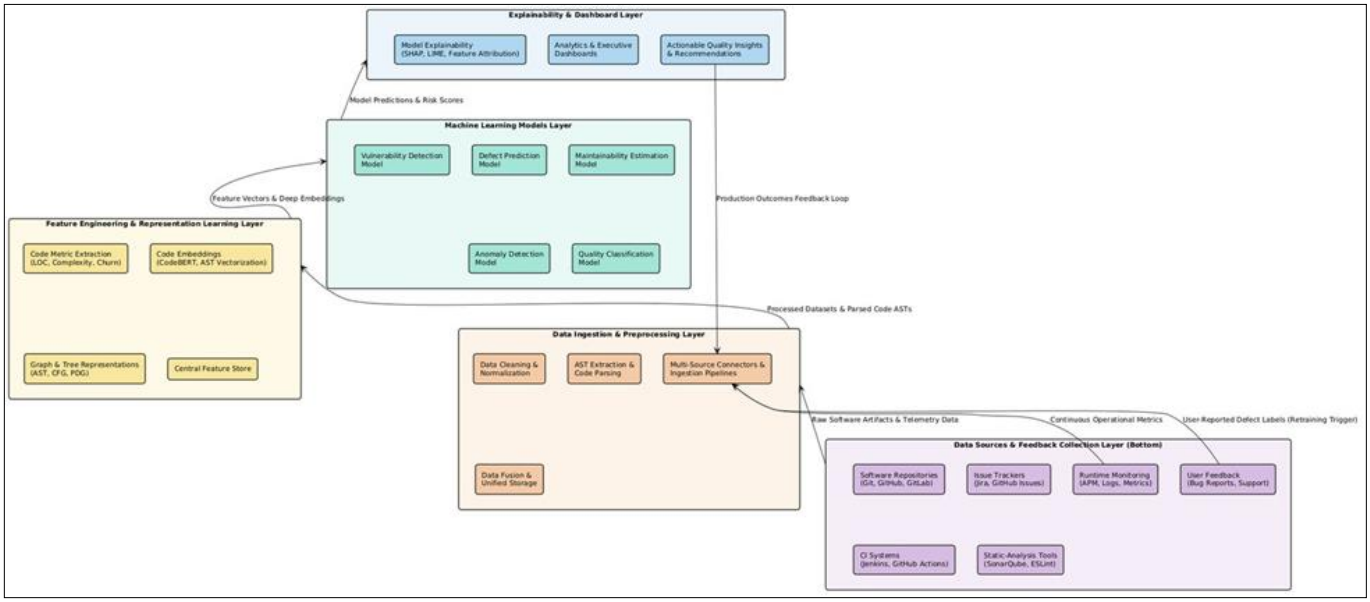
Proposed Framework

a. Framework Overview

The proposed framework consists of six layers:

1. Data acquisition.
2. Data preprocessing and integration.
3. Feature engineering and representation learning.
4. AI-based quality analysis.
5. Explainability and decision support.
6. Continuous feedback and model monitoring.

The framework is intended to operate within an iterative software development process. Quality predictions are generated when code is committed, reviewed, tested, released, and monitored in production. The system therefore supports both pre-release and post-release evaluation.



b. Data Acquisition and Integration

The first layer collects data from software engineering tools through application programming interfaces, event streams, database connectors, or repository analysis. Each record should be associated with a project, component, commit, developer action, test execution, or operational event.

Data integration requires a consistent identification strategy. For example, a defect reported in an issue-tracking system should be linked to the commit that corrected it, the files changed by that commit, the tests executed afterward, and any corresponding production incidents. This linkage enables the construction of historical training examples.

A unified data schema may include the following fields:

- Component identifier.
- Commit identifier.
- Timestamp.
- Lines added and deleted.
- Cyclomatic complexity.
- Coupling and cohesion measurements.
- Code duplication.
- Test coverage.
- Number of previous defects.
- Number of code-review revisions.
- Build and test outcomes.
- Security warnings.
- Runtime failures.
- User-reported incidents.

c. Data Preprocessing

Raw engineering data frequently contains missing values, duplicated records, inconsistent timestamps, mislabeled

defects, and changes unrelated to functional behavior. Preprocessing should therefore include:

- Removal of duplicate records.
- Normalization of numerical variables.
- Conversion of categorical variables into machine-readable representations.
- Imputation or explicit marking of missing values.
- Removal of personally identifiable information where appropriate.
- Temporal alignment of development and operational events.
- Verification of defect labels and quality outcomes.
- Detection of data leakage between training and testing sets.

Temporal alignment is especially important. A model predicting whether a newly modified component will contain a defect must use only information available before the prediction point. Using future defect reports or later test results would produce artificially optimistic performance.

d. Feature Engineering

Features should represent measurable factors associated with software quality. They may be divided into five groups.

1. **Product features:** These include lines of code, complexity, coupling, cohesion, inheritance depth, duplication, dependency count, and code churn.
2. **Process features:** These include commit frequency, number of developers modifying a component, review duration, number of review iterations, build frequency, and historical test failure rate.
3. **Testing features:** These include statement coverage, branch coverage, mutation score, test execution time, flaky-test frequency, and the proportion of changed code covered by tests.
4. **Operational features:** These include runtime exceptions, response-time percentiles, memory utilization, service availability, and incident frequency.
5. **Textual and semantic features:** These include terms from issue reports, requirements, review comments, documentation, and source-code representations.

Feature engineering should be guided by software engineering knowledge rather than relying only on automated selection. A statistically predictive feature may be a proxy for team size, organizational structure, or project maturity. Such relationships must be interpreted carefully before they are used for management decisions.

AI Models for Quality Evaluation

a. Defect Prediction

Defect prediction estimates the probability that a software component or change will contain one or more defects. Classification algorithms such as logistic regression, decision trees, random forests, gradient-boosting models, and neural networks can be applied to this task.

A binary defect-prediction model can be represented as

$$P(y = 1 \mid \mathbf{x}) = f(\mathbf{x}) \qquad (1)$$

Where $y = 1$ denotes a defective component, $\mathbf{x}$ is the feature vector, f and is the trained prediction function.

The output can be used to prioritize code reviews, allocate testing resources, and identify components requiring additional analysis. Because defective components are often less frequent than nondefective components, evaluation should not rely only on accuracy. Precision, recall, the F1-score, the area under the receiver operating characteristic curve, and the area under the precision-recall curve are more informative.

b. Maintainability Estimation

Maintainability estimation predicts the effort or risk associated with modifying and extending software. Relevant indicators include complexity, duplication, coupling, code churn, historical repair effort, documentation quality, and review activity.

A regression model can estimate maintenance effort as

$$\hat{m} = g(\mathbf{x}) \qquad (2)$$

Where $\hat{m}$ is the predicted maintenance effort g and is a regression model. Alternatively, maintainability can be divided into categories such as low, moderate, and high risk. These predictions can support technical-debt management. However, maintainability is partly contextual. A complex component may be acceptable if it implements a stable domain abstraction and is supported by comprehensive tests. Model outputs should therefore be treated as decision support rather than absolute judgments.

c. Vulnerability Detection

AI can assist security-quality evaluation by identifying patterns associated with common vulnerabilities. Models may analyze source code, dependency graphs, configuration files, commit histories, and security advisories.

Security models can be used to:

- Detect potentially vulnerable code patterns.
- Identify risky dependencies.
- Prioritize warnings from static-analysis tools.
- Associate code changes with historical security incidents.
- Recommend secure implementation patterns.

False positives and false negatives are significant concerns. Security predictions should be reviewed by qualified personnel, particularly in safety-critical or highly regulated systems.

d. Anomaly Detection

Unsupervised and semi-supervised learning can identify unusual patterns when labeled quality data are scarce. Examples include an unexpected increase in build failures, abnormal commit behavior, sudden growth in response time, or a change that modifies an unusually large dependency region.

Let represent the quality-related measurements observed at time. An anomaly detector computes an abnormality score:

$$a_t = h(\mathbf{x}_t, \mathcal{H}_{t-1}) \qquad (3)$$

Where is the history of observations before time. High values of indicate behavior that differs substantially from the established baseline.

Anomaly detection is valuable for early warning, but unusual behavior is not necessarily poor quality. The result should initiate investigation rather than automatically block a release.

e. Natural Language and Code Analysis

Natural language processing can classify issue reports, identify recurring failure themes, detect ambiguous requirements, and summarize review discussions. Code-

oriented language models can support vulnerability detection, code summarization, test generation, and semantic similarity analysis.
These methods should be validated against project-specific terminology and coding conventions. A model trained on public repositories may perform poorly on proprietary systems, domain-specific languages, or safety-critical code. Fine-tuning and domain adaptation may be necessary.

Evaluation Methodology

a. Experimental Design

An empirical evaluation should compare AI-based methods with established baselines. Suitable baselines include:

- Threshold-based static-analysis rules.
- Historical defect rates.
- Logistic regression using conventional metrics.
- Random selection of components for testing.
- Existing quality gates in the continuous integration pipeline.

The dataset should be divided chronologically rather than randomly whenever the goal is to simulate future prediction. For example, earlier releases may be used for training, a later release for validation, and the most recent release for testing. This design reduces information leakage and better represents operational use.

b. Evaluation Metrics

For binary classification, precision and recall are defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}} \qquad (4)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}} \qquad (5)$$

Where TP denotes true positives, FP denotes false positives, and FN denotes false negatives.
The F1-score is the harmonic mean of precision and recall:

$$\text{F}_1 = \frac{2\,\text{Precision}\,\text{Recall}}{\text{Precision+Recall}} \qquad (6)$$

For regression, mean absolute error, root mean square error, and coefficient of determination may be used. In addition to predictive performance, an operational evaluation should measure:

- Defects detected before release.
- Reduction in testing effort.
- False-alarm rate.
- Time required for developer response.
- Change-failure rate.
- Model inference latency.
- Developer acceptance.
- Stability across projects and releases.

c. Explainability and Calibration

A model can achieve high predictive performance while producing poorly calibrated probabilities. Calibration measures whether predicted probabilities correspond to observed frequencies. For example, among components assigned a defect probability of , approximately 80 percent should be defective if the model is well calibrated.
Explainability methods can identify the features contributing to a prediction. Local explanations describe an individual component or change, whereas global explanations describe general model behavior. Explanations should be presented in software-engineering terms, such as high churn, low test coverage, or repeated historical failures, rather than as opaque numerical scores.

Reference Architecture

The proposed architecture contains the following components:
Data connectors: Interfaces for repositories, issue trackers, CI platforms, static-analysis tools, observability systems, and user-feedback channels.
Quality Data Lake: A storage layer containing raw and processed engineering data with timestamps and traceability links.
Feature store: A managed repository of reusable, versioned features for training and inference.
Model-serving layer: Services that execute trained models when commits, pull requests, builds, or operational events occur.
Quality intelligence engine: A component that combines predictions from several models into quality indicators and risk rankings.
Explainability service: A service that provides evidence supporting each prediction, including influential features, similar historical examples, and uncertainty estimates.
Developer interface: Dashboards, pull-request comments, issue annotations, and release reports that communicate actionable findings.
Feedback and monitoring service: A component that records user responses, tracks model drift, measures actual outcomes, and initiates retraining when necessary.

The architecture should support human approval for consequential decisions. Automatic recommendations may prioritize testing or request additional review, but release blocking should generally require configurable policies and responsible human oversight.

Challenges and Limitations

a. Data Quality and Label Reliability

Defect labels are often incomplete or inconsistent. Some defects are never reported, while others are assigned to the wrong component or release. Labels derived from issue trackers may also reflect reporting behavior rather than actual defect frequency.

b. Class Imbalance

In many projects, defective changes are much less common than nondefective changes. A model that predicts no defect for every component may achieve high accuracy but provide no practical value. Resampling, cost-sensitive learning, threshold optimization, and precision-recall analysis can address this problem.

c. Concept Drift

Software systems evolve. Architectures, programming languages, teams, requirements, and testing practices may change over time. Consequently, the relationships learned by a model may become outdated. Continuous performance monitoring and periodic retraining are required.

d. Generalization Across Projects

A model trained on one project may not transfer effectively to another because of differences in coding standards, development processes, domain complexity, and defect-reporting practices. Cross-project prediction should therefore be evaluated separately from within-project prediction.

e. Explainability and Trust

Developers may reject predictions that appear arbitrary or conflict with domain knowledge. Explanations must be accurate, concise, and connected to actions that developers can take. Excessive warnings may produce alert fatigue and reduce trust in the system.

f. Privacy and Governance

Development data may contain confidential source code, employee-related information, customer data, or security-sensitive details. Data collection and model training must comply with organizational policies and applicable legal requirements. Access control, anonymization, audit logging, and secure model deployment are necessary safeguards.

g. Automation Bias

Users may overtrust AI recommendations, particularly when predictions are presented with unjustified confidence. AI-based quality evaluation should augment professional judgment rather than replace it. Interfaces should communicate uncertainty and allow users to challenge or correct model outputs.

Discussion

Data-driven AI provides an opportunity to move software quality evaluation from periodic inspection toward continuous risk assessment. The greatest value is likely to arise when several evidence sources are combined. For example, a quality system can give higher priority to a code change that exhibits high complexity, substantial churn, low changed-code coverage, and a history of defects. Such a decision is more informative than any individual metric.

AI can also improve resource allocation. Testing teams can focus on high-risk changes, reviewers can inspect components with unusual patterns, and project managers can identify areas where technical debt is accumulating. Runtime data can complete the feedback loop by revealing whether pre-release quality indicators correspond to production behavior.

Nevertheless, predictive performance alone is not sufficient. A useful system must reduce practical effort, produce actionable recommendations, and integrate with existing development workflows. It must also remain reliable under changing conditions. Model monitoring, traceability, and human review are therefore essential parts of the quality framework.

Future systems may combine symbolic software-engineering rules with learned models. Static-analysis constraints can provide reliable safeguards, while machine learning can identify patterns that are difficult to encode manually. This hybrid approach may offer a better balance between flexibility, accuracy, and explainability.

Conclusion

This paper presented a framework for data-driven software quality evaluation using artificial intelligence. The framework integrates heterogeneous software engineering data with machine learning, natural language processing, anomaly detection, explainability, and continuous feedback. It can support defect prediction, maintainability estimation, vulnerability detection, testing prioritization, and operational quality monitoring.

The effectiveness of such systems depends on reliable labels, appropriate temporal evaluation, project-specific adaptation, explainable predictions, and continuous monitoring. AI should be used as an evidence-based assistant within software engineering processes rather than as an unquestionable replacement for human expertise. Future research should investigate standardized benchmark datasets, cross-project transfer learning, uncertainty-aware quality prediction, human-AI collaboration, and rigorous industrial evaluations.

References

1. ISO/IEC. ISO/IEC 25010:2011 Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation, International Organization for Standardization, Geneva, Switzerland., 2011.
2. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering.,1996:22(10):751-761.
3. Khoshgoftaar TM, Allen EB, Crook K, Hudepohl J. Application of neural networks to software quality modeling of a very large telecommunications system. IEEE Transactions on Neural Networks.,1997:8(4):902-909.
4. Fenton NE, Neil M. A critique of software defect prediction models. IEEE Transactions on Software Engineering.,1999:25(5):675-689.
5. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering.,2012:38(6):1276-1304.
6. Catal C, Diri B. A systematic review of software fault prediction studies. Expert Systems with Applications.,2009:36(4):7346-7354.
7. Menzies T, Greenwald J, Frank A. Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering.,2007:33(1):2-13.
8. Kim M, Nam J, Yeon J, Song SJ, Kim S. Predicting faults from cached history. in Proc. 29th ACM/IEEE Int. Conf. Software Engineering., 2007, 739-748.
9. Harman M, Mansuri SA, Zhang Y, Sarro F. Investigating the use of machine learning for software engineering. Empirical Software Engineering.,2017:22:1-4.
10. Spinellis D. Tool writing: A forgotten art? IEEE Software.,2005:22(4):9-11.
11. Sarro F, Petrozziello A, Harman M. Multi-objective software effort estimation. in Proc. IEEE Int. Conf. Software Maintenance and Evolution., 2016, 1-10.
12. Breiman L. Random forests. Machine Learning.,2001:45:5-32.
13. Chen T, Guestrin C. XGBoost: A scalable tree boosting system. in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining., 2016, 785-794.
14. Ribeiro MT, Singh S, Guestrin C. Why should I trust you?: Explaining the predictions of any classifier. in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining., 2016, 1135-1144.
15. Lundberg SM, Lee SI. A unified approach to interpreting model predictions. in Advances in Neural Information Processing Systems.,2017:30:4765-4774.