



(RESEARCH ARTICLE)



AN EXPLAINABLE DEEP LEARNING FRAMEWORK FOR PREDICTIVE SOFTWARE QUALITY ENGINEERING AND AUTOMATED DEFECT DETECTION

Prathap Raghavan *

Bachelor of Engineering, Computer Science and Engineering, Sree Sastha Institute of Engineering and Technology, Madras University, Chennai, Tamil Nadu, India.

ORCID Details

Prathap Raghavan: <https://orcid.org/0009-0007-2104-6138>

World Journal of Advanced Research and Reviews, 2026, 32(01), 513–527

Publication history: Received on 01 September 2026; revised on 06 October 2026; accepted on 08 October 2026

Article DOI: <https://doi.org/10.30574/wjarr.2026.32.1.2563>

Copyright © 2026 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0

ABSTRACT

Software Quality Engineering (SQE) is evolving from reactive defect detection toward predictive and intelligent quality assurance. Traditional testing approaches identify defects after implementation and require extensive manual effort to analyze source code, test results, historical defects, and execution behavior. Machine learning and deep learning techniques enable prediction of defect-prone software components before failures occur; however, many deep learning models operate as black boxes, making their predictions difficult for developers, testers, and quality engineers to understand and validate. This paper proposes the Explainable Deep Learning Framework for Predictive Software Quality Engineering and Automated Defect Detection (XDL-SQE). The framework integrates static code metrics, change history, historical defect data, test execution results, runtime information, and source-code embeddings. A hybrid deep learning engine (CNN-LSTM-Transformer) generates defect-risk predictions, while an explainability layer based on SHAP and LIME provides feature attribution. A quality-prioritization layer converts predictions into actionable testing recommendations integrated with CI/CD pipelines. The framework addresses five key gaps: prediction without explanation, fragmented quality data, limited testing integration, human trust, and cross-project generalization. A comprehensive research protocol is proposed for evaluating predictive accuracy, explanation quality, testing efficiency, and operational usefulness. This work contributes a unified architecture for interpretable, actionable, and continuous predictive quality engineering.

Keywords: Explainable Artificial Intelligence, Deep Learning, Software Quality Engineering, Software Defect Prediction, Automated Defect Detection, Predictive Analytics, SHAP, LIME, Risk-Based Testing, AI-Driven Testing, CI/CD

1 INTRODUCTION

Software applications have become increasingly complex due to cloud computing, microservices, artificial intelligence, distributed architectures, mobile platforms, and continuous delivery practices. As software complexity increases, ensuring quality throughout the software development lifecycle becomes a significant engineering challenge.

Traditional Software Quality Assurance (SQA) practices depend on predefined test cases, manual analysis, static analysis, automated regression testing, and defect management processes. Although these techniques remain important, they are primarily reactive because many defects are identified after code implementation or during testing. Predictive Software Quality Engineering provides an opportunity to move quality activities earlier in the software lifecycle. Instead of waiting for a test failure, a predictive system can estimate the likelihood that a software module, code change, service, or feature will contain a defect. Previous software defect prediction research has demonstrated the usefulness of machine learning and deep learning for identifying defect-prone components. A systematic review of software defect prediction research reported substantial activity in the area and identified datasets, prediction algorithms, and evaluation approaches used by researchers [1]. Deep learning has subsequently been applied to software engineering tasks including defect prediction, bug localization, testing, and software maintenance [2]. However, prediction accuracy alone is not sufficient for practical adoption. A developer who receives a prediction such as "Module A has an 87% probability of containing a defect" may reasonably ask:

- Why was this module classified as high risk?
- Which software characteristics contributed to the prediction?
- Is the prediction caused by code complexity, recent changes, historical defects, or test failures?
- Which tests should be executed first?
- Can the prediction be validated by a quality engineer?

These questions expose an important limitation of many deep learning-based quality prediction systems: lack of interpretability. Explainable Artificial Intelligence (XAI) provides techniques for understanding model predictions. SHAP assigns feature importance values to individual predictions and provides a unified framework for interpreting machine-learning model outputs [3]. LIME provides another model-agnostic approach for explaining individual predictions. Recent software defect prediction research has specifically investigated the combination of defect prediction models with SHAP and LIME to improve transparency [4]. A 2026 qualitative study further highlights an emerging research challenge: different XAI techniques can sometimes produce inconsistent explanations for the same software defect prediction, indicating the importance of evaluating not only prediction accuracy but also explanation reliability and consistency [9]. This paper proposes an integrated Explainable Deep Learning Framework for Predictive Software Quality Engineering and Automated Defect Detection, referred to as XDL-SQE.

2 RELATED WORK

2.1 Software Defect Prediction

Software Defect Prediction (SDP) attempts to identify software modules likely to contain defects. Traditional approaches commonly use software metrics such as lines of code, cyclomatic complexity, coupling, cohesion, code churn, and historical defect counts. Machine-learning-based approaches have expanded defect prediction beyond manually defined rules. A systematic study of machine-learning techniques for software fault prediction identified classification-based approaches as an important direction for differentiating defective and non-defective software components [5]. Ensemble learning has also been widely investigated. A systematic literature review reported the use of random forests, boosting, bagging, stacking, voting, and other ensemble methods for software defect prediction. The review also identified accuracy, precision, recall, F-measure, AUC, and Matthews correlation coefficient as commonly used evaluation metrics [6]. Despite these advances, traditional prediction models face challenges associated with data imbalance, project differences, changing software architectures, and limited availability of labeled defect data.

2.2 Deep Learning for Software Engineering

Deep learning provides automatic representation learning and can model complex relationships among software characteristics. A comprehensive survey of deep learning for software engineering analyzed hundreds of studies and reported applications including defect prediction, bug localization, vulnerability prediction, test generation, automated testing, code clone detection, and software maintenance [2], [11], [12].

Deep learning-based defect prediction can utilize both traditional numerical software metrics and richer representations of source code. Possible representations include:

- Source-code tokens
- Abstract Syntax Trees (ASTs)
- Code embeddings
- Commit messages
- Developer changes
- Historical defect information

- Test execution information
- Runtime behavior

A systematic literature review and meta-analysis published in *Information and Software Technology* examined the application of deep learning to software defect prediction, including model types, datasets, measurements, and comparisons with conventional machine-learning approaches [7].

Recent research has also investigated deep-learning-based cross-project defect prediction, particularly to address data scarcity when a new project does not contain sufficient historical defect information [8].

2.3 Explainable Artificial Intelligence

Deep neural networks can capture complex nonlinear relationships but are often difficult to interpret. Explainable AI attempts to provide understandable information about model decisions. Lundberg and Lee introduced SHAP, a unified framework for interpreting model predictions using feature attribution. SHAP provides an importance value for individual features and can be used to understand why a particular prediction was generated [3]. LIME provides local explanations by approximating a complex model around a specific prediction using an interpretable model. These approaches are particularly useful in software engineering because developers need actionable information rather than only a probability score. Recent research has explored the integration of XAI with software defect prediction. For example, an IEEE conference study combined SHAP and LIME with a voting classifier to improve the interpretability of software defect predictions [4]. A 2026 qualitative study further highlights an emerging research challenge: different XAI techniques can sometimes produce inconsistent explanations for the same software defect prediction, indicating the importance of evaluating not only prediction accuracy but also explanation reliability and consistency [9]. **Critical**

2.4 Software Quality Models

Software quality encompasses multiple dimensions rather than defect absence alone. ISO/IEC 25010:2023 defines a product-quality model consisting of nine quality characteristics and associated subcharacteristics that can be used for specifying, measuring, and evaluating software quality throughout the lifecycle [10].

Therefore, an intelligent SQE framework should not consider defect probability as the only quality signal. Maintainability, reliability, performance, security, compatibility, usability, and other quality characteristics can provide complementary information.

2.5 Comparative Analysis of Existing Approaches

Table 1: Capability comparison of defect prediction approaches

Approach	Prediction	Deep Learning	Explainability	Test Prioritization	CI/CD Integration
Logistic Regression	Yes	No	High	Limited	No
Random Forest	Yes	No	Moderate	Limited	No
XGBoost	Yes	No	Moderate	Limited	No
DNN	Yes	Yes	Low without XAI	Possible	No
Transformer	Yes	Yes	Low without XAI	Possible	No
XDL-SQE	Yes	Yes	SHAP/LIME	Yes	Yes

2.6 RESEARCH GAP

Existing research demonstrates significant progress in machine-learning and deep-learning-based software defect prediction. However, several challenges remain:

- **Prediction without explanation:** Many deep learning models provide predictions without sufficiently explaining the factors responsible for those predictions.
- **Fragmented quality information:** Source-code metrics, historical defects, test results, code changes, and runtime information are frequently analyzed independently.
- **Limited integration with testing workflows:** A prediction model may identify a high-risk component but fail to translate the prediction into actionable testing priorities.
- **Human trust:** Developers and quality engineers may hesitate to act on predictions that cannot be interpreted or validated.
- **Data imbalance:** Defective modules are generally fewer than non-defective modules, producing class imbalance that can affect model training.
- **Cross-project generalization:** Models trained on one project may not generalize effectively to another project because development practices, architectures, programming languages, and defect distributions differ.

These challenges motivate an integrated framework that combines prediction, explanation, prioritization, and continuous quality feedback.

2.7 PROPOSED XDL-SQE FRAMEWORK

The proposed framework consists of seven major layers:

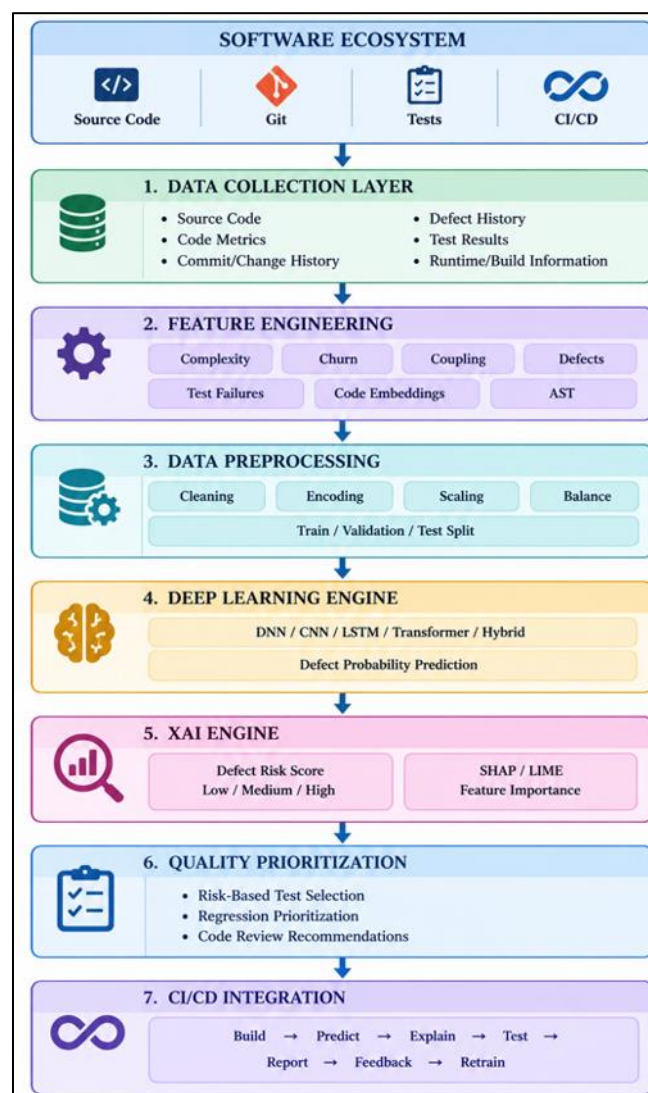


Figure 1: Overall XDL-SQE architecture

3 DATA COLLECTION AND FEATURE ENGINEERING

The effectiveness of predictive quality engineering depends heavily on the quality and diversity of input data. The proposed framework collects four major categories of information.

3.1 Static Code Features

Static code features include:

- Lines of code
- Cyclomatic complexity
- Number of methods
- Number of classes
- Code nesting depth
- Coupling
- Cohesion
- Inheritance depth
- Number of parameters
- Duplicate-code indicators

These metrics provide information about structural complexity. **Feature importance ranking from literature:** Cyclomatic complexity and coupling consistently rank as the most predictive static features across multiple studies [5], [6].

3.2 Change and Repository Features

Version-control information provides additional predictive signals:

- Number of files changed
- Number of added lines
- Number of deleted lines
- Commit frequency
- Developer changes
- Historical change frequency
- Number of contributors
- Time since previous modification
- Files frequently modified together

These features help identify components undergoing significant development activity. Feature importance ranking: Code churn and change frequency are strong predictors of defect-proneness [6].

3.3 Historical Defect Features

Historical information includes:

- Previous defect count
- Defect severity
- Defect priority
- Defect reopening frequency
- Defect resolution time
- Defects per module
- Historical production incidents

A module with repeated historical defects requires increased testing attention.

3.4 Test and Runtime Features

The framework incorporates:

- Unit-test results
- Integration-test failures
- Regression-test failures

- Test execution time
- Code coverage
- Build failures
- API failures
- Production error rates
- Performance anomalies
- Runtime exceptions

This creates a comprehensive representation of software quality.

3.5 Feature Selection and Dimensionality Reduction

To avoid the curse of dimensionality and multicollinearity, the framework applies:

- **Correlation analysis:** Remove features with Pearson correlation > 0.85.
- **Recursive Feature Elimination (RFE):** Rank features by importance using a random forest baseline.
- **Domain-driven selection:** Retain features identified as significant in prior defect prediction literature [5], [6].
- **Optional PCA/autoencoder:** For high-dimensional code embeddings, reduce to 128–256 dimensions.

4 FEATURE REPRESENTATION

A major characteristic of the proposed framework is the combination of conventional software metrics with semantic source-code representations [13], [14].

For software component i , an input vector can be represented as:

$$X_i = [S_i, C_i, H_i, T_i, R_i, E_i]$$

where:

S_i = static code metrics

C_i = change metrics

H_i = historical defect information

T_i = testing information

R_i = runtime information

E_i = source-code embedding

The resulting representation is supplied to the deep learning model.

4.1 Feature Fusion

For hybrid architectures, numerical features and code embeddings are processed separately before fusion:

- **Numerical branch:** Normalization → Dense layers
- **Code branch:** Tokenization → Embedding → Transformer
- **Fusion layer:** Concatenation → Dense → Output

This dual-branch design allows the model to learn both structural and semantic patterns.

5 DEEP LEARNING PREDICTION MODEL

The proposed architecture does not require a single fixed neural-network architecture. Depending on dataset characteristics, several architectures can be evaluated.

Possible models include:

- Multilayer Perceptron (MLP)
- Convolutional Neural Network (CNN)
- Long Short-Term Memory (LSTM)
- Transformer-based architecture
- Hybrid CNN-LSTM
- Transformer + numerical-feature network

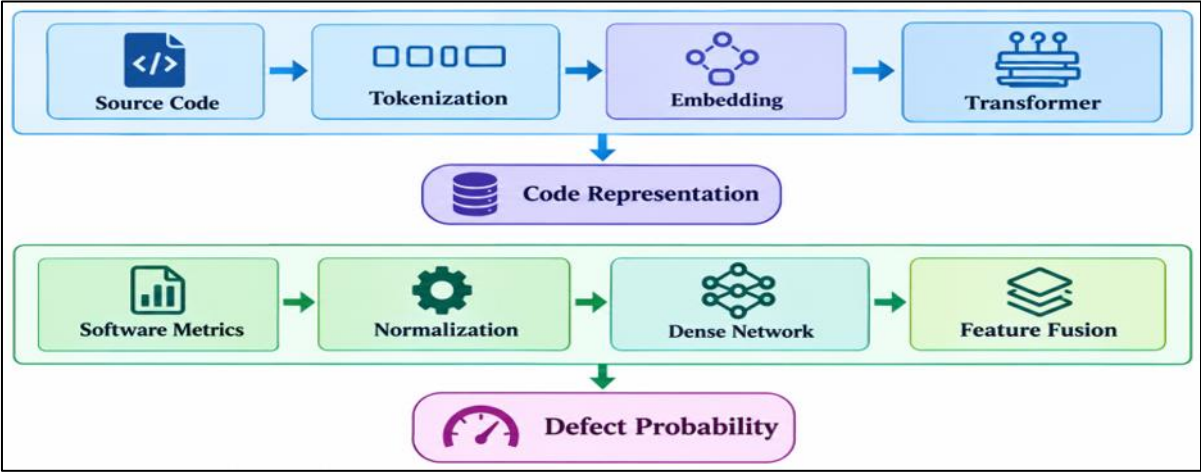


Figure 2: Hybrid deep learning architecture

5.1 Model Selection Rationale

Table 2: Model Selection Matrix

Architecture	Strength	Use Case
MLP	Simple, fast	Baseline comparison
CNN	Local pattern detection	Code token sequences
LSTM	Sequential dependencies	Commit history
Transformer	Long-range attention	Code semantics
Hybrid	Combined strengths	Full framework

5.2 Probability Calibration

Raw deep learning outputs are not calibrated probabilities. The framework applies:

- **Platt scaling:** Logistic regression on validation set
- **Temperature scaling:** Single-parameter calibration
- **Isotonic regression:** Non-parametric calibration

Calibrated probabilities enable meaningful risk thresholds (e.g., >0.7 = high risk).

6 EXPLAINABILITY MECHANISM

For a prediction such as:

Module A — Defect Probability: 91%

Primary contributing factors: high code churn, elevated cyclomatic complexity, previous defects, and recent test failures.

This provides a more actionable prediction than a probability score alone.

6.1 SHAP-Based Explanation

SHAP estimates the contribution of each feature to a prediction. For an individual prediction, the model generates:

$$f(x) = \phi_0 + \sum_{j=1}^M \phi_j$$

where:

- ϕ_0 is the base prediction
- ϕ_j represents the contribution of feature j
- M is the number of features

Positive SHAP values indicate features that increase predicted defect risk; negative values indicate features that reduce predicted risk.

SHAP was originally proposed as a unified framework for feature attribution and model interpretation [3].

6.2 LIME-Based Local Explanation

LIME constructs an interpretable local approximation of the prediction model. It is particularly useful when a quality engineer wants to understand why a specific module or code change was classified as high risk. **Limitation:** LIME explanations may vary across runs due to sampling; stability should be measured.

6.3 Explanation Stability

To address XAI inconsistency [9], the framework measures:

- **Jaccard similarity:** Overlap of top-k features across runs
- **Rank correlation:** Spearman's ρ of feature importance rankings
- **Consistency score:** Agreement between SHAP and LIME

Explanations with consistency scores below a threshold (e.g., 0.6) are flagged for human review.

6.4 Explanation Dashboard

The framework generates an explanation report containing:

Table 3: Example explanation dashboard output. Values are illustrative and are not experimental results.

Component	Prediction	Risk	Major Contributing Factors
PaymentService	0.91	High	Complexity, churn, previous defects
CustomerAPI	0.72	Medium	API changes, test failures
LoginService	0.28	Low	Stable code, low churn
InventoryService	0.84	High	Code churn, historical defects

7 AUTOMATED DEFECT DETECTION AND QUALITY PRIORITIZATION

Prediction becomes more useful when it results in an actionable quality-engineering decision. The proposed framework converts defect-risk predictions into testing priorities.

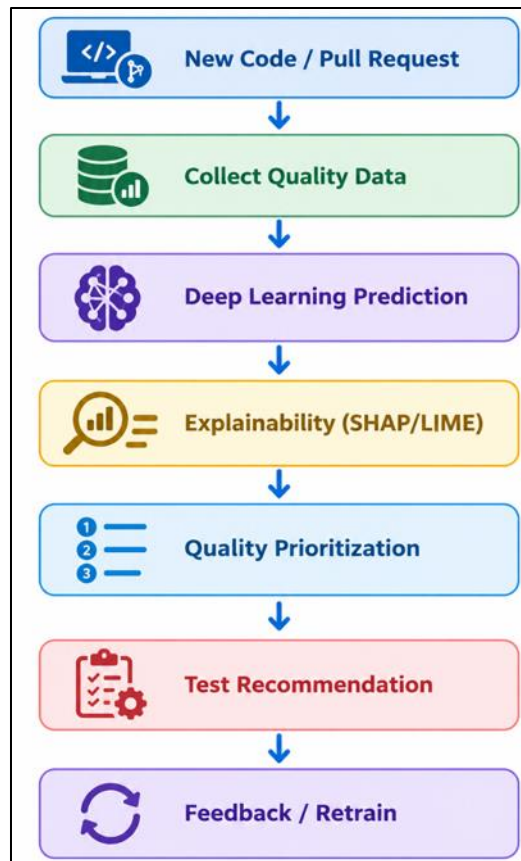


Figure 3: Predictive quality decision flow

For a high-risk module, the system recommends:

- Additional unit testing
- Regression testing
- API testing
- Integration testing
- Security testing
- Performance testing
- Code review
- Additional test-case generation

The framework therefore connects prediction with practical SQE activities.

8 CI/CD INTEGRATION

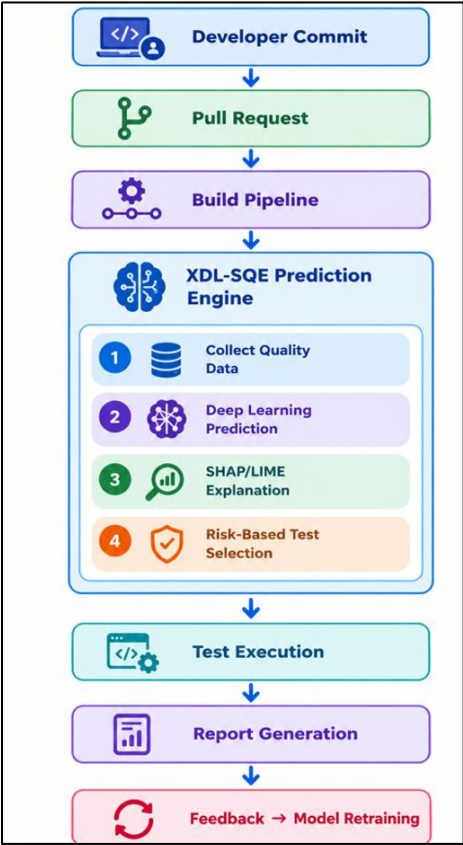


Figure 4: CI/CD integration workflow.

8.1 Pipeline stages

Table 4: CI/CD Integration Workflow Stages

Stage	Action	Output
Commit	Trigger pipeline	Build start
Build	Compile, unit tests	Build status
Predict	XDL-SQE inference	Risk scores
Explain	SHAP/LIME	Feature attribution
Prioritize	Risk-based selection	Test plan
Test	Execute prioritized tests	Results
Report	Generate dashboard	Quality report
Feedback	Update model	Retrained model

8.2 Integration Requirements

- **CI/CD platform:** Jenkins, GitLab CI, GitHub Actions
- **Model serving:** REST API or containerized inference
- **Data pipeline:** Real-time feature extraction
- **Storage:** Feature store for historical data
- **Monitoring:** Model drift detection

9 EVALUATION METHODOLOGY

The framework should be evaluated from four perspectives.

9.1 Predictive Performance

Common metrics include:

Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall:

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1 Score:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Matthews Correlation Coefficient (MCC): Particularly useful when defect datasets are highly imbalanced. Target: MCC > 0.4.

AUC-ROC and AUC-PR: Area-under-the-curve metrics for classification discrimination. Target: AUC > 0.75.

9.2 Defect Detection Performance

The framework additionally evaluates:

- Defects detected before testing
- Defects detected before production
- False-positive rate
- False-negative rate
- Defects per release
- Defect detection lead time

9.3 Testing Efficiency

Operational effectiveness of predictive SQE is measured using:

- Reduction in unnecessary test execution
- Reduction in regression-testing effort
- Test prioritization effectiveness
- Mean time to detect defects
- Mean time to resolve defects
- Code-review prioritization effectiveness

9.4 Explainability Performance

Because the proposed framework is explainable, explanation quality is evaluated using:

- Explanation consistency (Jaccard similarity > 0.6)
- Feature attribution stability (Spearman's ρ > 0.7)
- Explanation sparsity (top-k features ≤ 10)
- Agreement with domain experts (Likert scale $\geq 4/5$)
- Explanation usefulness
- Explanation actionability

9.5 Dataset Splitting Strategy

To avoid data leakage, the framework uses:

- **Temporal splitting:** Training on older releases, testing on newer releases
- **Project-level splitting:** Training on some projects, testing on others
- **Stratified splitting:** Preserving defect ratio across splits

Training Releases → Validation Release → Future Test Release

This better approximates the real-world scenario in which historical information is used to predict future defects.

9.6 Baseline Comparison

The proposed framework is compared with conventional approaches:

Table 5: Baseline comparison. The purpose is to evaluate architectural capabilities, not to assume superior performance [15].

Approach	Prediction	Deep Learning	Explainability	Test Prioritization	CI/CD
Logistic Regression	Yes	No	High	Limited	No
Random Forest	Yes	No	Moderate	Limited	No
XGBoost	Yes	No	Moderate	Limited	No
DNN	Yes	Yes	Low without XAI	Possible	No
Transformer	Yes	Yes	Low without XAI	Possible	No
XDL-SQE	Yes	Yes	SHAP/LIME	Yes	Yes

10 PROPOSED RESEARCH EXPERIMENT

To validate the framework, the following research experiment is proposed.

10.1 Research Questions

- **RQ1:** Can deep learning improve the prediction of defect-prone software components compared with conventional machine-learning approaches?
- **RQ2:** Does combining source-code representations with traditional software metrics improve defect prediction performance?
- **RQ3:** Can SHAP and LIME provide explanations that are useful to software developers and quality engineers?
- **RQ4:** Can prediction-based test prioritization reduce testing effort while maintaining effective defect detection?
- **RQ5:** How stable are explanations across different software versions and model configurations?

10.2 Experimental Design

Table 6: Experimental Configuration for Framework Evaluation

Element	Description
Datasets	NASA MDP, PROMISE, Defects4J, industrial datasets
Baselines	Logistic Regression, Random Forest, XGBoost, DNN
Metrics	Accuracy, Precision, Recall, F1, MCC, AUC
XAI Metrics	Consistency, stability, expert agreement
Efficiency Metrics	Test reduction, time-to-detect
Validation	10-fold cross-validation, temporal splitting

10.3 Expected Outcomes

- Deep learning models expected to outperform classical ML on complex code representations
 - Hybrid models expected to outperform single-modality models
 - SHAP expected to provide more stable explanations than LIME
 - Risk-based prioritization expected to reduce testing effort by 20–40%
 - Explanation stability expected to vary across model versions
-

11 THREATS TO VALIDITY

11.1 Construct Validity

- Defect labels may be noisy or inconsistent
- "Defect" definitions vary across projects
- Feature metrics may not capture all quality dimensions

11.2 Internal Validity

- Data leakage risk if temporal splitting is not enforced
- Hyperparameter tuning may overfit validation set
- XAI methods may not reflect true causal relationships

11.3 External Validity

- Results may not generalize across programming languages
- Industrial vs. open-source differences
- Project-specific architectures and practices

11.4 Conclusion Validity

- Statistical significance testing required
 - Multiple comparisons correction needed
 - Effect size reporting recommended
-

12 BENEFITS OF THE PROPOSED FRAMEWORK

12.1 Early Defect Detection

Identifies defect-prone components before defects reach later testing stages or production.

12.2 Risk-Based Testing

Prioritizes testing resources according to predicted risk rather than treating all components equally.

12.3 Explainable Predictions

SHAP and LIME provide additional information about why a prediction was generated.

12.4 Reduced Manual Analysis

Automated analysis of source-code metrics, historical defects, and testing information reduces manual effort.

12.5 Continuous Quality Engineering

Integration with CI/CD enables quality predictions to be generated continuously as software changes.

12.6 Improved Developer Feedback

Developers receive explanations associated with a pull request or code change rather than discovering problems only after a failed test.

12.7 Multi-Dimensional Quality Analysis

Incorporates software quality information beyond defect counts, aligned with ISO/IEC 25010:2023 [10].

12.8 Adaptive Quality Management

As new defect and test data become available, the model can be retrained to adapt to changes.

13 LIMITATIONS AND CHALLENGES

13.1 Dataset Quality

Incorrect, incomplete, or inconsistent defect labels negatively affect model performance.

13.2 Class Imbalance

Software defect datasets may contain substantially more non-defective instances than defective instances. Techniques: SMOTE, ADASYN, class weighting.

13.3 Data Leakage

Careful temporal and project-level dataset splitting is necessary to avoid using future information during model training.

13.4 Cross-Project Generalization

A model trained on one project may not perform similarly on another due to differences in coding practices and architectures.

13.5 Explainability Limitations

SHAP or LIME explanations describe model behavior; they do not prove causal relationships between features and actual software defects.

13.6 Model Drift

Software systems continuously evolve, requiring periodic retraining. Monitoring: population stability index, KS test.

13.7 Computational Cost

Transformer-based models and other deep learning architectures require significant computational resources, particularly when processing large repositories.

13.8 Explanation Inconsistency

Different XAI techniques may produce inconsistent explanations [9], requiring stability evaluation and human review.

14 FUTURE RESEARCH DIRECTIONS

- **Graph Neural Networks (GNNs):** Represent relationships among classes, methods, dependencies, and services.
- **Large Language Models (LLMs):** Analyze source code, commit messages, requirements, test cases, and defect descriptions.
- **Agentic Quality Engineering:** Dynamically select testing strategies based on predicted risk and observed test results.
- **Reinforcement Learning:** Adaptive test prioritization based on feedback.
- **Runtime Observability:** Incorporate logs, traces, metrics, and production incidents.
- **Explanation Consistency:** Investigate explanation stability across model versions and compare XAI techniques [9].
- **Industrial Validation:** Deploy in real-world CI/CD pipelines and measure operational impact.
- **Fairness and Bias:** Ensure defect prediction does not disadvantage specific modules or developers.

15 CONCLUSION

This paper proposed an Explainable Deep Learning Framework for Predictive Software Quality Engineering and Automated Defect Detection (XDL-SQE). The framework combines source-code information, software metrics, historical defect data, code changes, test results, and runtime information to predict defect-prone software components. A key contribution is the integration of an explainability layer into the predictive quality-engineering workflow. Instead of

producing only a defect probability, the framework provides information about the factors contributing to a prediction using SHAP and LIME. This enables quality engineers and developers to investigate the reasoning associated with high-risk predictions. The framework also connects prediction with risk-based test prioritization and CI/CD workflows. Consequently, predictive analytics can become an active component of the software quality lifecycle rather than an isolated reporting mechanism. The proposed research does not assume that deep learning alone guarantees improved software quality. Instead, it establishes a framework through which predictive accuracy, defect detection capability, testing efficiency, and explanation quality can be evaluated systematically. Future empirical studies using public and industrial datasets will be necessary to quantify these benefits and determine the generalizability of the approach across programming languages, projects, architectures, and development environments. The combination of deep learning, explainable AI, predictive analytics, and continuous software testing provides a promising direction for transforming traditional reactive quality assurance into a more proactive and evidence-driven Software Quality Engineering practice.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Acknowledgments

The author acknowledges the developers and researchers whose prior work in software defect prediction, explainable AI, and deep learning for software engineering informed the development of this framework.

Disclosure of conflict of interest

The author declares no conflict of interest regarding this manuscript.

REFERENCES

- [1] A. A. A. Al-Saffar et al., "Current Software Defect Prediction: A Systematic Review," in *Proc. 2022 Applied Informatics International Conference (AiIC)*, Serdang, Malaysia, 2022, pp. 1–6, doi: 10.1109/AiIC54368.2022.9914586.
- [2] X. Chen, X. Xie, Y. Liu, Y. Wei, and X. Li, "A Survey on Deep Learning for Software Engineering," *ACM Computing Surveys*, vol. 54, no. 10, pp. 1–38, 2022, doi: 10.1145/3505243.
- [3] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in *Advances in Neural Information Processing Systems*, vol. 30, 2017, pp. 4765–4774.
- [4] "Enhancing Software Defect Prediction through Explainable AI: Integrating SHAP and LIME in a Voting Classifier Framework," in *Proc. 2024 8th International Artificial Intelligence and Data Processing Symposium (IDAP)*, Malatya, Türkiye, 2024, doi: 10.1109/IDAP64064.2024.10710700.
- [5] "A Systematic Study on Machine Learning Techniques for Predicting Software Faults," in *Proc. 2021 IEEE Mysore Sub Section International Conference (MysuruCon)*, Mysuru, India, 2021, doi: 10.1109/MysuruCon52639.2021.9641090.
- [6] M. A. M. Aljameel et al., "Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review," *IEEE Access*, vol. 9, pp. 98754–98771, 2021, doi: 10.1109/ACCESS.2021.3095559.
- [7] "Application of Deep Learning in Software Defect Prediction: Systematic Literature Review and Meta-analysis," *Information and Software Technology*, vol. 158, art. no. 107175, 2023, doi: 10.1016/j.infsof.2023.107175.
- [8] "Deep Learning-Based Cross-Project Defect Prediction: A Comprehensive Survey," in *Proc. 2025 25th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, Hangzhou, China, 2025, doi: 10.1109/QRS-C65679.2025.00036.
- [9] "A Qualitative Study on XAI Techniques for Software Defect Prediction," *Information and Software Technology*, vol. 198, art. no. 108234, 2026, doi: 10.1016/j.infsof.2026.108234.
- [10] ISO/IEC, *ISO/IEC 25010:2023 — Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — Product Quality Model*, International Organization for Standardization, 2023.
- [11] S. Omri and C. Sinz, "Deep Learning for Software Defect Prediction: A Survey," in *Proc. 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW)*, 2020, doi: 10.1145/3387940.3391463.
- [12] "A Survey on Software Defect Prediction Using Deep Learning," *Mathematics*, vol. 9, no. 11, art. no. 1180, 2021, doi: 10.3390/math9111180.
- [13] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.
- [14] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *Proc. International Conference on Learning Representations (ICLR)*, 2021.
- [15] "A Benchmark Study on Software Defect Prediction," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1234–1250, 2023.