Software Failure Prediction and Efficiency Optimization Using AI/ML Techniques

Prof. Prasad Mathapati[1], Dr. S. G. Gollagi[2], Prof. Zebashireen Fahim Shaikh[2]

[1]Assistant Professor, Department of Computer Science & Engineering, KLS Gogte Institute of Technology, Belagavi

[2]Assistant Professor, SGBIT, Belagavi

DOI: https://doi.org/10.51244/IJRSI.2026.1307000335

Received: 05 August 2026; Accepted: 10 August 2026; Published: 18 August 2026

ABSTRACT

Software reliability remains a major concern in modern software engineering due to the increasing complexity of software systems and the rapid pace of development. Software Failure Prediction (SFP) aims to identify fault-prone modules before deployment, enabling organizations to reduce maintenance costs and improve system quality. Artificial Intelligence (AI) and Machine Learning (ML) techniques provide data-driven approaches for analyzing software metrics, defect repositories, and execution logs to predict failures. This review examines AI/ML-based approaches for software failure prediction and efficiency optimization, with explicit attention to benchmark datasets, dataset quality, preprocessing, class imbalance, feature selection, model families, validation strategies, and evaluation measures. Representative NASA and PROMISE/Jureczko datasets are characterized in terms of software-unit type, metric families, binary defect labels, and imbalance. The review emphasizes Precision, Recall, F1-score, ROC-AUC, and MCC in addition to accuracy and compares traditional ML, ensemble, and deep-learning approaches. It further discusses missing data, concept drift, and cross-project prediction as key factors affecting real-world generalization. The analysis indicates that no single model is universally optimal; robust software failure prediction requires dataset-aware preprocessing, leakage-safe validation, imbalance-aware evaluation, and an explicit trade-off among predictive performance, computational efficiency, and interpretability.

INTRODUCTION

Software reliability is a critical quality attribute, particularly in safety-critical, enterprise-scale, and cloud-based applications. Traditional testing approaches are often insufficient for identifying defects in increasingly complex software environments. As software projects generate large volumes of historical data, AI and ML techniques have become valuable tools for extracting meaningful patterns associated with software failures.

The primary objective of software failure prediction is to identify components that are likely to contain defects before deployment. Early detection enables developers to prioritize testing efforts, allocate resources effectively, and reduce project costs. This paper provides a comprehensive discussion of AI/ML techniques used for software failure prediction and their contribution to efficiency optimization.

Background and Related Work

Early software reliability studies relied on statistical and reliability growth models. With the availability of software repositories and improved computational capabilities, machine learning algorithms such as Logistic Regression, Decision Trees, Support Vector Machines, and Random Forests became widely adopted.

Recent research demonstrates that ensemble methods often outperform single classifiers because they reduce variance and improve generalization. Deep learning approaches, including Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, have also shown promising results for large and

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VII July 2026

complex datasets. However, their computational requirements and limited interpretability remain important concerns.

Research Gap Analysis

Although significant progress has been made in software failure prediction using Artificial Intelligence (AI) and Machine Learning (ML) techniques, several research challenges remain unresolved. Existing studies have primarily focused on improving prediction accuracy, while relatively less attention has been given to computational efficiency, model interpretability, and practical deployment in industrial environments. Furthermore, the rapid evolution of deep learning and explainable AI technologies has created new opportunities that are not comprehensively addressed in earlier surveys.

Table 1. Research Gap Analysis in Software Failure Prediction Studies

Authors	Technique Used	Major Findings	Research Gap
Fenton and Neil	Statistical Defect Prediction Models	Identified limitations of traditional prediction methods	Limited applicability to modern large-scale software systems
Menzies et al.	Data Mining and Static Code Metrics	Demonstrated effectiveness of code metrics	Lack of deep learning-based analysis
Lessmann et al.	Classification Model Benchmarking	Random Forest showed superior performance	Focused mainly on accuracy comparison
Catal and Diri	Systematic Literature Review	Summarized existing fault prediction approaches	Limited coverage of recent AI methods
Nam et al.	Cross-Project Prediction	Improved knowledge transfer across projects	Generalization issues remain unresolved
Tian et al.	Deep Learning Models	Improved prediction performance	High computational complexity and low interpretability
Recent DevOps Studies	Continuous Prediction Models	Support for CI/CD environments	Limited industrial validation
Explainable AI Studies	XAI Techniques	Enhanced model transparency	Limited adoption in software engineering
Federated Learning Research	Privacy-Preserving Learning	Improved data privacy	Very limited application in defect prediction
Ensemble Learning Studies	Hybrid AI Models	Improved accuracy and robustness	Increased model complexity and resource requirements

The analysis of existing literature reveals several important research gaps:

• Most studies prioritize prediction accuracy while overlooking computational efficiency and resource optimization.

• Limited research has been conducted on Explainable AI (XAI) techniques for software failure prediction.

• Cross-project prediction remains a challenging problem due to dataset heterogeneity and domain differences.

• Existing deep learning models often suffer from poor interpretability, reducing industrial adoption.

• Real-time software failure prediction integrated with DevOps pipelines is still an emerging area.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VII July 2026

• Standardized datasets and evaluation frameworks are lacking, making fair comparison difficult.

• Privacy-preserving approaches such as federated learning have received limited attention in software reliability prediction.

This survey addresses these gaps by providing a comprehensive review of traditional machine learning, ensemble learning, deep learning, optimization techniques, explainable AI, transfer learning, and federated learning approaches for software failure prediction and efficiency optimization.

Problem Statement

Despite significant advancements in software testing and quality assurance, software failures continue to cause service interruptions, financial losses, and reduced user satisfaction. Manual defect inspection is time-consuming and difficult to scale. Therefore, there is a need for automated, accurate, and scalable prediction models capable of identifying fault-prone software modules while simultaneously improving development efficiency and resource utilization.

METHODOLOGY

This paper adopts a structured review methodology to synthesize evidence from published software defect/failure prediction studies. The review focuses on the data sources, preprocessing procedures, model families, optimization strategies, evaluation measures, and deployment constraints reported in the literature. Because the present manuscript is a review rather than a new experimental study, it does not claim original experimental results or a newly collected industrial dataset. Instead, representative benchmark datasets reported in prior empirical studies are explicitly characterized so that the evidence base and limitations of reported results are transparent. Common benchmark families include the NASA Metrics Data Program (MDP) datasets and PROMISE/Jureczko repositories. These datasets generally represent software modules or classes with software metrics as input attributes and a binary defect-proneness label as the prediction target. The literature also contains cross-project datasets that combine projects with different domains and metric distributions.

Dataset Characteristics and Failure Types

The benchmark datasets used in the studies reviewed in this paper mainly support binary software defect prediction, where a software module is classified as defective/fault-prone or non-defective. Published empirical work reports NASA datasets such as CM1, JM1, KC1, KC2, MC1, MC2, MW1, and PC1, while PROMISE/Jureczko datasets include projects such as Ant, Camel, JEdit, Log4j, Lucene, Poi, Synapse, Tomcat, Velocity, Xalan, and Xerces. Predictor variables typically include size, complexity, Halstead, control-flow, and related static-code metrics. Dataset characteristics vary substantially in project size, number of metrics, programming language, defect prevalence, and release history; consequently, results obtained on one dataset should not automatically be generalized to another project.

Illustrative dataset evidence from the literature shows the effect of imbalance. One reported NASA-dataset study described CM1 with 498 instances and 49 defective instances (about 9%), JM1 with 10,885 instances and 2,106 defective instances (about 19.34%), and KC1 with 2,109 instances and 326 defective instances (about 15.4%). Such distributions demonstrate why accuracy alone can be misleading: a model can obtain high accuracy while failing to identify a substantial proportion of minority defective modules. The literature also cautions that different cleaned and original versions of NASA datasets can produce different findings, making dataset provenance and preprocessing details essential for reproducibility.

Preprocessing, Feature Selection, and Class Imbalance

Preprocessing should be performed without allowing information from the test set to leak into the training process. The recommended sequence is: (1) remove duplicate or inconsistent records, (2) inspect missing values and either impute or remove records using a documented rule, (3) remove constant or near-constant attributes where appropriate, (4) normalize or standardize features for scale-sensitive algorithms, (5) perform

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VII July 2026

feature selection within the training folds, and (6) apply class-rebalancing methods such as SMOTE only to the training data. Feature selection can reduce redundant metrics, computational cost, and overfitting. For tree-based ensembles, scaling is usually less critical, whereas Logistic Regression, SVM, and neural networks can be more sensitive to feature scale. Class imbalance should be treated as an experimental factor rather than as a preprocessing step whose benefit is assumed in advance. Prior empirical evidence shows that the effect of SMOTE and under-sampling depends on the classifier and the evaluation measure.

Evaluation Measures for Imbalanced Software Failure Prediction

The revised evaluation framework does not rely on accuracy alone. Accuracy is defined as (TP+TN)/(TP+TN+FP+FN), but it can be overly optimistic when defective modules form a small minority. Therefore, the reviewed studies should be interpreted using multiple complementary measures. Precision = TP/(TP+FP) indicates how many modules predicted as defective are actually defective. Recall = TP/(TP+FN) measures the ability to detect defective modules and is particularly important when missed defects are costly. F1-score is the harmonic mean of precision and recall. ROC-AUC summarizes discrimination across classification thresholds, while the Precision-Recall curve/AUC can provide additional insight when the positive class is rare. Matthews Correlation Coefficient (MCC) is also useful because it considers all four entries of the confusion matrix. Reporting the confusion matrix, class distribution, and at least Precision, Recall, F1, ROC-AUC, and MCC provides a more meaningful assessment than accuracy alone. The choice of the primary metric should reflect the operational objective; for example, high recall may be preferred when failing to identify a defective module has a high cost.

Comparative Analysis of Machine-Learning, Ensemble, and Deep-Learning Approaches

Traditional machine-learning methods such as Logistic Regression, Decision Trees, and SVM provide useful baselines because they are comparatively interpretable and computationally manageable. Random Forest and Gradient Boosting/other ensemble methods can capture nonlinear relationships and interactions among software metrics and often provide strong performance on tabular defect data. Deep-learning methods such as Artificial Neural Networks, CNNs, and LSTMs can learn more complex representations, but their advantages are more dependent on dataset size, representation, and experimental design. They also introduce higher computational cost and reduced interpretability. Accordingly, a fair comparison should use the same data partitions, preprocessing rules, class imbalance strategy, validation protocol, and evaluation metrics across model families rather than comparing isolated accuracy values reported under different experimental conditions.

Table 2. Comparative Characteristics of Prediction Approaches

Approach	Strengths	Limitations	Typical Data Need	Interpretability
Logistic Regression	Simple, fast, strong baseline	Limited nonlinear modeling	Small to medium tabular datasets	High
Decision Tree	Captures nonlinear rules; easy to visualize	Can overfit without pruning	Small to medium tabular datasets	High
SVM	Effective in high-dimensional spaces	Parameter/kernel selection; scaling needed	Small to medium datasets	Medium
Random Forest	Robust nonlinear ensemble	Higher computational cost than a single tree	Medium tabular datasets	Medium
Gradient Boosting	Strong performance on structured data	Sensitive to tuning; can overfit	Medium tabular datasets	Medium
ANN/CNN/LSTM	Learns complex representations	Higher data, compute, and tuning requirements	Larger or structured datasets	Low

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VII July 2026

Cross-project prediction deserves separate treatment because models trained on one project can face different metric distributions, coding practices, languages, and defect mechanisms in another project. Transfer learning, feature alignment, domain adaptation, and careful source-project selection can reduce this distribution shift, but reported performance should always distinguish within-project prediction from cross-project prediction. Concept drift is another practical issue: as software evolves, coding practices, architecture, dependencies, and defect patterns change. Time-aware validation and periodic model monitoring are therefore preferable to assuming that a model trained on historical releases remains valid indefinitely.

Empirical Evidence and Methodological Limitations. The evidence synthesized in this survey is empirical in the sense that it is drawn from published experiments on public and industrial software-defect datasets; however, the present paper does not claim to have independently reproduced those experiments. This distinction is important when interpreting statements such as 'Random Forest performs better' because performance depends on the dataset, validation strategy, class distribution, feature set, and parameter configuration. Published work has shown that automated parameter optimization can materially change defect-prediction results, and studies of class rebalancing have shown that conclusions can vary with the chosen metric and classifier. Therefore, the revised paper reports comparative findings as literature-level evidence rather than as universally valid rankings.

RESULTS AND DISCUSSION

Comparative analysis from the reviewed literature indicates that Random Forest and Gradient Boosting frequently provide strong performance on structured software-metric data because they can capture nonlinear feature interactions and reduce sensitivity to individual noisy predictors. Logistic Regression remains valuable as an interpretable baseline, while SVM can be effective for high-dimensional feature spaces when scaling and hyperparameters are carefully controlled. Deep-learning models can be competitive when sufficient data and appropriate representations are available, but their computational cost, training time, and reduced interpretability can limit adoption. These findings should not be interpreted as a universal ranking: the literature shows that dataset characteristics, validation strategy, parameter optimization, class imbalance treatment, and selected evaluation metrics can substantially alter model comparisons.

For imbalanced datasets, the preferred reporting set is Accuracy, Precision, Recall, F1-score, ROC-AUC, and preferably MCC together with the confusion matrix and class distribution. Recall should be emphasized when the practical objective is to minimize missed defective modules, while Precision is important when inspection resources are limited and false alarms are costly. ROC-AUC should be interpreted together with threshold-dependent measures rather than used as the sole indicator of practical usefulness.

Applications

Applications include safety-critical systems, healthcare software, aviation systems, enterprise applications, cloud computing platforms, distributed systems, and DevOps environments where proactive defect detection can improve operational reliability.

Challenges and Limitations

Key challenges include data quality issues, missing and inconsistent values, duplicate records, class imbalance, feature redundancy, model interpretability, dataset heterogeneity, concept drift, and limited cross-project generalization. Additional threats include data leakage during feature selection or resampling, differences between cleaned and original benchmark datasets, and inconsistent validation protocols across studies. Organizations must also address privacy, governance, and ethical considerations when using software repository data for predictive analytics. These issues mean that a high accuracy or AUC value should not be interpreted in isolation.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue VII July 2026

CONCLUSION

Finally, the experimental results should be statistically analyzed using appropriate statistical significance tests and effect-size measures. This will enable more reliable comparisons among competing AI/ML approaches and help identify models that provide the best balance between prediction performance, computational efficiency, interpretability, and generalization. Such a systematic evaluation can contribute toward developing reliable and practically deployable AI-driven software failure prediction systems.

Future research should also investigate concept drift, since software characteristics, development practices, dependencies, and defect patterns may change across project versions. Techniques such as incremental learning, transfer learning, domain adaptation, and drift detection can be explored to maintain prediction performance over time.

Another important direction is cross-project and cross-version validation. In cross-project experiments, models trained using data from one or more projects should be evaluated on previously unseen projects to determine their generalization capability. Similarly, cross-version validation should evaluate whether models trained on earlier software releases remain effective for subsequent versions. These experiments can provide a more realistic assessment of model robustness than random train-test splitting within a single dataset.

Feature-selection and missing-data handling strategies should also be systematically evaluated to determine their effect on predictive performance and computational efficiency. Hyperparameter optimization can be applied using consistent procedures across the compared models to reduce experimental bias and improve the reliability of the comparison.

Particular attention should be given to the highly imbalanced nature of software defect datasets, where fault-prone modules generally constitute a smaller proportion of the total software modules. Future experiments should therefore investigate cost-sensitive learning, SMOTE and other appropriate resampling techniques, class-weighted algorithms, and anomaly-detection approaches. Performance should be evaluated using Precision, Recall, F1-score, ROC-AUC, Precision-Recall AUC, Matthews Correlation Coefficient (MCC), and confusion matrices rather than relying on accuracy alone.

A comprehensive experimental framework can be developed to evaluate traditional machine-learning classifiers, ensemble-learning methods, deep-learning models, and hybrid approaches under consistent experimental conditions. Traditional classifiers such as Logistic Regression, Decision Trees, Support Vector Machines, and K-Nearest Neighbors can be compared with ensemble approaches including Random Forest, Gradient Boosting, and XGBoost. Deep-learning approaches such as Artificial Neural Networks, Convolutional Neural Networks, and Long Short-Term Memory networks can also be investigated. Hybrid models combining feature-selection techniques, ensemble learning, and deep-learning approaches may further improve prediction performance.

Future research will focus on conducting systematic experimental comparisons of AI/ML models for software failure and defect prediction using multiple publicly available software defect datasets. Datasets from repositories such as NASA PROMISE and other publicly available software engineering repositories can be considered to provide diversity in project size, software characteristics, programming languages, defect density, and software metrics.

AI and ML techniques provide useful data-driven support for identifying fault-prone software modules and prioritizing quality-assurance effort. The revised review shows that model performance cannot be judged by accuracy alone, particularly for imbalanced defect datasets. Dataset provenance, missing-data treatment, feature selection, class balancing, validation design, and cross-project generalization are equally important. Traditional ML, ensemble learning, and deep learning each offer different trade-offs between predictive performance, computational efficiency, and interpretability. Consequently, reliable software failure prediction requires a reproducible methodology and a multi-metric evaluation framework rather than a single universally preferred model.

REFERENCES

1. M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data Quality: Some Comments on the NASA Software Defect Data Sets," IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208-1215, 2013. doi: 10.1109/TSE.2013.11.
2. C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models," IEEE Transactions on Software Engineering, vol. 46, no. 11, pp. 1200-1219, 2020. doi: 10.1109/TSE.2018.2876537.
3. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated Parameter Optimization of Classification Techniques for Defect Prediction Models," Proceedings of ICSE, 2016. doi: 10.1145/2884781.2884857.
4. M. M. Ozturk et al., "Which type of metrics are useful to deal with class imbalance in software defect prediction?" Information and Software Technology, vol. 92, pp. 17-29, 2017. doi: 10.1016/j.infsof.2017.07.004.
5. N. E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 675-689, 1999. doi: 10.1109/32.815326.
6. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 485-496, 2008. doi: 10.1109/TSE.2008.35.
7. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An Empirical Comparison of Model Validation Techniques for Defect Prediction Models," IEEE Transactions on Software Engineering, vol. 43, no. 1, pp. 1-18, 2017. doi: 10.1109/TSE.2016.2584050.
8. F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models," IEEE Transactions on Software Engineering, vol. 43, no. 5, pp. 476-491, 2017. doi: 10.1109/TSE.2016.2599161.
9. J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models," IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 166-185, 2022. doi: 10.1109/TSE.2020.2982385.
10. M. Shepperd, D. Bowes, and T. Hall, "Researcher Bias: The Use of Machine Learning in Software Defect Prediction," IEEE Transactions on Software Engineering, vol. 40, no. 6, pp. 603-616, 2014. doi: 10.1109/TSE.2014.2322358.
11. B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models," Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, pp. 789-800. doi: 10.1109/ICSE.2015.91.
12. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated Parameter Optimization of Classification Techniques for Defect Prediction Models," Proceedings of ICSE, 2016. doi: 10.1145/2884781.2884857.
13. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276-1304, 2012.
14. C. Catal and B. Diri, "A Systematic Review of Software Fault Prediction Studies," Expert Systems with Applications, vol. 36, no. 4, pp. 7346-7354, 2009.
15. T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," IEEE Transactions on Software Engineering, vol. 33, no. 1, pp. 2-13, 2007.
16. D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software Fault Prediction Metrics: A Systematic Literature Review," Information and Software Technology, vol. 55, no. 8, pp. 1397-1418, 2013.
17. J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous Defect Prediction," IEEE Transactions on Software Engineering, vol. 44, no. 9, pp. 874-896, 2018.
18. Y. Kamei et al., "A Large-Scale Empirical Study of Just-In-Time Quality Assurance," IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 757-773, 2013.
19. T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process," Proceedings of ESEC/FSE, 2009.