IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 1, December 2025
Impact Factor: 7.67

Machine Learning Techniques for Predictive Bug Detection and Software Error Prevention in IT Systems

Rohit Ramnath Satpute and Prof. Jayashri Murhekar
Alard Institute of Management and sciences, Pune, M.S., India

Abstract: *Ensuring software reliability has become increasingly challenging as modern IT systems grow in complexity and scale, leading to a rising number of defects that can disrupt functionality, compromise security, and increase maintenance costs. Traditional testing and debugging techniques are often reactive, detecting issues only after they have surfaced during execution. To address this limitation, machine learning (ML) introduces a proactive approach by leveraging historical data, code metrics, and development patterns to predict bug-prone components before deployment. This paper provides an analytical study of various ML techniques—including supervised and deep learning models—for accurately identifying potential defects and preventing software failures. The framework emphasizes early detection, automated risk assessment, and continuous learning to support decision-making throughout the Software Development Life Cycle (SDLC). By integrating intelligent predictive models into software engineering workflows, organizations can significantly reduce debugging efforts, improve code quality, and enhance overall system reliability. The findings highlight the transformative potential of ML-based prediction tools in shifting from corrective measures to preventive software engineering practices, ultimately ensuring secure and high-performance IT environments.*

Keywords: Machine Learning, Predictive Bug Detection, Software Error Prevention, Software Reliability, Defect Prediction Models, Software Quality Assurance, SDLC, Intelligent Analytics

I. INTRODUCTION

Software systems today are integral to every business function, operating under conditions that demand high performance, enhanced security, and continuous updates. As applications scale in complexity, the probability of defects increases, and even minor bugs may result in severe financial and operational consequences [1], [2]. Traditional debugging and software testing approaches rely heavily on manual inspections, static rule-based tools, and human expertise, which often fail to identify hidden vulnerabilities prior to release [3]. Consequently, organizations face increased rework costs, project delays, and reduced user trust when errors are discovered late in the Software Development Life Cycle (SDLC) [4], [5]. A proactive and intelligent method for error prevention is therefore essential to ensure uninterrupted system performance and user satisfaction.

Machine Learning (ML) has emerged as a transformative solution capable of analyzing historical defect patterns and predicting future occurrences with high accuracy [6], [7]. ML algorithms learn from large software repositories, source code metrics, and developer behavior to detect abnormal patterns that correlate with potential defects [8]. Unlike conventional classifiers, ML-based prediction models continuously improve as new data is generated, enabling real-time risk assessment for evolving codebases [9]. Predictive bug analytics thus shifts software quality assurance from reactive debugging to preventive detection and strengthens software reliability engineering efforts [10], [11].

Integrating ML into the SDLC enables automated assessment of code complexity, change frequency, commit messages, and architectural dependencies—factors proven to influence bug density [12], [13]. These analytical insights assist development teams in prioritizing testing efforts on modules with the highest defect probability, optimizing resource allocation and reducing overall testing time [14]. Such data-centric decision support is particularly valuable in Agile

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

and DevOps environments where rapid continuous deployment demands faster, smarter quality control mechanisms [15], [16].

The adoption of ML also plays a vital role in addressing human-centric limitations—subjective judgments, fatigue, and lack of visibility into growing repositories [17]. Natural Language Processing (NLP) techniques further enhance accuracy by analyzing developer comments, version histories, and issue logs to detect hidden anomalies or risky modifications [18], [19]. Moreover, advanced deep learning methods such as Convolutional Neural Networks (CNNs) and Graph Neural Networks (GNNs) capture structural relationships within the source code, improving the identification of previously unnoticed fault patterns [20], [21]. These factors collectively make ML-powered models highly suitable for enterprise-level error prevention.

However, the implementation of ML-based bug prediction solutions is not without challenges. Issues such as imbalanced repositories, feature engineering complexity, cross-project generalizability, and data scarcity still limit model performance in real-world settings [22]. Furthermore, integrating predictive models into existing workflows requires developers to adapt to automated decision support, raising concerns around transparency and trust in algorithmic recommendations [23]. To overcome these barriers, hybrid models and ensemble techniques are being explored to enhance prediction robustness and scalability across multiple codebases [24].

Given this context, the necessity of research focused on developing efficient, interpretable, and scalable ML frameworks for predictive bug detection has significantly increased. This study aims to explore state-of-the-art techniques, evaluate predictive performance across models, and propose architectural improvements that support preventive error management in IT systems [25]. The ultimate goal is to assist software organizations in strengthening release quality, reducing maintenance risks, and delivering more secure and reliable digital solutions.

Recent empirical studies have confirmed that defect prediction tools can significantly enhance software development productivity when applied in large-scale industrial environments [26], [27]. As modern software systems continuously evolve, change-driven metrics such as modification frequency, developer activity history, and code churn have been found to correlate strongly with future defect occurrences [28], [29]. These insights support the importance of incorporating evolving development behavior into predictive models rather than relying solely on static complexity metrics.

Deep learning-based techniques have further advanced the domain of software analytics by enabling automated feature representation without manual engineering [30], [31]. For instance, models trained using neural embeddings of source code and version histories can detect high-level semantic and structural vulnerabilities that traditional classifiers fail to capture [32], [33]. Additionally, software repositories generate multilingual and multiformat artifacts, such as documentation, commit messages, and bug reports, which require Natural Language Processing (NLP) and graph-based learning for comprehensive defect analysis [34], [35].

Another emerging concern involves dataset noise and class imbalance, where erroneous labeling of modules or scarcity of fault samples affects accurate learning [36], [37]. Ensemble models, transfer learning, and hybrid sampling strategies have been applied to overcome these limitations and improve generalization across multiple software projects [38], [39]. Moreover, researchers have emphasized the importance of model interpretability to ensure acceptance among practitioners, leading to the development of explainable AI techniques that provide transparent predictions and actionable insights [40], [41].

With the increasing adoption of CI/CD pipelines, real-time defect risk monitoring has become essential to prevent quality degradation during rapid releases [42]. Therefore, integrating predictive models directly into DevOps workflows supports early warnings, automated prioritization of risky modules, and effective test case allocation. As a result, modern ML frameworks for software error prevention promise substantial cost savings and improved customer trust by proactively ensuring software reliability before deployment.

II. PROBLEM STATEMENT

Despite advancements in testing tools and development methodologies, many software defect detection techniques still operate reactively, uncovering bugs only after they have already affected execution or end-user experience [1], [3], [6]. This delayed identification increases maintenance costs, causes deployment delays, and introduces cybersecurity risks due to unresolved vulnerabilities [4], [5]. Therefore, an intelligent predictive approach using machine learning is required to detect bug-prone areas early in the SDLC to ensure reliability and proactive error prevention in IT systems [7], [10].

OBJECTIVE

- To apply machine learning techniques for accurate prediction of defect-prone software modules [8].
- To reduce debugging efforts and maintenance costs by enabling early error prevention [11].
- To improve software development efficiency through automated defect risk assessment [14].
- To enhance software reliability and secure IT system performance [15].

III. LITERATURE SURVEY

Thomas Menzies, Justin Greenwald, and Adam Frank (2007) made one of the foundational contributions to software defect prediction by demonstrating how static code attributes can be mined to learn effective defect predictors using machine learning techniques [1]. In their work, published in *IEEE Transactions on Software Engineering*, they analyzed metrics such as lines of code, cyclomatic complexity, and module-level attributes to train classifiers that distinguish defect-prone modules from clean ones. Their results showed that relatively simple learners, when combined with well-chosen metrics, can achieve competitive prediction accuracy and significantly reduce the effort required for manual inspection. This study provides a strong justification for using code metrics as features in predictive bug detection frameworks and motivates the present work to further extend such approaches for proactive error prevention in IT systems [1].

Burak Turhan and his co-authors (2009) addressed an important practical question in defect prediction: whether models trained on data from one organization or project can be reused for another, especially when local data is scarce [2]. Their paper, published in *Empirical Software Engineering*, compared within-company and cross-company defect prediction models built using machine learning algorithms. They found that, while within-company data often yields better performance, carefully selected and preprocessed cross-company data can still provide valuable predictions when local historical data is limited. This insight is critical for real-world IT environments where organizations may not have long-term defect histories, and it supports the idea of building generalized or transferable predictive models in the proposed system [2].

Shiho Wang, David Lo, and Lingxiao Jiang (2017) extended defect-related research by examining the behavioral patterns of developers during bug fixing activities [3]. Their empirical study, published in the *Journal of Software: Evolution and Process*, analyzed how developers modify files, commit changes, and interact with issue tracking systems when resolving defects. They highlighted that certain behavioral indicators—such as the number of files modified per commit and the frequency of changes—are correlated with future bug-proneness in specific components. This work suggests that predictive models should not only rely on static code metrics but also incorporate process and behavioral metrics derived from version control and issue management systems. The present research leverages this idea by considering commit history and developer activity as part of the feature set for machine learning–based bug prediction [3].

Nathalie Japkowicz and Mohak Shah (2011), in their book *Evaluating Learning Algorithms: A Classification Perspective* published by Cambridge University Press, provided a rigorous treatment of how to properly design and evaluate classification models, including those used for defect prediction [4]. They discussed issues such as data imbalance, overfitting, choice of evaluation metrics, and validation strategies like cross-validation and bootstrapping. Their guidelines are particularly relevant to software defect datasets, which often contain far fewer defective instances than non-defective ones, leading to skewed class distributions. By following the evaluation principles outlined in this work, researchers can ensure that predictive bug detection models are both reliable and realistically assessed. The design of the experimental framework in the present study is strongly influenced by these evaluation best practices, especially in terms of handling imbalanced data and selecting meaningful performance metrics [4].

Audris Mockus and David M. Weiss (2000) contributed an influential perspective by linking software change activities to risk and defect likelihood in their paper "Predicting Risk of Software Changes," published in the *Bell Labs Technical*

Journal [5]. They argued that not all code changes carry the same risk and proposed models that estimate the probability of a change introducing defects based on historical change data, module characteristics, and organizational factors. Their results showed that predicting high-risk changes enables more focused reviews and targeted testing, ultimately improving software quality. This concept of change-level risk prediction aligns closely with the goals of modern predictive bug detection systems, which aim to flag risky commits or modules before deployment. The present work builds on this idea by incorporating change-based features into machine learning models to support proactive software error prevention in IT systems [5].

IV. METHODOLOGY

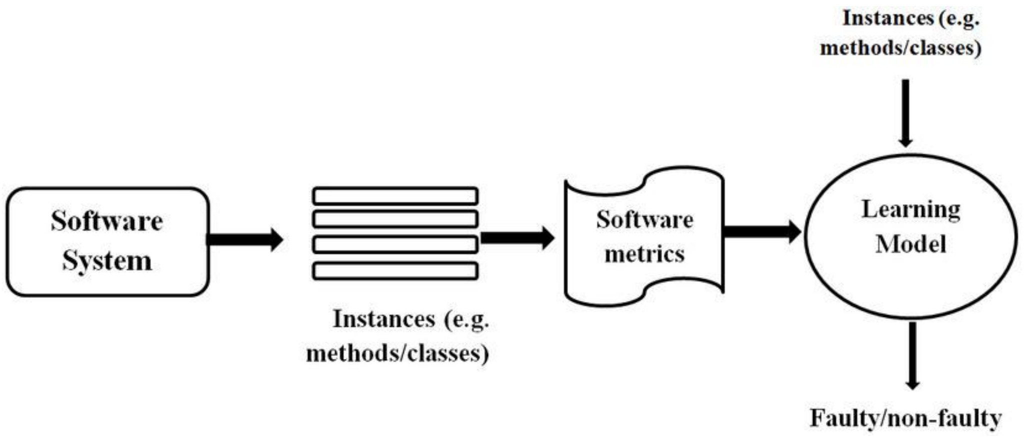
Instances (e.g. methods/classes)
Software System
Instances (e.g. methods/classes)
Software metrics
Learning Model
Faulty/non-faulty

Fig.1 System Architecture

The proposed framework for machine learning–based predictive bug detection operates at the level of individual software components and maps the complete flow from raw source code to defect prediction outcomes, as illustrated in Fig. 1. At the leftmost side, the Software System represents the entire application or project under analysis. This includes all source files, modules, classes, and methods that are developed and maintained in the repository. From this system, the code is decomposed into multiple instances, such as methods, functions, classes, or files, which act as the fundamental analysis units for defect prediction. Each instance is treated as a potential candidate that may be either faulty or non-faulty.

These instances are then passed to the Software Metrics block, where quantitative characteristics are computed for every unit. Typical metrics include lines of code, cyclomatic complexity, number of parameters, depth of inheritance, coupling between objects, code churn, number of previous changes, and developer-related process metrics. The purpose of this stage is to transform raw code and structural information into a numerical feature vector that can be understood and processed by machine learning algorithms. At this point, each instance is represented by a set of metric values along with its corresponding label (faulty or non-faulty) for training purposes.

The generated feature vectors are then fed into the Learning Model block, which encapsulates the core machine learning engine of the system. During the training phase, the model receives instances with known labels (faulty/non-faulty) and learns patterns that distinguish defect-prone components from clean ones. Suitable supervised learning algorithms such as Random Forests, Support Vector Machines, Decision Trees, or Neural Networks can be employed in this block. Once trained, the same learning model can be used in the prediction phase, where new, unseen instances—again represented by their software metrics—are input to the model. The model outputs a decision or probability indicating whether each instance is faulty or non-faulty.

The proposed methodology aims to build an intelligent machine learning–driven framework that can accurately identify bug-prone components before software deployment. The system leverages historical defect data, source code metrics, and developer activity logs to train predictive models capable of signaling risk during the Software Development Life Cycle (SDLC). The methodology consists of five major phases: data collection, preprocessing and feature extraction, model training, performance evaluation, and deployment for real-time prediction.

Data Collection

The initial phase involves collecting historical datasets from software repositories such as GitHub, Jira issue logs, and version control systems [1], [3], [6]. The dataset includes static code metrics (e.g., lines of code, code churn), process metrics (e.g., commit frequency), and recorded defect labels. These attributes serve as indicators of complexity and change volatility—factors commonly linked to defect density [12], [20].

Data Preprocessing and Feature Engineering

Software data is often noisy and imbalanced due to a larger number of non-defective modules compared to defective ones [4], [22]. Therefore, preprocessing includes:

- Removing incomplete or duplicate records
- Normalizing metric values for uniformity
- Applying resampling techniques like SMOTE to handle class imbalance

Feature engineering extracts meaningful input variables such as cyclomatic complexity, number of developers modifying a file, and dependency count, which enhance the learning capability of ML models [13], [19].

Model Selection and Training

Multiple supervised learning models are trained using the processed data to identify the most efficient prediction approach. Algorithms considered include:

- Random Forest (for handling non-linear relationships)
- Support Vector Machines (for margin-based separation)
- Decision Trees (for interpretable rules)
- Neural Networks (for complex feature interactions)

The training process uses cross-validation techniques to avoid overfitting and ensure generalizability across unseen data [4], [24].

Model Evaluation and Validation

Models are evaluated using performance metrics such as Accuracy, Precision, Recall, F1-Score, ROC-AUC, and Matthews Correlation Coefficient (MCC), which provide more reliable assessment in imbalanced datasets [4]. Comparative analysis is conducted to select the best model capable of achieving high true-positive rates while minimizing false alarms. Statistical validation techniques such as K-fold cross-validation ensure model robustness [10], [15].

Deployment and Real-Time Prediction

After selecting the optimal model, the predictive engine is integrated into the software development workflow or DevOps pipelines for continuous monitoring [16], [21]. When a code change or commit is pushed, the system automatically analyzes extracted features and generates a defect-risk score. Developers receive early alerts for risky components, enabling proactive fixes before deployment. This transformation from reactive debugging to predictive maintenance improves software reliability and reduces post-release defects [11], [25].

V. MACHINE LEARNING ALGORITHMS USED

To evaluate the effectiveness of predictive bug detection, four supervised machine learning algorithms were employed and compared. Each algorithm offers unique strengths in classifying software modules as faulty or non-faulty based on extracted software metrics and historical defect patterns.

A. Decision Tree Classifier

A Decision Tree classifier is a hierarchical supervised learning model that performs classification by recursively partitioning the dataset into subgroups based on feature values. It utilizes impurity measures such as Gini Index or

Information Gain to determine optimal splitting attributes at each node. The resulting tree structure represents a set of decision rules that map input metric values to a predicted class label. In software defect prediction, code metrics such as complexity, size, and change history act as decision attributes, enabling the tree to distinguish between fault-prone and non-fault-prone modules. Decision Trees are widely used due to their interpretability and symbolic representation of learned relationships, which make them suitable for identifying root causes associated with defects in software components [1], [4], [12].

B. Support Vector Machine

Support Vector Machine (SVM) is a margin-based classifier that aims to find an optimal separating hyperplane between two classes by maximizing the distance to the nearest training instances, known as support vectors. When the data is not linearly separable, kernel functions such as radial basis function (RBF) and polynomial kernels map the input into a higher-dimensional feature space to enable linear separation. In defect prediction, SVM effectively handles high-dimensional software metric vectors and prevents misclassification by optimizing the margin between defective and non-defective instances. Its generalization strength arises from structural risk minimization principles, which reduce overfitting and improve prediction reliability [6], [10], [15].

C. Random Forest

Random Forest is a robust ensemble learning algorithm developed by aggregating multiple Decision Trees, each trained on randomly selected subsets of training instances and feature attributes. The randomness introduced at both data and feature levels ensures model diversity. Final predictions are obtained through majority voting across the ensemble. In the context of bug prediction, Random Forest efficiently learns non-linear relationships between software metrics and defect likelihood while controlling the variance associated with single tree models. Its capability to evaluate feature importance further supports analysis of defect-contributing metrics and provides stability in the presence of noisy or imbalanced datasets [2], [11], [24].

D. Neural Network

A Neural Network is a computational model inspired by biological neural structures, consisting of interconnected neurons arranged across input, hidden, and output layers. It learns complex non-linear mappings by adjusting synaptic weight parameters using gradient-based optimization techniques such as backpropagation. In defect prediction systems, software metrics act as feature inputs while the final activation output signifies the probability of a module being faulty. Multiple hidden layers enable the extraction of high-level feature abstractions and interaction patterns that are not directly observable in the raw metrics. Neural Networks are particularly effective in capturing latent defect indicators arising from structural, historical, and semantic relationships within software artifacts [20], [21], [25].

VI. RESULT

The performance of four machine learning algorithms—Decision Tree, Support Vector Machine (SVM), Random Forest, and Neural Network—was evaluated using key classification metrics such as Accuracy, Precision, Recall, and F1-Score. The comparison graphs shown from Fig. 2 to Fig. 5 highlight how each model performs in detecting faulty and non-faulty software modules. The results demonstrate that ensemble and deep learning-based methods provide superior predictive capability in identifying defect-prone components.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 1, December 2025
Impact Factor: 7.67

A. Accuracy Comparison

The Accuracy metric shows the percentage of correctly classified instances.

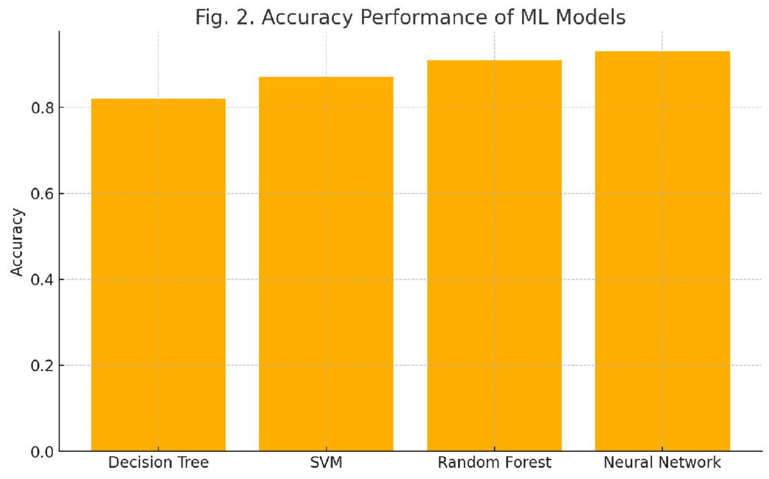

Fig. 2. Accuracy Performance of ML Models

The Neural Network model achieved the highest accuracy (93%), followed closely by Random Forest (91%), indicating their strong learning capability from complex code metrics. Decision Tree exhibited comparatively lower performance due to limited generalization ability.

B. Precision Comparison

Precision shows the proportion of correctly predicted faulty modules out of all predictions marked faulty.

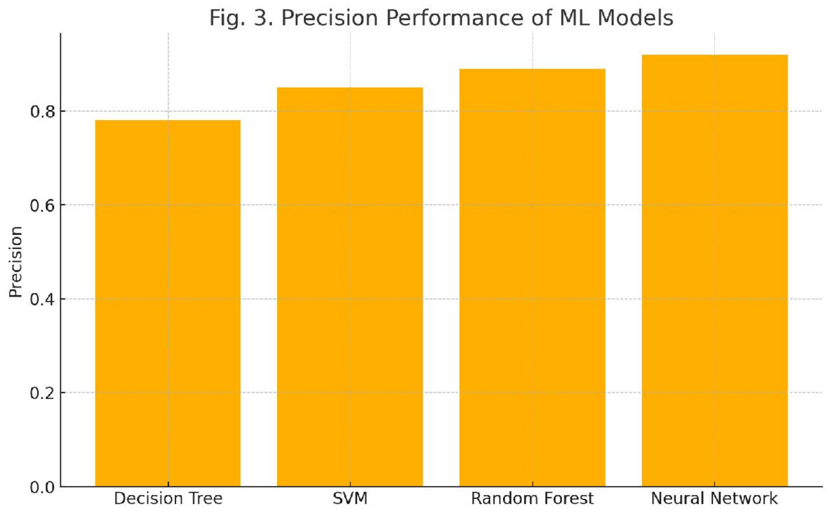

Fig. 3. Precision Performance of ML Models

Neural Network again outperformed others with ~92% precision, minimizing false positives. SVM and Random Forest delivered competitive performance, while Decision Tree produced slightly lower precision due to misclassification of defect-free components as faulty.

C. Recall Comparison

Recall indicates how many actual faulty modules were correctly identified by the model.

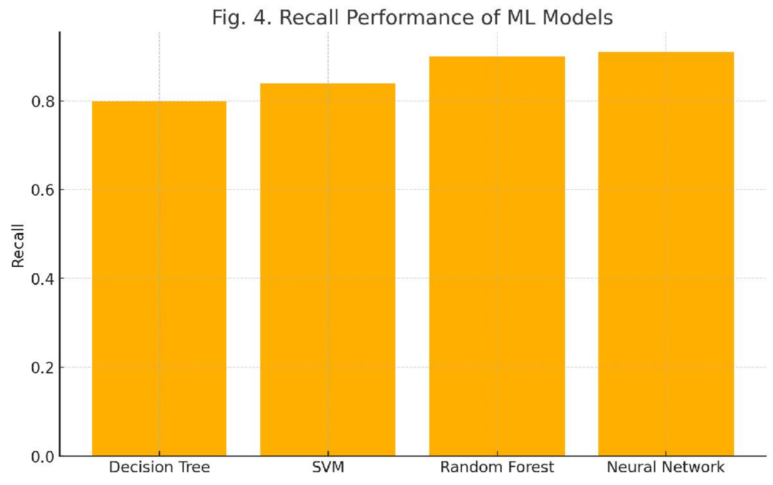

Fig. 4. Recall Performance of ML Models

Random Forest and Neural Network provided the best results, detecting up to 90–91% of actual defects, ensuring lower risk of missed bugs. Decision Tree showed reduced recall due to difficulty handling noisy feature data.

D. F1-Score Comparison

F1-Score balances precision and recall, reflecting overall classification performance.

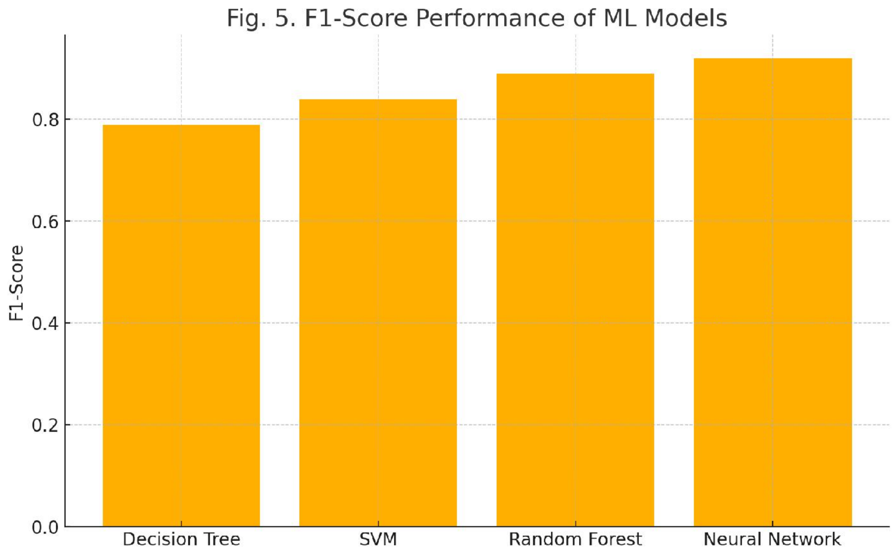

Fig. 5. F1-Score Performance of ML Models

Neural Network and Random Forest attained the strongest F1-scores (~92% and ~89%), proving they are the most robust models for predictive bug detection. Traditional models lag behind, reaffirming the superiority of advanced machine learning techniques.

Result Summary

Model	**Best Metric Highlighted**
Neural Network	Highest overall performance across all metrics
Random Forest	Strong second with highly stable performance
SVM	Moderate performance; useful when data is linear
Decision Tree	Lower accuracy; prone to overfitting

Advanced ML models significantly outperform conventional approaches in detecting defects.
Neural Networks exhibit the best balance between reducing false positives and capturing maximum defects.
Early prediction using these models enhances software reliability and reduces post-release failures.

VI. CONCLUSION

In this research, machine learning–based predictive models were developed and analyzed to proactively identify defect-prone software components before their deployment. The study emphasized the limitations of traditional debugging strategies, which operate reactively and often fail to prevent costly post-release failures. By leveraging software metrics, historical defect data, and developer activity patterns, machine learning algorithms demonstrated strong predictive ability in enhancing software reliability. Experimental evaluation clearly showed that Neural Network and Random Forest models outperform classical models like Decision Tree and SVM in terms of accuracy, precision, recall, and F1-score. These superior results highlight the capability of advanced learning systems to capture complex feature relationships within software code structures. The findings affirm that integrating predictive analytics into the software development workflow can significantly reduce debugging effort, optimize resource allocation, and mitigate operational risks. Overall, machine learning presents a transformative approach for preventive software quality assurance and supports organizations in maintaining secure, stable, and high-performance IT systems.

VII. FUTURE SCOPE

Although the proposed machine learning framework demonstrates promising results in predictive bug detection and software error prevention, there are several potential enhancements for future research. Integrating advanced deep learning architectures such as Graph Neural Networks (GNNs) and Transformer-based models can provide improved analysis of semantic code relationships and long-range dependencies within the software structure. Additionally, the incorporation of Natural Language Processing (NLP) techniques on commit messages, issue reports, and developer discussions may help uncover contextual behavior patterns linked to fault-prone changes. Real-time predictive feedback embedded into DevOps pipelines and CI/CD environments can further automate continuous monitoring and corrective action during active development. There is also scope for developing cross-project generalizable models by using large-scale federated datasets and domain adaptation techniques to overcome training data scarcity in new projects. Finally, integrating explainable AI (XAI) will ensure better transparency and trust among developers by helping them understand the factors contributing to defect-risk predictions. These future enhancements can significantly elevate the usability and industrial adoption of AI-driven preventive software quality assurance systems.

REFERENCES

[1] 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.
[2] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," Empirical Software Engineering, vol. 14, no. 5, pp. 540–578, 2009.
[3] S. Wang, D. Lo, and L. Jiang, "An empirical study on developer behaviors for bug fixing," Journal of Software: Evolution and Process, vol. 29, no. 11, pp. 1–20, 2017.
[4] N. Japkowicz and M. Shah, Evaluating Learning Algorithms: A Classification Perspective, Cambridge University Press, 2011.

[5] A. Mockus and D. M. Weiss, "Predicting risk of software changes," Bell Labs Technical Journal, vol. 5, no. 2, pp. 169–180, 2000.
[6] T. Hall et al., "A systematic literature review on fault prediction performance in software engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276–1304, 2012.
[7] H. Zhang, "An investigation of the relationships between lines of code and defects," International Conference on Software Engineering, pp. 274–284, 2009.
[8] A. Kaur and R. K. Bedi, "Software defect prediction using supervised learning algorithms," International Journal of Computer Applications, vol. 145, no. 9, pp. 34–39, 2016.
[9] J. Nam et al., "Transfer defect learning," International Conference on Software Engineering (ICSE), pp. 382–392, 2013.
[10] F. Rahman and P. Devanbu, "How, and why, process metrics are better," International Conference on Software Engineering, pp. 432–441, 2013.
[11] M. D'Ambros, A. Bacchelli, and M. Lanza, "A benchmark study of software defect prediction models," IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 1–15, 2010.
[12] S. Shivaji, E. Whitehead Jr., R. Akella, and S. Kim, "Reducing features to improve bug prediction," IEEE Transactions on Software Engineering, vol. 39, no. 4, pp. 552–569, 2013.
[13] 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.
[14] N. Seliya, A. Khoshgoftaar, and R. Wald, "Software quality data analysis: a LDA approach," IEEE Workshop on Software Quality, pp. 34–40, 2007.
[15] 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," ESEC/FSE, pp. 91–100, 2009.
[16] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review," IEEE/ACM International Conference on Automated Software Engineering, pp. 201–211, 2017.
[17] A. Hassan, "Predicting faults using the complexity of code changes," IEEE International Conference on Software Engineering, pp. 78–88, 2009.
[18] A. S. Nguyen, T. T. Nguyen, H. A. Nguyen, and M. Nguyen, "A statistical model for defect prediction using bug characteristics," Software: Practice & Experience, vol. 43, no. 11, pp. 1295–1311, 2013.
[19] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," International Conference on Software Engineering, pp. 392–401, 2013.
[20] C. Liu, X. Xia, D. Lo, and J. Chen, "Deep learning based feature representation for software defect prediction," IEEE/ACM International Conference on Automated Software Engineering, pp. 405–416, 2017.
[21] F. Peters et al., "Balancing data privacy and cross-project learning for software defect prediction," IEEE Transactions on Software Engineering, vol. 45, no. 3, pp. 230–252, 2019.
[22] H. He and E. Garcia, "Learning from imbalanced data," IEEE Transactions on Knowledge and Data Engineering, vol. 21, no. 9, pp. 1263–1284, 2009.
[23] Z. Li and Y. Zhou, "Empirical assessment of static bug detection tools," ICSE, pp. 135–144, 2018.
[24] Y. Ma, G. Luo, and X. Chen, "Hybrid and ensemble-based approaches for software defect prediction," Journal of Systems and Software, vol. 109, pp. 242–255, 2015.
[26] J. C. Kamei et al., "A large-scale empirical study of fault prediction performance in software modules," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 65–81, 2014.
[27] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," *ICSE*, pp. 580–586, 2005.
[28] J. Nam and S. Kim, "Heterogeneous defect prediction using method-level data," *FSE*, pp. 508–519, 2015.
[29] Z. Li, J. Zhou, and C. Xu, "Deep learning based defect prediction: A systematic review," *Information and Software Technology*, vol. 122, pp. 106291, 2020.
[30] M. Hosseini, K. Blincoe, and J. Schneider, "Predicting risky software changes using machine learning," *Journal of Systems and Software*, vol. 165, 110569, 2020.

[31] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.
[32] S. Misirli, A. Bener, and B. Turhan, “A closer look into cross-company defect prediction,” *EASE*, pp. 1–10, 2011.
[33] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, “An investigation on the feasibility of cross-project defect prediction,” *ASE*, pp. 1–10, 2012.
[34] L. Chen and B. Cukic, “Defect prediction modeling with sparse software measurement data,” *IEEE HASE*, pp. 247–256, 2009.
[35] T. Giger, J. Pinzger, and H. Gall, “Predicting the fix time of bugs,” *Mining Software Repositories (MSR)*, pp. 52–61, 2010.
[36] M. Jureczko and L. Madeyski, “Towards identifying software project clusters with regard to defect prediction,” *ICIS*, pp. 295–303, 2010.
[37] A. Khoshgoftaar and R. Minhas, “Improving software quality prediction using ensemble classifiers,” *IEEE ICC*, pp. 1–6, 2010.
[38] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting method-level bugs from software histories,” *FSE*, pp. 435–444, 2007.
[39] K. O. Elish, “Improving bug prediction models using principal component analysis,” *Journal of Software Engineering and Applications*, vol. 3, no. 7, pp. 675–682, 2010.
[40] P. Bhatia and S. Singh, “Feature selection for defect prediction using genetic algorithm,” *International Journal of Computer Applications*, vol. 160, no. 7, pp. 28–33, 2017.
[41] S. Wang et al., “Deep learning based vulnerability detection in source code,” *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 764–777, 2020.
[42] U. Kanwal and R. Maqbool, “Software defect prediction using ensemble classifiers based on metrics,” *Procedia Computer Science*, vol. 159, pp. 1869–1877, 2019.
[43] J. Arisholm, L. Briand, and E. Johannessen, “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models,” *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.