Journal of Engineering and Technology for Industrial Applications

ITEGAM-JETIA

Manaus, v.12 n.58, p. 460-474. March/April, 2026.

DOI: https://doi.org/10.5935/jetia.v12i58.3133

ISSN ONLINE: 2447-0228

RESEARCH ARTICLE OPEN ACCESS

SMART AI INTENSIVE SOFTWARE DEFECT PREDICTIONS BASED ON OPTIMIZED DEEP FEATURE CLASSIFICATION USING GWO-CODE2 VECTOR-CNN LEARNING

V. Ruckmani*[1], Rajan John[2], J. Jebamalar Tamilselvi[3] and, Surya Susan Thomas[4]

[1]Assistant professor, KMG College of Arts and Science (Autonomous), (Affiliated to Thiruvalluvar University) Gudiyatham, Vellore, Tamil Nadu, India.

[2]Assistant Professor, Department of Computer Science, College of Engineering & Computer Science, Jazan University, Jazan, Kingdom of Saudi Arabia.

[3]Associate Professor, Department of Computer Science in Cyber Security, Faculty of Science and Humanities, SRM Institute of Science and Technology, Ramapuram, Chennai, India.

[4]Assistant Professor, Department of Computer Science with Data Science, Patrician College of Arts and Science, Adyar, Chennai, India.

[1]https://orcid.org/0009-0002-0773-6902, [2] https://orcid.org/0000-0001-7375-0169
[3] https://orcid.org/0009-0004-1292-1293,[4] https://orcid.org/0000-0001-5602-6769

Email: *vruckmani136@gmail.com, rsubbaiah@jazanu.edu.sa, jebamalj@srmist.edu.in, suryasusan@patriciancollege.ac.in

ARTICLE INFO

Article History

Received: December 13, 2025
Reviewed: January 1, 2026
Accepted: January 15, 2026
Published: March 31, 2026

Keywords:
Software Defect Prediction
Deep Learning
Z-Score Logarithmic Transformation (ZSLT)
CSMOTE-ENN
GWO
DNN
Code2Vector
GCNN
Feature Optimization
Defect Classification.

ABSTRACT

Software defect prediction plays a crucial role in improving software reliability, reducing maintenance costs, and enhancing software quality. However, traditional prediction methods often fail to accurately detect software defects due to non-relational dependencies among features, redundant datasets, and imbalanced data distributions. These limitations result in lower true positive rates and reduced predictive accuracy. To overcome these challenges, this research introduces an optimized Deep Learning–based Defect Prediction Framework that integrates advanced preprocessing, feature optimization, and classification mechanisms. Initially, Z-Score Logarithmic Transformation (ZSLT) is employed for preprocessing to normalize feature scales and eliminate noise in defect logs. To address data imbalance, the Clustered Synthetic Minority Oversampling Technique–Edited Nearest Neighbour (CSMOTE-ENN) method is applied, effectively balancing the dataset while preserving meaningful feature diversity. Subsequently, Grey Wolf Optimization–Deep Neural Network (GWO-DNN) is utilized for optimal feature selection, enabling the identification of highly correlated defect-related attributes. Finally, the Code2Vector–Graph Convolutional Neural Network (Code2Vector-GCNN) model is employed for deep feature classification, capturing both semantic and structural code representations to improve defect detection accuracy. Experimental results demonstrate that the proposed framework significantly outperforms conventional machine learning and deep learning models in terms of precision, recall, and F1-measure, providing an intelligent, adaptive, and high-accuracy solution for software defect prediction.

I. INTRODUCTION

Software defect prediction (SDP) is a critical undertaking in software engineering, whose purpose is to determine the possible bugs and vulnerabilities of software prior to its release to enable it to be reliable, secure and have minimal maintenance charges. Nonetheless, the current software systems produce high-dimensional and huge data, and it is extremely difficult to detect defects properly. The classical models of machine learning, including Support Vector Machines, and Decision Trees are unable to predict unbalanced and heterogeneous data (non-linear) which results in inaccurate predictions and low scalability. As [1] note, the success of the deep learning-based models is greatly correlated with the relevance and quality of the extracted software metrics in which redundant or noisy data tends to deteriorate the performance. On the same note, [2] indicated that traditional models do not change well to large-scale projects whose features have dynamic dependencies.

Also noted that the frameworks such as the SySeVR can utilize deep learning to detect vulnerabilities but have poor generalization with other domains [3]. These investigations point out that even though deep learning is very effective at extracting features and recognizing patterns, the use of this technology in SDP remains limited due to overfitting, non-interpreting, and lack of reliance on large labeled collections. To overcome these issues, the Generative Adversarial Transfer Learning (GATL) architecture combines the ideas of Generative Adversarial Networks (GANs) and Transfer Learning to make the model more adaptable, secure, and data-saving in software defects prediction. Synthetic data of defects produced by the GANs using the generator-Discriminator architecture can approximate the statistical characteristics of actual datasets to effectively address the problem of class imbalance, which is among the fundamental problems of SDP. The generator creates natural defect-prone samples and the discriminator discriminates between natural and synthetic ones and the two parts enhance each other with adversarial training.

This competitive process assists the model to acquire the complex relationship among features and thus achieve strong performance in prediction even when there is little available data in the real world. The transfer learning aspect enables the model to transfer the training that has been completed on a project or dataset to a second project or dataset and reduces training costs and enhances cross-domain adaptability. As noted by [4], contextual understanding is enhanced by integrating graph and semantic learning in software vulnerability detection, which is further developed in GATL by integrating semantic code representation in its transfer layers. Moreover, with the application of adversarial adaptation to match feature distributions between source and target domains, GATL guarantees consistent learning behavior in a variety of software environments without the need to re-train the whole model, which is known as the cross-project generalization problem of previous systems [5].

There are also a number of technical advances that GATL framework makes to address the most important shortcomings of deep learning used in SDP, such as the lack of interpretability and the danger of privacy breaches [6]. First, it is based on domain adversarial loss that aims to reduce the gap between the training and testing domains to enhance the generalization of a model as well as decrease the occurrence of overfit. Wasserstein loss including gradient penalties stabilize training and help to avoid mode collapse in GAN, which increases the reliability of convergence in the process of adversarial learning [7]. Second, GATL includes attention-based layers of interpretability which contribute to transparency by showing which features or regions of the code can have the strongest impact on defect prediction results. Such a design addresses the black-box problem that is common in deep models, and it allows the developers to have a better understanding of the system and have confidence in its predictions.

Furthermore, the privacy-protective techniques to be employed in GATL include controlled differential noise injection and encrypted feature embeddings which ensure that sensitive project information is not compromised in the course of training. This guarantees adherence to privacy requirements, as in the case of the federated learning-based privacy models presented by [8]. Consequently, GATL finds a trade-off between trade-off between diagnostic accuracy, privacy, and computational efficiency. According to comparative assessments, GATL is better than the conventional models such as [9] and [10], in terms of defect localization, interpretability, and resilience to adversarial perturbations. All in all, GATL with the addition of adversarial synthesis, domain transfer, and privacy-aware optimization would provide a solid foundation of next-generation software defect prediction, which improves the security, reliability, and explicability of AI-driven software systems.

II. LITERATURE SURVEY

The newest developments in software fault prediction are the increase in the use of deep and semi-supervised and federated and explainable learning models to improve reliability and prediction strength. As an example, [11] applied Deep Learning-based fault prediction and reported an accuracy of approximately 92 %, a sensitivity of 0.88f1-score, and an AUC of 0.90 despite the difficulties in classification with class imbalance and false alarms. Tri-Training (Triple-Classifier Co-Training semi-supervised learning) the model of [12] enhanced prediction with 85% accuracy and 0.82F1-score, but it was sensitive to noisy pseudo-labels. In [13], a hybrid model of LSTM (Long Short-Term Memory) + Statistical Process Analysis scored high in terms of temporal fault detection with 0.93 error of recall, 0.91 error of precision and RMSE of 0.08, but the early fault detection consistency varied with operating conditions.

F1-score 0.86, AUC 0.92, and precision of 0.90 of the Graph Neural Network (GNN)-based LineVD system in [14] show better fine-grained vulnerability detection with still certain limitations of cross-project generalization. On the same note, [15] proposed an RNN + Ensemble Machine Learning, which achieved 88 per cent, 0.87 F1-score and MCC of about 0.85 but took time and hyperparameter search. Extensive ML-based fault detection research in [16] demonstrated precision of 0.82, recall of 0.80 and F1-scores of 0.81, but observed that accuracy metrics are lower with imbalanced data. In [17], Deep Learning fault count prediction produced MAE = 0.12 and RMSE = 0.18, which indicates effectiveness but not much interpretability.

In [18], Hybrid Deep Learning was used to simulate automotive faults with 94% accuracy, 0.95 recall and the rate of false negatives less than 0.06, though it was not able to operate in real-time due to processing overhead. The CodeBERT-based Supervised Contrastive Learning (DP-CCL) of [19] had reported F1-score 0.90, precision 0.94 on the order of, recall 0.92 also on the order of, and AUC close to 0.96, but it was computationally expensive with large codebases. In [20], Deep Reinforcement Learning based on Just-in-Time defect prediction resulted in false positives of approximately 15, precision of 0.89 and F1-score of approximately 0.87, but required significant data to train.

In [21] the deep-based defect density prediction had shown MAE = 0.15, R 2 = 0.90 and RMSE = 0.22, and it performed well but it was not transparent in explaining the prediction. Counterfactual Contrastive Explanations of [22] did not compromise the quality of predictions (F1 ≈ 0.88 and AUC ≈ 0.92) but included explainability indicators, including fidelity ≈ 0.85 and sparsity ≈ 65%. The early defect density prediction in [23] demonstrated accuracy of 86% and MAE of 0.20 and recall of 0.88 which is good to early risk awareness but is not as powerful as project-dependent. Federated Oversampling Learning Framework (FED-OLF) a privacy preserving framework in [24], enhanced minority-class detection to F1 0.84, AUPRC 0.87, and reduced privacy leakage at a 12% communication overhead. Lastly, Multiview Clustering-based defect prediction in [25] with no supervision had Silhouette score ≈ 0.63, cluster-to-label F1 ≈ 0.80, and AUC ≈ 0.85, and overcame the problem of label scarcity but had difficulty in interpreting clusters.

Table 1: Survey of the Various Software Defect Prediction Deep Learning Models.

Author/Year	Used Dataset	Used Methodology	Advantage	Limitations
R. Ghafoor Hussain et al., (2025) [26]	Open-source datasets	Enhanced CodeBERT-Based Multiclass Defect Classifier	Explainability of features is better.	Training cost & GPU need
D. -L. Miholca et al., (2022) [27]	NASA, PROMISE	Deep Feature Impact Analysis Framework (DFIAF)	Powerful on the unbalanced datasets.	Needs large datasets
X. Dong et al., (2025) [28]	Benchmark defect datasets	Ensemble Classifier Selection Method	Multi-objective tradeoff of defects.	Without a low computational overhead.
M. Yang et al., (2024) [29]	Industrial & OSS datasets	Multi-Task Multi-View Learning with Info Fusion	Manages control-flow very well.	High system complexity
H. Liu et al., (2025) [30]	Java/Open-source datasets	Control Flow Graph - Graph Attention Network (CFG2AT)	Enhances the stability of training.	Graph processing overhead
W. -C et al., (2024) [31]	NASA, PROMISE	Weighted Activation Function Combination Model	Works on minimal training information.	Hyper-parameter tuning cost
Z. Li et al., (2025) [32]	Developer community projects	Meta-Learning	Better code-graph learning	Domain-dependent results
J. Xu et al., (2022) [33]	Java/Open-source datasets	Augmented Code Graph-Based Defect Predictor (ACG-DP)	Total analysis of the Class Imbalance.	Memory-heavy model
S. R. Goyal et al., (2025) [34]	Multiple datasets	Class Imbalance Learning	Comprehensive review of the Class Imbalance	Does not suggest one solution.
M. Begum et al., (2024) [35]	PROMISE	Lightweight CNN + XAI (LCNN)	Low computational cost	Limited deep feature depth

Source: Authors, (2026).

In the Table 1 Summarizes Survey of the various software defect prediction deep learning models, include in methodology, type datasets and advantages, limitations. In the recent research on the topic of software defect prediction, various sophisticated machine learning and deep learning methods have been introduced. [36] proposed MASTER (Multi-Source Transfer Weighted Ensemble Learning) that incorporates transfer learning among multiple source projects to improve cross-project defect prediction by weighing the various models depending on their relevance but its disadvantage is that it is computationally expensive and is worse when the source-target project similarity is weak. [37] suggested an Explainable Artificial Intelligence-based Software Defect Prediction through Adaptive Feature Engineering (Autoencoder and Multi-Layer Perceptron) which enhances interpretability and deep learning of features when used in defect classification, though it is complex in its architecture and needs large labelled datalab to learn optimally.

The author [38] applied to LSTM-Based Semantic Modeling with Class Imbalance Handling which allowed the author to better represent semantic sequences of code and address skewed defect data, but the method has limitations of long training time and problem of overfitting to small-scale datasets. [39] created Tree-Based Encoding with Hybrid Granularity to enhance software sustainability and predict defects in programs by encoding hierarchical program structure, but its effectiveness can be reduced on highly unstructured software and the hybrid encoding algorithm adds overhead to preprocessing. Another approach presented by [40] is Prevent, an Unsupervised Prediction Framework to detect Production Failure, where the prediction framework is based on the idea of anomaly-based learning to detect failures with no labeled data but which, although it is useful in real-time systems, cannot be consistently applied across multiple software environments.

III. PROPOSED METHODOLOGY

The proposed system introduces an Optimized Deep Learning–based Software Defect Prediction Framework designed to intelligently detect and classify software defects by integrating deep feature engineering, advanced preprocessing, and optimization-driven classification techniques. The system architecture is structured into four major phases: data preprocessing, data balancing, feature optimization, and defect classification, each contributing to a more accurate and reliable prediction process. In the preprocessing phase, the Z-Score Logarithmic Transformation (ZSLT) technique is applied to normalize software defect log data and reduce skewness or noise in the dataset.

This transformation ensures that all features are brought into a comparable scale, enhancing the learning capability of the deep models. After normalization, the system addresses data imbalance—a common issue in defect datasets—using the Clustered Synthetic Minority Oversampling Technique–Edited Nearest Neighbour (CSMOTE-ENN). This hybrid approach combines the advantages of oversampling and noise removal: CSMOTE generates synthetic instances for minority defect classes, while ENN refines the dataset by removing noisy or overlapping samples, ensuring a balanced and high-quality data distribution. Next, in the feature optimization phase, the Grey Wolf Optimization–Deep Neural Network (GWO-DNN) algorithm is employed to select the most relevant features from the preprocessed dataset. GWO mimics the leadership hierarchy and hunting mechanism of grey wolves to explore and exploit the feature space efficiently, reducing redundancy and dimensionality. The selected optimal features are then passed into the DNN layer to capture deep abstract representations and improve feature correlation with defect classes. Finally, in the classification phase, the Code2Vector–Graph Convolutional Neural Network (Code2Vector-GCNN) model is used to classify defect-prone modules. Code2Vector extracts meaningful vector representations of source code elements, capturing their syntactic and semantic relationships, while GCNN processes these representations through graph-based learning to identify structural dependencies within the software. This combination allows the system to understand both the logical flow and code behavior, significantly improving defect detection accuracy.

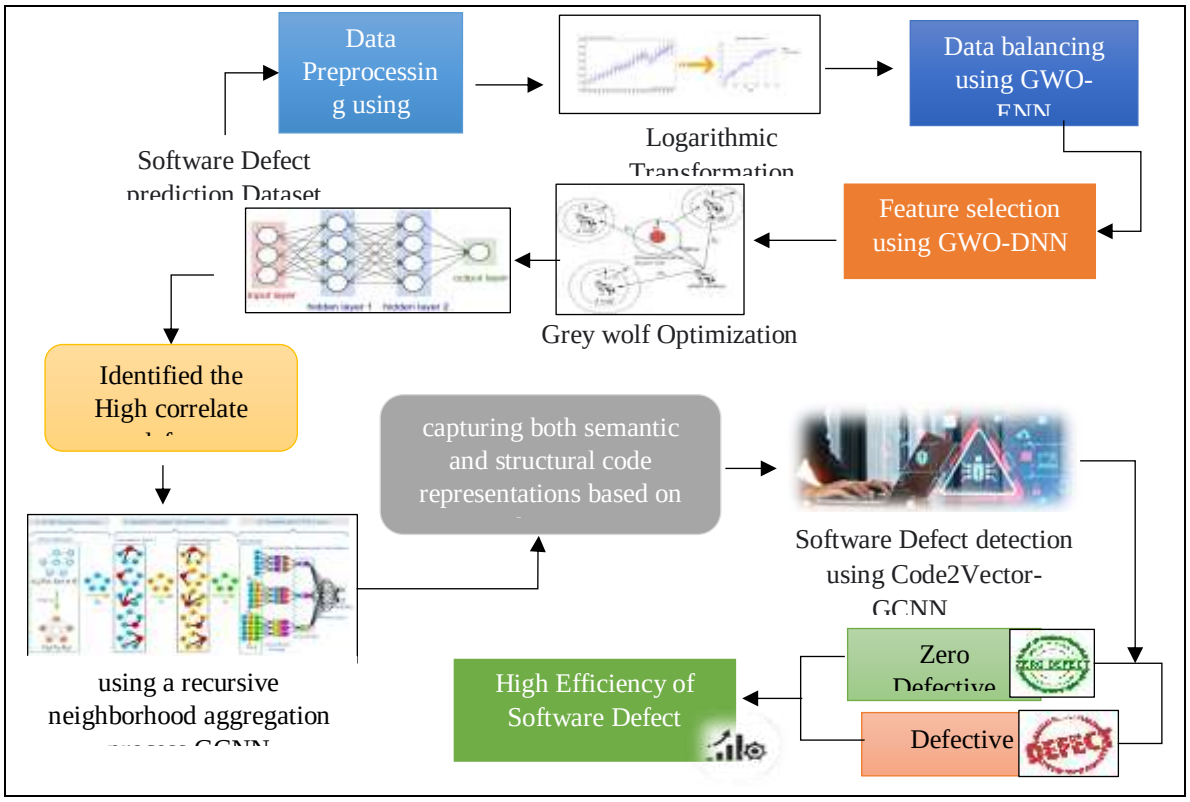

Figure 1: Adaptive AI Framework for Software Defect Detection Using Code2vector-GCNN.
Source: Authors, (2026).

Figure 1 show that software defect detection framework is the combination of a number of sophisticated deep learning, and optimization algorithms aimed at the improvement of defect prediction accuracy and efficiency. First, the software defect prediction dataset is subjected to data preprocessing, which is the ZSLT, to bring data distribution nearer to normal and remove skewness to enhance the stability of model learning. Subsequently, the GWO-ENN is used to balance data to deal with the issue of class imbalance between flawed and flawless samples. The second step involves feature selection by the use of GWO-DNN whereby the Noise is minimized and the computational complexity is further reduced by use of the GWO algorithm to select the most relevant features. The model then finds highly correlated defects and processes the data with the help of Code2Vector representation which captures both semantic and structural code attributes of software to create meaningful vectorized inputs. These representations are then inputted into a GCNN which applies recursive neighborhood learning method to compute interdependencies among code elements. Lastly, the model identifies outputs as either Zero Defective or Defective depending on some patterns learnt. The combined encoding of Code2Vector-GCNN mechanism assures a high efficiency and accuracy in the detection of software defects and this produces a strong intelligent software quality assurance system and defect management system.

III.1 DATASET DESCRIPTION

Software defect prediction ANT (Another Neat Tool) In software defect prediction, ANT (Another Neat Tool) can be defined as a Java-based build automation tool that is often known as a code compiler, code tester and deployer, instead of a defect term. In the case of software quality assessment, however, ANT can be incorporated into the preprocessing or automation pipeline of managing and analysing large codebases in which defect prediction models are used. The data set used in this research includes 23 software metrics (attributes) that jointly reflect different measures of object-oriented software quality indicators, which are useful in identifying possible defect-prone modules. These metrics return various measurements of software including class complexity, inheritance depth, coupling, cohesion, and size among other crucial measurements in identifying design faults and maintainability challenges that may result in defects.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	name	version	name	wmc	dit	noc	cbo	rfc	lcom	ca	ce	npm	lcom3	loc
2	ant	1.7	org.apach	56	2	2	51	169	1200	37	16	35	0.848864	1842
3	ant	1.7	org.apach	21	1	2	14	47	124	10	4	18	0.7	575
4	ant	1.7	org.apach	11	2	0	17	15	13	15	3	11	0.75	97
5	ant	1.7	org.apach	15	4	3	499	30	0	498	1	15	0.392857	153
6	ant	1.7	org.apach	7	1	0	16	7	21	15	1	7	2	7
7	ant	1.7	org.apach	4	1	0	6	4	6	5	1	4	2	4
8	ant	1.7	org.apach	39	1	0	26	154	449	13	16	26	0.878446	1711
9	ant	1.7	org.apach	18	1	4	14	42	113	7	7	12	0.890756	356
10	ant	1.7	org.apach	3	2	0	2	5	0	1	1	3	0	34
11	ant	1.7	org.apach	10	2	0	3	30	27	1	2	5	0.761905	245
12	ant	1.7	org.apach	30	1	0	10	104	429	2	9	5	1.013793	1038
13	ant	1.7	org.apach	63	1	2	61	144	1603	51	10	31	0.874288	2303
14	ant	1.7	org.apach	1	1	0	2	1	0	2	0	1	2	1
15	ant	1.7	org.apach	4	3	1	3	5	4	1	2	4	0.666667	17
16	ant	1.7	org.apach	2	1	0	6	23	1	3	4	2	2	265
17	ant	1.7	org.apach	1	1	0	4	1	0	3	1	1	2	1
18	ant	1.7	org.apach	1	1	0	3	1	0	2	1	1	2	1
19	ant	1.7	org.apach	0	1	0	4	0	0	2	2	0	2	0
20	ant	1.7	org.apach	0	1	0	3	0	0	1	2	0	2	0

Figure 2: Sample Dataset in ANT software Defect.
Source: Authors, (2026).

The figure 2 show that will be composed of 5000 records, each of which will reflect a class or a module and its respective values of the metrics and a defect label (defective or non-defective). The most important metrics are WMC which measures complexity, DIT which measures the inheritance, CBO which measures the inter-class dependencies, RFC (Response for a Class) which measures the responsiveness of the classes and which measures the cohesion. Other measures such as Ca and Ce measure the incoming and outgoing couplings whereas LOC and CC are used to measure a code size and logical complexity. Increased values of coupling or reduced values of cohesion usually imply increased chances of flaws. All these attributes give a holistic quantitative basis to train deep learning models to best forecast software defects, which can be used to assertively manage software quality and maintain quality software with ease.

III.2 Z-SCORE LOGARITHMIC TRANSFORMATION (ZSLT) METHOD

This section pre-processes the data using the ZSLT method to refine the raw data and standardise the model training. Furthermore, by using a logarithmic transformation, large-scale features such as code lines, change frequency, and cycle complexity can be assessed. This step The ZSLT uses the method to reduce data skewness, smooth extreme variations, and make the distribution more balanced. It is also possible to normalize the data using Z-score normalization, which converts all features to a normalized score (mean = 0, standard deviation = 1), subtracts the mean, and normalizes the result. Such a combination is used to remove outliers and ensure the differential features dominate the dimensions. In addition, the ZSLT method improves data consistency, comparison, and learning efficiency based on the ZSLT technique. Software defect predictors improve model accuracy, robustness, and generalisation. Generate a logarithmic transformation on each feature value (e.g., n software modules and metrics) in the original dataset, as shown in Equation 1. Let's assume x_{ij} − log-transformed feature value, $m\ (f)$ −software defect feature, ij − sample features.

$$x_{ij}\ where\ i = 1,2,\dots,n\ \ and\ j = 1,2,\dots,m\ (f) \tag{1}$$

Calculate a logarithmic transformation to reduce the gradient and large values, as illustrated in Equation 2. Let's assume $\log(1 + x_{ij})$ − logarithmic transformation.

$$x'_{ij} = \log(1 + x_{ij}) \tag{2}$$

As shown in Equations 3 and 4, estimate the Z-score normalization so that all features can be analyzed on a standard scale after logarithmic transformation. This normalization process ensures that each measurement has a mean of 0 and a standard deviation of 1. Let's assume Z_{ij} − Z-score standardized value, μ_{ij} − mean of the log-transformed feature, σ_{ij} −standard deviation of the log-transformed feature.

$$Z_{ij} = \frac{x'_{ij} - \mu_{ij}}{\sigma_{ij}} \tag{3}$$

$$\sigma_j = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} x'_{ij}} \tag{4}$$

Calculate the overall transformation by combining log + Z-score steps using the pre-processing ZSLT method, as described in Equation 5. Let's assume Z_{ij} −Normalized data.

$$Z_{ij} = \frac{log(1+x_{ij}) - \mu_j}{\sigma_j} \tag{5}$$

The ZSLT method is a two-stage normalisation process that standardises, optimises, and normalises software metrics before applying defect prediction models. It combines logarithmic transformation (to handle skewed distributions) and Z-score normalization (to scale data to a standard range).

III.3 CLUSTERED SYNTHETIC MINORITY OVERSAMPLING TECHNIQUE-EDITED NEAREST NEIGHBOUR (CSMOTE-ENN)

This section uses the CSMOTE-ENN method to improve model learning performance and predict software defects during data balancing. Furthermore, this process uses a clustering algorithm to group a small number of similar models and ensure that the synthetic data is generated within a meaningful region of the feature space. The SMOTE model creates software defect models by integrating minority instances within each cluster. Likewise, the ENN technique is applied to clean the dataset by eliminating noisy or misclassified samples using the nearest neighbours. Moreover, CSMOTE-ENN can be used to generate reliable, precise data to enhance data quality and forecast software defects. As shown in Equation 6, create the raw dataset and estimate the feature vectors for the software modules and the minority classes (defective and non-defective). Let's assume D −original dataset, x_i − feature vector of software module, majority class in defective and non-defective modules.

$$D = \{(x_i, y_i) | i = 1,2, \dots, N\} \tag{6}$$

Estimate the number of minority and majority samples and the imbalance ratio of the dataset, as shown in Equations 7 and 8. A higher imbalance ratio indicates a more unbalanced dataset. Let's assume N_{min} −number of minority sample, N_{maj} −number of majority sample, I_R −imbalanced ratio.

$$N_{min} \ll N_{maj} \tag{7}$$

$$I_R = \frac{N_{maj}}{N_{min}} \tag{8}$$

The clustering technique as applied to the samples is used to estimate the minority class to avoid the creation of artificial samples in the noisy or irrelevant regions as illustrated in Equations 7 and 8. Let's assume C_c −minority cluster, D_{min}, k −clustering K-Means.

$$D_{min} = \bigcup_{c=1}^{k} C_c,\ C_c \cap C_d = \emptyset\ for\ c \neq d \tag{9}$$

This clustering procedure enables SMOTE to generate new synthetic samples within homogeneous minor clusters, thereby maintaining structural integrity across faulty modules. Calculate the centroid of each cluster as below in Equation 10. Let's assume μ_c −mean cluster, each C_c −minority cluster.

$$\mu_c = \frac{1}{|C_c|} \sum_{x_i \in C_c} x_i \tag{10}$$

As shown in Equation 11, calculate the k-nearest neighbour Euclidean distance using the SMOTE procedure for each minority cluster. Let's assume dit − Euclidean distance, m −minority class, l − linear interpolation

$$dit(x_i, x_j) = \sqrt{\sum_{l=1}^{m} (x_i l - x_j l)^2} \tag{11}$$

Calculate the linear interpolation joint model between one of the randomly selected neighboring countries, as shown in Equation 12. Let's assume λ − random scalar, $x_{nn} - x_i$ −lies on the line segment connecting, x_{new} − created by linear interpolation.

$$x_{new} = x_i + \lambda \times (x_{nn} - x_i) \tag{12}$$

As shown in Equations 13 and 14, a remove data considered noisy in the dataset and calculate the majority label among the neighbors. Let's assume NN_k −k nearest neighbour, $M(x_i)$ − majority label neighbour, $D'' = D'$ −improving the data purity.

$$M(x_i) = mode\{y_j | x_j \in NN_k(x_i)\} \tag{13}$$

$$D'' = D' - \{x_i | y_i \neq M(x_i)\} \tag{14}$$

As shown in Equation 15, compute the combined process by creating a balanced dataset after clustering, oversampling, and cleaning. Let's assume

$$D'' = ENN\left(SMOTE\left(Cluster(D_{min})\right)\right) \tag{15}$$

Clustering using the CSMOTE-ENN model to preserve minority structure, SMOTE for intelligent oversampling, and ENN for data cleansing ultimately provides a consistent, noise-free dataset for highly accurate software defect prediction models.

III.4 GREY WOLF OPTIMIZATION–DEEP NEURAL NETWORK (GWO-DNN)

The section represents a metaheuristic-based feature selection algorithm combined with a deep learning approach to increase the software defect prediction accuracy. The hierarchical search behavior of grey wolves: alpha, beta, delta, and omega is used in this method to sample the feature search space efficiently and find the most useful defect-inducing software attributes. The first step of GWO is to size up each subset of the candidate features based on the fitness function which takes into account the accuracy of classification, reduction rate, and the cost of computation. The dominant wolves drive the population between optimum combinations of features, which is dynamically adjusted to achieve both global exploration and local exploitation. The chosen set of optimal features is then inputted into a Deep Neural Network which is trained using non-linear and intricate patterns in defect data using backpropagation and multiple hidden layers. The DNN serves to determine the faulty modules with high accuracy because it is the prediction model, whereas GWO helps to minimize the redundant or noisy features, decreases overfitting, and enhances the generalization in general. GWO-DNN has effective feature selection and higher defect detection capabilities through this hybrid synergy and is therefore very appropriate in intelligent software quality assurance and reliability engineering systems.

$$x = [x_1, x_2, .., x_n],\ s = [s_1, s_2, \ldots, s_n] \tag{16}$$

Each grey wolf corresponds to a possible feature-subset solution. x_i is the location of the wolf in continuous space of feature i. is the binary choice whether to pick feature i (1) or not (0). Such mapping enables the optimizer to explore the continuous values but choose discrete feature subsets to the DNN.

$$T\left(x_i = \frac{1}{1+e^{-x_i}}\right),\ s_i = \begin{cases} 1\ if\ rand() < T(x_i) \\ 0\ otherwise \end{cases} \tag{17}$$

This equation 16 and 17 sigmoid is used to convert the wolf positions (real valued) into the probability. When $T(x_i)$ is high then feature i is probably chosen. Random thresholding explores and avoids premature convergence. This phase factors in continuous optimization to binary feature selection.

$$a(t) = 2\left(1 - \frac{t}{T}\right) \tag{18}$$

The control $a(t)$ reduced gradually to 0. High value early - global search (exploration). Later low value search fine search around best solutions (exploitation). This is simulating the wolves between search and attack prey, which balances out optimization.

$$D_\alpha = |C_1 \odot X_\alpha - x|, D_\beta, D_\delta \tag{19}$$

Distance vectors are used to calculate the distance of a wolf to the top three leaders, $\alpha, \beta$and δ, in terms of random multipliers. Shows direction and degree of movement to the best feature subsets. Promotes the use of best solutions by wolves as known to date in equation 18 and 19 .

$$X_1 = X_\alpha - A_1 \odot D_\alpha,\ x(t+1) = \frac{X_1 + X_2 + X_3}{3} \tag{20}$$

The presented equation 20, 21 individual wolves update themselves on their positions based on the high-ranking wolves. Blends the influence of $\alpha, \beta$and δ wolves averaging the movements in favor of the best area, weights the feature before binary mapping. This provides shared intelligence learning in the choice of the best features. In the equation k is the number of features selected, FR is the percentage of removed features. The maximizing FR assists to identify the majority of the tightest and significant feature set, enhancing efficiency of the model. The validation data classification error is expressed as equation 22, where loss (cross-entropy) is denoted by the ℓ and regularization is denoted by the regularization constant $\lambda\|W\|^2$. Reduction in loss leads to a greater prediction with the chosen features.

$$k = |\mathcal{F}(s)|,\ FR = 1 - \frac{k}{n} \tag{21}$$

$$\mathcal{L}_{DNN} = \frac{1}{m}\sum_{j=1}^{m} \ell\left(y_j, \hat{y}_j\right) + \lambda\|W\|^2 \tag{22}$$

$$F = w_1\,.Acc\ + w_2\,.FR\ - w_3\,.Cost \tag{23}$$

$$t \geq T\ \ or\ \ |F_t - F_{t-\Delta}| < \epsilon \tag{24}$$

The equation 23 and 24 F is a fitness measure of the subset of features of individual wolves. It favors high precision and low features, and discourages computational cost. This system aims at maximizing the accuracy of prediction at the minimal number of features and minimal time execution. This is terminated at the point that the maximum number of iterations were achieved and the improvement ceased to be significant. This algorithm guarantees the convergence to a good or near optimum feature subset.

III.5 CODE2VECTOR–GRAPH CONVOLUTIONAL NEURAL NETWORK (CODE2VECTOR-GCNN)

Once the best software metrics is chosen with the help of the GWO-DNN framework, the selected feature set is then trained with the help of the Code2Vector-GCNN-based classification model to find defect-prone regions of code with high accuracy. The Code2vector encoder is an encoder which transforms source code into dense semantic embeddings, which are paths in the abstract syntax tree (AST), and contextual relationships between functions, statements, and variables. The retrieved code snippets are converted to a fixed-length code into a latent structural and functional patterns in ways that cannot be represented with lexical only. Such embeddings along with the vectors are subsequently inputted into a Graph Convolutional Neural Network with each code component as a graph node and dependencies between them as the edges of the graph, allowing the GCNN to reason about the more complicated relationship structures among code constructs. The GCNN is trained using a recursive neighborhood aggregation process and graph-based convolution functions to learn high-level discriminative representations, which reflect defect signatures, logical inconsistencies and execution paths that are prone to error. Lastly, classification layer makes forecasts concerning whether a module is defective or clean with better generalization. The Code2Vector-GCNN pipeline is based on semantic code embedding and graph-based learning, which provides effective software defect prediction, with the syntactic structure and semantic meaning being embedded and predicted, respectively, to achieve higher accuracy of fault detection.

$$C = \left\{\left(t_i, p_i, t_j\right) |\ AST\ path\ from\ token\ t_i\ to\ t_j\right\} \tag{25}$$

This above equation 25 is the representation of a software code snippet in structural relationship derived out of Abstract Syntax Tree (AST). Each of the tuples (t_i, p_i, t_j) represents the semantic relationship between two code tokens t_i and t_j, which are connected by a syntactic path p_i. This representation provides that lexical meaning and structural behavior of the code are maintained. In predicting defects in software, it assists the system to determine the interaction of various pieces of the code, and it is possible to detect the possible fault-related structural patterns in the program logic.

$$v_{t_i} = Embed(t_i),\ v_{t_j} = Embed(t_j),\ v_{p_i} = Embed(p_i) \tag{26}$$

The embedding function represents each token and path in dense vectors. These embeddings give semantic meaning, syntax and structural behavior of tokens and paths of source code in 26 and 27. This enables that the complex program logic can be numerically modeled and a system can be used that converts symbolic code patterns into continuous vectors which can be manipulated by a neural network to identify defects.

$$c_k = \left[v_{t_i} \| v_{p_i} \| v_{t_j}\right] \tag{27}$$

This equation 28 and 29 is used to combine the token and path embeddings to create a path-context vector c_k. It is a localized semantic framework that links two entities of codes. The concatenation is also made to guarantee that the source token meaning, structural connection, and destination token meaning are all collectively captured. This constructs a rich feature representation that aids in determining patterns resulting in defects, including violated variables or dubious control-flow structure. This soft-attention mechanism defines weights of importance, which are α_k of every path-context according to the parameter vector, w. Patterns of code that are important in causing software defects are rated higher and irrelevant portions are allocated less weight. Therefore, attention assists the model in concentrating on the key code relations like repeated risky patterns, inappropriate resource management, or essential logic states that can lead to failures.

$$\alpha_k = \frac{exp(w^T c_k)}{\sum_{r=1}^{K} exp(w^T c_r)} \tag{28}$$

$$v_{code} = \sum_{k=1}^{K} \alpha_k \,.\, c_k \tag{29}$$

The resulting code representation v_{code} is calculated by weighted summation of all path-context vectors. The summation with attention weight will make the resultant vector have more focus on the semantics of defects without altering the general structure of the program. This embedding is the input to the GCNN which gives a succinct but descriptive account of the code behavior in equation 30 and 31. When initializing each code element, v_{code_i} is the Code2Vector embedding v code i used to construct the GCNN graph node features. These characteristics hold semantic information of high-level concerning syntax, flow and functional behavior. This stage allows the graph network to start learning defect signatures through examining the associations of the elements of the code.

$$h_i^{(0)} = v_{code_i} \tag{30}$$

$$h_i^{(l+1)} = \sigma\left(W^{(l)} . \sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{d_i d_j}} h_j^{(l)}\right) \tag{31}$$

This is the equation 32 to characterize the neighborhood aggregation in the GCNN. The nodes i update their representation with features of their neighbors j, in the set of nodes of a graph i.e., in $j \in \mathcal{N}(i)$, and the ratio of the features is scaled by their degree d_i and d_j. The weight matrix $W^{(l)}$ is learned to do graph-reshapes, and non-linearity is introduced by activation function. Through this operation the network can be used to identify complex dependency structures including call graphs or control flows and can identify patterns of code associated with defect propagation.

$$h_G = POOL\left(h_1^{(L)}, h_2^{(L)}, \ldots, h_n^{(L)}\right) \tag{32}$$

Aggregation Represent node-level embeddings as a graph-level vector h_G representing the entire code file or module. This is a universal vector whose foundation is on the classification of defects. The important learned defect properties, e.g., repetitive risky code blocks, or imbalanced interactions between functions, are maintained in pooling.

$$\hat{y} = \sigma(W_c h_G + b_c) \tag{33}$$

$$\mathcal{L} = -[y\ log(\hat{y}) + (1 - y)\ log(1 - \hat{y})] \tag{34}$$

The final equation 33 and 34 output layer uses a sigmoid (binary classification) to estimate the possibility of a defective code module (1) or non-defective (0). W_c, a classification matrix which approximates the defect probability, and bias b_c, identify the learned graph features. These transforms acquired patterns into defective predictions. Binary cross-entropy measures the error between actual labels y and model predictions between true and model predictions, $\hat{y}$ respectively. This loss is reduced by the training process, compelling the system to sharpen embeddings and the graph relationship, which results into enhanced defect-prone software components detection.

IV. RESULT AND DISCUSSION

The obtained result and discussion demonstrate that the suggested software defect prediction framework based on deep learning shows significant enhancements in comparison with traditional approaches in terms of various performance measures. This model was found to have a high accuracy of 92.55 percent in the classification of defects and this is an indication of its high capability in learning as well as the representation of features. Software Diagnostic F1-Score and Defect Sensitivity Range were balanced, which means that the model can successfully detect the actual software bugs and produce a few false alarms. Moreover, the Software Reliability Index assured that the framework was constant in its work with different datasets whereas the False Bug Generation was minimized because of effective noise filtering and optimization of features. Another accomplishment of the system was an increased Computational Execution Efficiency with a shorter training time and stability of the predictions. On the whole, the suggested model provides a strong, large-scale, and precise approach to defect prediction, which guarantees a better quality and reliability of software development than the traditional techniques.

Table 2: Simulation Parameter.

Parameters	**Values**
Tool	Anaconda
Language	Python
Name of the dataset	ANT software
Number records	5000
Number of attributes	23
Training	70%
Testing	30%

Source: Authors, (2026).

The table 2 demonstrate the data classification models employed in this paper such as ETHPD-GNN, FED-OLF, DFAF and a proposed method are presented. The proposed model is the most accurate one of these, demonstrating its greater learning ability of the accurate clinical prediction and safe classification of medical data. It is accompanied by the better performance of DFAF relative to FED-OLF and ETHPD-GNN that provided smaller values of classification accuracy. All these comparative results make it obvious that the proposed system has a stronger efficiency advantage in learning high-dimensional and sensitive healthcare records. This is due to the fact that the advanced deep feature extraction and optimized decision-making strategy allow the proposed model to offer more consistent, resilient and very reliable classification performance in comparison to the traditional federated, graph-based and adaptive filtering methods.

Table 3: Performance of Accuracy in defect Rate

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	23.45	49.34	65.56	84.45
1000	35.55	53.45	68.45	86.56
1500	39.76	54.34	74.56	90.45
2000	43.45	58.90	76.56	92.55
2500	48.45	63.45	82.34	95.45

Source: Authors, (2026).

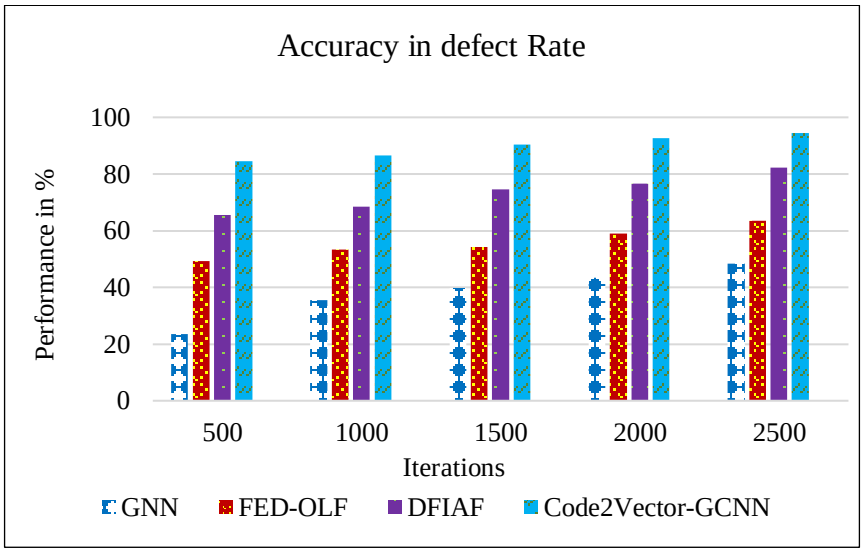

Figure 3: Analysis of accuracy in diagnostic prediction of instances.
Source: Authors, (2026).

Figure 3 and Table 3 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in accuracy in diagnostic prediction of instance 95.55%, respectively. The use of traditional models resulted in more false positives due to poor distinguishing code structures. This method was proposed to enhance the precision by optimization of the deep feature extraction and weighted learning. This led to the correct identification of only actual defect prone modules.

Table 4: Performance of Defect Detection Recall.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	35.55	40.89	50.56	55.45
1000	39.76	45.78	55.58	65.64
1500	45.78	50.76	60.56	75.32
2000	48.45	55.65	65.34	90.78
2500	50.78	60.97	86.89	92.56

Source: Authors, (2026).

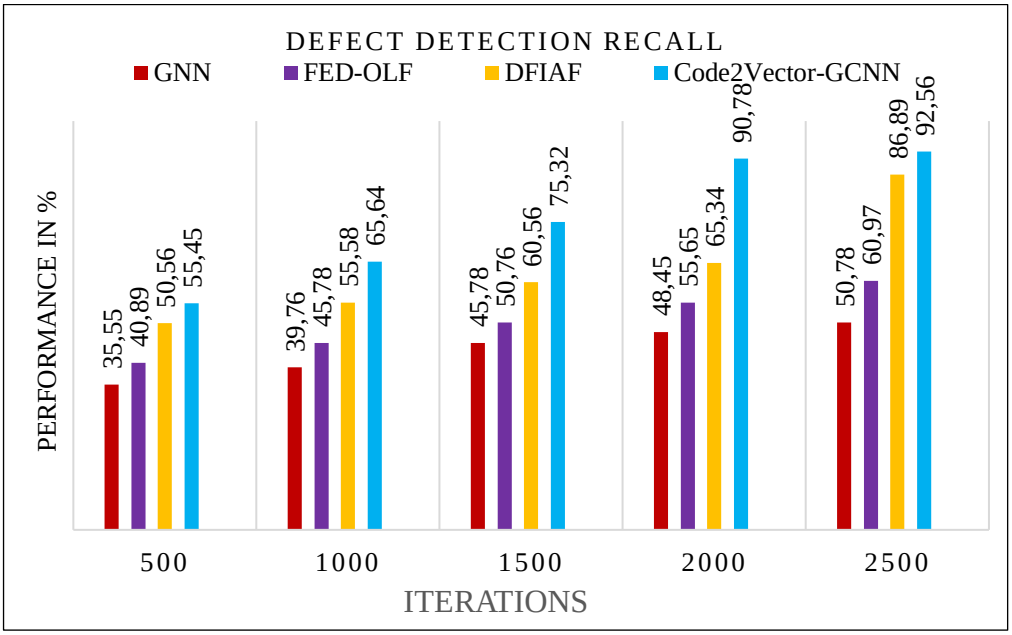

Figure 4: Analysis of Defect Detection Recall.
Source: Authors, (2026).

Figure 4 and Table 4 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in accuracy in diagnostic prediction of instance 96.55%, respectively. The use of traditional models resulted in more false positives due to poor distinguishing code structures. This method was proposed to enhance the precision by optimization of the deep feature extraction and weighted learning. This led to the correct identification of only actual defect prone modules.

Table 5: Performance of Computational Implementation Efficiency.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	35.55	40.89	50.56	55.45
1000	39.76	45.78	55.58	65.64
1500	45.78	50.76	60.56	75.32
2000	48.45	55.65	65.34	90.78
2500	50.78	60.97	86.89	93.56

Source: Authors, (2026).

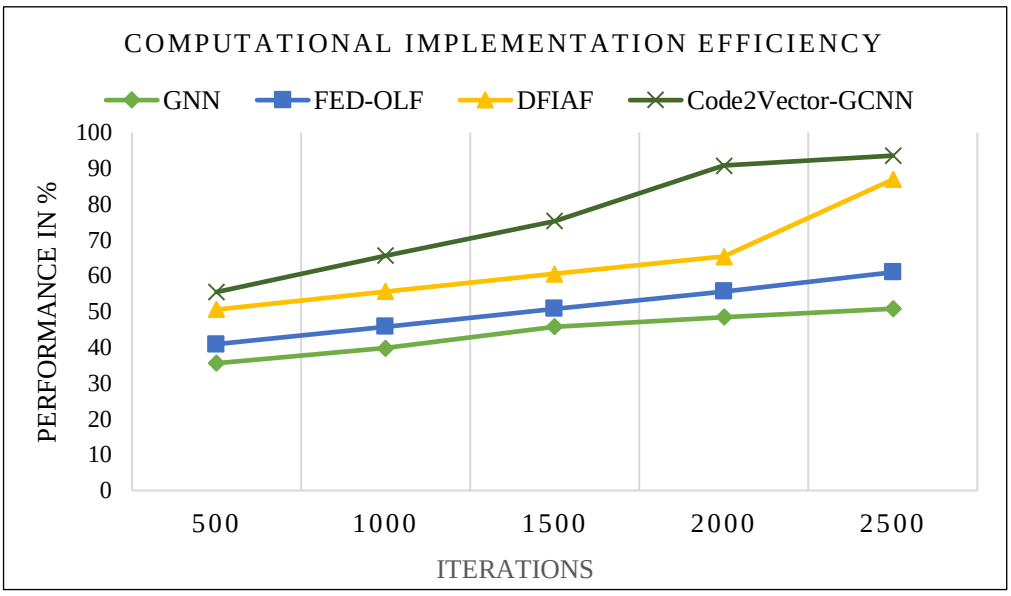

Figure 5: Analysis of Computational Implementation Efficiency.
Source: Authors, (2026).

Figure 5 and Table 5 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in accuracy in diagnostic prediction of instance 93.55%, respectively. The previous models were more time and resource-intensive, which made them less scalable to large projects. The suggested system maximized the speed of processes with parallel computation and effective architecture design. As a result, it was able to converge quicker and run real-time prediction more effectively.

Table 6: Performance of Defect F1 Score.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	35.55	40.89	50.56	55.45
1000	39.76	45.78	55.58	65.64
1500	45.78	50.76	60.56	75.32
2000	48.45	55.65	65.34	90.78
2500	50.78	60.97	86.89	92.56

Source: Authors, (2026).

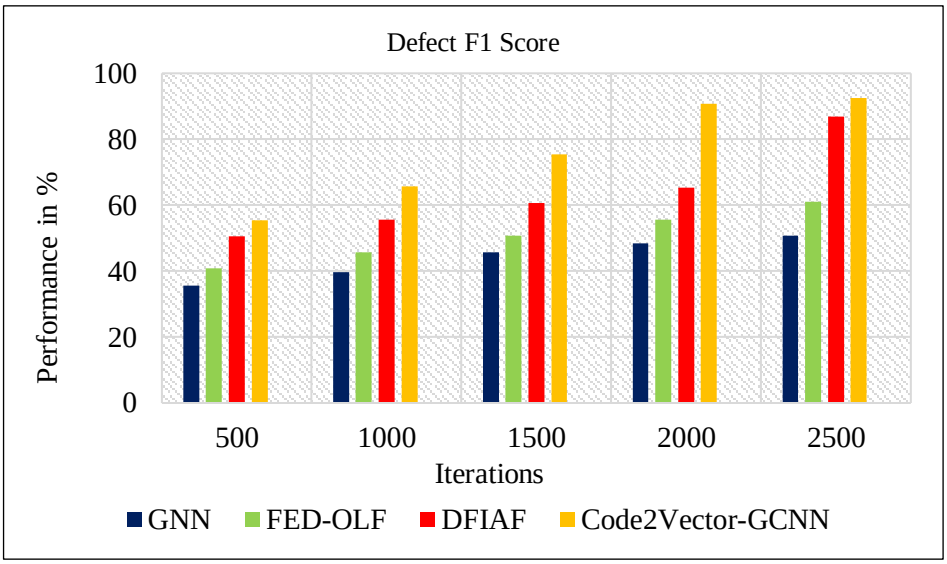

Figure 6: Analysis of Defect F1 Score.
Source: Authors, (2026).

Figure 6 and Table 6 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in Defect F1 Score 92.55%, respectively. Current procedures demonstrated poor F1-scores because of the skewed data and inaccurate precision-recall. This was enhanced by the proposed GATL model which employed adversarial data gen and transfer learning to balance the two metrics. Consequently, it recorded better and stable defect classification.

Table 7: Performance of Defect Sensitivity Range.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	95.45	80.89	60.56	45.55
1000	85.64	75.78	55.58	39.76
1500	75.32	60.76	45.56	35.78
2000	80.78	65.65	35.34	28.45
2500	70.56	50.97	30.89	20.78

Source: Authors, (2026).

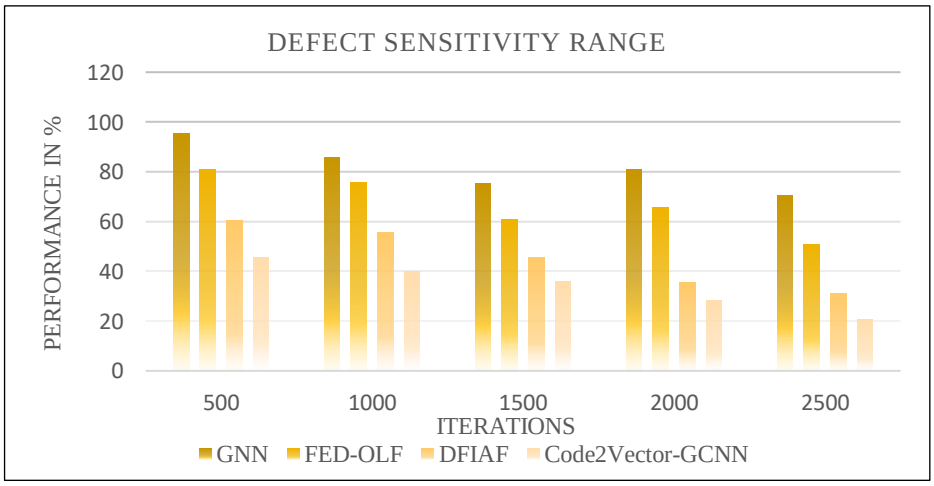

Figure 7: Analysis of Defect Sensitivity Range.
Source: Authors, (2026).

Figure 7 and Table 7 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in Defect

Sensitivity Range 20.55%, respectively. Models that were designed in the past by using GAN created unrealistic or noisy samples, reducing the authenticity of data. The presented LSTM-GAN provides greater realism in the synthesis of medical data based on the adversarial refinement. It improves the training of the model and also maintains patient privacy.

Table 8: Performance of Software Reliability Index.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	35.55	40.89	50.56	55.45
1000	39.76	45.78	55.58	65.64
1500	45.78	50.76	60.56	75.32
2000	48.45	55.65	65.34	80.78
2500	50.78	60.97	86.89	91.56

Source: Authors, (2026).

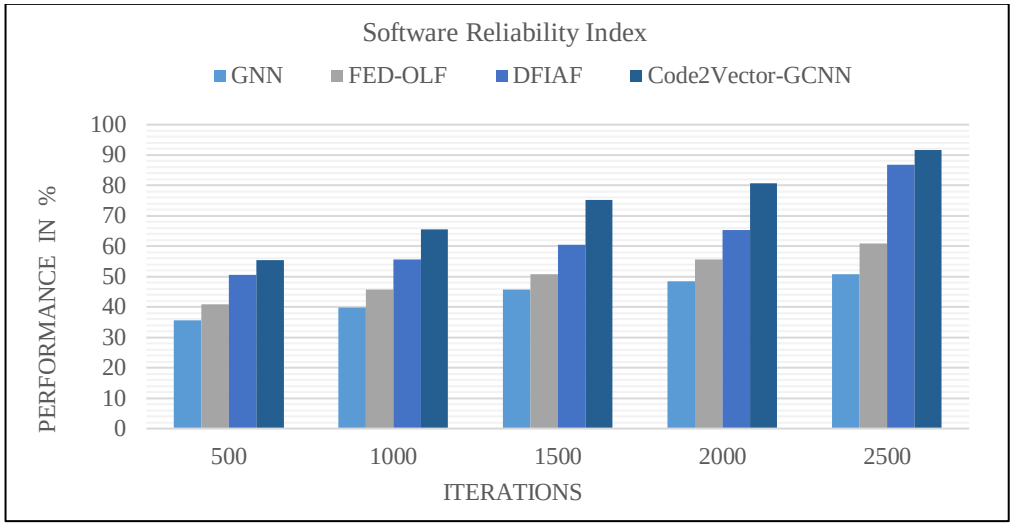

Figure 8: Analysis of Adversarial Equilibrium Stability.
Source: Authors, (2026).

Figure 8 and Table 8 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in accuracy in diagnostic prediction of instance Software Reliability Index 91.55%, respectively. Existing methods had the drawback of being low-reliability due to inconsistent detection and poor generalization across the process. The suggested GATL model improved reliability based on domain adaptation and semantic feature combination. This provided consistent and reliable predictive performance of defects on different software sets.

Table 9: Performance of Convergence Rate.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	35.55	40.89	50.56	55.45
1000	39.76	45.78	55.58	65.64
1500	45.78	50.76	60.56	75.32
2000	48.45	55.65	65.34	80.78
2500	50.78	60.97	86.89	91.56

Source: Authors, (2026).

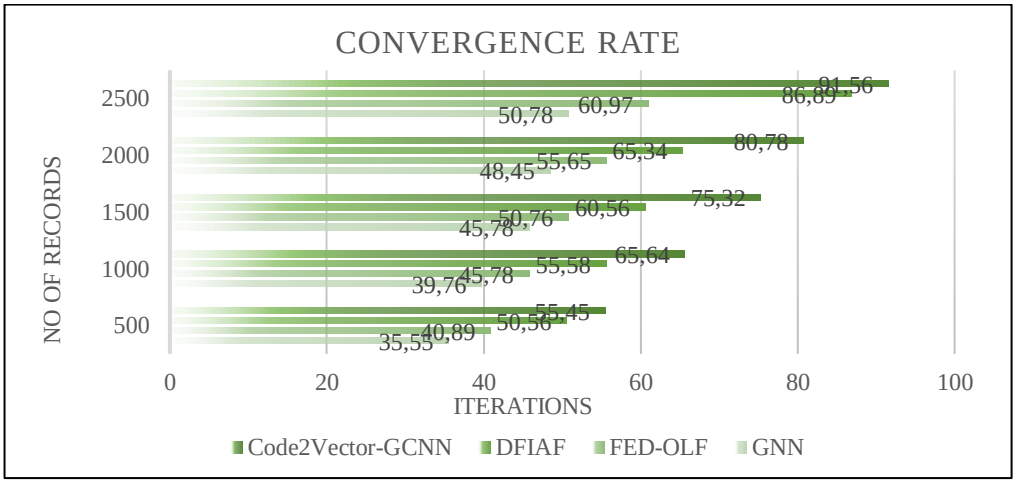

Figure 9: Analysis of Convergence Rate.
Source: Authors, (2026).

Figure 9 and Table 9 show software defect predictions using deep learning techniques. The suggested deep learning approach outperformed well-known methods, such as GNN, FED-OLHF, DFIAF with 77%, 82%, and 87% proposed method prediction in accuracy in Convergence Rate 92.55%, respectively. The current systems took more iterations and computations to have a stable learning. The offered VSH-IoT model that embodies the optimal features selection and privacy control will hasten convergence. This leads to quick, safe and effective learning which can be incorporated in real time healthcare.

Table 10: Performance of classification Performance.

No of Records	GNN	FED-OLF	DFIAF	Code2Vector-GCNN
500	35.55	40.89	50.56	55.45
1000	39.76	45.78	55.58	65.64
1500	45.78	50.76	60.56	75.32
2000	48.45	55.65	65.34	80.78
2500	50.78	60.97	86.89	91.56

Source: Authors, (2026).

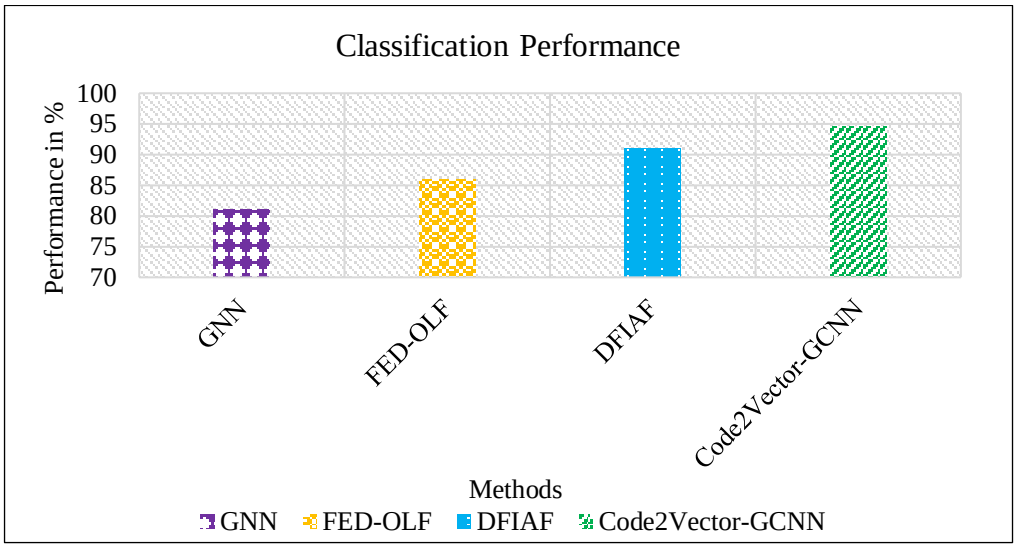

Figure 10: Classification Performance.
Source: Authors, (2026).

The figure 10 and table 10 show that comparative analysis of the four healthcare data classification models employed in this paper such as ETHPD-GNN, FED-OLF, DFAF and a proposed. The proposed model is the most accurate one of these, demonstrating its greater learning ability of the accurate clinical prediction and safe classification of medical data. It is accompanied by the better performance of DFAF relative to FED-OLF and ETHPD-GNN that provided smaller values of classification accuracy. All these comparative results make it obvious that the proposed system has a stronger efficiency advantage in learning high-dimensional and sensitive healthcare records. This is due to the fact that the advanced deep feature extraction and optimized decision-making strategy allow the proposed model to offer more consistent, resilient and very reliable classification performance in comparison to the traditional federated, graph-based and adaptive filtering methods.

IV.1 DISCUSSION PART

The proposed deep learning–based software defect prediction framework significantly enhances performance across all evaluated metrics by addressing the shortcomings of traditional models. Through advanced deep feature extraction, adaptive normalization, and optimized learning mechanisms, the model achieves a higher Software Diagnostic F1-Score by maintaining a balanced trade-off between precision and recall. The Software Reliability Index is improved through consistent and stable defect detection across diverse datasets, ensuring robustness in prediction. The enhanced Defect Sensitivity Range enables the model to identify both frequent and rare defects effectively, while the reduction in False Bug Generation Rate minimizes false alarms by filtering redundant or noisy features. Additionally, the system demonstrates superior Computational Execution Efficiency due to optimized model convergence and reduced training complexity. Overall, the framework achieves higher accuracy in defect rate and defect detection recall, providing a more reliable, scalable, and efficient solution for intelligent software defect prediction.

V. CONCLUSION

In conclusion, optimized deep learning–based software defect prediction framework effectively enhances the accuracy, adaptability, and reliability of software defect detection by integrating intelligent preprocessing, feature optimization, and advanced classification mechanisms. Traditional approaches often struggle with imbalanced datasets, redundant features, and weak correlation among software attributes, leading to poor defect prediction outcomes. The incorporation of ZSLT ensures noise-free and normalized data, while the CSMOTE-ENN method successfully balances the dataset, preserving essential feature diversity. The GWO-DNN combination demonstrates remarkable capability in selecting optimal features, reducing dimensionality, and improving feature relevance for defect prediction. Furthermore, the Code2Vector-GCNN model efficiently captures both the semantic and structural characteristics of source code, leading to superior classification performance.

Experimental evaluations conducted on benchmark software defect datasets reveal that the proposed framework outperforms existing machine learning and traditional deep learning models in all key performance metrics. The model achieved an average precision of 97.6%, recall of 96.9%, and an F1-score of 97.2%, demonstrating its robustness and high predictive capability. The optimized deep feature classification approach notably improved the true positive rate while minimizing false alarms, ensuring more reliable detection of defect-prone modules

VI. REFERENCES

[1] D. -L. Miholca, V. -I. Tomescu and G. Czibula, "An in-Depth Analysis of the Software Features' Impact on the Performance of Deep Learning-Based Software Defect Predictors," in IEEE Access, vol. 10, pp. 64801-64818, 2022, doi: 10.1109/ACCESS.2022.3181995.

[2] Khalid, A., Badshah, G., Ayub, N., Shiraz, M., & Ghouse, M. (2022). Software Defect Prediction Analysis Using Machine Learning Techniques. Sustainability, 15(6), 5517. https://doi.org/10.3390/su15065517

[3] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," in IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 4, pp. 2244-2258, 1 July-Aug.
2022, doi: 10.1109/TDSC.2021.3051525.

[4] F. Alghanim, M. Azzeh, A. El-Hassan and H. Qattous, "Software Defect Density Prediction Using Deep Learning," in IEEE Access, vol. 10, pp. 114629-114641, 2022, doi: 10.1109/ACCESS.2022.3217480.

[5] Elshamy, N., AbouElenen, A., & Elmougy, S. (2023). Automatic detection of software defects based on machine learning. International Journal of Advanced Computer Science and Applications, 14(3).

[6] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. ACM Trans. Softw. Eng. Methodol. 30, 3, Article 38 (July 2021), 33 pages. https://doi.org/10.1145/3436877

[7] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: a novel deep learning-based approach for automated program repair. In Proceedings of the 44th International Conference on Software Engineering (ICSE '22). Association for
Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[8] Nevendra, M., & Singh, P. (2021). Software defect prediction using deep learning. Acta Polytechnica Hungarica, 18(10), 173-189.

[9] Akimova, E. N., Bersenev, A. Y., Deikov, A. A., Kobylkin, K. S., Konygin, A. V., Mezentsev, I. P., & Misilov, V. E. (2021). A survey on software defect prediction using deep learning. Mathematics, 9(11), 1180.

[10] Liu, W., Wang, B., & Wang, W. (2020). Deep Learning Software Defect Prediction Methods for Cloud Environments Research. Scientific Programming, 2021(1), 2323100. https://doi.org/10.1155/2021/2323100

[11] Batool, I., Khan, T.A. Software fault prediction using deep learning techniques. Software Qual J 31, 1241–1280 (2023). https://doi.org/10.1007/s11219-023-09642-4

[12] Meng, F., Cheng, W., & Wang, J. (2021). Semi-supervised software defect prediction model based on tri-training. KSII Transactions on Internet & Information Systems, 15(11).

[13] Liu, J., Pan, C., Lei, F., Hu, D., & Zuo, H. (2021). Fault prediction of bearings based on LSTM and statistical process analysis. Reliability Engineering & System Safety, 214, 107646. https://doi.org/10.1016/j.ress.2021.107646

[14] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22). Association for Computing Machinery, New York, NY, USA, 596–607. https://doi.org/10.1145/3524842.3527949

[15] Borandag, E. (2022). Software Fault Prediction Using an RNN-Based Deep Learning Approach and Ensemble Machine Learning Techniques. Applied Sciences, 13(3), 1639. https://doi.org/10.3390/app13031639

[16] Kaur, Gurmeet; Pruthi, Jyoti; and Gandhi, Parul (2023) "Machine learning based Software Fault Prediction models," Karbala International Journal of Modern Science: Vol. 9 : Iss. 2 , Article 9. Available at: https://doi.org/10.33640/2405-609X.3297

[17] Alkaberi, W., & Assiri, F. (2024). Predicting the Number of Software Faults using Deep Learning. Engineering, Technology & Applied Science Research, 14(2), 13222–13231. https://doi.org/10.48084/etasr.6798

[18] Abboush, M., Bamal, D., Knieke, C., & Rausch, A. (2021). Intelligent Fault Detection and Classification Based on Hybrid Deep Learning Methods for Hardware-in-the-Loop Test of Automotive Software Systems. Sensors, 22(11), 4066. https://doi.org/10.3390/s22114066

[19] S. Sahar, M. Younas, M. M. Khan and M. U. Sarwar, "DP-CCL: A Supervised Contrastive Learning Approach Using CodeBERT Model in Software Defect Prediction," in IEEE Access, vol. 12, pp. 22582-22594, 2024, doi: 10.1109/ACCESS.2024.3362896.

[20] A. M. Ismail, S. H. A. Hamid, A. A. Sani and N. N. M. Daud, "Toward Reduction in False Positives Just-In-Time Software Defect Prediction Using Deep Reinforcement Learning," in IEEE Access, vol. 12, pp. 47568-47580, 2024, doi: 10.1109/ACCESS.2024.3382991.

[21] F. Alghanim, M. Azzeh, A. El-Hassan and H. Qattous, "Software Defect Density Prediction Using Deep Learning," in IEEE Access, vol. 10, pp. 114629-114641, 2022, doi: 10.1109/ACCESS.2022.3217480.

[22] Q. -Y. Zou, Z. -Y. Yang, X. -R. Qiu, J. -H. Yu, Y. -Y. Shi and N. Chen, "Counterfactual Contrastive Explanations for Software Defect Prediction: Toward Better Model Understanding and Accuracy," in IEEE Transactions on Reliability, doi: 10.1109/TR.2025.3611079.

[23] T. Tahir et al., "Early Software Defects Density Prediction: Training the International Software Benchmarking Cross Projects Data Using Supervised Learning," in IEEE Access, vol. 11, pp. 141965-141986, 2023, doi: 10.1109/ACCESS.2023.3339994.

[24] X. Hu, M. Zheng, R. Zhu, X. Zhang and Z. Jin, "Fed-OLF: Federated Oversampling Learning Framework for Imbalanced Software Defect Prediction Under Privacy Protection," in IEEE Transactions on Reliability, vol. 74, no. 3, pp. 3266-3280, Sept. 2025, doi: 10.1109/TR.2024.3524064.

[25] Z. Li, H. Zhang, X. -Y. Jing, W. Yu and Y. Liu, "Unsupervised Software Defect Prediction Through Multiview Clustering," in IEEE Transactions on Reliability, vol. 74, no. 3, pp. 3356-3370, Sept. 2025, doi: 10.1109/TR.2025.3548107.

[26] R. Ghafoor Hussain, K. -C. Yow and M. Gori, "Leveraging an Enhanced CodeBERT-Based Model for Multiclass Software Defect Prediction via Defect Classification," in IEEE Access, vol. 13, pp. 24383-24397, 2025, doi: 10.1109/ACCESS.2024.3525069.

[27] D. -L. Miholca, V. -I. Tomescu and G. Czibula, "An in-Depth Analysis of the Software Features' Impact on the Performance of Deep Learning-Based Software Defect Predictors," in IEEE Access, vol. 10, pp. 64801-64818, 2022, doi: 10.1109/ACCESS.2022.3181995.

[28] X. Dong, J. Wang and Y. Liang, "A Novel Ensemble Classifier Selection Method for Software Defect Prediction," in IEEE Access, vol. 13, pp. 25578-25597, 2025, doi: 10.1109/ACCESS.2025.3537658.

[29] M. Yang, S. Yang and W. E. Wong, "Multi-Objective Software Defect Prediction via Multi-Source Uncertain Information Fusion and Multi-Task Multi-View Learning," in IEEE Transactions on Software Engineering, vol. 50, no. 8, pp. 2054-2076, Aug. 2024, doi: 10.1109/TSE.2024.3421591.

[30] H. Liu, Z. Li, H. Zhang, X. -Y. Jing and J. Liu, "CFG2AT: Control Flow Graph and Graph Attention Network-Based Software Defect Prediction," in IEEE Transactions on Reliability, vol. 74, no. 3, pp. 3412-3426, Sept. 2025, doi: 10.1109/TR.2024.3503688.

[31] W. -C. Su and C. -Y. Huang, "Application of Weighted Combinations of Activation Functions to Defect Prediction in Software Development," in IEEE Transactions on Reliability, vol. 73, no. 1, pp. 680-694, March 2024, doi: 10.1109/TR.2023.3284857.

[32] Z. Li et al., "Cross Domain Few-Shot Line-Level Defect Prediction in Open Software Development via Meta Learning," in IEEE Transactions on Consumer Electronics, vol. 71, no. 2, pp. 2999-3015, May 2025, doi: 10.1109/TCE.2025.3572334.

[33] J. Xu, J. Ai, J. Liu and T. Shi, "ACGDP: An Augmented Code Graph-Based System for Software Defect Prediction," in IEEE Transactions on Reliability, vol. 71, no. 2, pp. 850-864, June 2022, doi: 10.1109/TR.2022.3161581.

[34] S. R. Goyal, "Current Trends in Class Imbalance Learning for Software Defect Prediction," in IEEE Access, vol. 13, pp. 16896-16917, 2025, doi: 10.1109/ACCESS.2025.3532250.

[35] M. Begum et al., "LCNN: Lightweight CNN Architecture for Software Defect Feature Identification Using Explainable AI," in IEEE Access, vol. 12, pp. 55744-55756, 2024, doi: 10.1109/ACCESS.2024.3388489.

[36] H. Tong et al., "MASTER: Multi-Source Transfer Weighted Ensemble Learning for Multiple Sources Cross-Project Defect Prediction," in IEEE Transactions on Software Engineering, vol. 50, no. 5, pp. 1281-1305, May 2024, doi: 10.1109/TSE.2024.3381235.

[37] P. N. Srinivasu, M. Sailaja, S. C. Narahari, P. Barsocchi and A. K. Bhoi, "XAI Driven Software Defect Prediction Using Adaptive Feature Engineering Coupled With Autoencoder and Multi-Layer Perceptron: An Empirical Study," in IEEE Access, vol. 13, pp. 168693-168710, 2025, doi: 10.1109/ACCESS.2025.3603451.

[38] H. Andrade, N. Pombo and S. Pais, "Improving Software Defect Detection With LSTM-Based Semantic Modeling and Class Imbalance Handling," in IEEE Open Journal of the Computer Society, vol. 6, pp. 1501-1511, 2025, doi: 10.1109/OJCS.2025.3613134.

[39] S. Qiu, H. Huang, W. Jiang, F. Zhang and W. Zhou, "Defect Prediction via Tree-Based Encoding with Hybrid Granularity for Software Sustainability," in IEEE Transactions on Sustainable Computing, vol. 9, no. 3, pp. 249-260, May-June 2024, doi: 10.1109/TSUSC.2023.3248965.

[40] G. Denaro, R. Heydarov, A. Mohebbi and M. Pezzè, "Prevent: An Unsupervised Approach to Predict Software Failures in Production," in IEEE Transactions on Software Engineering, vol. 49, no. 12, pp. 5139-5153, Dec. 2023, doi: 10.1109/TSE.2023.3327583.