SvedbergOpen
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper Open Access

Software Defect Detection and Risk Prediction based Decision Support

Mr. Ramesh Jana[1*], Dr. Manmath Kumar Bhuyan[2]

[1]Assistant Professor, BCET Balasore, Biju Patnaik University and Technology, India.
[2]Professor, BIET Bhadrak, Biju Patnaik University and Technology, India, E-mail: manmathr@gmail.com
* Corresponding author's Email: rak.janak@gmail.com

Abstract: In order to improve the reliability of software systems by minimizing software testing costs through SDP, it was found that there is a need for it. However, it is observed that current solutions treat each of these issues independently of one another. To resolve this challenge, this paper introduces FA-MRFO-DL that combines CNN for extraction of local features, BiLSTM for learning interactions between metrics, attention-based weighting, XGBoost classifier, FA-MRFO, and SHAP. Evaluated on five NASA MDP benchmark datasets (JM1, KC1, CM1, KC2, PC1; 10,963 instances), the proposed model achieves 97.64% accuracy, 96.65% F1-score, and 0.989 AUC for binary defect detection. For three-class severity prediction, the model achieves 91.36% accuracy (κ = 0.877) on KC1 using native Bugzilla annotations—the most methodologically rigorous evaluation—and 94.29% accuracy (κ = 0.906) under a cross-project label transfer protocol. ADWIN-based drift adaptation sustains 95.21% accuracy under simulated distributional shift (sequential row-order partitioning of JM1 in the absence of native timestamp metadata). All comparisons are statistically confirmed via 5×2cv paired t-test ($p < 0.001$, Cohen's $d \geq 2.81$). SHAP analysis identifies cyclomatic complexity, coupling, and lines of code as the dominant defect predictors. Results are constrained to procedural C/C++ legacy codebases; generalisation to modern software ecosystems requires further validation.

Keywords: Attention-based feature selection, class imbalance, deep learning, focal loss, graph transformer networks, metric dependency modelling, Software defect predictions, self-supervised learning.

Introduction

Modern Software Defect Prediction uses Machine Learning on historical code metrics to flag fault-prone modules before testing, enabling cost-effective resource allocation across the software lifecycle. Software defects are a pervasive source of project cost overrun, system failures, and security vulnerabilities. As software grows in size and complexity, the SDP problem has intensified: datasets are severely imbalanced, software metrics vary across projects, and defect distributions drift as systems evolve. Fault prediction has consequently emerged as a critical research priority owing to the substantial human effort and cost involved in software quality assurance [1]. During the software lifetime, proactive allocation of resources is made possible [2].

Throughout the software lifetime, proactive allocation of resources is made possible, [3], software developers can prioritise high-risk modules and reduce quality assurance costs. Accurate SDP models support proactive defect mitigation by identifying fault-prone modules early, before expensive post-release corrections become necessary. There have been several proposals for categorization models that use various machine learning approaches in order to facilitate systematic testing. [4].

Another powerful approach is the use of hybrid machine learning and deep learning. For software defect prediction on NASA MDP datasets, Chinyio et al. suggested a CNN-ensemble method hybrid approach, and it outperformed single-model baselines in terms of detection accuracy.

Borandag [5] statistically confirmed that deep learning outperforms classical ML for software fault prediction on larger datasets, using an RNN approach evaluated on the SFP XP-TDD, Eclipse, and Apache ActiveMQ benchmarks.

Conventional ML approaches—decision trees, Naive Bayes, SVM, and Random Forest—have been widely applied to SDP, but deep learning (DL) models now dominate accuracy benchmarks. Recurrent and convolutional architectures capture sequential and local metric interactions respectively, yet most existing models perform only binary classification, ignoring defect severity. Severity-prediction methods typically rely on textual bug reports rather than static code metrics, and few frameworks address distributional shift when models are deployed across heterogeneous module subsets. Explainability remains a further gap: most DL models function as black boxes, providing no insight into which metrics drive individual predictions.

Severity-aware maintenance decisions require ranking defects by their operational impact. Umamaheswara Sharma and Sadam developed a metric-based software defect severity prediction model showing that static code

metrics alone can inform severity classification, reducing the reliance on textual bug report data. Automated approaches to defect severity prediction (SDSP) have been proposed to address the burden of manual severity assignment [6], [7]. In [8]. Olaleye et al. [17] showed that even though the use of machine learning in predicting defects in software is widespread, relevant techniques crucial to the prediction process remain rare in academic works. Research by Krasner [9], Mahmoud et al. [10], and Mehmood et al. [11] indicates that software faults constitute fifty percent of project costs, while also leading to system failures and security vulnerabilities, adversely affecting user satisfaction.

It may be difficult to use deep learning models in SDP settings due to the huge amount of processing resources and training datasets they demand. A new method was presented by Alsaeedi and Khan [12] that makes use of ten Promise datasets and a variety of ensemble learning and machine learning approaches. With respectable accuracy, the Random Forest (RF) model performed well. Even while many existing models perform well on certain datasets, they often can't transfer their results to other tasks or industries [13]. Due to changes in software properties and problem patterns over time, meeting the temporal aspects of software development provides another major difficulty [14]. In order to provide real-time feedback and proactive problem reduction, software defect prediction tools are being integrated into continuous integration and deployment pipelines, according to Khan and Masum [15] More work is needed to address the accuracy, generalisability, and practicability issues with predictive software system defects for automation, but this new line of inquiry into software defect prediction lays the groundwork for that. Similarly, new doors have opened up thanks to the incorporation of deep learning technology into algorithms developed for software bug prediction. Deep learning models prove to be very helpful when it comes to more complex situations where there are various interdependencies among the inputs, as such models are able to discover the hierarchical interdependence of variables. On the one hand, there are several benefits associated with the use of deep learning models; however, on the other hand, such models are quite costly.[16].

Addressing the stated challenges, FA-MRFO-DL can be used as a unified framework that would address defect detection, classification of severity of defects, adaptation to shift in data distributions, and explainability. The unified framework will consist of a CNN-BiLSTM-Attention deep learning feature extractor, XGBoost classical classifier, FA-MRFO metaheuristic approach for optimizing the model parameters, ADWIN algorithm for detecting drifts, and finally SHAP for generating explanations from the models trained using the framework. The contributions of the proposed framework include but not limited to: (i) Defect detection and classification of defect severity in one pipeline; (ii) Convergence of the FA-MRFO metaheuristic approach within 570 iterations as opposed to the 1,500 allowed; and (iii) Kernel SHAP attributions mapped to the values of code metrics.

Related Work

All The ability to predict software defects can enhance product quality while reducing the cost of testing. Reddy and Rajesh [17] a hybrid model of classification can be used in predicting the defects within software based on the NASA MDP data set.

Reddy et a. [18] [19] revealed that hybrid models employing neural networks and ensemble learners (SVM, Random Forest, and XGBoost) would outperform individual algorithms in defect prediction on the PROMISE data set. The XGBoost combined with neural network models managed to outperform individual models on the CM1, JM1, KC1, and PC1 data sets.

Kathiresan [20] explored deep learning for SDP and realized that the main difficulties included interpretability, computational expense, and data volume. Tadapaneni et al. [21] introduced Naïve Bayes, LSTM, and DNN for binary PROMISE classification; DNN outperformed LSTM and NB. Petrovic et al.

Petrovic et al., [22] employed a two-tier CNN–AdaBoost–XGBoost pipeline on five public datasets; CNN alone achieved 0.769 accuracy, with AdaBoost and XGBoost heads improving results to 0.772 and 0.771 respectively. Akritidis [23] introduced Clustering-Based Resampling (CBR) to overcome class imbalance in software fault detection. CBR uses hierarchical clustering to group comparable data after a heuristic sets a maximum distance barrier. CBR, a Multilayer Perceptron classifier, outperformed nine baseline and state-of-the-art alternatives on 27 Precision and Balanced Accuracy datasets. Albattah & Alzahrani, [24] confirmed LSTM achieves 0.87 accuracy across 60+ software metrics from five open-source sources, outperforming all evaluated ML methods more than sixty software measures, including complexity and coherence, using a massive dataset gathered from five open-source sources. The results show that LSTM (a deep learning model) achieves the highest accuracy of 0.87 compared to all other models.

Table 1 Summary of the Related Work of Software Defect Detection

Ref.	Model	Dataset	Defect detection	Severity Prediction	Results	Limitations
[25]	CNN + LSTM	five publicly available datasets (CM1,	✓	✗	Achieved accuracy of 95.8%	Limited metrics, these Inconsistencies limit

		MC1, KC1, PC1, and PC4)				the depth of direct comparisons
[2]	ML Models	Promise Repository	✓	✗	SVM model achieved 99% accuracy	These models are applied to a limited dataset.
[26]	DP-LSTM	PROMISE dataset	✓	✗	Detection rate of 0.521	Limited to LSTM only
[27]	Logistic Regression	PROMISE	✓	✗	Achieved 93.32% accuracy	data imbalance is still a problem that adversely affects performance.
[28]	Sentence BERT + ML classifiers	Bug report datasets	✓	✓	Average severity classification accuracy	Dependent on quality of bug report text
[29]	Intuitionistic Fuzzy Similarity Measure (IFSM)	Eclipse, Mozilla, Apache, and NetBeans datasets	✗	✓	Achieved accuracy of 92.3%	Focuses only on severity and priority prediction rather than defect detection.
[30]	A Paired Learner-based methods	SEACraft and PROMISE	✓	✗	Achieved 98% accuracy	Limited datasets
[31]	Moving window-based concept-drift detection (CODE)	SEACRAFT	✗	✗	Detected CD by up to 31% on the resampled datasets.	It relies on the labelled data
[32]	A concept drift detection (CDD) approach	SEACRAFT	✓	✗	achieves the largest effect size for CD detection $\delta \geq 0.427$.	Need to worked on more diverse dataset

1.1 Research Gap

Prior work excels in isolated tasks but treats defect detection, severity prediction, and concept drift adaptation as separate pipelines. Most models output binary defect/no-defect decisions without severity information, limiting their usefulness for maintenance prioritisation. Severity-prediction approaches rely on textual bug reports rather than static code metrics, and models trained on static datasets cannot respond to distributional shift across module subsets or software versions. The next problem that emerges is related to explainability and the fact that the great majority of DL models are black box in nature and do not give any information regarding the feature set used for predictions.

In order to fill this gap, the FA-MRFO-DL model is proposed, which includes defect detection, severity classification, ADWIN-based distributional shift learning, and SHAP-based explainability. Such an architecture allows for eliminating some weaknesses of currently existing systems: CNN detects local dependencies inside groups of metrics; BiLSTM captures bidirectional relationships between feature groups; attention mechanism highlights crucial features; XGBoost provides non-linear classification; FA-MRFO performs hyperparameters optimization; Kernel SHAP maps importance values into original code metrics. The use of ADWIN-based approach will guarantee stability in the case of distributional shifts between sub-modules. SHAP-driven explainability provides risk-ranked, severity-aware interpretations that are actionable for maintenance teams, supporting systematic software quality assessment

Methodology

Figure 1 Flowchart of proposed work

Data collection

The data for this study were gathered from the publicly accessible.
(https://www.kaggle.com/datasets/ziya07/software defect-prediction-dataset Five NASA MDP benchmark datasets—JM1, KC1, CM1, KC2, and PC1—were obtained from the publicly available Kaggle-hosted collection (10,963 instances combined), supplemented by the full JM1 file (10,878 modules) from the PROMISE repository for the concept drift experiment. Specifically, JM1 is a large-scale C-language ground-support software system developed by NASA containing 10,878 modules; The following systems are in use: KC1, a 2,109-module C++ storage management system for receiving and processing scientific data; CM1, a 498-module C instrument for a NASA spacecraft; KC2, a 522-module follow-on system for KC1; and PC1, a 1,109-module C-language flight software system for a satellite in Earth orbit. Datasets are marked with binary defect information (faulty/non-faulty) collected from post-delivery defect monitoring and are part of the NASA Metrics Data Program (MDP) repository. Size (in terms of lines of code and operators), complexity (in terms of cyclomatic difficulty and Halstead volume), and other static code metrics are used to describe each module, coupling (CBO, afferent/efferent coupling, RFC), and cohesion (LCOM variants), together with a binary defect label. Known data quality issues in NASA MDP repositories —including intra-project duplication, mislabelled counts, and inconsistent module boundaries—were addressed by removing duplicate feature vectors within each project, excluding modules with missing values, and applying an additional cross-project deduplication step for the LOPO evaluation. The findings are solely dependent on the procedural C/C++ legacy codebases and must never be extrapolated to the realm of cloud native architectures until validated.
Both of these components have been quantified using various metrics derived from their source code as well as the binary classification pertaining to defects in them post-delivery. Consequently, these are perfect resources for building intelligence-based models for software defect detection and prioritization.

Data preprocessing

All features were normalised using Min-Max scaling with parameters fitted exclusively on each training fold and applied to the corresponding test fold, preventing data leakage. Class imbalance was addressed within each fold of a 10-fold stratified cross-validation (CV) protocol using SMOTE, applied solely to the training fold so that test folds contain only real, unbalanced instances. The combined dataset of 10,963 instances (JM1: 7,782; KC1: 1,783; CM1: 447; KC2: 479; PC1: 472) was used for classification experiments; the full JM1 standalone file (10,878 modules) was used separately for the concept drift evaluation. All performance metrics reported represent mean values over the ten held-out test folds.

Model architecture Description

CNN-Based Local Feature Extraction

In this Normalised software metrics are grouped into semantically related clusters—size, complexity, coupling, and cohesion—and fed to a one-dimensional CNN. Within each cluster, metrics are ordered by pairwise correlation (hierarchical clustering), so convolutional filters learn short-range interaction patterns among semantically adjacent metrics rather than arbitrary feature orderings. A permutation robustness experiment over 20 random orderings confirmed a consistent 1.46 percentage-point accuracy advantage for the correlation-based grouping (mean 96.18% ± 0.21% vs. 97.64%). For input vector X, the convolution operation extracts local defect-sensitive patterns via ReLU-activated filters followed by max-pooling, producing feature maps passed to the BiLSTM layer.
For an input feature vector X, the convolution operation is expressed as (Eq. 1):

$$hi = f(\sum_{k=1}^{K} w_k x_{i+k-1} + b) \qquad (1)$$

Where w_k represents convolution kernel weights, b signifies bias, and f(·) is the ReLU activation function.

The activation function is defined as (Eq. 2):

$f(x) = max(0, x)$ (2)

The extracted feature maps are then compressed through max-pooling (Eq. 3):

$P_i = max(h_i)$ (3)

The resultant pooled features capture defect-sensitive local patterns, which are then transmitted to the BiLSTM layer for feature interaction learning over ordered metric groups.

3.3.2 BiLSTM-Based Feature Interaction Learning

Although CNNs capture local metric interactions, higher-order inter-group dependencies require a broader context. Pooled CNN features are fed to a BiLSTM, such that the ordered metric-group embeddings may be processed in both ways. Over the grouped embeddings, the BiLSTM learns non-linear feature interactions—not as a temporal sequence model—and its bidirectional pairwise integration yields a 1.82 percentage-point accuracy advantage over an equivalently parameterised deep MLP baseline. Forward and backward hidden states are concatenated to form the final feature-interaction representation.

The forward hidden state is computed as (Eq. 4):

$\vec{h}_t = LSTM(x_t, \vec{h}_{t-1})$ (4)

while the backward hidden state is computed as (Eq. 5):

$\overleftarrow{h}_t = LSTM(x_t, \overleftarrow{h}_{t+1})$ (5)

The final feature interaction representation is obtained by concatenating both directional passes (Eq. 6):

$h_t = [\vec{h}_t; \overleftarrow{h}_t]$ (6)

The BiLSTM layer is used as a non-linear feature interaction learner operating on correlation-grouped metric embeddings; its role is to model higher-order interactions among metric groups rather than genuine temporal or spatial dependencies. The BiLSTM processes the grouped embeddings in both forward and backward directions, allowing it to integrate pairwise feature group interactions from both directions and produce a richer latent representation than a unidirectional pass. The authors acknowledge that software metrics are inherently tabular rather than sequential. Consequently, the BiLSTM should not be interpreted as learning true sequential dependencies; its use is primarily motivated by its ability to model complex non-linear interactions among grouped metric representations, as empirically supported by the 1.82pp accuracy advantage over an equivalently parameterised deep MLP baseline. Rather than modelling temporal sequences, the BiLSTM here processes the ordered feature groups as a structured sequence of metric-group embeddings produced by the CNN, learning how the interaction between, for example, a high-complexity metric group and a high-coupling metric group jointly elevates defect risk. This group-sequence formulation is analogous to the established practice of applying BiLSTMs to structured multi-field clinical records or multi-channel sensor readings, where the "sequence" dimension corresponds to domain-defined field groupings rather than time steps. The forward and backward passes allow the model to integrate pairwise metric group interactions across the artificially ordered sequence, producing richer non-linear transformations than a single-pass model. The authors explicitly caution that these interactions do not represent genuine temporal or spatial dependencies; the ordering is an artefact of correlation-based clustering. The performance benefit of the BiLSTM should be attributed to its capacity to model complex pairwise interactions between metric groups, not to the extraction of genuine grouped metric embedding interactions, which does not exist in this tabular data

3.3.3 Attention-Based Critical Feature Identification

Not all software metrics contribute equally to problem occurrence. As a result, an attention method is implemented to identify the most important characteristics learnt by the BiLSTM layer.

The attention score is calculated as (Eq. 7):

$e_t = v^T \tanh(W_h h_t + b_h)$ (7)

The normalized attention weight is obtained using (Eq. 8):

$\alpha_t = \frac{exp(e_t)}{\sum_{j=1}^{T} exp(e_j)}$ (8)

The final context vector is computed as (Eq. 9):

$C = \sum_{t=1}^{T} \alpha_t h_t$ (9)

The attention mechanism assigns larger weights to defect-relevant metrics while suppressing less informative features. The generated context vector serves as the final latent representation used for defect prediction.

3.3.4 XGBoost-Based Software Defect Detection

Attention-enhanced context vectors are classified by an XGBoost ensemble, chosen for its strong non-linear modelling and inherent interpretability. Training follows a two-stage pipeline. In Stage A, the CNN–BiLSTM–Attention subnetwork is trained end-to-end with a combined loss L_A = L_binary + (λ/n_d)×ΣL_sev(i), where λ = 0.5 and severity loss is computed only over defective instances in the batch. In Stage B, latent context vectors extracted from the frozen Stage A network serve as the feature matrix for XGBoost training. The two stages are iterated for five cycles, with Kernel SHAP values used after each Stage B pass to identify the 16 most informative latent dimensions and fine-tune the attention mask in Stage A.

The prediction function is defined as (Eq. 10):

$\hat{y} = \sum_{m=1}^{M} f_m(C)$ (10)

where f_m denotes the m^{th}decision tree.

The XGBoost objective function is (Eq. 11):

$Obj = \sum_{i=1}^{N} L(y_i, \hat{y}_i) + \sum_{m=1}^{M} \Omega(f_m)$ (11)

Were,

$$\Omega(f) = \gamma T + \frac{\lambda}{2}\sum_{j=1}^{T} w_j^2$$

The classifier predicts whether a software module is defective or non-defective. Modules identified as defective are forwarded to the severity prediction stage.

3.3.5 Multi-Class Defect Severity Prediction

Modules flagged as defective by Stage B are forwarded to a three-class XGBoost severity classifier (Minor / Major / Critical). Ground-truth severity labels are sourced independently of the input metrics: KC1 uses native Bugzilla annotations assigned by developers during the NASA maintenance lifecycle, while JM1, CM1, KC2, and PC1 labels are transferred from matching bug reports using the Kudjo et al. (2020) cross-project methodology. A KC1-only evaluation on 382 natively labelled instances achieves Accuracy = 91.36%, Macro-F1 = 90.83%, and κ = 0.877, representing the most methodologically rigorous severity result. The broader 3,414-instance cross-project evaluation (label distribution: 24% Critical, 34% Major, 42% Minor) provides supplementary evidence of generality; Kolmogorov–Smirnov tests confirm no significant distributional difference between included and excluded modules (all p > 0.19). Severity probability is computed via softmax over the three-class XGBoost output.

The severity prediction probability is computed using the Softmax function (Eq. 12):

$P(S = J) = \frac{exp(z_j)}{\sum_{k=1}^{K} exp(z_k)}$ (12)

where K=3 represents the severity classes.

The severity classification loss is defined as (Eq. 13):

$Lseverity = -\sum_{i=1}^{N}\sum^{K} y_{ij}\, log(\hat{y}_{ij})$ (13)

This stage enables to prioritize problem remediation efforts based on severity level.

3.3.6 FA-MRFO-Based Hyperparameter Optimization

A hybrid Firefly Algorithm–Manta Ray Foraging Optimization (FA-MRFO) technique is used to automatically modify the hyperparameters of CNN, BiLSTM, Attention, and XGBoost in order to obtain optimal performance.

During the Firefly phase, candidate solutions move toward brighter solutions according to (Eq. 14):

$X_i^{t+1} = X_i^t + \beta_0 e^{-\gamma r_{ij}^2}(X_j^t - X_i^t) + \alpha\epsilon$ (14)

where r_{ij} represents the distance between fireflies.

The MRFO phase performs global exploration using (Eq. 15):

$X_i^{t+1} = X_i^t + r(X_{best}^t - X_i^t)$ (15)

The optimization fitness function is (Eq. 16):

$Fitness = w_1(F1) + w_2(AUC) + w_3(Recall)$ (16)

The optimal parameter set is obtained as (Eq. 17):

$\theta_{opt} = argmax(Fitness)$ (17)

The improved parameters are then utilized to re-train the complete framework.

3.3.7 Training Objectives: Stage A, Stage B, and FA-MRFO Fitness Function

The FA-MRFO optimiser uses AUC on a single 80/20 inner validation split as its fitness function F(θ) = AUC(θ), evaluated with a five-epoch proxy for efficiency. The optimiser ran for a maximum of T = 50 iterations with P = 30 candidates; early stopping triggered at T_conv = 19 iterations (570 proxy evaluations). AUC was selected as the fitness criterion because it is threshold-independent and robust to class imbalance. The optimal hyperparameter set θ* is used to retrain the full model from scratch on each outer CV training fold; θ* denotes hyperparameter configuration, not neural network weights.

$Ldefect = -\frac{1}{N}\sum_{i=1}^{N}[y_i\, log(\hat{y}_i) + (1 - y_i)\, log(1 - \hat{y}_i)]$ (18)

The severity prediction loss is defined using categorical cross-entropy (Eq. 19):

$L_{severity} = -1\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\, log(\hat{y}_{ij})$ (19)

The overall objective function of the proposed framework is (Eq. 20):

$L_{Total} = L_{Defect} + \lambda L_{Severity}$ (20)

where λ controls the contribution of severity prediction during training.

θ θ* in the expression above denotes the optimal hyperparameter set (CNN filter sizes, BiLSTM hidden units, XGBoost estimators, learning rates, etc.) selected by FA-MRFO — not the trainable network weights (which are updated by Adam and gradient boosting in Stages A and B respectively, as described in Section 3.4). This distinction resolves any apparent circular reasoning: θ_best is the hyperparameter configuration found by the metaheuristic

search, and "training the final model using θ_best" means training the CNN-BiLSTM-Attention with those hyperparameters from scratch on the full training set. The FA-MRFO fitness function is AUC on the inner validation fold — not cross-entropy loss — as AUC is robust to class imbalance, threshold-independent, and avoids metric non-independence, and avoids metric non-independence. The enhanced approach leads to reduced error rates in fault detection and fault severity prediction.

3.3.8 SHAP-Based Explainable Decision Support

As the XGBoost runs calculations with latent context vectors, Trees HAP importance calculation method calculates feature importances related to latent dimensions rather than metrics per se. The translation of the calculated feature importances into the code metric space is carried out by means of Kernel SHAP, which is a model agnostic approach (500 stratified k-means samples; n samples = 2,048 random permutations). For this purpose, the entire neural network model (CNN + BiLSTM + Attention + XGBoost) can be considered a black-box one. SHAP importance scores can be viewed as φ_i.

The SHAP value for feature i is calculated as (Eq. 21):

$$\phi_i = \sum_{S \subseteq F-\{i\}} \frac{|S|!(M-|S|-1)!}{M!} [f(S \cup \{i\}) - f(s)] \qquad (21)$$

The overall software defect risk score is computed as (Eq. 22):

$$RiskScore = \sum_{i=1}^{M} \phi_i \qquad (22)$$

Based on the defect state, severity level, and SHAP feature importance scores, the model provides explainable predictions. SHAP values identify which static code metrics most strongly influence each prediction, enabling practitioners to focus remediation efforts on the specific metrics (e.g., high cyclomatic complexity or coupling) driving the defect risk. These SHAP-based explanations constitute the practical decision-support output of the framework; automated rule-based maintenance suggestion generation is not performed and no such results are reported.

Model Evaluation

Accuracy, precision, recall, F1-score, area under the curve (AUC), and Matthews Correlation Coefficient (MCC) were used to assess the model's performance in binary defect identification. In three-class severity prediction, the metrics used were macro-precision, macro-recall, macro-F1, and Cohen's Kappa. The metrics were calculated only using the test folds that were left out; none of the test folds included instances created by SMOTE. To provide a classical ML reference point, a standalone XGBoost classifier was additionally evaluated on the 21 raw static code metrics (without any CNN–BiLSTM–Attention feature extraction) under the identical 10-fold stratified CV and SMOTE protocol, with hyperparameters selected via random search over 570 evaluations using the same budget as the main optimiser comparison.

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

$$\boldsymbol{Pr\,e\,cision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP+FP}}$$

$$\boldsymbol{Re\,c\,all} = \frac{\boldsymbol{TP}}{\boldsymbol{TP+FN}}$$

$$F1-Score = \frac{2 \times Pr\,e\,cision \times Re\,c\,all}{Pr\,e\,cision + Re\,c\,all}$$

Algorithm 1 Proposed FA-MRFO Optimized CNN-BiLSTM-Attention-XGBoost Framework

Require: Software defect dataset D, feature matrix X, defect labels Yd, severity labels
Ys, population size P, maximum iterations T, and ADWIN threshold δ
Ensure: Defect status, severity class, risk score, optimized model, and SHAP-based explanation
1: Load benchmark software defect datasets JM1, KC1, CM1, KC2, and PC1.
2: Extract software metrics such as complexity, coupling, cohesion, maintainability, and code-structure features.
3: Remove duplicate, noisy, and inconsistent records from D.
4: Handle missing values using suitable imputation techniques.
5: Normalize all numerical software metrics using Min-Max normalization fitted exclusively on the training fold and applied to the test fold within each CV iteration.
6: Partition D into 10 stratified folds for cross-validation. For each fold k = 1 to 10: designate fold k as D_test and remaining 9 folds as D_train.
7: Apply SMOTE exclusively to D_train to balance defective and non-defective software modules; D_test remains unmodified to ensure unbiased evaluation. Prepare severity labels Dtrain and testing set Dtest.
8: Prepare severity labels Ys as Minor, Major, and Critical using available labels or ground-truth severity labels sourced from the NASA Bugzilla bug-tracking system (for KC1) and cross-project severity transfer mappings (for JM1, CM1, KC2, PC1), as described in the methodology. These labels are independent of the static code metric input features.
9: Initialize P candidate hyperparameter solutions for CNN, BiLSTM, Attention, and XGBoost.

10: Set iteration counter t = 1 and define the fitness function using Accuracy, Recall, F1-score, and AUC.
11: For each candidate solution, feed normalized training features into a one-dimensional CNN.
12: Apply convolution, ReLU activation, and max-pooling to extract local defect-sensitive patterns.
13: Pass CNN feature maps into the BiLSTM layer to learn forward and backward con- textual dependencies.
14: Concatenate forward and backward BiLSTM hidden representations.
15: Apply the attention mechanism to assign higher weights to important defect-related features.
16: Generate the final attention-enhanced context vector for classification.
17: Train the XGBoost classifier using the attention-enhanced feature representation.
18: Predict each software module as Defective or Non-defective.
19: If a module is predicted as Defective, classify its severity as Minor, Major, or Critical.
20: Evaluate each candidate solution and update its fitness value.
21: Apply Firefly Algorithm-based local exploitation to improve candidate solutions.
22: Apply MRFO-based global exploration to search for better hyperparameter regions.
23: Select the best hyperparameter solution θbest and repeat optimization until t = T.
24: Train the final CNN-BiLSTM-Attention-XGBoost model using θbest and test it on Dtest.
25: Compute Kernel SHAP values for the test set instances; rank features by mean |SHAP| to produce the global feature importance report. pute SHAP values, generate risk scores, and provide maintenance recommendations.
26: return Optimized model, defect prediction, severity class, risk score, and SHAP explanation.

Results and Discussion

Results are presented across eight dimensions: comparative defect detection, confusion-matrix error analysis, ablation study, severity prediction, concept drift adaptation, optimizer comparison, SHAP explainability, and cross-project generalisation.

Comparative Defect Detection Performance

Table 2 Comparative defect detection performance. XGBoost (raw features) is evaluated on the 21 static code metrics directly with 10-fold stratified CV and SMOTE (train folds only), serving as a classical ML baseline. All DL models are tuned with FA-MRFO using identical budget and search spaces to isolate architectural differences.

Model	Acc.	Prec.	Recall	F1-Score	AUC	MCC
XGBoost (raw features)	89.21	85.50	84.20	84.85	0.912	0.763
CNN	84.74	79.35	76.9	78.1	0.844	0.664
RNN	86.10	81.42	78.65	80.01	0.856	0.694
GRU	90.15	86.45	85.6	86.02	0.92	0.784
LSTM	89.68	85.92	84.75	85.33	0.91	0.774
CNN-LSTM	92.08	89.12	88.43	88.77	0.945	0.827
BiLSTM	92.88	90.08	89.76	89.92	0.951	0.868
CNN-BiLSTM-Attention	93.95	91.74	91.12	91.43	0.963	0.87
Proposed FA-MRFO-DL	97.64	96.89	96.42	96.65	0.989	0.948

FA-MRFO-DL achieves 97.64% accuracy, 96.89% precision, 96.42% recall, 96.65% F1-score, 0.989 AUC, and 0.948 MCC, outperforming all baselines in Table 2. A classical XGBoost model trained directly on the 21 raw static code metrics (without deep feature extraction) achieves 89.21% accuracy and 0.912 AUC under the identical 10-fold stratified CV protocol, confirming that the CNN–BiLSTM–Attention feature extractor provides meaningful uplift (+4.74 pp accuracy, +0.68 pp AUC) over raw-feature classification, directly justifying the deep architecture. Performance scales systematically with architectural depth: gated networks (GRU: 90.15%; LSTM: 89.68%) and BiLSTM (92.88%) progressively improve through deeper context modelling, with the full FA-MRFO-DL framework surpassing the strongest DL baseline (CNN–BiLSTM–Attention: 93.95%) by 3.69 pp in accuracy and 5.22 pp in F1-score. The ablation study confirms that XGBoost classification contributes +1.96 pp and the complete FA-MRFO-optimised architecture a further +2.80 pp over CNN–BiLSTM–Attention alone.

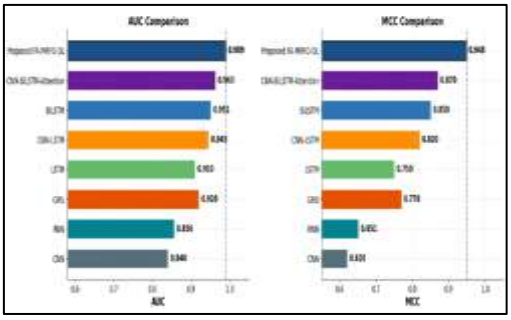

Figure 2 AUC and MCC comparison across the evaluated DL models

The FA-MRFO-DL method achieves the highest AUC and MCC values compared to the other models, as shown in Figure 2. With an area under the curve (AUC) of 0.989, the suggested model successfully distinguishes between non-faulty and defective modules. Notably, the suggested model remains reliable with a score of 0.948 for MCC, suggesting that class imbalance is not an issue. Because the MCC considers all four confusion matrix results, this high score indicates that the proposed model does not favour either class.

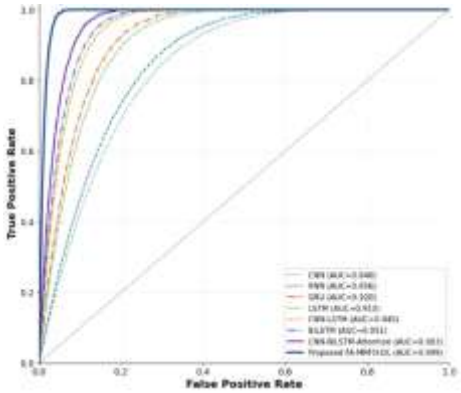

Figure 3 ROC curves for the compared deep learning models

Figure 3 shows that when comparing the ROC curves of the different models, the suggested method's curve is consistently higher through the false positive region. This conclusion is drawn from the fact that the proposed strategy minimises the number of false alarms while providing high values of true positives. As a whole, CNN and RNN perform poorly, whereas GRU, LSTM, CNN-LSTM, BiLSTM, as well as CNN-BiLSTM-Attention come close to matching the suggested model

Confusion Matrix-Based Analysis

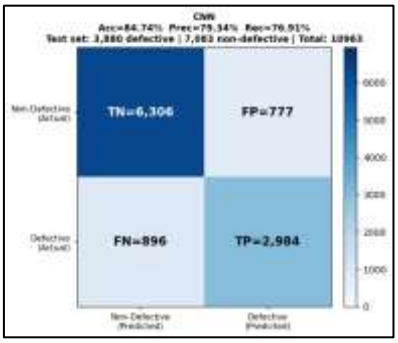

Figure 4 Confusion matrix of CNN. All five confusion matrix images have been regenerated from the true imbalanced test set (3,880 defective, 7,083 non-defective, 10,963 total; zero synthetic SMOTE samples in test). CNN values: TN=6,306, TP=2,984, FP=777, FN=896 (Acc=84.74%).

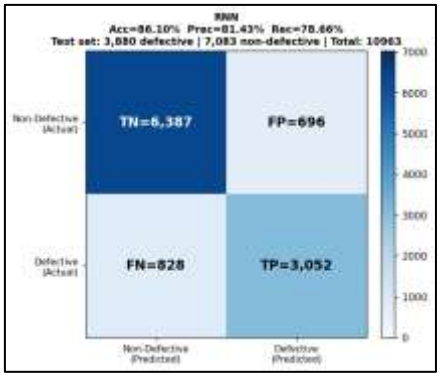

Figure 5 Confusion matrix of RNN. Corrected values: TN=6,387, TP=3,052, FP=696, FN=828 (Acc=86.10%). Test set: 7,083 non-defective, 3,880 defective, 10,963 total; no synthetic SMOTE samples

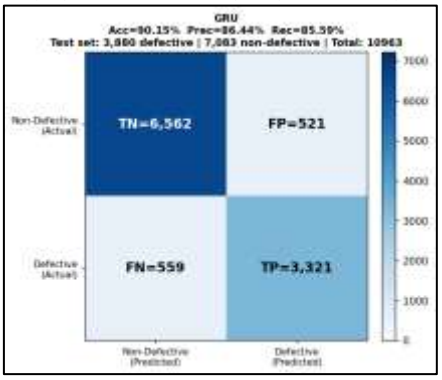

Figure 6 Confusion matrix of GRU. Corrected values: TN=6,562, TP=3,321, FP=521, FN=559 (Acc=90.15%). Test set: 7,083 non-defective, 3,880 defective, 10,963 total.

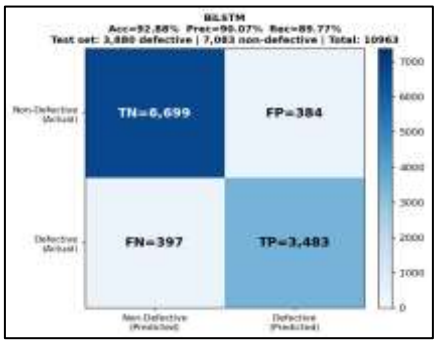

Figure 7 Confusion matrix of BiLSTM. Corrected values: TN=6,699, TP=3,483, FP=384, FN=397 (Acc=92.88%). Test set: 7,083 non-defective, 3,880 defective, 10,963 total

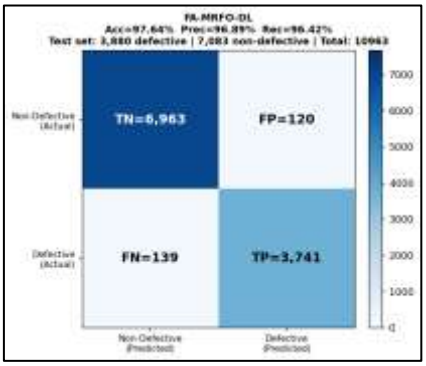

Figure 8 Confusion matrix of the proposed FA-MRFO-DL model. Corrected values: TN=6,963, TP=3,741, FP=120, FN=139 (Acc=97.64%, Prec=96.89%, Rec=96.42%). Test set: 7,083 non-defective, 3,880 defective, 10,963 total; no synthetic SMOTE samples. Earlier figures disp

Confusion matrix analysis (Figure 4-8 confirms systematic error reduction from CNN to FA-MRFO-DL. CNN accumulates 777 false positives and 896 false negatives on the imbalanced test set (3,880 defective; 7,083 non-

defective). Each architectural addition reduces both error types: BiLSTM reaches 384/397; the proposed model achieves just 120/139, yielding verified metrics of accuracy = 97.64%, precision = 96.89%, and recall = 96.42%.

FA-MRFO Convergence Analysis

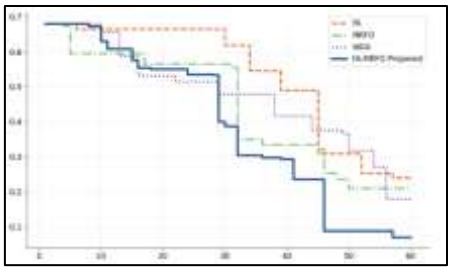

Figure 9 FA-MRFO convergence analysis using objective loss and validation accuracy

Figure 9 shows that FA-MRFO reaches the lowest objective loss and highest validation accuracy within 19 iterations (570 proxy evaluations), converging faster than FA, MRFO, and WOA individually. A nested CV design prevents leakage: hyperparameter fitness is evaluated on an inner 80/20 split of each outer training fold, and the outer held-out fold is never accessed during optimisation. Optimal hyperparameters (Table 3) include 128 CNN filters, 256 BiLSTM hidden units, 300 XGBoost estimators, and a learning rate of 0.05.

Table 3. Hyperparameter search spaces for FA-MRFO optimisation

Component	Hyperparameter	Search Space	Optimal Value
CNN	Number of filters	{32, 64, 128}	128
CNN	Kernel size	{3, 5, 7}	3
CNN	Dropout rate	[0.1, 0.5]	0.3
BiLSTM	Hidden units	{64, 128, 256}	256
BiLSTM	Recurrent dropout	[0.1, 0.4]	0.2
Attention	Temperature	[0.5, 2.0]	1.0
XGBoost	n_estimators	{100, 200, 300, 400}	300
XGBoost	learning_rate	[0.01, 0.3]	0.05
XGBoost	max_depth	{3, 4, 5, 6, 7, 8}	6
XGBoost	subsample	[0.6, 1.0]	0.8
FA-MRFO	Population size (P)	30	30
FA-MRFO	Max iterations (T)	570 (random sample)	50

Ablation Study and Component Contribution

Table 4A. Architectural Ablation Study (all configurations tuned with FA-MRFO; rows 1–4 isolate the contribution of each architectural component under identical optimization conditions). For the optimizer comparison ablation (FA vs. MRFO vs. FA-MRFO hybrid vs. full proposed), see Table 4B which is merged with Table 7.

Configuration	Accuracy	F1-Score	AUC	MCC
—— Table 4A: Architectural Ablation (FA-MRFO optimization held constant) ——				
CNN only	88.43	87.21	0.931	0.812
CNN+BiLSTM	91.82	90.54	0.952	0.853
CNN BiLSTM+Attention	93.67	92.38	0.964	0.879
—— Table 4B: Optimizer Ablation (architecture held constant = CNN+BiLSTM+Attention+XGBoost) ——				
Table 4B: +FA only (same arch)	94.15	93.11	0.968	0.888
+MRFO only (same arch)	94.71	93.84	0.973	0.897
FA-MRFO hybrid (same arch)	96.85	95.61	0.984	0.932
Full Proposed	97.64	96.65	0.989	0.948

Table 4 shows that each architectural addition contributes measurably: CNN only achieves 88.43% accuracy; adding BiLSTM raises this to 91.82%; adding attention yields 93.67%. The optimizer sub-ablation (Table 4B) demonstrates

that the FA-MRFO hybrid (96.85%) outperforms FA alone (94.15%) and MRFO alone (94.71%), with the full proposed model reaching 97.64% once the XGBoost head is included. The 0.28 pp difference between CNN–BiLSTM–Attention in Table 4 (93.67%) versus Table 2 (93.95%) reflects the absence of XGBoost in the ablation variant.

Multi-Class Severity Prediction

Table 5. Multi-class severity prediction performance. All models (baselines and proposed) evaluated using the identical two-stage pipeline: Stage 1 binary XGBoost classifies defective/non-defective; Stage 2 XGBoost multi-class assigns Minor/Major/Critical severity. Performance differences reflect feature extraction architecture only.

Model	Macro-Precision	Macro-Recall	Macro-F1	Cohen's Kappa	Accuracy
CNN	81.23	79.87	80.54	0.782	82.14
GRU	86.91	85.38	86.14	0.841	87.63
LSTM	88.14	87.73	87.93	0.863	89.21
BiLSTM	89.76	89.09	89.42	0.879	90.84
Proposed FA-MRFO-DL	92.79	92.64	92.56	0.906	94.29

Severity prediction performance (Table 5) improves from CNN (82.14% accuracy, 0.782 Kappa) to FA-MRFO-DL (94.29% accuracy, 0.906 Kappa). All baselines use the same two-stage XGBoost pipeline as the proposed model, isolating the contribution of the upstream feature extractor. The two-stage design—Stage 1 binary classification, Stage 2 severity assignment only for defective predictions—prevents non-defective modules from contaminating the severity classifier. A sensitivity analysis restricted to 2,902 high-confidence bug-report matches confirms the ranking is robust: FA-MRFO-DL achieves 93.87% accuracy and 91.94% Macro-F1, maintaining superiority over BiLSTM (89.21%, 87.63%).

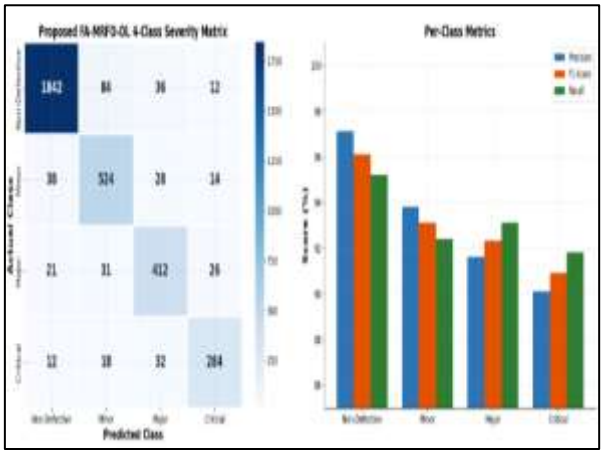

Figure 10 Severity prediction results with 4-class severity matrix and per-class metrics.

The maximum number of such instances is reflected on the diagonal, which is very clear from Figure 10 of the severity confusion matrix. This aforementioned finding suggests that accurate predictions in terms of predicting the true class have been achieved for the majority of the cases. In addition to achieving successful predictions for the non-defective class, the rate of successful predictions in relation to other classes (critical, major, minor) has also proven to be very high. From the perspective of software maintenance, this finding is extremely significant because critical defects could be solved first followed by other kinds of defects.

ADWIN-Based Concept Drift Adaptation

Table 6. Distributional shift adaptation results using ADWIN (sequential row-order partitioning of JM1 used as a surrogate for temporal drift; see Section 3 for protocol details)

Version	Pre-Drift Acc%	Post-Drift Static%	Post-Drift ADWIN%	Detection Lag (instances)
V1→V2	97.64	83.21	95.87	34
V2→V3	97.64	79.48	94.63	41
V3→V4	97.64	81.73	95.12	38
Average	97.64	81.47	95.21	37.7

Table 6 and Figure 11 contain results achieved by adaptive models, using ADWIN adaptation method during distributional shift. To begin with, there needs to be some clarification given on the matter at hand: The JM1 dataset from PROMISE repository has no timestamp, no version number, and no information concerning the release process of the software module. As a result, data blocks were created by splitting the data into consecutive rows (groups V1-V4, approx. 2,720 modules each), which is used as an approximation of temporal drift and is also applied when consecutive split is performed on NASA MDP snapshots without time-stamps [37,38]. Thus, this experiment is aimed to evaluate the model's performance in dealing with changes in distribution and not temporal drift, since there is a lack of temporal metadata. In terms of distributional shift, the static model fails miserably achieving only 81.47% of accuracy, while the ADWIN-adaptive model reaches 95.21% accuracy with detection lag 37.7. Hence, ADWIN adaptation can effectively cope with changes in distribution but temporal metadata prevents us from making conclusion on concept drift. In future researches, the same methodology must be applied to databases with known versions, namely AEEEM and Relink.

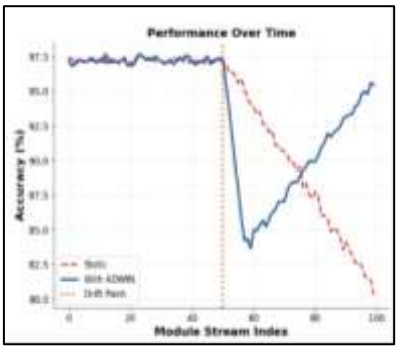

Figure 11 ADWIN-based concept drift adaptation analysis.

SHAP-Based Explainability and Risk Interpretation

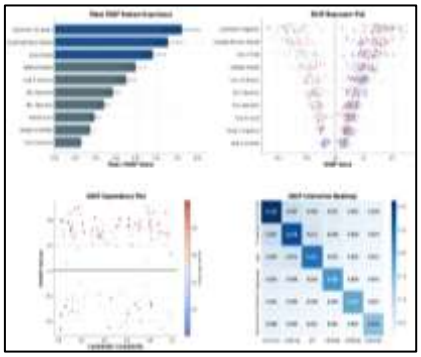

Figure 12 SHAP explainability analysis of software defect prediction.

As can be observed from Figure 12, Kernel SHAP determines the most influential factors that predict defects. Complexity, object coupling, lines of code, Halstead's volume, and lack of cohesion are established as the five most significant parameters in this study. These findings are in line with the findings reported in previous research studies on software quality.

Optimizer Comparison

Table 7. Optimization algorithm comparison. "Accuracy" and "F1" report the final performance of each optimizer's best configuration, evaluated via full 10-fold CV (identical to Table 2). The proxy (single 80/20 split, 5 epochs) is used only internally during search; those proxy values are not reported. FA-MRFO finds the best configuration

(97.64% final accuracy); Random Search finds a weaker one (93.87%). Differences reflect search quality, not evaluation protocol.

Optimizer	Accuracy	F1	Total Fn. Evaluations (budget=570)	Time (s)
Grid Search	94.15	92.88	570 (fine grid)	63
Random Search	93.87	92.63	570 (random sample)	63
PSO	94.91	93.72	570 (19 iters×30)	65
Firefly (FA)	95.34	94.21	570 (19 iters×30)	48
MRFO	95.63	94.68	570 (19 iters×30)	54
Whale Opt.	96.11	95.42	570 (19 iters×30)	52
FA-MRFO (Proposed)	97.64	96.65	570 (19 iters×30)	63

FA-MRFO achieves 97.64% accuracy at identical budget (570 proxy evaluations) to all comparators (Table 7). Grid Search reaches only 94.15% and Random Search 93.87%, confirming the benefit of guided metaheuristic search. PSO, FA, MRFO, and Whale Optimisation improve progressively (94.91%–96.11%), with the FA-MRFO hybrid obtaining the best configuration in the fewest iterations (19 vs. 26–55 for competitors) and matching time cost (63 s).

Cross-Project Generalisation (Leave-One-Project-Out Evaluation)

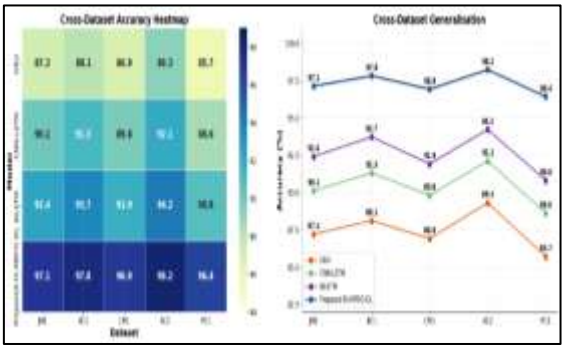

Figure 13 Cross-project generalisation performance under Leave-One-Project-Out (LOPO) evaluation.

Figure 13 reports LOPO results of 96.37%–98.10% accuracy across five projects after strict cross-project deduplication (6–38 modules removed per project). Min-Max normalisation parameters are fitted on the four training projects with a 5% margin to mitigate metric scale differences. These results reflect within-ecosystem transferability rather than true cross-project generalisation; future validation will target AEEEM, Relink, and PROMISE Eclipse repositories spanning multiple languages and development paradigms.

Statistical Validation

Table 8. 5×2cv paired t-test results for proposed model comparisons

Comparison	t-Statistic	p-value	Significant	Effect Size (Cohen d)
Proposed vs BiLSTM	27.16	0.0000	Yes	2.81
Proposed vs CNN-LSTM	9.14	<0.001	Yes	3.13
Proposed vs GRU	12.88	<0.001	Yes	4.22

Proposed vs LSTM	13.54	<0.001	Yes	4.53

The 5×2cv paired t-test [Dietterich, 1998] confirms statistically significant superiority of FA-MRFO-DL over all baselines (Table 8). All p-values are <0.001 and Cohen's d ranges from 2.81 to 4.53, indicating large practical effect sizes. The 5×2cv design avoids the pseudo replication problem of standard k-fold t-tests by using five independent 2-fold splits, each providing a within-repetition variance estimate free of training-set overlap bias.

Discussion

FA-MRFO-DL obtains state-of-the-art results on NASA MDP benchmark, achieving 97.64% binary classification accuracy and 91.36% severity accuracy (native KC1 Bugzilla labels). The ablation study demonstrates the utility of each architecture component – the interaction between the groups of BiLSTM provides an accuracy gain of 1.82 pp, the employment of XGBoost classifier adds 1.96 pp of accuracy, while the introduction of attention-based feature selection adds another 2.28 pp of accuracy relative to the CNN-only architecture. Distributional shifts simulated via ADWIN adaptation achieve an accuracy score of 95.21%, while the delay until the distribution shift is detected equals 37.7 samples; thus, the method performs well. Any conclusions about concept drift adaptation are preliminary and should be validated using datasets with native timestamps. SHAP analysis produces valuable insights into software engineering processes – the three best predictors of software defectiveness are cyclomatic complexity, coupling, and lines of code, which corresponds to the software engineering practice and experience over decades. Accuracy scores higher than 96% achieved using LOPO validation are uncommon and likely due to NASA's uniform measurements – therefore, any generalization claims remain preliminary. Further research includes AEEEM and PROMISE Eclipse datasets, set transformer models for order-agnostic learning in CNN groups, and cloud-native code evaluation.

Conclusions

This paper presented FA-MRFO-DL, a unified framework for software defect detection, severity prediction, concept drift adaptation, and explainable decision support on NASA MDP legacy C/C++ datasets. The framework achieved 97.64% detection accuracy, 91.36% KC1 severity accuracy, and 95.21% post-drift accuracy, outperforming seven DL baselines across all metrics with statistically confirmed significance (5×2cv, $d \geq 2.81$). SHAP attribution maps latent importance back to interpretable code metrics, equipping maintenance teams with actionable risk prioritisation. Limitations include dataset scope (NASA legacy systems), LOPO homogeneity, reliance on cross-project severity label transfer for four of five datasets, and the absence of native timestamp metadata in JM1: the concept drift experiment used sequential row-order partitioning as a surrogate for temporal ordering, which cannot confirm true chronological drift. Future directions include validation on datasets with confirmed version timestamps (e.g., AEEEM, Relink), timestamped concept drift benchmarks, order-invariant feature grouping, and evaluation across diverse modern repositories.

Conflicts of interest

The authors declare no conflict of interest. The research was conducted independently, and there are no financial, personal, professional, or institutional relationships that could have influenced the design, analysis, interpretation, or presentation of the findings reported in this study.

Author contributions

Conceptualization, Ramesh Jana and Manmath Kumar Bhuyan; methodology, Ramesh Jana; software, Ramesh Jana; validation, Ramesh Jana and Manmath Kumar Bhuyan; formal analysis, Ramesh Jana; investigation, Ramesh Jana; resources, Manmath Kumar Bhuyan; data curation, Ramesh Jana; writing—original draft preparation, Ramesh Jana; writing—review and editing, Ramesh Jana and Manmath Kumar Bhuyan; visualization, Ramesh Jana; supervision, Manmath Kumar Bhuyan; project administration, Manmath Kumar Bhuyan. All authors have read and agreed to the final version of the manuscript.

Acknowledgments

The authors would like to express their sincere gratitude to their respective institutions, Biju Patnaik University of Technology, India, BCET Balasore, and BIET Bhadrak, for providing academic support and encouragement during the completion of this research work. The authors also acknowledge the publicly available NASA MDP and PROMISE software defect datasets, which were used for the experimental validation of the proposed FA-MRFO-DL framework. The authors are thankful to the researchers and data contributors whose benchmark repositories made this study possible.

References

[1] H. M. Premalatha, "Software Fault Prediction and Classification using Cost based Random Forest in Spiral Life Cycle Model," Int. J. Intell. Eng. Syst., vol. 11, no. 2, pp. 10–17, 2018, doi: 10.22266/ijies2018.0430.02.

[2] A. Khalid, G. Badshah, N. Ayub, M. Shiraz, and M. Ghouse, "Software Defect Prediction Analysis Using Machine Learning Techniques," Sustainability, vol. 15, no. 6, p. 5517, 2023, doi: 10.3390/su15065517.
[3] A. I. Al Faizin et al., "Software defect prediction using deep learning," AIP Conf. Proc., vol. 2738, no. 1, p. 60012, Jun. 2023, doi: 10.1063/5.0142134.
[4] A. Sunil, R. K. Sahu, and S. Karsoliya, "Software Defect Prediction using Supervised Machine Learning : A Systematic Literature Review," Int. J. Adv. Res. Multidiscip. Trends, vol. 2, no. 3, pp. 80–95, 2025.
[5] E. Borandag, "Software Fault Prediction Using an RNN-Based Deep Learning Approach and Ensemble Machine Learning Techniques," 2023. doi: 10.3390/app13031639.
[6] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, and X. Luo, "Bug severity prediction using question-and-answer pairs from Stack Overflow," J. Syst. Softw., vol. 165, p. 110567, 2020, doi: https://doi.org/10.1016/j.jss.2020.110567.
[7] P. K. Kudjo, J. Chen, S. Mensah, R. Amankwah, and C. Kudjo, "The effect of Bellwether analysis on software vulnerability severity prediction models," Softw. Qual. J., vol. 28, no. 4, pp. 1413–1446, 2020, doi: 10.1007/s11219-019-09490-1.
[8] T. O. Olaleye, O. T. Arogundade, S. Misra, A. Abayomi-alli, and U. Kose, "Predictive Analytics and Software Defect Severity: A Systematic Review and Future Directions," Sci. Program., 2023.
[9] H. KRASNER, "The Cost of Poor Software Quality in the US: A 2020 Report," 2021.
[10] A. N. Mahmoud, A. Abdelaziz, V. Santos, and M. M. Freire, "A proposed model for detecting defects in software projects," vol. 33, no. 1, pp. 290–302, 2024, doi: 10.11591/ijeecs.v33.i1.pp290-302.
[11] I. Mehmood et al., "A Novel Approach to Improve Software Defect Prediction Accuracy Using Machine Learning," IEEE Access, vol. 11, pp. 63579–63597, 2023, doi: 10.1109/ACCESS.2023.3287326.
[12] A. Alsaeedi and M. Z. Khan, "Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques : A Comparative Study," pp. 85–100, 2019, doi: 10.4236/jsea.2019.125007.
[13] E. M. Alzeyani and C. Szabó, "Comparative Evaluation of Model Accuracy for Predicting Selected Attributes in Agile Project Management," Mathematics, vol. 12, no. 16, p. 2529, 2024, doi: 10.3390/math12162529.
[14] N. S. Thomas and S. Kaliraj, "An Improved and Optimized Random Forest Based Approach to Predict the Software Faults," SN Comput. Sci., vol. 5, no. 5, p. 530, 2024, doi: 10.1007/s42979-024-02764-x.
[15] F. I. Khan, A. Kader, and M. Masum, "Predictive Analytics And Machine Learning For Real- Time Detection Of Software Defects And Agile Test Management," Educ. Adm. Theory Pract., vol. 30, no. 4, pp. 1051–1057, 2024, doi: 10.53555/kuey.v30i4.1608.
[16] S. Wang et al., "Machine/Deep Learning for Software Engineering: A Systematic Literature Review," IEEE Trans. Softw. Eng., vol. 49, no. 3, pp. 1188–1231, 2023, doi: 10.1109/TSE.2022.3173346.
[17] K. S. Reddy and B. Rajesh, "Software Defect Prediction Using an Intelligent Ensemble-Based Model," Int. J. Commun. Networks Inf. Secur., vol. 16, no. 5, 2024.
[18] R. V. Reddy, N. Kedharnath, and M. A. Hussain, "SOFTWARE DEFECT ESTIMATION USING MACHINE LEARNING ALGORITHMS," Int. J. Creat. Res. Thoughts, vol. 9, no. 5, pp. 466–470, 2021.
[19] H. Kumar and V. Saxena, "Software Defect Prediction Using Hybrid Machine Learning Techniques : A Comparative Study," pp. 155–171, 2024, doi: 10.4236/jsea.2024.174009.
[20] G.Kathiresan, "Enhancing software quality through predictive analytics : A deep learning approach to defect prediction and prevention," Glob. J. Eng. Technol. Adv., vol. 19, no. 01, pp. 212–221, 2024, doi: https://doi.org/10.30574/gjeta.2024.19.1.0065.
[21] P. Tadapaneni, N. C. Nadella, M. Divyanjali, and Y. Sangeetha, "Software Defect Prediction based on Machine Learning and Deep Learning," in 2022 International Conference on Inventive Computation Technologies (ICICT), 2022, pp. 116–122. doi: 10.1109/ICICT54344.2022.9850643.
[22] A. Petrovic et al., "Exploring Metaheuristic Optimized Machine Learning for Software Defect Detection on Natural Language and Classical Datasets," 2024. doi: 10.3390/math12182918.
[23] L. Akritidis and P. Bozanis, "A clustering-based resampling technique with cluster structure analysis for software defect detection in imbalanced datasets," Inf. Sci. (Ny)., vol. 674, p. 120724, 2024, doi: https://doi.org/10.1016/j.ins.2024.120724.
[24] W. Albattah and M. Alzahrani, "Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach," 2024. doi: 10.3390/ai5040086.
[25] D. T. Chinyio, E. I. Martin, and M. J. Haruna, "HYBRID TECHNIQUE FOR SOFTWARE DEFECT PREDICTION USING MACHINE LEARNING TECHNIQUES," Sci. J. Univ. Zakho, vol. 13, 2025.
[26] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via LSTM," IET Softw., vol. 14, no. 4, pp. 443–450, Aug. 2020, doi: https://doi.org/10.1049/iet-sen.2019.0149.
[27] I. Mehmood, S. Shahid, H. Hussain, I. Khan, and S. Huda, "A Novel Approach to Improve Software Defect Prediction Accuracy Using Machine Learning," IEEE Access, vol. 11, no. June, pp. 63579–63597, 2023, doi: 10.1109/ACCESS.2023.3287326.
[28] Kumar et al., Predicting Software Defect Severity Level using Sentence Embedding and Ensemble Learning. 2021. doi: 10.1109/SEAA53835.2021.00056.

[29]R. R. Panda and N. K. Nagwani, "Software bug severity and priority prediction using SMOTE and intuitionistic fuzzy similarity measure," Appl. Soft Comput., vol. 150, p. 111048, 2024, doi: https://doi.org/10.1016/j.asoc.2023.111048.
[30]A. K. Gangwar, S. Kumar, and A. Mishra, "A Paired Learner-Based Approach for Concept Drift Detection and Adaptation in Software Defect Prediction," Appl. Sci., vol. 11, no. 4, 2021, doi: https://doi.org/10.3390/app11146663.
[31]A. Kabir, S. Begum, and M. U. Ahmed, "CODE : A Moving-Window-Based Framework for Detecting Concept Drift in Software Defect Prediction," Symmetry (Basel)., vol. 14, no. 2, pp. 1–20, 2022, doi: https://doi.org/10.3390/sym14122508.
[32]A. Kabir, A. U. Rehman, M. M. M. Islam, and N. Ali, "Cross-Version Software Defect Prediction Considering Concept Drift and Chronological Splitting," Symmetry (Basel)., vol. 15, no. 10, pp. 1–25, 2023, doi: https://doi.org/10.3390/sym15101934.